An Activity Coordination System

An Activity Coordination System

Michael Karr
Thomas E. Cheatham

May 21, 1993

Harvard University
and
Software Options, Inc.
Cambridge, Mass. 02138

Contents

1 Introduction

This paper presents an overview of an activity coordination system that is currently operational on unix based systems. The purpose of this system is to support combinations of people and computers engaged in a wide variety of activities. Some of the activities that the system has supported are:

- An editor/reviewer activity in which a proposed reviewer accepts or declines some review task and, having accepted, eventually completes the review or else decides not to do it.
- Modification and testing of some program module with respect to some proposed set of changes and acceptance of the result by a change control board.
- Release of a software system.
- Generation of a request for proposals.

Activities may take place over long periods of time, sometimes months or years. Activities may also take place over large amounts of space in the sense that the participants may be at sites scattered world wide. And, the communication media may vary widely, from using local area networks, internets, dial-up, e-mail, and so on. An important feature of the activity coordination system is its open architecture that, over time, will accomodate new data types, hardware architectures, protocols, and comunication media.

Central to the system is the so-called *activity description* that specifies the various aspects of some activity and, when *instantiated*, produces a program or programs whose execution enacts the activity specified, providing communication and coordination among the several participants in the activity and dealing with the persistence of the activity over time.

The next section discusses the simple editor/reviewer activity and is followed by a section containing a detailed presentation of the activity description for this activity. Following this, we provide a brief discussion of *transaction graphs*, the formalism that underlies activity descriptions and provides the basis for the implementation of the activity coordination system. We then discuss some of the implementation issues that have been addressed. The section following then describes how extensions to the system are accomodated and we close with a discussion of the current status of the system and some future plans.

2 An Example — The Editor/Reviewer Activity

Consider an activity that involves an editor who wishes to have some document reviewed and a reviewer who is asked to review that document. When the activity is initiated, the

first issue is whether the reviewer accepts the reviewing task or declines it. If the reviewer's response is to decline the review, the editor is informed and that is the end of the activity. If the reviewer's response is to accept the review, the editor is informed and the reviewer then eventually does one of the following:

- The reviewer completes the review, so informing the editor, or
- the reviewer decides that doing the review is impossible, and the editor is informed of this decision.

Before the reviewer responds, the reviewer may receive a reminder that the review is due, to which the reviewer is to respond with an estimate of when the review will be completed. Additionally, the editor may decide to cancel the review before the reviewer responds and in this case the activity terminates.

From the editor's point of view, the first issue is whether the reviewer accepts or declines doing the review. If the latter, the activity is over (and presumably a new activity with another reviewer will ensue). If the reviewer accepts, the editor is informed and the due date is noted so that the reviewer can be reminded if necessary.

Several things can then transpire:

- The due date arrives and a reminder is sent to the reviewer.
- The editor sees that the review is done or that the reviewer has quit and will not do the review. If either event transpires the activity terminates.
- The editor gets a response from the reviewer after the reviewer is reminded that the review is due and resets the due date.
- Finally, the editor can decide to cancel the review by this particular reviewer and the activity then terminates.

The overall activity takes three parameters, the names of the editor and reviewer and the due date of the review.

3 Activity Descriptions

An *activity description* plays several roles in the activity coordination system:

- It is a program whose execution models the enactment of the activity described,
- it is a window through which the current state of an activity in progress may be viewed, and

- it is a window through which a participant in an activity may interact with that activity.

Visually, an activity description takes the form of a graph whose nodes are sub-activities and whose arcs depict the sorts of communications that may take place among the sub-activities.

3.1 The Reviewer's Activity

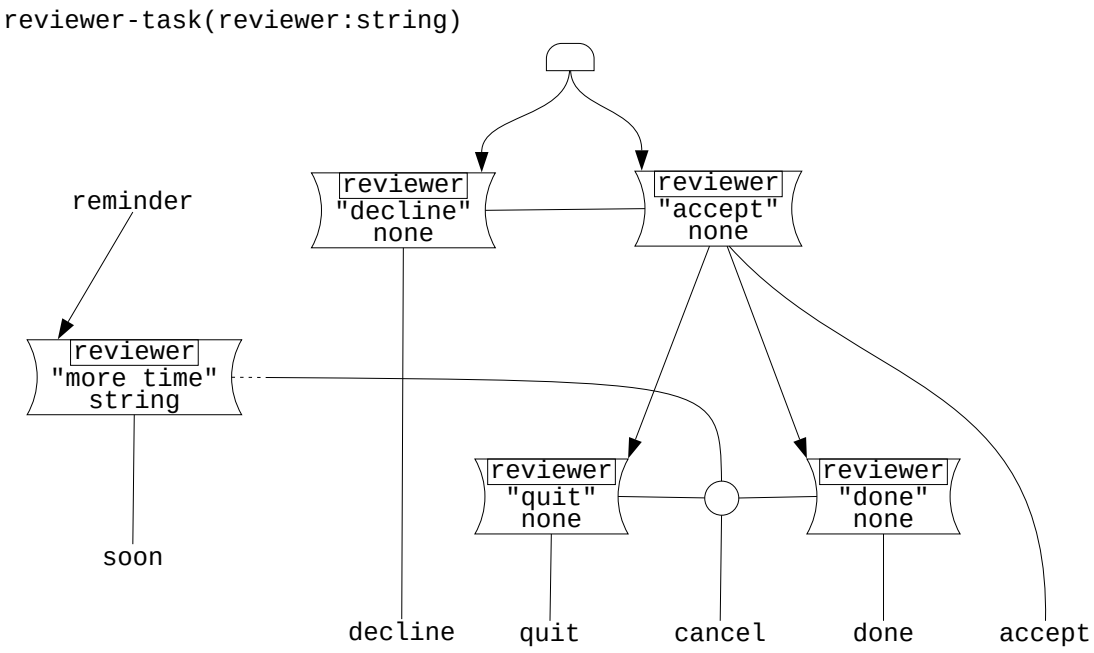

Figure 1: The Reviewer's Activity

An activity description for the reviewer activity described above is shown in figure 1. The first line, `reviewer-task(reviewer: string)`, identifies the activity and specifies that, when instantiated, the activity will have a parameter named `reviewer` that is a `string` naming the person chosen as the reviewer. The several occurences of `reviewer` in the sub-activities in figure 1 refer to that same person.

The half rounded node at the top is an *initiate* node and its function is to send a value down each of the two directed arcs emanating from it when the activity is instantiated. It is often the case that the actual value sent on an arc is not of interest, only the fact that a value is sent being important, and that is the case with this initiate node. The type, `none`, is provided for this purpose and there is a single value of this type called `synch`. Thus, this particular initiate node sends `synch` to the two nodes below it. These nodes are *user interaction* nodes. A user interaction node has, in general, a set of input arcs, a set of output

arcs, a set of *contention* arcs that can be either *exclusion* arcs or *reset* arcs, and a user. The behavior of a user interaction node is as follows:

1. It is quiescent until values have arrived on all its input arcs.
2. It then becomes *active* and informs the user that a response from that user is requested, remaining in the active state until the response is provided or the node is *disabled*.
3. When the user provides a response, the user interaction node attempts to "grab" all its exclusion arcs. If successful, it becomes *enabled* and sends values on all its output arcs. If unsuccessful, it becomes *disabled* and does not send values on any of its output arcs. In either event, the user is informed that the response requested is no longer relevant and the user interaction node becomes quiescent, awaiting a possible new wave of inputs, whence the behavior is repeated.

When the topmost user interaction nodes in the reviewer activity have received the `synch` value on their input arcs, they become active and inform the user indicated by `reviewer` that the node is active and awaiting a response from that user. The `"decline"` and `"accept"` fields of those nodes suggest what the user is told, namely that the user can decline or accept doing the review. These nodes are connected with an exclusion arc, the undirected arc between them. When the user responds, one of the two nodes will become enabled and the other disabled. The enabled node will send values out on its output arcs. The specification `none` in these nodes is the type of value sent, and thus each node will send the value `synch` if it is enabled.

We could think of the reviewer as providing the response in two different ways. One would be for the reviewer to essentially say either *accept* or *decline* and the other would be for the user to have two "buttons", one for accepting and the other for declining the reviewing task. It is the latter that is modeled here and the undirected arc between the two user interaction nodes depicts the "contention" between the two possible responses in the sense that if the reviewer pushes both buttons at the same time, one will "win" and the other will "lose".

If the decline response wins, the left user interaction node will output `synch` on the "dangling" arc that is labeled `decline` and the right node will output nothing. Each dangling arc must eventually be connected to something and, in this case, that is an arc in the editor's activity as we shall see presently. If the accept response wins, the left node will output nothing and the right node will output `synch` on each of its three output arcs. Two of these will provide the inputs to the lower right pair of user interaction nodes, activating them, and the third will inform the editor of the fact that the reviewer has accepted the reviewing task via the dangling arc labeled `accept`. In any event, both the topmost user interaction nodes will then become disabled in the sense that a response from the reviewer to accept or decline the review will be ignored.

The behavior of the two user interaction nodes in the lower right part of figure 1 is very much like that of the two user interaction nodes preceeding them, informing the reviewer that the responses *quit* and/or *done* are awaited, indicating that the reviewer has decided to reject the reviewing task or has completed it.

The leftmost user interaction node is enabled when an input arrives on the arc labeled `reminder` and it queries the reviewer for a new estimate of the date when the review will be done, sending that date to the editor on the arc labeled `soon`. Note that the three lowest user ineraction nodes are all connected by undirected *contention* arcs, via the *exclusion* node that is depicted as a circle. As we noted earlier, from the point of view of a particular user interaction node a contention arc is either a reset arc or an exclusion arc and the two are distinguished by the arc becoming dotted near a node if that node views the arc as a reset arc. Thus, the leftmost of the three lower user intraction nodes views the contention arc as a reset arc while the other two view their contention arcs as exclusion arcs. If the leftmost user interaction node attempts to "grab" its exclusion arcs it will succeed since it has none, only a reset arc. If one of the two other user interaction nodes attempts to "grab" its exclusion arcs, it will have to win over the other two. The point, in this particular activity, is that if the reviewer responds to the "more time" request we want the other two user interaction nodes remain in the state they are in, actively awaiting a quit or done response from the reviewer, but if the reviewer responds with quit or done all three of the user interaction nodes need to be "shut down" and the exclusion arcs provide for this.

3.2 The Editor's Activity

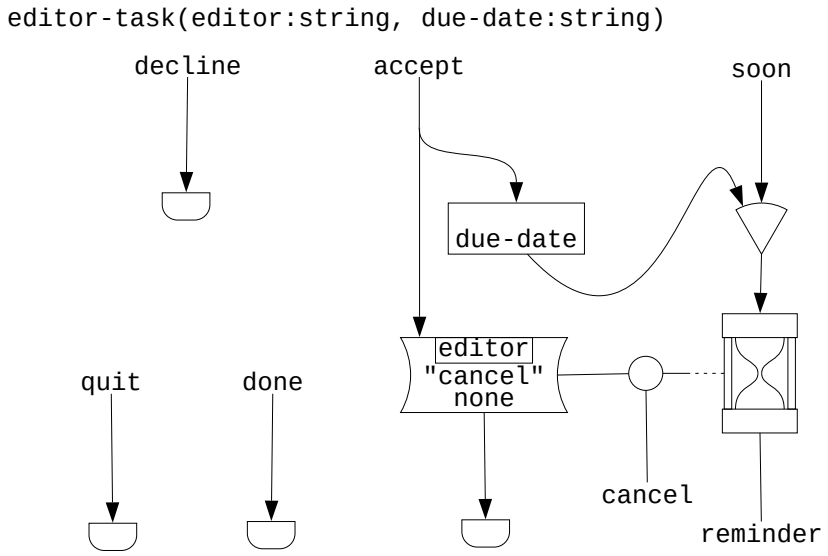

Figure 2: The Editor's Activity

The editor's side of the activity is presented in figure 2 and it takes two parameters, the `editor` and the `due-date`, specifying the name of the person playing the role of editor and the date that the review is due back from the reviewer. The first thing that will happen in this activity is that `synch` will arrive on the arc labeled `accept` or the arc labeled `decline`,

indicating which response the reviewer made to doing the review. The value on the `decline` arc is input to the half oval node that is a *terminate* node. Its sole role is to simply record the value received. In this example, we have chosen to have four terminate nodes, the one eventually receiving a value indicating the way in which the activity terminated because the value, `synch`, received by each is the same. An alternative protocol could have a single terminate node that received on of four different values to distinguish among the ways in which the activity terminated.

If `synch` arrives on the `accept` arc, it is "split" and sent to both the rectangle node and to the user interaction node with query `"cancel"`. When the rectangle node has all its inputs it sends its outputs out, in this case a single output that is specified as the value of the parameter, `due-date`. The triangular *merge* node to which the due date is sent simply sends that value on its output arc, in this case to the timer node below it. In general, a merge node immediately sends out on its output arc each value received on any of its input arcs. In this example, the other possibile input is a new date, received on the `soon` arc, that is the reviewers response when requesting more time.

The user interaction node for the editor provides for the editor deciding to cancel the review by the current reviewer and this node contends with the timer node and whatever nodes are connected via the contention arc labeled `cancel` (that will be connected to the contention arc labeled `cancel` in the reviewer activity).

The timer node becomes "armed" when a time arrives on its input arc and it does nothing until that time arrives. It then attempts to grab all of its exclusion arcs and, if succesful, sends `synch` on its output arc. Thus this particular timer will send `synch` out when the time for which it is set arrives, because it has no exclusion arcs, only a reset arc. The timer may later become re-armed when another time arrives on the arc labeled `soon`.

If a "grab" is attempted by the exclusion node (the circle betwen the user interaction node and the timer) the timer will become disabled and the user interaction node will either be enabled or be disabled, depending on whether it wins the grab or one of the two user interaction nodes in the reviewer activity connected via the arc labeled `cancel` wins the grab. In any event, the result will eventually be that one of the three lowest terminate nodes receives `synch`, the one receiving it indicating why the activity terminated.

3.3 The Combined Editor/Reviewer Activity

The activity description for the combined editor/reviewer activity is shown in figure 3 and it indicates that the combined activity has three parameters, the editor, reviewer, and the due date. The two oval nodes are *condensation* nodes that refer, respectively, to the editor activity description and the reviewer activity description presented above. The labels inside the ovals in figure 3 near its border correspond to the labels of the "dangling" arcs in those two activity descriptions and the arcs between the ovals indicate which dangling arcs in one connect to which dangling arcs in the other.

The combined activity description is equivalent to one that has the nodes of both the editor and reviewer descriptions but with the dangling arcs eliminated by connecting the appropriate nodes directly. The reason for using two separate descriptions is that by separating

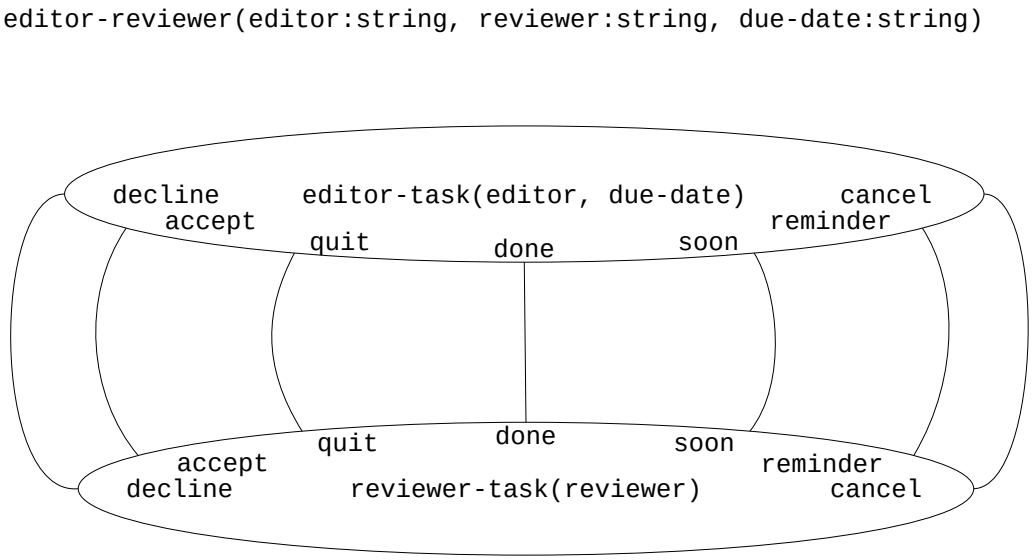

Figure 3: The Editor and Reviewer

the descriptions, the reviewer's and editor's sides of the activity can be viewed separately. Also, if the descriptions are combined, the resulting graph becomes somewhat cluttered.

Obviously deciding which node goes in which description is somewhat arbitrary. For example the terminate nodes that reflect termination because of some response by the reviewer could have been included in the reviewer's activity and the timer could have been placed in either activity.

3.4 Viewing Activity Descriptions

If an activity description has been instantiated it is possible to view that particular instantiation of the activity as it executes. For example, a view on the reviewer's activity takes the form of the graph in figure 1 with a few differences:

- The actual parameters used to instantiate the activity description replace the parameter names (with color used for emphasis),
- active nodes (those that have received values and have not yet sent outputs or been disabled) are highlighted using color, and
- active user interaction nodes are sensitive in the sense that they can be mouse selected resulting in the appearance of a box that can be used to provide some response that the user wishes to make.

Thus, as soon as the reviewer's activity is instantiated it can be viewed and the accept/decline nodes will initially be highlighted. The reviewer can then accept or decline by

a simple gesture. Any authorized user other than the reviewer can also view the activity, but of course only the reviewer is allowed to provide a response.

4 Transaction Graphs

The mathematical formalism that underlies activity descriptions and provides a specification for the implementation of the activity coordination system is called *transaction graphs*. We here provide a brief sketch of transaction graphs, referring the interested reader to [1] for details. Transaction graphs essentially play the role of a "machine language" for supporting the high level language provided by activity descriptions.

A transaction graph is a graph with nodes and undirected arcs and each node, ν, has some associated state, S_ν. At any point in time each node, ν, in a transaction graph *shows* a value, $V_{\nu,\alpha}$, on each arc, α, impinging on ν that the node, ν', at the other end of arc α *sees*. A *transaction* is as follows: A node, ν, that sees a value, $V_{\nu',\alpha}$, on arc α to node ν' may change its state and show a new value, $V'_{\nu,\alpha}$ to ν'. If, at the same time, ν' also attempts to change its state and show ν a new value on arc α, then one of ν and ν' will win and one will lose. This definition of a transaction leads to the serializability of transactions, that is, the execution of a transaction graph does not allow two nodes to change the values shown on some arc simultaneously, but instead one must precede the other and the choice of the winner is non-deterministic.

Some changes that are required to the state of a node in a transaction graph are not reasonable to model in transaction graphs themselves, a good example being an alarm clock that "goes off" (that is, changes some state) at some particular time. Thus, transaction graphs accomodate the notion of out-of-graph messages — changes to the state of some node that, effectively, occur spontaneously.

Given two nodes ν and ν' connected by an arc α, ν and ν' have *protocols* $P_{\nu,\alpha}$ and $P_{\nu',\alpha}$, where $P_{\nu,\alpha}$ describes the behavior on arc α from νs point of view and vice versa. The set, P, of protocols has a transposition operator, $T : P \rightarrow P$, where $p^{TT} = p$ for all $p \in P$ and, in general, p^T produces the protocol for the other end of the arc that has protocol p. While the activity descriptions presented earlier suggest that some arcs in an activity description are directed (those depicted by arcs with barbs at one end) this is not the case at the level of transaction graphs. The reason is that there are many different protocols involving various handshakes between nodes that result in a behavior that we think of as associated with directed arcs; the activity descriptions used for user interaction nodes and others with directed arcs use one particular protocol, but other protocols for directing arcs may also be provided. Another sort of protocol that is provided is that for mutual exclusion, used on exclusion arcs. Again there are many possible protocols for mutual exclusion and one of the possible extensions of the activity coordination system is to add new mutual exclusion protocols that might be appropriate for certain applications.

Protocols are realized by so-called τ functions that take as parameters the value seen on some arc of some node and the state of that node. A τ function outputs a proposed change to the state and a new value proposed to be shown on the arc. Thus, if $\tau_{\nu,\alpha}$ is the τ function

for node ν on arc α, we have $\tau_{\nu,\alpha} : S_\nu \times V \rightarrow 2^{S_\nu} \times V'$, where V and V' are the sets of values seen and shown on arc α. A node, ν, may also have an out-of-graph function $\omega_\nu : S_\nu \rightarrow 2^{S_\nu}$ that can change its state.

Transaction graphs provide a formalism for stating and proving such properties as liveness, freedom from deadlock, and the like that are very important to be able to demonstrate when dealing with concurrent and distributed computations.

Another important property of transaction graphs is the set of transaction graph preserving transformations that are possible. Consider two nodes ν_1 and ν_2 that share no arcs. We can "pinch" these two nodes into a node ν_{12} whose arcs are those of ν_1 plus those of ν_2, whose state is $S_{\nu_1} \times S_{\nu_2}$, and whose behavior on arcs is that of ν_1 or ν_2 as appropriate. Eliminating ν_1 and ν_2 from the graph $\mathcal{G}$ and adding ν_{12} yields a transaction graph $\mathcal{G}$' with the same behavior and properties of $\mathcal{G}$ but with one less node.

As another example of a graph preserving transformation, consider a node with an arc to itself (perhaps as the result of previously pinching two nodes). We can "shrink" that node, removing the self-arc, pulling the semantics inside the node as an out-of-graph state change.

Pinching and shrinking allow a transaction graph to eventually be reduced to a single node.

Such transformations are important in the world of activity descriptions, there permitting coalescing of nodes to simplify an activity description by having fewer nodes. One application of this is to transform an activity description containing lots of detail to a simpler one that elides much of this detail and provides the basis for the activity being viewed, say, by a high level manager who is concerned with appraising a global picture uncluttered by local details.

5 The Activity Coordination System

The Activity Coordination System provides an implementation of transaction graphs and the graphic tools used to construct and view them. It provides:

- A library of τ and ω functions that realize a variety of protocols for arcs and out-of-graph messages,
- a library of instantiation functions that construct nodes and arcs and initialize the state of each node,
- a manager that calls the τ and ω functions associated with a node whenever the state of a node is altered, and
- an activity description editor that provides for creating new activity descriptions and for viewing executing activity descriptions.

Although the above sounds reasonably straightforward, there are a number of considerations that complicate matters:

- Activities may be executed over very long periods of time, often involving weeks or months.
- An activity may execute at several different sites that are connected via a variety of communication media.
- Data must be transported among different sites and among different hardware architectures.

In order to obtain the program whose execution models the enactment of some activity, the activity description for that activity is *instantiated* to produce one or more *activations* — think of these as executable programs plus some current state — that then commence executing. The instantiation process involves binding actual parameters, linking with appropriate libraries, and producing the executable(s). A situation in which more than one executable is required is an activity that spans more than one site so that an executable is required for each site.

Each site has an *activation server*, a program that manages activations at that site and communicates with activation servers at other sites as required. Additionally, each site has an *activations plex* and an *instantiations plex*, data structures that contain the appropriate records for each activation and instantiation at that site. At any point in time, a given activation can be *awake* or be *sleeping* and most activations are sleeping most of the time. We can think of the activation server as a program that awakens an activation that then does a few transactions and goes back to sleep. The activation server must update the activations plex as apropriate, in particular capturing the state of a sleeping activation so that the execution of that activation can be later resumed and information about its state can be provided to other activations that request it. For example, if an activity is to be viewed at some site other than that at which it executes, the activation server at the viewing site would request the activation server at the executing site to provide the state of the activation that is required to present the view.

5.1 The Dynamic Type System

The current implementation is based on a *dynamic type system* that provides a rich variety of data types and the data transport mechanisms required to deal with disparate hardware architectures. Another problem that the dynamic type system addresses is that of dealing with state changes. Recall that the changes proposed to the state of some node may have to be aborted because of conflict between two nodes. That is, if two nodes connected by an arc propose changes simultaneously, one will win and one will lose and the loser's proposed changes must be retracted. Since the loser might have meanwhile proposed other changes to the state that do go through, getting the correct state cannot be handled simply by "rolling back" the state to some previous value.

The dynamic type system provides a delta mechanism and proposed state changes are represented as deltas to the state. Those deltas that go through are eventually applied to the state and those that do not are discarded.

The dynamic type system provides several basic types like integer, boolean, string, and none and type constructors like vector, product, disjoint-union, and set. For each, there is an internal representation appropriate for a particular target architecture as well as a "transport representation" that is independent of hardware architecture. This representation is used in communicating values between activations, possibly at different sites. The representation uses character strings to represent values that are then converted to the appropriate internal representation in a hardware dependent manner. Additionally the dynamic type system provides functions for interconverting values between the representation used in the dynamic type system and the "native" representation used by the particular target architecture, for example between an integer value and a C `int`.

There is also the type, any, that provides for any sort of value and it is particularly useful for dealing with some node that passes around arbitrary values that it never looks at and whose type it does not need to know.

While our present implementation provides the dynamic type system through C programs, any programming language may be used to host activities so long as a compatible dynamic type system is provided for that language.

The dynamic type system must be realized independently on each different type of hardware architecture so that the internal representations for integers, reals, and the like are used when data is converted from the transport representation to the internal representation for the particular target architecture. In most cases, of course, the differences are completely trivial (on unix systems amounting to using the appropriate `scanf` function).

5.2 Activation Servers

An activation server has the task of dealing with activations at the same site and activation servers at other sites. It is realized using remote procedure call (RPC) mechanisms. In the unix implementation, an activation server is contacted using a canonical host/port mechnism that isolates sites. Each site has a data structure called the *sites plex* that records the canonical host/port for itself and all other sites and enables communication with other sites independently of how those sites are configured so long as the computer at the site that is running the activation server can be contacted using the canonical host/port mechanism.

5.3 Other Tools

The other tools that are required are those to do with the editing and viewing of activity descriptions. These tools use the X window system to provide turf for graphs and capture various gestures like mouse gestures on various nodes or arcs or other gestures like keystrokes. Additionally, the ghostview postscript interpreter is used to actually produce the pixels that are viewed. The normal way that activity descriptions are created or modified is with the *activity description editor*. This is a graphical editor that provides menus of available nodes and arcs and an activity description is created by selecting nodes and arcs, placing them on the screen, and supplying various parameters that specialize the nodes and/or arcs in

various ways. For example, if a user interaction node is required, the icon for that node type is selected and placed in the graph under construction. At that point, such things as the number of input arcs and the types of values received on them, the user to be contacted (usually a parameter of the activity description but possibly a constant or a value received dynamically), and the number of output arcs and the types of values sent on these arcs are specified.

Having placed one or more nodes, arcs between nodes can then be established by appropriate mouse gestures.

Thus, the activity description "programmer" needs to understand the underlying "execution model" and the semantics of each node type (most activity descriptions require only a dozen or so node types) and each arc type (most activity descriptions require only a handfull of arc types) in order to create programs based on activity descriptions.

In the future there are a number of tools that deal with reasoning about activity descriptions envisioned. These would include identifying cul de sacs, the absence of deadlock, and so on.

6 Extending the Activity Coordination System

The activity coordination system has an open architecture and we believe that this is very important since new applications may require new functionality and it should be relatively easy to extend the system to provide this functionality. Some of the axes along which extensions are probably mandated are:

- The addition of new protocols,
- the addition of new node types,
- the addition of new types of data,
- the accomodation of new communication media,
- the accomodation of new kinds of hardware architectures, and
- the addition of new tools for reasoning about activity descriptions.

In the paragraphs below we comment on doing these sorts of extensions.

6.1 New Protocols

Providing a new protocol for transactions between nodes means writing a new τ function that embodies that protocol and adding it to the library of τ functions. To do this sort of extension requires detailed knowledge of the dynamic type system and techniques for constructing deltas. For the current system such functions are usually written in C (though,

in principle, any language whose compiler produces an executable that can be linked with C produced executables would do). We are currently preparing a document that describes the current τ function library, and expect that this will help anyone who is writing a new τ function by providing a library of paradigms for writing τ functions.

6.2 New Node Types

To provide a new node type requires finding or creating the appropriate τ functions, writing a function that will instantiate a node of the new type, and "introducing" the new node type to the activity description editor. We have evolved a style of specifying nodes that, we believe, will usually make the introduction of new node types quite simple.

It is often convenient to think of a node as consisting of one or more *node part*s where each node part operates more or less independently. An example is the functionality for dealing with a wave of inputs on a set of input arcs. For this particular node part, the functionality is that when the τ functions for the input arcs have seen values on all the arcs then whatever actions are appropriate are taken. The node part for input waves does the bookeeping to determine when the complete wave has arrived and then calls a *coupling* function that provides for the appropriate actions, usually informing the other parts of the node that the wave of inputs has arrived. Thus to specify an input wave node part for some new node type, we simply find or write the appropriate coupling function for the node type being defined. Again, there is a library of node parts and coupling functions and it usually suffices to mix and match elements of this library in order to develop a new node type. The instantiation function for the new node type is usually a function that simply calls the instantiation functions for the node parts used.

A new node type must also be "introduced" to the activity description editor so that that new type can be used to create and view activity descriptions. The issues that must be addressed include specifying the icon that represents the node, providing the queries that elicit such information as the number of arcs and type of values on those arcs when an instance of the node type is installed in an activity description, and specifying what part of the state is presented in a view and the format for depicting that state. The activity description has a library of templates for such purposes and defining a new node type usualy means at most adding one or more new templates.

6.3 New Data Types

To add a new data type to the dynamic type system requires extending that system with functions for manipulating values of the new type and functions that provide the appropriate delta mechanisms for these values.

To do this sort of extension obviously requires detailed knowledge of the dynamic type system.

6.4 New Communication Media

The present implementation provides for LAN, internet, e-mail, and dialup communications but it is certainly possible that some new media (or new protocol on an existing media) might be desired. This sort of extension would typically require writing the functions required and installing them as new RPCs that can be serviced by the activation server.

7 Status and Future Plans

The activity coordination system described in this paper is operational and and has been used extensively by a group at TRW. The current system is Unix based and implemented in C and Lisp. Based on the experience of others it would appear that proting the system to, for example, Windows 95 or NT based systems would be quite straigtforward.

There is a library of node types that includes twenty or so entries, including the so-called procedure based nodes that await receiving a wave of inputs, perform some computations, and then send out a wave of outputs that result from those computations. The computations performed are completely arbitrary in the sense that they are specified by naming one or more functions that are to be applied to the inputs received in order to produce the outputs to be sent. Thus, the procedure based node is really itself a library of nodes in the sense that, by adding new functions to do the input/output mapping, new behaviors can be provided.

There is also a library of τ functions providing a number of different protocols that can be used for such things as directing arcs, mutual exclusion, and so on.

There are four major directions in which development would be required in order to have a production quality system. One of these is to provide for a PC based system. Another is to develop an extensive library of activity descriptions and gain experience with their use in a variety of activities.

The third direction involves various extensions of the present activity coordination system. One of the issues we have investigated is that of dealing with the logs produced as an activity is enacted. These logs provide the history of the activity, but presently in a form that only a computer wants to see. We plan to investigate various ways of presenting this history in human oriented form, for example, essentially replaying the history by annotating the activity description for the activity over time.

The fourth direction is developing a suite of tools for reasoning about various aspects of activity descriptions like insuring the absencce of deadlocks and cul de sacs.

Another issue that we have investigated is that referred to as *cutover*, by which we mean changing an activity description *and* one or more "in progress" instantiations of it. This is important because the rules governing real human activities often change in the midst of the activity. The essential idea in providing a cutover capability is to use two facts. First, a node in an instantiation can have its state changed, without affecting any neighbors, so long as its exposed values remain the same. Second, because subgraphs are semantically equivalent to complicated nodes, the previous statement applies to entire subgraphs as well as single nodes, with the understanding that exposed values on the boundary arcs of the

changed subgraph do not change.

References

[1] Mike Karr *Transaction Graphs: A Sketch Formalism for Activity Coordination*, Technical Report, Software Options, Inc., April 1990.

[2] Michael Karr and Thomas Cheatham *A Solution to the ISPW-6 Software Process Modeling Example*, Proceedings ISPW-6, Hokaido Japan, Oct 1990.