# 10
# Cerberus - a tool for debugging distributed algorithms

*F. Carter and A. Fekete*
*Department of Computer Science, University of Sydney*
*Madsen Building F09, University of Sydney 2006, Australia.*
*email:* fekete@cs.su.oz.au

**Abstract**

Distributed applications are hard to program. They are particularly prone to subtle race conditions, deadlocks, or similar errors in the underlying distributed algorithm. This paper describes a tool which can assist the designer in debugging a distributed algorithm early in the software lifecycle. The tool takes a high-level abstract description of the algorithm, and an even more abstract requirements specification; it simulates an execution until a discrepancy arises between algorithm and specification; it then assist the developer to explore backwards and forwards through the execution till the error is understood.

**Keywords**

Distributed programs, debugging, simulation, formal methods

## 1 INTRODUCTION

Programming a distributed application is even more difficult and error-prone than writing other software systems. One source of difficulty is the lack of programmer experience with the languages and tools used in coding: the domain is relatively new, and standards are still developing, so different systems are incompatible and many unnecessary obstacles are placed in the programmers' path. Whole books have been written to assist with these arcane details (Stevens, 1990). Distributed applications however differ from tasks in other domains in that even before the coding phase is reached, the design phase is often affected by fundamental algorithmic errors, which are hard to detect, and are not amenable to being fixed with small corrections. For example, in a sequential program, a common mistake in algorithm design is an "off-by-one" loop, which fails to check the last element of a sequence; this is easily detected by testing with a range of extreme inputs, and is fixed by changing the termination test in the loop. In contrast, a distributed algorthm may have a "race condition", where the error is revealed only when a particular pattern of message delays occurs, and the correction requires a completely new approach. The greater difficulty of design for distributed algorithms is clearly evident in the high rate of errors among papers written by experts and accepted in prestigous journals. As as extreme case, Knapp (1987) discusses the sad sequence of incorrect algorithms for detecting deadlock.

I. Jelly et al. (eds.), *Software Engineering for Parallel and Distributed Systems*

Because a distributed application is so vulnerable to errors in algorithm design. we believe that the development of these applications would benefit from a tool that allows the debugging of the algorithm itself, in a rather abstract early form. The designer should be able to explore the executions of the underlying algorithm intended for their system; only once a sound design is chosen would coding take place. This paper describes a tool of this sort. It is called Cerberus and a first version has been implemented.

It is important for the reader to distinguish the sort of tool we describe, which is used for debugging a distributed algorithm at the design phase, from those which can assist in debugging a deployed distributed system after the coding is completed. The latter sort of tool must itself be a distributed program, collecting information at multiple sites in a network, and attempting to determine whether or not certain global conditions are satisfied. Because remote information is always out-of-date, a debugger for distributed systems is very hard to build. Babaoglu and Marzullo summarise the theory behind these systems in chapter 4 of the book edited by Mullender (1993).

The view of software development in this paper is based on top-down refinement: the designer starts from a requirements specification of the service the system is expected to provide to its clients. As described by Fekete (1993), this service specification will generally be presented as a global, abstract, state transition system. Next the designer decides on a fundamental algorithm that will provide this service. For example, the algorithm might involve a token traversing the network, or it might be based on a replicated state machine (Schneider, 1990). This basic algorithm is described as a collection of separate abstract state transition systems, one (for each site in the network) representing the part of the system at one site, and others (such as buffers) which provide inter-site communication. In Cerberus, both requirements specification and proposed distributed algorithm are presented in a particular syntax which is based on a semantic model called Input-Output Automata (Lynch and Tuttle, 1989), which has been extensively used in research papers for describing distributed algorithms. The Cerberus tool is used to detect errors in the basic algorithm. Once no more errors are discovered, and the designer is confident in the correctness of the algorithm, the individual site components can be further refined to efficient code in a conventional programming language. This requires converting the state-transition description used by Cerberus to flow-of-control in a language like C; one must also replace abstract data structures like sets by efficient implementations.

The top-down style of development supported by Cerberus is in contrast to the more common bottom-up building of distributed applications, where the designer takes individual components that already exist, and combines them in different configurations to meet various goals (perhaps writing additional clients to make calls on the components). This bottom-up style of composition is expected by recent standards such as CORBA or Microsoft's OLE Component Object Model; it is also supported by prototype tools such as the Software Architect's Assistant (Ng et al, 1995).

Non-determinism is a key feature of distributed algorithms, and a central reason for the frequency of major errors in their design. Even though each node in the system is completely predicatble in its responses to messages, the whole system has a very large set of executions, each corresponding to a particular pattern of unpredicatable message delays. Since the system has no control over these delays, an algorithm is considered correct only if every possible execution produces the desired outcomes. The core of the Cerberus tool is to simulate one execution of the distributed algorithm (if no error is detected in this, another execution is simulated). The tool allows the designer to control the execution

directly, by repeatedly selecting the next action to occur from among those enabled at the current state; alternatively the execution may develop without the designer's intervention, with appropriate randomisation controlling the pattern of message delays etc. As the algorithm's execution is simulated, Cerberus also follows the transitions in the requirements specification. An error is detected when the algorithm takes an action not allowed in the requirement (this violates a *safety condition*) or when no action is possible by the algorithm at a time when the specification can produce output (this is *deadlock*). The organisation and interface of Cerberus has been inspired by a previous simulation tool for the Input/Output Automaton formal method, called Spectrum (Goldman, 1990).

The key contribution of Cerberus, which distinguishes it from general simulation tools, lies in what happens after an error has been identified (that is, once an incorrect execution has been found). Cerberus allows the designer to explore the execution history, trying to pin down the part of the algorithm that needs fixing. Usually the algorithmic error is in a step that occurred long before the system finally took a step not allowed by the specification. For example, the detected step is often triggered by the arrival of a particular message at a node which is in a particular state; however the error might be in the decision to send that message, or in the decision to enter the particular state (or the problem might lie further back, in the sending of the message that caused the node to enter that state). Cerberus provides the ability to jump backwards and forwards through the execution that has revealed an error: for example, one can move to the most recent step of a given node.

This paper shows how Cerberus is used. In Section 2 we explain the language used to represent both the requirements specification, and the distributed algorithm. In Section 3 we explain the facilities provided by the system and how they are implemented. In Section 4 we work through a (contrived) example. In Section 5 we summarize our conclusions.

## 2 DESCRIBING TRANSITION SYSTEMS

The intellectual foundation for Cerberus is the Input/Output Automaton formal method (Lynch and Tuttle, 1989) invented by Lynch and her colleagues as a semantic model used in presenting and verifying distributed algorithms. The framework has been found to be widely applicable, with simple extensions to deal with time-dependent algorithms, shared memory algorithms, several different types of modularity, and even impossibility proofs. The formal method is based on representing each component as a state-transition system, with potentially infinite state space, and where transitions are named and also classified as inputs, outputs or internal steps. A collection of components can be composed, with synchronisation provided by the fact that identically-named transitions must be taken simultaneously in all components.

As described by Lynch and Tuttle (1989), the Input/Output Automaton method is semantic; the states and transitions may be described using all the techniques of mathematics. In a software tool such as Cerberus, it is essential that a fixed syntax be used to present an automaton. This section describes the language we have chosen. As an example, we give the code in Figure 1 which models a unidirectional error free communication channel. Further examples written in the language are found in Section 4.

*Classes of Automata:* In representing a distributed algorithm, it is usual to have many similar automata in the system; commonly, the processing at each node follows the same principles (which indeed apply no matter what the network topology). Thus in Cerberus,

```
typedef MsgType   : tup(from,val : integer;);
typedef PktType   : tup(type : integer; msg : MsgType;);
typedef PktQueue  : seq of PktType;

automata channel()
state_vars
    buffer : PktQueue;
initial  buffer := seq();
begin
    input send(channel : integer; pkt: PktType;)
    restrict
        channel == AutomataNum;
    begin
        buffer := buffer + seq(pkt);
    end

    output receive(channel : integer; pkt : PktType;)
    restrict
        channel == AutomataNum;
    pre
        if
            (#buffer != 0) -> pkt == buffer[1];
        fi;
    begin
        buffer := buffer[2..#buffer];
    end
end
```

**Figure 1** An I/O Automata class to model a communication channel

we describe a generic template which we call a class of automata, and then we simulate an algorithm in which many instantiations of this class coexist, in a particular configuration.

The class name is declared in a similar method to many programming languages syntax for declaring a procedure or function, i.e an identifier with a set of typed parameters enclosed in brackets. There is also an implicit argument to the automata that does not need to be declared, this is 'AutomataNum' which provides a unique integer identifier for each automata that is included in the final simulated system. In the example above, this implicit parameter is the only parameter.

*State:* The Input/Output Automaton formal method allows an arbitrary, possibly infinite, state space. In Cerberus, the state space is always given as a Cartesian product, based on a collection of named, typed, state variables. Each state of the automaton is described by giving a particular value of the correct type to each variable. Because Cerberus aims to support debugging of algorithms early in the lifecycle, it is important to allow them to be described in a rather abstract style, more common in specification languages than in common programming languages. Experience in describing many algorithms shows the usefulness of abstract, complex data types, such as a set of sequences of pairs, each of which has a string and a set of integers. To cater for these needs, three type constructors were include in the language: sequences, sets, and tuples. In the example above, there is a

single state variable called buffer; its value is a sequence of tuples. Each constructed type has the usual operations: for example `seq()` denotes the constant empty sequence, and `#v` denotes the number of entries in the sequence which is the value of variable `v`.

*Transitions:* In the formal method, an automaton can change between states in discrete steps. There is a transition relation that defines the allowable steps. There are names given to the possible steps (these names are crucial in defining the way automata interact). Each name is refered to as an action, and each action is classified either as an input (meaning that it is controlled by the environment rather than by the automaton itself), as an output (under the autonomous control of the automaton, and able to be detected by the environment), or as internal (under autonomous control, but unable to be detected by other components). In the formal model, each action may label an arbitrary set of transitions. However, it is normal for many related actions to all be used to label transitions which can occur in similar states, and for which the state after the action is determined as a function of the state before it. In Cerberus, we define parameterised action templates. The possible values of the parameters are of course limited by their types, but the algorithm designer can also provide an extra restriction. For example, the code of Figure 1 shows that each channel automaton has many input actions, one for each possible value of pkt; however, the first parameter in the action name is fixed to be identical to the AutomataNum of this component. The restriction clause is not arbitrary: an action class parameter may only be used on the left hand side of an equality operator or the left side of the "element of" operator (written with the `[=` symbol). This allows the simulator to efficiently decide which parameter values should be considered.

It is fundamental in the formal method that input actions can occur at any time as they are controlled by the environment rather than by the automaton. However output and internal actions are under local control, so these actions have a precondition, introduced by the `pre` keyword, which is a series of boolean expressions*. We say an action within a class is enabled when the conjunction of expressions in this section are true. Action parameters can be mentioned in this section, under the same limitations as in the `Restrict` section; these two sections jointly decide which actions within an action class will be allowed to occur in a given state.

When an action occurs, the component must change state. In Cerberus, the new state is determined by executing a body of code enclosed `begin` and `end` keywords. This code, through assignments, defines the way in which the state variables are to be updated to move from the current state to the next state when an action from this class is executed. Within the code, one can refer to any state variables, to the parameters of the action template, and also to local temporary variables (these are meaningful only within the code, and they do not keep their value in the automaton state for use in later transitions). The code should alter values of local or state variables, but not of parameters of the action and automaton. The code may also contain assignments which are executed conditionally; we use the syntax `if` *boolexp* `->` *code boolexp* `->` *code* ... `fi`. The meaning is that each *boolexp* in turn is evaluated; whichever is the first that evaluates to true; the corresponding *code* is executed†

*For convenience in translating existing algorithm models from the research literature, which are written without specific syntax, we provide syntactic sugar for implication written as `if` *boolexp* `->` *boolexp* `fi`.

†There is a deceptive similarity between this syntax and that used for implication in preconditions; this design error should be corrected in future versions.

*Composition:* A distributed algorithm will be represented as an interacting collection of automata. Once the automata classes necessary to construct the system to be simulated have been defined and compiled, instances of these classes need to be created so that a complete distributed system can be simulated. The number and type of each automata within the simulation is defined in a configuration file. This file also contains values with which to instantiate the class parameters for each automaton in the system.

In the formal method, it is the action names which determine whether or not two components in a system can influence one another. This carries over to Cerberus, in contrast to systems like the Software Architect's Assistant (Ng et al, 1995) where each component uses local names and explicit binding is done in a configuration language. In Cerberus, nodes which communicate synchronously can be created by having action names which contain parameters which identify the neighbors; the values of the parameters are set in a configuration file, so the instances which are intended to communicate synchronously are given matching action names.

Asynchronous communication is expressed by introducing a channel automaton between each pair of neighbours: Each node has as a parameter a set that contains the unique identifiers of all of the communication channels upon which it is to send messages. The `send` actions that place messages on the communication channels contain one of the identifiers from this set as a parameter, the other parameter is the message to be sent. Later the channel interacts synchronously with the other node in a `receive` action, where again the action in the destination node is parameterised with the identifier of the channel.

## 3 OVERVIEW OF THE CERBERUS TOOL

The tool includes a translator to transform the description of each automata class into a C source file. The C source file is then compiled to an object file. Once all of the automata classes required for the simulation have been compiled they are linked with the simulator and a configuration file. The configuration file specifies the number of instances of each automata class in the simulation and gives values to the parameters of the automata instances.

*Specification:* To specify the required (correct) behaviour of the system, an automaton that contains the same interface as the implementation must also be provided. The specification automaton models the correct behaviour of the system by only enabling those output actions that correspond to the correct results being passed to the users of the service. The specification automaton should contain the input and output class actions used by the algorithm under investigation, when it interacts with the clients. The specification automaton should be able to model the correct behaviour of the system regardless of the number of users of the service being provided. Hence, a set that contains the identifiers of each of the users of the service will almost always be a parameter for this automaton.

It is important to remember that the specification automata should contain all the actions that communicate with the client regardless of the fact that these actions may be contained in components that are physically distributed from one another. This style of specification, with a global automaton to characterize the desired service, is discussed by Fekete (1993).

*Simulation:* Once Cerberus is running, with linked code for the algorithm's components and for the specification, the first activity is to generate one of the possible executions

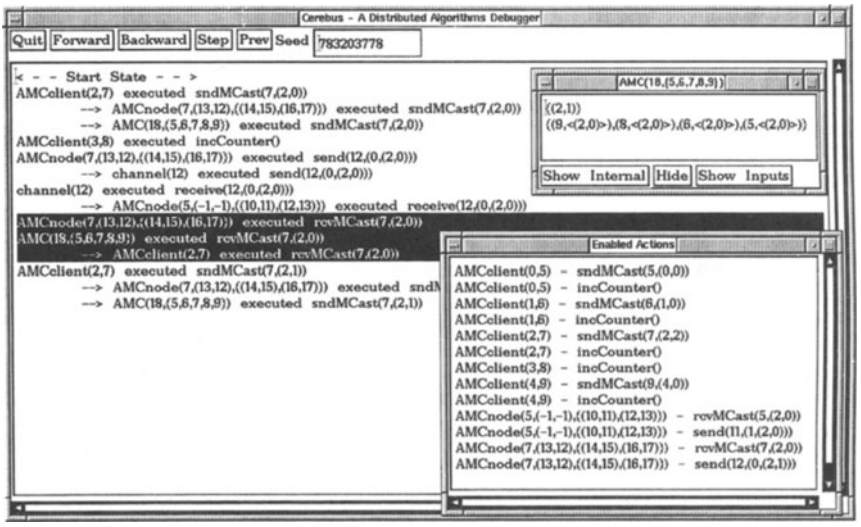


**Figure 2** Cerberus Main Window

of the algorithm being modeled. The execution is developed step by step. At each step, the possible enabled actions are calculated. To be more precise, the system considers each action template which is an output or internal action of any component; it decides which (if any) values of the parameters in that template make both restrict clause and precondition true. Then the user may choose one of the enabled actions (they are displayed in a popup window, seen in the lower right in Figure 2); alternatively, the next action may be selected randomly from those enabled. Whichever mechanism is used, once an action is chosen, it is compared with those that the specification allows: an error reported if the action is not appropriate in the specification.

Once an action has been selected and determined not to cause a violation of the specification then the simulation can take the corresponding transition. This amounts to performing the assignments in the effect code, with the action parameters instantiated to the values that enable the action. Also, if any other components have the same action class name as input, and the chosen parameters satisfy the corresponding `restrict` section, then each of these components also executes the transition determined by its own code. The specification also takes a transition if that is appropriate.

When there are no enabled locally controlled actions within the automata that are modelling the implementation, then there are two possibilities. If the specification automaton contains enabled actions then an error is reported. This situations indicates deadlock has occurred: the specification automaton has interactions that are unfinished, however the algorithm is unable to continue. On the other hand, if the specification automaton also contains no enabled actions then a valid end state has been reached and the simulation terminates without an error. Another execution should now be simulated, until the designer is confident that no errors are present.

*Execution display:* When the execution of the system is stopped for any reason, be it a

breakpoint or an error, details of the actions that were executed to arrive in the current state are displayed in the main window. Also, at any point where the execution is paused the state variables of each automaton in the system may be hidden or displayed. Any displayed state variables will be updated after the execution of an action. Figure 2 shows a screen shot of the main window of the system with the enabled actions window and a window containing the state variables of an automaton.

*Breakpoints:* When a window is opened to display the values of the state variables of an automaton, breakpoints are placed on the execution of any action within the automaton. Whenever the simulation is running freely and it chooses an action with a break point set upon it, then the simulation is paused. A description of the action and any related actions is also printed in the main window. Breakpoints are effective in both forwards and backwards traversal of the simulation and leave the system in the state after the execution of the action in both cases. Buttons on the state display window allow the breakpoints to be removed from either (or both) of the Internal and Input actions.

*Execution Exploration:* A set of buttons along the top of the window allow the user to control the direction of execution and the number of steps the simulation will take. The buttons `Step` and `Prev` cause the execution to take a single step either forwards or backwards. If the algorithm is to step forward then the action it executes is dependent upon whether or not the current state is the last in the sequence of states that have been visited. If it is, then the action that is executed is determined by random selection as previously explained. If the current state has been reached by executing the program in reverse then the action to be executed will be one executed when this state was first visited. If the execution of the original action in this state caused an error then no further progress in this direction is allowed.

If the direction of execution is in reverse then the values of the state variables in the state preceding the current one are restored.

The buttons `Forward` and `Backward` again execute the system in the respective direction, however instead of executing a single event they continue executing until the action executed meets a breakpoint.

## 4 AN EXAMPLE - ATOMIC MULTICAST

To demonstrate the way Cerberus is used, we have taken a simple algorithm from the paper by Fekete (1993). This algorithm provides the communication service called *atomic broadcast* (or sometimes also called totally ordered broadcast). This is an important building block in management of replicated data.

The algorithm can be described informally as follows. The network topology must form a tree, with each node aware of the connection to its parent. When a node receives a request from a client to broadcast a message to all other nodes, this request is firstly sent along the parent connections until it arrives at the root of the tree. The root then sends the message to all of its children and adds the message to a FIFO queue of messages to be delivered to the client. On receipt of message from its parent, each node behaves in the same fashion. This solution achieves the essential property that all clients receive the messages in the same order. it does this by allowing one node, the root of the tree, to make the decision as to the order in which messages will be received.

To show the Cerberus tool in use, a bug was added to the algorithm. To model this

situation using I/O Automata, three automata classes were used. The automata class `AMCclient(node)` models the clients which are using the service. The single parameter shown for this class specifies the identifier of the protocol node with which this client communicates, to request services and receive results. Clients make requests that a message be broadcast upon the network using the Output actions `sndMCast(node, (from,msg))` where parameter `from` contains the identifier of the client that is broadcasting the message, and `msg` contains the information to be sent to each of the other clients. In this simple example, the information is simply an integer that is incremented for each message that the client sends. The `node` parameter allows the automata that are modeling the service providers to only communicate with one client.

The clients accept delivery of a message when they execute the input action `rcvMCast(node, (from,msg))`

The protocol agent process at one node which is providing the Atomic Multicast is represented by another automaton of the class `AMCnode(parent, children)` part of whose code is shown in Figure 3. The third automata class that is used in the simulation models the communication channel that provides an error free, in-order transmission of messages between neighbouring nodes in the network.

The topology of the network in which this simulation was performed has 5 nodes. The root of the tree has Client 0; its children have Client 1 and Client 2 respectively. Client 1 is at a leaf, while Client 2's node has two children with Client 3 and Client 4 respectively. Each node also has a protocol agent whose automaton identifier is 5 more than the corresponding client.

The correctness of the algorithm is specified by providing an automata that embodies the desired behaviour of the service to be provided. It contains the same interface to the clients, i.e the actions to accept a multicast request from the client and to deliver the message to the client. The possible behaviours of the specification automata are the set of actions that represent all possible sequences of send and receive events that are allowed for an atomic multicast with the topology of the implementation automata.

The specification automata maintains two state variables. The first state variable is a set, which contains messages that have been sent by a client and have not yet been delivered to any client. The second state variable is actually a collection of FIFO queues, one for each node. The messages on a queue represent those which must be delivered to the client associated with the node. When the specification automata accepts an action for multicast it is added to the set of undelivered messages. There are two possible places from which a message can be chosen for delivery. At any time the head of the queue associated with a node may be delivered to that node (and removed from the corresponding queue). If the queue of messages awaiting delivery to a client by a node is empty, then any message in the undelivered set may be chosen and be delivered by that node. The effect of executing this action is to remove the chosen message from the undelivered set, and also to append it to the FIFO queue associated with every node except the node which delivered the message with the execution of the action.

## 4.1 Simulating The System

Once the automata classes (representing both algorithm and specification) had been compiled and linked with the simulator, an execution was generated randomly. Very quickly an error is reported by the system. The message below is generated and the action that

```
automata AMCnode(parent : socket; children : socketSet;)
var
    i : socket;
    N : integer;
state_vars
    rcvdQueue : MsgQueue;
    pending   : NodeQueue;
initial rcvdQueue := seq(); pending := seq();
begin
    input sndMCast(node : integer; msg : MsgType;)
    restrict  node == AutomataNum;
    begin
        if
            parent.from == NONE -> begin
                forall i in children -> do
                    pending := pending + seq(QType(i.to,PktType(RECV,msg)));
                od;
                rcvdQueue := rcvdQueue + seq(msg);
            end
            true -> begin
                pending := pending + seq(QType(parent.to,PktType(SEND,msg)));
                rcvdQueue := rcvdQueue + seq(msg); /* BUG ! ! ! */
            end
        fi;
    end

    output rcvMCast(node : integer; msg : MsgType;)
    /* OMITTED */

    input receive(channel : integer; pkt : PktType;)
    restrict
        N := 0;
        (channel == parent.from) ||
        ((forall i in children -> do
            if
                channel == i.from -> N := N + 1;
            fi;
        od) && FALSE) || (N > 0);
    begin
        if
            (pkt.type == SEND) -> begin
                if
                    (parent.to != NONE) -> pending := pending + seq(QType(parent.to,pkt));
                    true -> begin
                        rcvdQueue := rcvdQueue + seq(pkt.msg);
                        forall i in children -> do
                            pending := pending + seq(QType(i.to,PktType(RECV,pkt.msg)));
                        od;
                    end
                fi;
            end
            true -> begin
                rcvdQueue := rcvdQueue + seq(pkt.msg);
                forall i in children -> do
                    pending := pending + seq(QType(i.to,pkt));
                od;
            end
        fi;
    end

    output send(channel : integer; pkt : PktType;)
    /* OMITTED */
end
```

**Figure 3** Abbreviated Source for the Atomic Multicast Node

caused the automata to change to the state in which this action is enabled is printed.
`ERROR: specification can not execute rcvMCast(7,(2,0))`

This leads us to examine the state variables of the specification automata and of AM-Cnode(7). From the displays we saw that the node that tried to execute the unallowed action contained two messages to be delivered (2,0) and (2,1). Looking at the state variables of the specification automata we saw that node 7 should have been delivering the two messages (3,0) and (3,1). We also observed that the set of messages awaiting their first delivery in the specification automata includes the message (2,0) which the node in the algorithm was about to deliver. Thus the error must have been earlier, leading to message (2,0) entering the queue incorrectly.

We would now like to begin looking for the cause of the discrepencies in the state variables. The first thing we tried was to go back and look at the the delivery of any messages to the clients. We could achieve this by displaying the window that contains the node 7 state variables and choosing to not display Input actions of the specification automaton. Now the execution will pause each time the specification automata executes an Output action: these actions correspond to the delivery of messages to the clients. As the specification contains such actions for all the nodes within the network we were able to leap back through each delivery of each message to a client. On doing this we encountered only two actions that delivered messages, node 8 delivers the message (3,0) and then (3, 1).

To observe the state variables before and after the execution of an action, we use a breakpoint on that action, and then step back one statement using the button provided. In both actions of node 8, the transformation of the state variables in both the specification automata and the implementation node are as expected. The node removes the delivered message from the queue of nodes awaiting delivery and makes no other changes. The specification removes the delivered message from the set of undelivered messages and adds copy of it to each of the message queues inside the specification automata except for the queue associated with the node where the message was delivered.

The deliveries of the messages all appear to behave in the expected manner, so we decided to investigate the sending of the messages next. We did this by displaying the state of clients 2 and 3 (those that were senders of the two messages involved). Displaying these components sets breakpoints on their output actions. When each action is found in the execution, we examine the state variables of the corresponding node. When we do this for the action that transmits a message from client 2 we saw that this action adds the message (2,0) to the queue of messages to be delivered to the client, but in the specification it was not waiting to be delivered back to the client. We step through the actions executed and find that this value remains at the head of the queue until the last state before the illegal action was executed. This reveals the bug: client 2 is not the root of the tree and therefore should not add a message until it has been sent to the root of the tree first. After this part of the code has been corrected, the algorithm works as the specification says it should.

## 5 CONCLUSION

We have described a tool that allows the designer to experiment with a distributed algorithm in an abstract form, early in the software lifecycle. By simulating the algorithm

and continuously comparing its behavior to that of a service specification, errors can be found. The key contribution of this work is that the designer can explore the inappropriate execution, moving forwards and backwards in large stages as they seek the specific aspect of the algorithm that must be fixed.

The prototype we have constructed is far from complete. In future work, we are looking to expand the flexibility of the breakpoint mechanism. At present, we set breakpoints which allow us to jump forwards or backwards to the next action or next output of a given automaton. A natural extension would be to move along the causal dependencies; thus from any point one would have a choice of moving to the next action of that automaton or else to the send (or receive) corresponding to the current recieve (or send).

Our design envisages more powerful support for exploration. We plan to allow a breakpoint to be set independently on any variable in any component's state. Thus the user could pause the execution at the next step which leads to a change in the value of that variable. In the example of section 4, as soon as we found that there was an inappropriate value in the queue in node 7, one could jump backward directly to the action that placed the incorrect value there.

Another useful facility would be the ability to give a relationship between the state of algorithm and that of specification, and use the failure of this relationship as a condition to stop the execution's simulation. Thus sort of relationship (called a "refinement" or "abstraction mapping") has been very helpful in proving protocols correct; we expect it would also be helpful in finding errors.

## REFERENCES

Fekete, A. (1993), Formal Model Of Communication Services: A Case Study, *IEEE Computer* **26**(8):37–47.

Goldman, K. (1990) *Distributed Algorithm Simulation Using Input/Output Automata.* PhD Dissertation, MIT Laboratory for Computer Science.

Holzmann, G. (1991) *Design and Validation of Computer Protocols.* Prentice-Hall.

Knapp, E. (1987) Deadlock Detection in Distributed Databases, *ACM Computing Surveys*, **19**(4):303–328.

Lynch, N. and Tuttle, M. (1989) An Introduction to Input/Output Automata, *CWI-Quaterly*, **2**(3).

Mullender, S. (1993) *Distributed Systems (2nd edition).* Addison Wesley.

Ng, K., Kramer, J., Magee, J. and Dulay, N. (1995) The Software Architect's Assistant – A Visual Environment for Distributed Programming *Proceedings of 28th Hawaii International Conference on System Sciences* vol II, 254–263.

Schneider, F. (1990) Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial *ACM Computing Surveys* **22**(4):299–319.

Stevens, W. (1990) *Unix Network Programming.* Prentice Hall.