Implementation of an Activity Coordination System

Robert Morris
Software Options, Inc.
22 Hilliard Street
Cambridge, Mass. 02138

Abstract

An activity coordination system helps its users describe, guide, and monitor processes in which they are involved. The activity coordination system described here is based on a formalism suited to the modular description of complex distributed activities. An activity, once described, may be executed as a distributed computation, organizing the interaction of the activity's participants. The system supports activities that last a long time, allows them to be examined as they execute, and may be extended to provide for new types of activities.

1 Introduction

This paper[1] describes the status of a project to provide computer-based support for the coordination of human activities. We aim to describe the structure of activities, not their content. For instance, we can model "wait until the Project Manager and the Design Engineer agree on a design," but not "produce a design that meets some given requirements." Such *activity coordination* plays a central role in the Knowledge Based Software Assistant [3] model.

As an example activity, consider an auction. The auctioneer opens bidding and then listens for bids from the participants. He remembers the current high bid, and whenever he receives a bid that is higher, he announces it to all the participants. The participants listen to the auctioneer's announcements, and periodically make bids. At some point the auctioneer announces that the auction is over.

We would like to be able to describe such an activity in an intuitive way. Once described, we would like to automate the mundane tasks of moving information around and deciding what happens next. This automation should not rely on the continuous operation of any single computer, since activities may last for weeks or months. We would like the coordination to be an integral and natural part of users' daily activities. We would like to be able to examine and modify the structure of an ongoing activity without interrupting it. These activities are often geographically distributed, and we would like the system to provide transparent communication. Finally, we would like the system to be extensible.

Section 2 of this paper describes the abstract model underlying this work. As an example, section 3 shows how to describe an auction with the model. Sections 4 and 5 describe how we implement the model. Section 6 describes how we describe activities and how they are initialized, and section 7 outlines our plans for facilities to monitor ongoing activities. Section 8 describes how the implementation may be extended by users. Section 9 reviews related work.

2 Transaction Graphs

We describe activities using a formalism called *transaction graphs*. See [4] for a complete description and motivation for the formalism; this section relates only what is needed to understand the implementation.

A transaction graph consists of a graph of nodes connected by undirected arcs. Each node has a state consisting of arbitrary data. Each node exposes a value to the node at the other end of each of its arcs. At each point in time, the state of the graph as a whole consists of the state of each node and the value exposed in each direction over each arc.

Each node has a *transition function* associated with each arc incident on it. The function for a particular node and arc maps the node's state and the value it sees from the arc to a new state and a new value to

[1] This work was supported by a Phase II SBIR contract (contract number N66001-88-C-7008), with funds provided by the Naval Ocean Systems Center, and is based on work funded by Rome Laboratory, contract number F30602-87-D-092.

expose along the arc. It may not change the value exposed along any other arc. We call the process of changing the state and exposing a new value along an arc a *transaction* along that arc.

Each transaction occurs instantaneously in the transaction graph model, so that the graph as a whole proceeds serially from one state to the next. This helps keep the model simple at the expense of some complexity in the implementation, as described in later sections.

As described in [4], a single transaction graph node can do the work of any collection of nodes. We call such nodes *condensation* nodes. This invites hierarchical specification of graphs, a subject elaborated in section 6.

The transaction graph formalism lends itself to distribution by simplifying communication and avoiding centralized components. It provides a simple programming model, and it allows natural division of large systems into subgraphs of nodes connected with arcs.

3 Auction Example

The auction example outlined above can be implemented as a transaction graph with a node for the auctioneer, a node for each participant, and an arc connecting each participant to the auctioneer.

Let the auctioneer node number its arcs from zero. The auctioneer's state consists of a boolean flag indicating whether the auction is over, the value of the highest bid seen so far, and the number of the arc from which the high bid came. The flag starts out as false, the highest bid starts out as zero, and the arc number of the high bid starts out as -1. The auctioneer will expose two boolean flags and a number along each arc, and expects to see a number from each arc.

The auctioneer has the same transition function on each arc. This function interprets the number it sees from each arc as a bid. If the bid on arc i is higher than the highest bid seen so far, and the auction is not yet over, the transition function changes the state to reflect the new high bid and arc number. The value the transition function exposes along each arc i consists of a boolean flag which is true only if the highest bid so far came from arc i, a boolean which is true when the auction is over, and the value of the highest bid seen so far.

Each participant node has a single arc leading to the auctioneer and no state. Each participant exposes a number along its arc, and expects to receive two boolean flags and a number from the arc. The transition function can tell when the auction is over, whether the node is the high bidder, and the value of the highest bid by looking at the value it sees from the arc. If the auction is not over, and this node is not the highest bidder, the transition function may choose to bid by exposing along the arc a value greater than the current highest bid.

A distributed implementation of this example must face the race condition in which the auctioneer closes the bidding, but one of the participants makes another bid before receiving the auctioneer's announcement. In many cases, including this, our system may be programmed as if transactions occurred instantaneously. Section 5 explains how our implementation maintains this illusion as well as situations in which the illusion breaks down.

4 Execution

A transaction graph is executed by repeatedly calling each node's transition functions, updating the state and propagating arc values after each call. Our implementation of the transaction graph model therefore addresses the management of node state, the scheduling of transition function calls, and communication. This section describes the execution of single nodes, and the next the communication issues raised by executing multiple nodes concurrently.

Most of the functionality described in this section and the next is provided by *activation* programs. An activation program contains transition functions and code to manage nodes. A single activation process can manage more than one node at a time; communication among these nodes is efficient since it involves no network traffic.

If activations were ordinary processes running under some operating system, they would consume considerable resources for the duration of an activity and be vulnerable to host shutdowns. To avoid this we use a facility called the Very Long Execution Mechanism, or VLEM [11], which provides persistent execution under UNIX. The VLEM allows computers that can share disk files to be grouped into *sites*. A VLEM server always runs on each computer at each site, and the servers of a site cooperate in managing a number of *long-running processes*. Each long-running process has a program associated with it. A long-running process is *awake* if that program is running; in this case, the VLEM keeps track of how to contact the running program. A long-running process can ask the VLEM to put it to sleep: the process supplies the VLEM with its state before it exits, and the VLEM records that state in a shared disk file. A long-running process that is asleep consumes no resources other than the recorded state. Long-running processes at a site may exchange messages via the VLEM servers, or receive them from programs outside the VLEM. When a message arrives for a sleeping process, the VLEM starts up the program, supplies it with its recorded state, and gives it the message and any more that arrive while the process is awake. The VLEM provides a degree of reliability at sites with more than one computer by allowing a long-running process to wake up on a computer other than the one on which it last went to sleep.

The nodes of an executing transaction graph may be spread over a number of sites. The information about each node is stored in the state of some long-running process. The program for each such long-running process is the activation program mentioned above. If a site has more than one type of computer, there must be an activation program for each.

When a node exposes a new value over an arc, the value is sent to the node at the other end of the arc in one of three ways. If the target node is managed by the same long-running process as the source, that process only needs to update internal tables to communicate the new value. If the target node is managed by some other long-running process at the same site, the new value is sent in a message through the VLEM servers. A message to a node at some other site is sent to any VLEM server at that site by any available network facility. Section 5 discusses how the implementation achieves reliable communication even when it cannot wholly depend on the networks it uses.

The transaction graph formalism does not specify the order in which transition functions are called, and more than one change to the state or incoming arc values may occur between successive calls to a transition function. Transition functions must be designed with this in mind. The implementation only guarantees that if a node's state changes, all the node's transition functions will eventually be called, and if an incoming arc value changes, the relevant transition function will eventually be called. The actual procedure is that an activation cycles through all of a node's transition functions until it completes a full cycle in which no function proposes a change to the node's state. If an activation controls more than one node, it interleaves the cycles for the different nodes. Running activations also periodically check the VLEM system to see if they have any messages waiting. When a running activation has no more messages to receive and all the nodes it manages are idle, it supplies the state information about its nodes to the VLEM and asks to be put to sleep.

5 Distribution

As mentioned above, a transaction graph system can be distributed: the data describing the graph, and the execution of the graph, may be divided among a number of networked computers. This allows geographically distributed users to participate in the same activities, and also lets the system scale to large sizes without centralized bottlenecks. Distribution comes at the cost of some complexity in the implementation, since the transaction graph formalism assumes instantaneous reliable communication.

Network messages are not reliable. Most applications can make use of protocols, such as TCP/IP [9], that provide reliable transport in the face of unreliable networks. But these protocols usually store state in the host operating system's transient memory, and thus consume resources and are vulnerable to host shutdowns.

Since activities may have long lives, the implementation provides its own reliable transport mechanism. Activations resend each message until they receive an acknowledgment; they record the state required to resend and filter out duplicate messages in the long-running process state along with other information about nodes. The implementation currently uses TCP/IP as the underlying network protocol, but it could use a less reliable medium such as electronic mail instead.

Another impact of distribution is that messages take time to be delivered, which can give rise to inconsistent computations. Consider the case raised in the auction example, in which a bidder raises his bid at the same time that the auctioneer closes the bidding. Each sends its message, and then receives the message from the other. They will then disagree about the order of events: the bidder thinks he bid before the close, and the auctioneer thinks he closed the auction before the bid.

In order to avoid such problems, the implementation delivers the messages along each arc serially. If a node receives a message from an arc on which it has sent a message, but not yet received an acknowledgment, a violation of serialization has occurred. The node at the other end of the arc will see a similar situation. In order to preserve serial communication over the arc, one of the nodes will ignore the other's message; they use a fair algorithm to decide which without further communication. The node whose message is ignored must also retract the state change made by the transition function as part of the transaction. To avoid actually retracting state changes, the transition functions do not modify the state directly, but propose changes with *deltas*.

The delta mechanism is an extension to normal programming language type systems that allows changes to values to be described. We provide deltas as part of a library of functions that implement a run-time type system for C. When one of a node's transition functions starts a transaction along an arc, it provides a delta to the node's state. The activation sends the new arc value to the destination node, remembers the proposed delta, and has the option of going to sleep until it receives an acknowledgment from the destination node. If the transaction succeeds, the delta is applied to the node's state, yielding the desired new state. If the transaction is canceled, because the node at the far end of the arc started a simultaneous transaction, the delta is discarded.

The implementation cannot provide the illusion of instantaneous transactions if a transition function interacts with the world outside the transaction graph, because events in the real world usually cannot be retracted. A transition function that affects the outside world should modify its node's state without proposing a new arc value; such a change will never be retracted since no arc transaction is involved.

If the implementation waited for each arc transaction to be acknowledged before allowing a node to continue with other transactions, some concurrency within the node would be lost. The auctioneer node, for instance, could announce the latest high bid to all the participants at the same time if it did not have to wait for each acknowledgment before proceeding to send to the next participant. If no transaction were ever canceled, the values from any sequence of transactions on the different arcs of a node could be exposed concurrently. The state resulting from the concurrent execution would be the state produced by the last transaction in the corresponding sequential execution. However, if one of a sequence of transactions *is* canceled, the transaction's changes to the state will be omitted, so subsequent transactions may see a different state than if it had not been canceled, and thus may expose different arc values. Once a group of transactions has been started concurrently, it is too late to change the exposed values. Thus any sequence that is to be run concurrently must be such that canceling any subset of the transactions leaves a sequence that exposes the same values on the remaining arcs. This would not be true if one of the transactions changed the state in a way that affected the exposed value computed by a later transaction. The implementation could check all the different combinations of canceled transactions before letting a group of them proceed concurrently, but the complexity of this grows rapidly with the number of transactions. Instead, we sacrifice some of the simplicity of the transaction graph model in return for efficient concurrency.

The implementation requires that transition functions never change any information in the state to which a concurrent transaction might refer, thus allowing concurrency but avoiding expensive computation. A transition function cannot directly determine whether any transactions from its node are outstanding, but in practice it has proved easy to deduce this information. For instance, the auctioneer might set a flag in the state for each arc as part of the transaction announcing the latest high bid along that arc; it would defer altering the state to reflect a new higher bid until all the flags were true, and thus until all the transactions were complete.

Transition functions specify state changes with deltas, which simplifies the resolution of the multiple new states proposed by concurrent transactions; the implementation just applies the deltas to the original state. Deltas also make it easier to write transition functions that avoid altering the state in ways that conflict with concurrent transactions. Most deltas make relative changes to parts of the state, rather than just replacing them: one kind of delta, for instance, adds an increment to an integer in the state. This could be used if the state held a bank balance and transactions represented transfers of money; the transactions could run concurrently by using incremental deltas despite affecting the same part of the state.

We also use deltas to decrease our use of network resources. The messages along an arc are implemented as a series of deltas to a value maintained by the nodes at either end, and the instantiation process (see the next section) ensures that both nodes start with the same value.

6 Activity Descriptions

Transaction graphs provide an abstraction mechanism to help people who wish to describe activities with transaction graphs.

A *parameterized activity description* is a named function that takes arbitrary parameters as arguments and produces a piece of a transaction graph. Some arcs may *dangle* outside this subgraph. A *primitive* activity yields a subgraph containing a single transaction graph node, and a *complex* activity yields a subgraph with multiple nodes. A primitive parameterized activity description describes a transaction graph node by supplying its transition functions and initial state. Any node in the subgraph provided by a complex activity may refer to another parameterized activity description, specifying its name, some parameters to pass to it, and the network name of a site at which to execute the activity.

Parameterized activity descriptions correspond to condensation nodes, and thus can be used without understanding their internal structure. A parameterized activity description is much like a procedure: larger activity descriptions that want to make use of it need only understand how to pass it parameters and how to use the dangling arcs of the resulting transaction graph. The parameter can be used for any purpose; for instance, an activity description for an auction could take the number of bidders as one of its parameters.

An activity description with no dangling arcs may be *instantiated* to form an executable transaction graph. The input to the instantiation process is an activity description and some parameters. Instantiation involves expanding subactivities, by calling their activity descriptions, until all nodes are primitive. The result is a transaction graph that can be executed as described in section 4. At each level of expansion, parameters are passed to subactivities, and the dangling arcs of subactivities are connected as specified by the surrounding activity.

Instantiating a large transaction graph may take a long time and involve many computers; we simplify the problem by using transaction graphs. A graph of *supervisor* nodes performs instantiation, calling the parameterized activity description functions and maintaining databases describing currently instantiated transaction graphs. The supervisor graph has one node at every site that might take part in an activity, and must be built by hand when the activity coordination system is first installed. Each supervisor node listens for instantiation requests from the outside world, as well as messages from other supervisors. Suppose the supervisor on computer a is instantiating a parameterized activity description that requests that some subactivity execute on computer b. The supervisor on a will send a message to the supervisor on b containing the name of the subactivity and the parameters. At some later time, the supervisor on b will return a message describing the dangling arcs of the subactivity. Meanwhile, the supervisor on a might instantiate other subactivities, collecting dangling arc information. When it is ready, the supervisor on a will send messages to the supervisor on b telling it where to connect its dangling arcs.

One transaction graph can create another by sending a message to a node in the supervisor graph. However, since an executing transaction graph has a fixed form and cannot have dangling arcs, the existing graph and the new graph cannot communicate with arcs. We will relax this restriction in the future by providing a *cutover* mechanism to replace parts of an executing transaction graph. For now, one graph can watch the other using the viewing mechanism described in the next section.

Parameterized activity description functions are compiled into activation programs, just like transition functions, so that the supervisor nodes can call them. This requirement is less strict than it might seem, because we plan to provide a standard activity description that takes a description of a subgraph as its parameter, and instantiates that subgraph.

7 Viewing

We plan that the nodes of a graph may be *viewed*, which means that the activation controlling the node sends updates to any observers when the node's state changes. The purpose of views is to let users monitor the progress of activities, to let one activity audit another, to record the history of an activity, and to recover from failures. These views will be subject to projection, which will summarize the state of subgraphs, supplying only the desired relevant information.

Viewers of a large transaction graph may want to see it in terms of the hierarchy of parameterized activity descriptions with which it was originally described. This hierarchy is not needed to execute a graph, so the supervisor nodes record it separately during instantiation for later use by viewers.

Some applications of viewing require that views be consistent: despite concurrency and delays in communication, changes to a view should always be presented in an order in which they might have happened. A partial order may be imposed on all the arc transactions in an executing transaction graph in the following way. Within each node, transactions that are not concurrent are ordered as they occur. Concurrent transactions, as described in section 5, always have effects consistent with some sequential order. Note that any transaction that exposes a new arc value occurs in the orders of the nodes at both ends of the arc. A partial order for the whole graph is the union of the orders for each node. We plan to provide a distributed implementation of this scheme to aid viewing.

8 Extensibility

The transaction graph system can be used at three levels. Ordinary users will view and interact with activities that are already under way. The implementation uses the facilities of E-L [1] to allow executing nodes to interact with users.

The designer of an activity assembles it from an existing library of activities. We plan to provide a visual graph building tool to assist this assembly.

At the lowest level, the library of primitive activities can be extended with transition functions written in C. Each new activity requires a parameterized activity description. Each new primitive activity also requires some transition functions. These functions can provide arbitrary functionality, such as accessing databases. We have developed a run-time library to support these functions and aid them in conforming to the transaction graph formalism. The delta mechanism described in section 5 is included in this library, and may also be extended.

A separate paper [5] describes a transaction graph with extensions that helps coordinate a software development cycle. It uses *procedure-based* activities to support a data-flow-like programming style; these activities wait for a message from each incoming arc, call an arbitrary procedure with these messages as arguments, and send the procedure's results over outgoing arcs.

9 Related Work

Our work shares with that of Osterweil [8] the goal of describing and enacting activities. Osterweil, with Taylor et al [10], uses extensions to Pascal-like languages to describe activities, thereby taking advantage of existing software development tools and expertise. We take a more visual approach that we hope will clarify the structure of complex activities and aid distribution.

Kellner [7] advocates a visual model along with a number of other requirements for activity coordination. Our work will fulfill many of them: it provides some useful viewpoints of activities, supports multiple levels of abstraction, offers formally defined constructs that are computable and capable of being analyzed, and allows descriptions to be reused.

Katayama [6] uses a functional programming language to describe activities. A function describes each activity, and subactivities may be invoked with function calls. Function arguments and return values allow communication between activities. Functions may be evaluated as soon as their arguments are ready and have little access to data other than their arguments, so that activities can often be scheduled concurrently. Like ours, this model encourages hierarchical activity descriptions. Unlike ours, it imposes this hierarchy on the communication of the executing activity, and makes it difficult for an activity to maintain long-term state.

Barghouti and Kaiser [2] use rules to describe activities and a shared database to communicate and maintain state. A sequence of rules may be *fired* to execute an activity. Firing a rule involves checking a precondition on the database, performing some action in the outside world, and then changing the database. Barghouti and Kaiser propose to make their system concurrent. They lock items in the shared database to detect conflicts between the rule sequences of concurrent activities. When a conflict occurs, a separate set of rules decides whether the conflict is permissible, and if not, how to repair it. Our work lacks a convenient shared database and distributes more naturally for that reason; we believe this will be appropriate for large systems.

10 Conclusion

We are developing an activity coordination system based on the transaction graph formalism. This system allows users to describe activities by building graphs with simpler activities as nodes, and with arcs indicating communication between subactivities. The graphs are then executed on a network of computers, and can interact with users as the activity progresses. The transaction graph system is extensible, and can accommodate activities that last a long time and are geographically distributed.

References

[1] "E-L User's Manual", Tech. Report, Software Options, Inc., continuously updated.

[2] N. Barghouti and G. Kaiser. "Multi-Agent Rule-Based Software Development Environments," in the *Proceedings of the Fifth Conference on*

Knowledge-Based Software Assistant, 1990, sponsored by Rome Air Development Center, Griffiss Air Force Base, New York.

[3] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich. "Report on a Knowledge-Based Software Assistant", Final Technical Report RADC-TR-83-195, Rome Laboratory, Griffiss Air Force Base, NY 13441, August 1983.

[4] M. Karr. "Transaction Graphs: A Sketch Formalism for Activity Coordination", RADC-TR-90-347, December 1990.

[5] M. Karr and T. Cheatham. "A Solution to the ISPW-6 Software Process Modeling Example", Tech. Report, Software Options, Inc., September 1990.

[6] T. Katayama. "A Hierarchical and Functional Software Process Description and its Enaction," in the *Proceedings of the 11th International Conference on Software Engineering*, IEEE, 1989.

[7] M. Kellner. "Representation Formalisms for Software Process Modeling," in the *Proceedings of the Fourth International Software Process Workshop*, May 1988.

[8] L. Osterweil. "Software Processes are Software Too," in the *Proceedings of the 9th International Conference on Software Engineering*, IEEE, 1987.

[9] J. Postel. "Transmission Control Protocol", RFC 793, SRI Network Information Center, Menlo Park, CA, September 1981.

[10] R. Taylor, F. Belz, L. Clarke, L. Osterweil, R. Selby, J. Wileden, A. Wolf, and M. Young. "Foundations for the Arcadia Environment Architecture," in the *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1988.

[11] A. Zatsman and M. Karr. "Very Long Execution Mechanism," Tech. Report, Software Options, Inc., September 1989.