# I/O Automaton Based Simulation of Selected Distributed Algorithms

by

Aparna Mini Gupta

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fullfillment of the Degree of
Bachelors of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1990

Signature of Author: ______________________________
Department of Electrical Engineering and Computer Science
May 24, 1990

Certified by: ______________________________
Nancy Lynch
Professor, Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: ______________________________
Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses

# I/O Automaton Based Simulation of Selected Distributed Algorithms

by

Aparna Mini Gupta

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 1990 in partial fulfillment of the
requirements for the Degree of
Bachelors of Science in Computer Science and Engineering

## ABSTRACT

This paper describes three distributed algorithms–the Hirshberg-Sinclair leader election algorithm, Peterson's leader election algorithm, and Dijkstra's shortest paths algorithm–using the I/O automaton model. This model has been developed and used as a means of formally describing and reasoning about distributed algorithms. The specifications have been coded into the Spectrum simulation system, which allows the user to simulate the execution of the algorithms and gain an intuitive understanding of them. The paper discusses three points. The first one is the ease of translating the algorithms into this model and into the Spectrum programming language and what is gained by each description. The second is possible changes to the Spectrum interface which would enhance its ease of use and utility. And the final one is recommendations for further studies facilitated by both methods of description.

Thesis Advisor: Nancy Lynch
Title: Professor, Department of Electrical Engineering and Computer Science

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

As distributed computing has become more commonplace within the field of computer science, demand for efficient, correct algorithms has increased. Efficiency analysis and correctness proof of algorithms have, of course, long been issues in sequential computing. Yet even in this more straightforward arena, correctness is often argued intuitively rather than being proved using a more formal verification method. In distributed computing intuition does not work as well since the human thought process lends itself more readily to thinking about events sequentially. It is difficult to keep track of issues such as the inherent nondeterminism of events, variable message delays along a network, different process speeds, and possible process failure. As a result, seemingly correct distributed algorithms have been implemented, and found to be flawed at a later date.

Despite these factors, the majority of the work on distributed algorithms still contains only informal arguments for correctness. Unfortunately, previously available proof methods for distributed algorithms were more obscure than their sequential counterparts. The proofs were only more complex and did not support intuitive understanding of the algorithms.

As a solution to this problem, Lynch and Tuttle introduced the *I/O automaton* model [LT86]. The model claims to enable one to do rigorous proofs of correctness which follow the informal arguments of the designers more closely. The model allows both hierarchical and modular reasoning, and supports the composition of automaton, which corresponds to using multiple algorithms concurrently. This paper describes the Hirshberg-Sinclair leader election algorithm, Peterson's leader election algorithm, and Dijkstra's shortest paths algorithm using the I/O automaton model to demonstrate the facilities of the model. Such algorithms commonly form the basis of much more complicated systems, and uncertainty about the correctness of such a building block means that the correctness of the entire system would be left under question.

To further demonstrate the power of the I/O automaton model, Goldman has developed Spectrum, a simulation system which allows the user to execute code based on the I/O automaton model [Gola] [Golb]. This not only facilitates debugging the algorithm, but can in fact be very useful in gaining an intuitive understanding of the actual working of the algorithm and analyzing message and time complexity.

## 1.2 I/O Automata

Each component, or process, in the algorithm's high level description will be modeled as an I/O automaton, a mathematical object similar to a traditional automaton. An I/O automaton is unique, however, in that it allows an infinite state set. The automaton's actions can be of one of three types, *input*, *output*, or *internal.* The internal and output actions are generated by the automaton and transmitted to the environment, whereas input actions are generated by the environment and transmitted to the automaton. An automaton can place restrictions on the output and internal actions, but may not block input actions. Thus they are said to be *input enabled.* Handling of the input, once accepted, is not constrained by the model.

I/O automata may be nondeterministic. The greater the nondeterminism, the clearer the description and the greater the field of algorithms it describes. In addition, I/O automata may be composed to form another I/O automaton.

### 1.2.1 Definition

The following definition has been adapted from Lynch's descriptions of the I/O automaton as presented in the lecture notes compiled for M.I.T.'s Distributed Algorithms course [LG88].

S–the *action signature*–is an ordered triple consisting of three pairwise-disjoint sets of actions, *in(S)*, *out(S)*, and *int(S)*, referring to the *input*, *output*, and *internal* actions of S, respectively. The *external actions* consist of both actions in in(S) as well as those in out(S), while the internal actions are those in int(S). The *actions* of S are the combination of both external and internal actions and are written *acts(S)*.

An *I/O automaton* A consists of the following five components:

- an action signature, *sig(A)*,
- a set of *states*, *states(A)*, possibly infinite,
- a nonempty set *start(A)* $\subseteq$ states(A) of *start states*,
- a transition relation *steps(A)* $\subseteq$ states(A) $\times$ acts(sig(A)) $\times$ states(A), with the property that for every state s$'$ and input action $\pi$ there is a transition (s$'$,$\pi$,s) in steps(A), and
- an equivalence relation *part(A)* on local(sig(A)), having at most countably many equivalence classes.

A finite *execution* of A is a sequence $s_0, \pi_1, s_1, \pi_2, ..., \pi_n, s_n$ where $s_0$ is the start state. A *fair execution* of an automaton A is defined to be an execution $\alpha$ of A such that if $\alpha$ is finite, then no action of C is enabled in the final state of $\alpha$ for each class C in part(A). It is also possible to have fair infinite executions, but this paper is concerned only with the finite case. A fair execution gives 'fair turns' to each class of part(A).

## 1.3 Spectrum

There are two elements to running a simulation in Spectrum–the language which is used to define *automaton types* and a *configuration* which is a collection of *automaton instances*, composed or connected by directed edges. An automaton type defines the signature, states, transition relation, and action partition of an automaton. This definition is good for each instance of that automaton type in the configuration. The signature, transition relation, and classes may be different for two instances of the same automaton type since the program may reference information present in the configuration.

The configuration defines relationships between automaton instances as if each instance were a node in a graph Any given node may also be composed of several other nodes. The user defines the set of instances, their type, a string name for each, the compositional hierarchy, and a set of directed edges between instances. A unique automaton id is assigned to each instance by the system.

An optional element of the configuration is a mapping from integer variables which are part of an automaton's state information to either the center or the rim of the geometric shape representing each automaton instance. Eight colors are presently available and may be referenced by the integers zero to seven. For values of the variable greater than seven, the element is colored with seven's color. Additionally, there is a statement in the language which allows the user to assign an integer, representing a color, to an edge between any two automaton instances. Both of these facilities can be used flexibly to show different aspects of the algorithm during simulation. An example of this feature is explained for each of the algorithms discussed.

The code and a configuration must both be present in order to perform an execution of the algorithm. Appendix D shows the look of the interface with both the setup and simulate menus down.

The simulator is always in either CONFIGURE or SIMULATE mode. The code and a configureation must both be present in order to perform an execution of the algorithm. In order to loal a code or configuration file, one must be in CONFIGURE mode. In this mode one may also construct a configuration using the mouse. Once a configuration has been constructed it may be saved for later use. There is never any need to define a configuration textually. Variables are mapped to process objects by openning up the object with the mouse. One may then select the desired integer variables from a list of those that make up the process's state.

Once the necessary information is in palce there are several options for running a simulation. These options are listed under the SIMULATE pulldown menu. One may run the simulation without any pauses, pause at every step, or pause at user defined intervals. Additionally, one may choose to give processes turns in order or at random. Appendix D shows the look of the interface with both the SETUP and SIMULATE menus down.

# Chapter 2

# Hirsberg-Sinclair Leader Election Algorithm

## 2.1 Description

The Hirshberg-Sinclair algorithm [HS80] was the first algorithm to show that it is possible to obtain a worst-case message complexity of $O(nlogn)$, an improvement on Le Lann-Chang-Roberts leader election algorithm with a worst-case message complexity of $O(n^2)$. The Hirshberg-Sinclair algorithm has the following properties:

- Does not assume that the number of nodes in the ring is known.
- Involves *bidirectional* message passing.
- Works asynchronously.

- Uses an unbounded identifier set.
- Elects the node with the maximal identifier as the leader.

The difference between this algorithm and the Le Lann-Chang-Roberts algorithm is that this algorithm assumes bidirectional message passing capability.

The way that the Hirshberg-Sinclair algorithm, as presented in [LG88], works is as follows. Each process in the ring sends out a message to its left and to its right. Initially this message is sent only to its immediate neighbors. Every time the sending process is returned a message that it initiated, it initiates another message back in the same direction, except this time it sends the message twice as far.

If a process receiving a message ever finds that the id of the process that ininitiated the message is greater than its own id, then it realizes it cannot be elected and stops initiating messages. Similarly, if the id of the process receiving a message is higher than that of the process that initiated the message, then rather than continuing to propagate the same message it turns the message back to the initiating process, telling the initiating process that it cannot be elected and should thus stop initiating messages.

When a message from a given process makes it all the way around the ring without being turned around, it means that this process must be the leader. Of course, all these events are occurring asynchronously, but by the time a process is elected the leader, all other processes will have died–stopped initiating messages.

## 2.2 In Terms of the I/O Automaton Model

The Hirshberg-Sinclair algorithm can be described in a straightforward, concise manner using the I/O automaton model. The model facilitates a simple, modular description of the algorithm. The description uses one automaton type, which will be named PROC for process.

```
in(S)={forward_left((chan:automaton_id,to_go:integer,id:integer)),
       forward_right((chan:automaton_id,to_go:integer,id:integer)),
       kill_left((chan:automaton_id,id:integer)),
       kill_right((chan:automaton_id,id:integer)),
       ok_left((chan:automaton_id,id:integer)),
       ok_right((chan:automaton_id,id:integer))}

out(S)={forward_left((chan:automaton_id,to_go:integer,id:integer)),
       forward_right((chan:automaton_id,to_go:integer,id:integer)),
       kill_left((chan:automaton_id,id:integer)),
       kill_right((chan:automaton_id,id:integer)),
       ok_left((chan:automaton_id,id:integer)),
       ok_right((chan:automaton_id,id:integer))}

int(S)={ }
```

Figure 2.1: Action Signature S for Hirshberg-Sinclair

The action signature, S, is shown in figure 2.1. All actions are external, of which there are two types, forward and back. *Forward left* and *forward right* are the forward actions, while the remaining ones are back actions. The forward actions are used when a message is on its way out from the initiating process, while the back actions are used when a message is on its way back to the initiating process.

The *chan* is the automaton id of the process to whom the message is directed, *to go* is the distance around the ring that the message has yet to go, where any given process is a distance of one from an immediate neighbor. And *id* is the unique id of the process that initiated the message. The *to go* variable is not needed for messages that are on their way back to the initiating process since these message will know they have reached their destination when the id of the process receiving a message is the same as the id carried by the message.

The actions occur in left/right pairs since the events in either direction out from a process are parallel. The output forward actions are simply used to propagate a message until a process receives a forward action as input and decides to turn it around. This can happen in two way. The first way is that the id of the initiating process turns out to be higher than the id's of all the other processes that the message encounters on the way out from the initiating process. In this case, either the *ok left* or *ok right* action is output by the final process in the chain. The second way is that the id of one of the processes that the message encounters on the way turns out to be higher than the id of the initiating process. In this case, the process with the higher id immediately turns the message around with either a *kill left* or *kill right* action.

If a process receives an ok action as input with the id carried in the message equivalent to its own id, and it has not yet been killed by a message it received from its neighbor on the other side, it initiates a new message. This message is now sent out twice as far as the previous one in the direction from which the input came.

These actions continue until a process receives a forward message carrying an id that is equal to its own. This means that a message that the process initiated has made it all the way around the ring without encountering a process with a higher id, in which case it would have been turned into a back message. Upon receipt of such a message the execution of the algorithm should terminate. The process will elect itself the leader by assigning its status to be "elected", and all other processes will be dead.

```
id:integer = unique arbitrary integer
status:string = ''waiting''
distance_left:integer = 1
distance_right:integer = 1
pending_fl:set((automaton_id,integer,integer)) = {(in(self()),id,1)}
pending_fr:set((automaton_id,integer,integer)) = {(out(self()),id,1)}
pending_kl:set((automaton_id,integer)) = { }
pending_kr:set((automaton_id,integer)) = { }
pending_ol:set((automaton_id,integer)) = { }
pending_or:set((automaton_id,integer)) = { }
```

Figure 2.2: Start State $S_0$

In order to accomplish this, the state information shown in figure 2.2 needs to be kept at every process with the accompanying initial values. The information keeps track of the following. *Id* holds the process's own unique id number. Such id's are necessary for any leader election algorithm. In this particular algorithm, of course, the process with the highest id is the one elected. *Status* is initially "waiting". If it comes to know that there is a process in the ring with a higher id then it assigns itself to be "dead". On the other hand, if it learns that it's own id is the highest of any process in the ring, then it assigns itself to be "elected". The latter assignment should be the final step that occurs in the execution of the algorithm. The two distance variables keep track of how far out from the process in either direction the most recently initiated message was sent in either the left or right direction. Initially both of these values are one.

The pending sets are all sets of messages waiting to go out from the given process. Initially the only nonempty sets are *pending fl* and *pending fr*, *Fl* for 'forward left' and *fr* for 'forward right'. These hold the messages waiting to go out a distance of one from the process. Self() refers to the automaton id of the process itself. In(self()) refers to the left-hand neighbor of the process, and similarly out(self()) refers to the right-hand neighbor of the process.

By defining the action signature and process state in such manner, the actual action descriptions become straightforward. Only the actions for messages out to the left from the initiating process will be discussed here, but the actions in the opposite directions are exactly parallel. The full code listing appears in Appendix A.

The input actions are described in figure 2.3. *Forward left* may handle an incoming message in any one of four ways. If the id of the message is its own id, it elects itself the leader. Otherwise, if the id of the message is greater than its own, it kill itself. If *to go* has not yet reached one, it sends the message forward to its left-hand neighbor. If it has reached one, then this process is the one that needs to turn the message around, performing an *ok right* output action. Conversely, if none of the former conditions hold, then the id of the process must be greater than the id of the message, so a *kill right* message is sent back to the initiating process.

If a process receives a *kill right* input, it checks to see if the id of the message matches

```
forward_left((self(),to_go,id))
effect: if id=s'.id then
           s.status=''elected''
        elseif s'.id<id and to_go>1 then
           s.status=''dead''
           s.pending_fl=s'.pending_fl U {(in(self()),to_go-1,id)}
        elseif s'.id<id and to_go=1 then
           s.status=''dead''
           s.pending_or=s'.pending_or U {(out(self()),id)}
        else
           s.pending_kr=s'.pending_kr U {(out(self()),id)}

kill_right((self(),id))
effect: if s'.id=id then
           s.status=''dead''
        else
           s.pending_kr=s'.pending_kr U {(out(self()),id)}

ok_right((self(),id))
effect: if s'.id=id and s'.status=''waiting'' then
           s.distance_right=s'.distance_right*2
           s.pending_fl=s'.pending_fl U {(in(self()),s.distance_right,s.id)}
        elseif ~(s'.id=s.id) then
           s.pending_or=s'.pending_or U {(in(self()),id)}
```

Figure 2.3: Input Actions

its own id. If it does, then it kills itself. Otherwise it propagates the message to its right-hand neighbor.

Similarly, *ok right* first checks if the id of the message matches its own id. If it does and the process has not been killed since the message was first sent out, then it initiates another message to its left-hand neighbor. This message is to travel twice as far as the last message it initiated in this direction. If the message has not yet reached the initiating process, it propagates the message to its right-hand neighbor. The one additional possibility is that the id of the message matches the process's own id, but the process is now dead. In this case no action needs to be taken.

```
forward_left(x)
precondition: x is an element of s'.pending_fl
effect: s.pending_fl=s'.pending_fl-x

kill_right(x)
precondition: x is an element of s'.pending_kr
effect: s.pending_kr=s'.pending_kr-x

ok_right(x)
precondition: x is an element of s'.pending_or
effect: s.pending_or=s'.pending_or-x
```

Figure 2.4: Output Actions

The output actions are described in figure 2.4. These are all similar to one another and very straightforward. Each output action is performed only if there is a message waiting in the appropriate pending set. If so, it arbitrarily selects one of them, and outputs it to the environment. When combined, these actions implement the Hirshberg-Sinclair leader election algorithm in a modular, intuitive fashion.

## 2.3 Simulation

The Spectrum code for the simulation is listed in Appendix A. It is obtained directly from the I/O automaton code and thus should not require additional explanation. Certain additional variables may exist due to constraints imposed by the language. For example, a tuple cannot be defined in one statement, so a temporary variable needs to be created, the elements assigned one at a time, and then the temporary variable inserted into the set. The only differences between the higher level description and the Spectrum code are such long forms required for the Spectrum language.

What does require explanation, however, is the second listing of Spectrum code for the Hirshberg-Sinclair algorithm included in Appendix A. The first listing works correctly, but does not give informative integer variables to map to the colors available for the simulation. Certain modifications have been made to the code to produce an intuitive feel for the algorithm when it is actually executed by the simulator.

First a *stat int* variable has been added. This variable maps an integer value to each of the three possible states of the automaton. In the ring configuration this integer variable is further mapped to the middle of the object representing a process in the configuration. As a result, we can see whether any given automaton instance is waiting, dead, or elected while the simulation is executing, simply by looking at the color at the center of each process object. Presently the colors are mapped so that it is green while waiting, black when dead, and red if elected.

Also, an integer variable, *max* has been added to the state of the automaton. *Max* holds the value of the most significant digit, zero to seven, of the highest id it has yet seen. *Max* is mapped to the rim of each process object.

Finally, in order to see the leader process successfully send its message further and further around the ring, an additional automaton type LEADER has been created. This automaton type is exactly identical to the PROC automaton type with the exception that it is assigned what is known by the user to be the highest id of any of the automata in the ring.

The id's of all PROC types will be assigned such that they lie between 0 and 700. The id of the LEADER type will be assigned such that it is greater than 700. Thus not only will the LEADER be the process elected since it has the highest id number, but it will also be the only process whose id number has 7 as the most significant digit. Through *max* the user will be able to see this id propagate further and further around the ring. If the color of the rim of any of the process objects changes to purple for 7, it must have received a message from the LEADER.

The processes themselves do not distinguish between a LEADER process and a PROC process, however, and thus are not given any information that would allow it to somehow cheat. By making these changes, one is actually able to see the algorithm progress–the higher id's taking over, killing the appropriate processes along the way. If the process with the highest id were to share its most significant digit with another process, the user would not be able to see the one process's id propagate around the ring as clearly.

# Chapter 3

# Peterson Leader Election Algorithm

## 3.1 Description

While the Hirshberg-Sinclair algorithm requires bidirectional message passing in order to achieve $O(nlogn)$ performance, Peterson developed an algorithm that would achieve the same message complexity using only unidirectional message passing [Pet82]. The algorithm has the following properties:

- Assumes that the number of nodes in the ring is unknown.
- Involves *unidirectional* message passing.
- Works asynchronously.

- Uses an unbounded identifier set.

- Elects any node as the leader.

The Peterson algorithm, as presented in [LG88] works as follows. Each process is initially active. By the point an execution of the algorithm terminates, each process has either become a relay, or has been elected leader of the ring. Relays act as dummy processes, only passing along messages that they receive. The active processes are the ones still in contention for the leader position. Once a leader has been elected, the execution will terminate.

The algorithm is divided into asynchronously determined phases. In each phase the number of active processes is divided in at least half, so *logn* is an upper bound on the number of requisite phases.

In the first phase each process sends its id two steps clockwise, allowing each process to compare its own id to the id's of its two counterclockwise neighbors. The process remains alive only if the id of its closest active neighbor was the highest of the three. In this case, it adopts its neighbors id and continues with the next phase. If not, it becomes a relay.

At least half of the processes must become relays after each phase since only a given process or its most closest active neighbor can stay active, not both. And of course, in each phase at least the closest active clockwise neighbor of the process with the highest id will remain active. Finally, all but one of the processes will have become a relay, and in the next phase this process will be elected the leader.

## 3.2 In Terms of the I/O Automaton Model

Again, a concise, intuitive description of the algorithm is constructed using the I/O automaton model. The one automaton type is named PROC for process.

```
in(S)={send_high((chan:automaton_id,phase:integer,id:integer))
       send_low((chan:automaton_id,phase:integer,id:integer))}

out(S)={send_high((chan:automaton_id,phase:integer,id:integer))
        send_low((chan:automaton_id,phase:integer,id:integer))}

int(S)={check_status(),
        do_wh(),
        do_wl()}
```

Figure 3.1: Action Signature S for Peterson

This time there are three internal actions as well as the external actions, as can be seen in figure 3.1. *Send high* is used when a process sends its own id out to its closest active clockwise neighbor. Upon receiving such an input action, the process modifies its own state information appropriately, and does a *send low* with the same id to *its* closest active clockwise neighbor. Upon receiving a *send low* action, the process will modify its own state information appropriately, but does not need to propagate the same id any further.

The *chan* is the automaton id of the process to whom the message is directed. *Phase* is the asynchronous phase to which the message belongs. Initially it will be zero. And it is the id of the process that initiated the set of messages with a *send high* action.

Each process waits to receive one input of each type such that the phase counter coincides with its own current phase. If the process has sent out $n$ *send high* messages, then its phase will be $n-1$. Once it has receive two messages with phase equal to $n-1$, the internal *check status* action is performed. *Check status* will either decide that the process is to remain active, or turn it into a relay. If it is to remain active, the process enters the next phase and initiates another *send high* message.

Additionally, since the phases are asynchronous, it is possible for a process to receive an input message with phase $m > n-1$, where again, $n-1$ is the process's own current phase. In such a case the message is buffered within the process until the process's becomes $m$, and it is ready to handle the message. *Do wh* and *do wl* are the two internal actions that process buffered messages at the appropriate times, where *wh* stands for 'waiting high' and *wl* stands for 'waiting low'. They are exactly symmetrical to the two input actions, only they take messages from an internal buffer rather than from the external environment.

An execution of the algorithm terminates when a process receives its own id back as the result of either a *send high* or *do wh* action. At this point this process will be elected the leader, and all other processes should have become relays.

The information required to be held in the state of each process is shown in figure 3.2. The purpose of each variable is as follows. *id* holds the process's own unique id number. *Status* is either "waiting", "relay", or "elected". *Phase* is equivalent to $n-1$, where $n$ is the number of *send high* messages that the process has initiated.

*Received* keeps track of the number of messages that the process has received for

```
id:integer = unique arbitrary integer
status:string = ''waiting''
phase:integer = 0
received:integer = 0
high:integer, undefined
low:integer, undefined
to_sh:set((automaton_id,integer,integer)) = { }
to_sl:set((automaton_id,integer,integer)) = { }
waiting_high:set((automaton_id,integer,integer)) = { }
waiting_low:set((automaton_id,integer,integer)) = { }
check_waiting:string = ''no''
```

Figure 3.2: Start State $S_0$

its current phase. Once *received* equals two, the process knows to perform the internal action *check status.* At this time, *high* will contain the id of the process's closest active counterclockwise neighbor in the current phase, and *low* will contain the id of the next closest active process in the current phase.

*To sh* is a set of messages waiting to be output through the *send high* action, and to sl is a set of messages waiting to be output through the *send low* action.

*Waiting low* and *waiting high* are buffers for messages received with phase counters higher than that of the process at the time. When the process may be ready to handle buffered input, *check waiting* is set to be first "high" and then "low". When it is "high", *do wh* is enabled, when it is "low", *do wl* is enabled, and when it is "no", neither is enabled.

Again, though more state information is required for the Peterson algorithm than was required for the Hirshberg-Sinclair algorithm, the actions themselves are straightforward. The code is listed in Appendix B.

```
send_high((self(),p,id))
effect: if s'.status=''relay'' then
           s.to_sh=s'.to_sh U {(out(self()),p,id)}
        elseif id=s'.id then
           s.status=''elected''
        elseif p=s.phase then
           s.high=id
           s.received=s'.received+1
           s.to_sl=s'.to_sl U {(out(self()),p,id)}
        elseif s'.status=''active'' then
           s.waiting_high=s'.waiting_high U {(self(),p,id)}

send_low((self(),p,id))
effect: if s'.status=''relay'' then
           s.to_sl=s'.to_sl U {(out(self()),p,id)
        elseif p=s.phase then
           s.low=id
           s.received=s'.received+1
        elseif s'.status=''active'' then
           s.waiting_low=s'.waiting_low U {(self(),p,id)}
```

Figure 3.3: Input Actions

The input actions are described in figure 3.3. *Send high* may handle an incoming message in any one of four ways. The first way occurs when the process has become a relay. In this case it simply propagates the message in tact to the next clockwise neighbor. The second possibility is that the id of the message matches its own id. If so, then the process elects itself the leader. At this point all other process should already be relays, and thus execution of the algorithm will terminate. This leaves two possibilities, either the phase of the message matches the processes own phase and needs to be handled right away, or the phase of the message is greater than the phase of the process and needs to be buffered until it can be handled. If the process is ready to handle the message, then it sets *high* to be the id of the message, and increments the *received* counter, as well as preparing a message to be output via the *send low* action to the immediate clockwise neighbor. If the process is not ready to handle the message then it simply insert the message in tact into the *waiting high* buffer.

*Send low* can be handled in three possible ways. If the process has become a relay, then the message is propagated in tact to the next clockwise neighbor. Otherwise either the phase of the message matches the phase of the process or it is greater. If it matches, then *low* is set to be the id of the message and the *received* counter is incremented. Otherwise the message is buffered in tact in  waiting low.

```
send_high(x)
precondition: x is an element of s'.send_high
effect: s.send_high=s'.send_high-x

send_low(x)
precondition: x is an element of s'.send_low
effect: s.send_low=s'.send_low-x
```

Figure 3.4: Output Actions

The output actions are described in figure 3.4. They perform the same function as the output actions from the Hirshberg-Sinclair algorithm. Each output action is performed only if there is a message waiting in the appropriate pending set. If so, it arbitrarily selects one of them, and outputs it to the environment.

The three internal actions are described in figure 3.5. *Check status* is activated when a messages corresponding to the current process phase have been received from both the closest active counterclockwise neighbor and the next closest active counterclockwise neighbor and the status of the process is still "active".

In such a case, *received* is reinitialized to zero. *Check waiting* is set to "high" since it is possible that messages for the next phase are already buffered within the process state. And the id of the closest active counterclockwise neighbor is compared to the other two id's. If it is not the greatest of the three than the process becomes a relay. If it is the greatest of the three, however, then the process adopts the highest id as its own, increments the phase counter, and prepares to initiate another *send high* message to its immediate clockwise neighbor.

*Do wh* is only enabled when *check waiting* equals "high". It sets *check waiting* to "low" and then looks for an element in the buffer with a phase counter equal to the present phase counter of the process. If such a message is found, it is first deleted from the buffer and then handled as if it had just been received by the process through a *send high* action. The only difference is that if such a message is found, then by definition the phase of the message equals the current phase of the process, and the possibility that they are not equal does not need to be handled.

Similarly, *Do wl* is only enabled when *check waiting* equals "low". It sets *check waiting* to "no" and then looks for an element in the buffer with a phase counter equal to the present phase counter of the process. If such a message is found, it is first deleted from the buffer and then handled as if it had just been receive by a *send low* action, except for two modifications. The first modification is the same as that for *do wh*–the case where the two phases do not match does not need to be handled. The second difference is that if the process is a relay, then it is necessary to enable *do wh* again since it will not be enabled by *check status*.

## 3.3 Simulation

The Spectrum code for the simulation of Peterson's algorithm is listed in Appendix B. Again, it follows directly from the I/O automaton code and thus should not require additional explanation. The only differences are the long forms required for the Spectrum language.

As before, two listings of Spectrum code appear in the appendix. The first one is the minimal code that will work correctly on its own. The second listing has been produced by enhancing the first one for visual purposes. When run with the proper configuration, it should aid in giving the user an intuitive feel for the algorithm.

Only one addition was made for this case, and that was to add a *stat int* variable to the state $S$. This variable is equal to 7 for purple when the process is waiting, 0 for black when the process has become a relay, and 4 for green if the process is elected. This variable was mapped to the center of each object representing a node in the configuration.

The preexisting phase variable was mapped to the rim of each node object. This way one is able to tell how many processes were still active after each phase, and how many phases were necessary before the leader was elected.

```
check_status()
precondition: s'.received=1 and s'.status=''active''
effect: s.received=0
        s.check_waiting=''high''
        if (s'.high > max(s'.id,s'.low) then
           s.id=s'.high
           s.phase=s'.phase+1
           s.to_sh=s'.to_sh U {(out(self()),s.phase,s.id)}
        else
           s.status=''relay''

do_wh()
precondition: s'.check_waiting=''high''
effect: s.check_waiting=''low''
        if there exists an element x of s.waiting_high s.t.
         x.phase=s'.phase then
              s.waiting_high=s'.waiting_high-x
           if s'.status=''relay'' then
              s.to_sh=s'.to_sh U {(out(self()),x.phase,x.id)}
           elseif x.id=s.id then
              s.status=''elected''
           else
              s.high=x.id
              s.received=s'.received+1
              s.to_sl=s'.to_sl U {(out(self()),x.phase,x.id)}

do_wl()
precondition: s'.check_waiting=''low''
effect: s.check_waiting=''no''
        if there exists an element x of s.waiting_high s.t.
         x.phase=s'.phase then
           s.waiting_low=s'.waiting_low-x
           if s'.status=''relay'' then
              s.to_sl=s'.to_sl U {(out(self()),x.phase,x.id)}
              s.check_waiting=''high''
           else
              s.low=x.id
              s.received=s'.received+1
```

Figure 3.5: Internal Actions

# Chapter 4

# Dijkstra BFS Shortest Paths Algorithm

## 4.1 Description

This shortest paths algorithm [Awe], referred to as the Dijkstra algorithm due to its similarity to the algorithm Dijkstra presented in [DS80], is included to serve as a contrast to the leader election algorithms, and show that the I/O automaton model is also appropriate for such graph algorithms. The algorithm is classified according to the following properties:

- Assumes that the size and structure of the network are unknown.
- Involves *bidirectional* message passing across links.
- Works asynchronously.

- Works with both directed and undirected graphs.
- Creates a shortest paths tree, rooted at a predetermined source node.

The same algorithm will work on either a directed or undirected graph without any changes. The directed graph problem still requires bidirectional communication across link. If it is operating upon an undirected graph, nodes are still referred to as having children and parents. If there is an edge between $a$ and $b$ in graph $G$, then $a$ and $b$ are said to be both one another's parent and child.

The algorithm begins at a predetermined source node. During execution, the algorithm build a shortest path tree, rooted at this node. Initially the tree is empty, and upon termination of the algorithm it is the complete BFS shortest paths tree. At any given time during the execution, the present tree rooted at the source is a sub-tree of the final one, and from there on the tree can only grow.

The source node acts to both initiate messages and to synchronize the execution. Successive levels of the shortest paths tree are determined in a breadth first manner. The source node begins by sending out messages to find the first level of the tree. Once it knows that this level has been completed, it initiates messages to add nodes at the next level of the tree. This continues until the source knows that all of the nodes in the graph have been added to the tree, and at this point the execution of the algorithm terminates.

The source node discovers nodes at layer $l$ in the tree by sending out messages to all nodes at layer $l-1$ in the tree. As described earlier, if the source node is working on layer $l$, this means that all nodes at layers $n < l$ have already been added to the tree. Once the nodes at layer $l - 1$ receive the messages sent out from the source they send out exploratory messages to all of their children. Each child that has not yet been added to the tree is hooked on at this layer, choosing the sender of the exploratory message as its parent, and sending back a positive acknowledgment to the probe. If the child has already been added to the tree that it sends back a negative acknowledgment to the probe. Upon receiving a negative acknowledgment, the process removes the node from its set of children.

Once a node receives responses to all the messages it sent out, it sends a response to its own parent telling the parent whether or not any new nodes were added at layer $l$. Once the source node collects responses from all its children, it determines whether any new nodes were added to the shortest paths tree. If not, then all the nodes must already have been added, meaning the tree is complete, and the execution terminates. Otherwise, the the source begins on the $l - 1$ iteration.

## 4.2 In Terms of the I/O Automaton Model

The I/O automaton description of the Dijkstra algorithms requires two automaton types, one named SOURCE and the other PROC. As described in the previous section, the two automata type are necessary since the source performs a different function in the algorithm than the other processes. Since the source node is the one to initiate messages, it will be discussed first.

### 4.2.1 Automaton Type SOURCE

```
in(S)={ack((chan:automaton_id,from:automaton_id,n:integer)),
       back((chan:automaton_id,new?:boolean))}

out(S)={probe((chan:automaton_id,from:automaton_id)),
        forward((chan:automaton_id))}

int(S)={make_pending(),
        terminate()}
```

Figure 4.1: Action Signature S for Dijkstra SOURCE

The action signature S for the SOURCE automaton is given in figure 4.1. If the source is the only node in the graph, then the execution will terminate immediately. Otherwise it begins be ouputting a *probe* to each of its children, and the children send back *ack*'s to the *probe*'s. All the responses to *probe*'s by the source should be positive, meaning they all take the source to be their parent and add themselves to the shortest paths tree. At this point *make pending* prepares another set of messages to go out. The *forward* output actions sends these messages out, and the *back* input action collects the responses. Once all responses have been collected, either the *make pending* or *terminate* internal action is performed depending on whether or not any new nodes were found at the previous level.

The *chan* variable is, as in the previous two cases, the automaton id of the process to whom the message is directed. *From* is the automaton id of the process who sent the message. This variable is required for *probe*'s and *ack*'s, so that a process knows who to make its parent, or who to remove from its set of children, when appropriate. *N*, in integer form, and *new?*, in boolean form, record whether any new nodes were added to the tree at the current level along the path from which the message is returning.

```
status:string = ''waiting''
new?:boolean = false
distance:integer = 1
pending:set(automaton_id) = children(self())
returned:integer = 0
```

Figure 4.2: Start State $S_0$

The start state for any process of type SOURCE is shown in figure 4.2. *Status* remains "waiting" until the execution is ready to terminate, at which time it becomes "done". *New?* is true if and only if at least one node was added to the shortest paths tree at the current level. *Distance* records the level of the tree that is currently being explored. *Pending* is a set of automaton id's of the source's children to whom messages are waiting to be delivered through one of the output actions. And *returned* keeps track of the number of responses received to messages sent out at the current level.

```
ack((self(),f,n))
effect: s.new?=true
        s.returned=s'.returned+1

back((self(),n))
effect: s.new?=s'.new? or n
        s.returned=s'.returned+1
```

Figure 4.3: Input Actions

The input actions for the source, shown in figure 4.3, are particularly simple. In both cases, the response counter, *returned* is incremented, and *new* is set to show whether any new nodes have yet been found at the current level. When receiving an *ack* the source can assume that *new* will be true, since all of the source's children will take the source to be their parent. In the case of the *back* action, *new?* is set to true if at least one of the responses that came back was also true. Beginning with the value equal to false, and then continuing to update it by performing OR operations upon it and the response received, accomplishes this objective.

```
probe((x,f))
precondition: x element of s'.pending
              s'.distance=1
              f=self()
effect: s.pending=s'.pending-x

forward((x))
precondition: x element of s'.pending
              s'.distance>1
effect: s.pending=s'.pending-x
```

Figure 4.4: Output Actions

The output actions, shown in figure 4.4, do slightly more than the output actions for the two leader election algorithms. Both actions reference the same pending set. The messages sent to these automata are *probe*'s if the first level is currently being explored, and simple *forward* messages otherwise.

The internal actions are described in figure 4.5. When responses have been collected from all of the source's children, either *make pending* or *terminate* is executed, depending on whether or not any new nodes were added to the tree at the iteration just completed.

```
make_pending()
precondition: s'.returned=size(children(self()))
              s'.new?=true
effect: s.returned=0
        s.distance=s'.distance+1
        s.new?=false
        s.pending=children(self())

terminate()
precondition: s'.returned=size(children(self()))
              s'.new?=false
effect: s.status=''done''
```

Figure 4.5: Internal Actions

*Make pending* does the majority of the work for the source. If new nodes were added to the tree at the iteration just completed, then *make pending* increments *distance*, which keeps track of the level of the tree currently being explored and reinitializes the other state variables. If no new nodes were found, then *terminate* assigns the source's status to be "done", and at this point each node should know its parent and children in the shortest paths tree.

### 4.2.2 Automaton Type PROC

The action signature S for the PROC automaton is given in figure 4.6. The functions of the input actions *ack* and *back* as well as the output actions *probe* and *forward* are the same as the corresponding actions for the SOURCE automaton. The processes do not need the internal actions of the source automaton, since these were actions that

```
in(S)={probe((chan:automaton_id,from:automaton_id)),
       forward((chan:automaton_id))},
       ack((chan:automaton_id,from:automaton_id,n:integer)),
       back((chan:automaton_id,new?:boolean))}

out(S)={probe((chan:automaton_id,from:automaton_id)),
        forward((chan:automaton_id))},
        ack((chan:automaton_id,from:automaton_id,n:integer)),
        back((chan:automaton_id,new?:boolean))}

int(S)={ }
```

Figure 4.6: Action Signature S for Dijkstra PROC

initiated messages and served as synchronizers, functions unique to the source. The PROC's do, however, require two additional input actions and two additional output actions.

The first action performed by a PROC automaton will be to receive a *probe* input. As a result of any such input, it will return an *ack* to the sender. If the *ack* is the first one sent, then it is positive, meaning the node has adopted the sender of the probe as the node's parent. All subsequent *ack*'s will be negative, meaning the process already has a parent, and the sender should delete the node from the sender's set of children.

In subsequent phases, the process will receive *forward* messages from its parent. The first time it receives such a message, it sends out *probe*'s to all of its children and receives back *acks* that it handles appropriately. At subsequent stages the process simply propagates the *forward* messages to all of it children in the shortest paths tree, and then waits to collect responses from all of them in the form of a *back* input. Once all the responses have been collected the process itself performs a *back* output action.

```
new?:boolean = false
distance:integer = 0
parent:automaton_id, undefined
children:set(automaton_id) = children(self())
pending:set(automaton_id) = children
returned:integer = 0
to_ack:set(automaton_id) = { }
back_ok:boolean = false
```

Figure 4.7: Start State $S_0$

The start state for each process of type PROC is shown in figure 4.7. *new?* is true if and only if the process knows of at least one node having been added to the shortest paths tree at the current level. *Distance* records the $l - m$ where $l$ is the level of the tree that is currently being explored, and $m$ is the nodes own level in the tree. if $m$ is unknown, then *distance* will be zero.

*Parent* is set to the automaton id of the process's parent node in the tree as soon as one has been adopted. *Children* is initially the set of automaton id's of the children of the node in the configuration. By the time *distance* equals two, *children* will contain the automaton id's of only the children of the node in the shortest paths tree.

*Pending* is a set of automaton id's of the source's children to whom messages are waiting to be delivered through one of the output actions, *probe* or *forward.* And *returned* keeps track of the number of responses received to messages sent out at the current level.

*To ack* is a set of automaton id's of processes to whose probes the automaton has yet to respond through an *ack* output action. And *back ok* is true if and only if a *forward* input has been received, but a *back output* has not yet been sent.

```
probe((self(),f))
effect: s.to_ack=s'.to_ack U f

forward((self()))
effect: s.pending=s'.children
        s.back_ok=true

ack((self(),f,n))
effect: if n=0 then
           s.new=true
           s.returned=s'.returned+1
        else
           s.children=s'.children-f

back((self(),n))
effect: s.new=s'.new or n
        s.returned=s'.returned+1
```

Figure 4.8: Input Actions

The input actions for PROC's, shown if figure 4.8 are an extension of the input actions to the source. Upon receiving a *probe* the automaton id of the sender is added to the set of automaton id's of processes to whose probes the automaton has yet to respond through a *ack* output action.

If a *forward* input is received then, the process knows that it must send either a *probe* or *forward* message to all of its children, so their automaton id's are inserted into *pending*. *Back ok* is set to true since a *forward* message has been received, but a *back* message has not yet been returned.

There are two possible outcomes when an *ack* is received by a process. The first one occurs when the sender has adopted the process as its parent and is thus sending back a positive acknowledgment. In this case $n$ will equal zero. *New* is set to true, since the process knows that a child was added to the shortest paths tree in this iteration, and the counter of the number of children who have responded is incremented by one. Conversely, the second outcome occurs when the *ack* is a negative acknowledgment. This means that the sender has already adopted a different node as its parent. In this case the sender's automaton id is removed from the process's set of children.

The *back* input behaves in exactly the same manner as the input action *back* for the SOURCE.

*Probe* and *forward*, two of the output actions, shown in figure 4.9, are the same as the corresponding output actions of the source. *Ack* is output if a probe has been received to which a response has not yet been output. $S'.distance$ will equal zero if and only if the node does not yet have a parent. In this case a positive *ack* is sent back, *distance* is set to one, and the sender of the probe is set to be the parent of the process. If the process already has a parent then a negative *ack* is sent back. In either case, the automaton id of the process to whom the acknowledgment is being sent is deleted from the set *to ack*.

## 4.3 Simulation

Two listings of Spectrum code for the Dijkstra algorithm appear in Appendix C. The first one is the minimal code that will work correctly on its own. The second listing contains three enhancements that should help give the user an intuitive feel for the algorithm.

The first enhancement is an additional integer variable, *in tree*, to the state of both the SOURCE and the PROC automaton types. *In tree* is 0 before the node has been added to the shortest paths tree, and 7 after. This variable is then mapped to the center color of the configuration object representing a process in Spectrum. The center will appear black before the node has been added to the shortest paths tree and purple afterwards.

The second enhancement was to add another integer variable, *level*, which stores the node's level in the shortest paths tree, the root being at level zero. This variable is mapped to the rim color of the configuration object representing a process in Spectrum. All nodes at the same level in the tree will have the same rim color. Since there are only eight colors available, if there are more than eight levels in the tree, then the *level* variable starts back at zero. This variable is not meant to accurately hold the level of the node in the tree, but is meant simply as a visual device.

Finally one line was added to the code. This line colors the edges in the configuration that are also edges in the shortest paths tree. If an edge is not in the tree, then it remains black, otherwise it becomes purple.

```
probe((x,f))
precondition: x elament of s'.pending
              s'.distance=1
              f=self()
effect: s.pending=s'.pending-x

forward((x))
precondition: x element of s'.pending
              s'.distance>1
effect: s.pending=s'.pending-x

ack((x,f,n))
precondition: x element of s'.to_ack
              f=self()
              n=s'.distance
effect: s.to_ack=s'.to_ack-x
        if s'.distance=0 then
           s.distance=1
           s.parent=x

back((x,n))
precondition: s'.returned=size(s'.children)
              x=s'.parent
              n=s'.new?
      s'.back_ok
effect: s.returned=0
        s.distance=s'.distance+1
        s.new?=false
        s.back_ok=false
```

Figure 4.9: Output Actions

# Chapter 5

# Conlclusion

## 5.1 Comments on Experience with Spectrum

While Spectrum is a powerful tool for simulating distributed algorithms, it is surprisingly easy to use. This ease of use is due to several factors. First of all, the window is very well laid out. It looks clear and unencumbered. Additionally, the system is driven, for the most part, by mouse input. It is especially convenient to be able to create a configuration with the mouse, and save it for later use. One does not need to handle specifying a configuration through some obscure method in a file.

This system is still relatively new, however, and there are certain enhancements that may make it more powerful as a tool and also easier to use. Certain changes are already in the planning stages, such as adding a scheduler. Such a scheduler would make it possible to do time complexity analysis. Also, syntactic sugar for assignments and computations would make the code listings more readable, particularly the more complicated the algorithms implemented. Discussed below are some additional ideas that could be useful.

### 5.1.1 Loading Files

Loading in files can be somewhat cumbersome at times. Especially since Spectrum is an interpreted language, one can have to attempt reloading the same file many times consecutively. In such cases it would be helpful if the previous file name specified were saved as a default value for later use.

The ability to set paths would make loading files even more convenient. These paths would be checked by the system when a file name was specified. This way if one were running from a directory other than the one where the files were stored, a long path name would not have to be typed even once, much less time are time. This addition would be advantageous for other than just the obvious reason. The top of the Spectrum display shows the code and configuration files presently loaded, including the path. There is a limited amount of space for these names. When either name is too long, the right-hand side, the filename itself, is cut off. Also, this addition would make it more viable to have the interface give the user a menu of possible files to load. The user would then simply click on the desired file name.

### 5.1.2 Feedback to Input

There are small number of cases where the system's feedback to user input could be better. First of all, if the user types an underscore at the input line, as part of the name of a file, the underscore is not visible.

More importantly, when an simulation of an algorithm is begun, the user should receive some sort of message saying that the algorithm is executing. If the either the user has not mapped variables to the colors of the nodes, or the variables mapped do not change until a fair way into the execution, there is no immediate sign that the execution has in fact begun. The only way the user can know that the program is running is to open up an automaton instance and see if the state is changing.

Lastly, if the user is in SIMULATE rather than CONFIGURE mode, the items under SETUP are not available. The system still allows one to bring down the SETUP menu, however, and does not given any message if the user attempts to load or save a file. It should be made clearer that these choices are presently unavailable.

### 5.1.3 Changing Configurations

When a user load a configuration file, its name is displayed in the upper right-hand corner of the Spectrum window. If the configuration is then saved under a different name, possibly after modifying it, the displayed file name does not change. It would seem reasonable that it should since the configuration the user is now working with is the one that was just saved.

In terms of ease of changing the configuration, it would be helpful if the *Delete* option allowed one to erase all or some section of the elements displayed in the configuration window. The user could click the mouse button and then drag it, creating an enclosing rectangle. When the button was released, everything within the rectangle would be deleted.

An *Undo* option could also come in handy, especially if the previously mentioned addition were made, and the user could delete large amounts of information at once. It is easy to forget that one is in *Delete* mode, rather than say *Connect* mode, and accidentally erase information that would be cumbersome, if not difficult, to reconstruct.

### 5.1.4 Execution

The ability to manually change state information while running the code would provide greater flexibility in studying executions. This enhancement would allow the user to pinpoint problems or play with variations of the algorithm without actually changing the code before knowing what change is necessary or preferred.

## 5.2 Further Studies Facilitated by Descriptions

The I/O automaton model has allowed a formal, intuitive description of the three algorithms. As a result, it would be a natural next step to set up an invariant and prove correctness on the high level I/O automaton code, as well as proving time and message complexity bounds. Additionally, once the scheduler is added to the system, one can study the time complexity behavior one actually sees, given maximum message delivery and process step times. The performace using the simulator can then be compared against the average and worst case performance that would be expected for the algorithm.

Finally, the system also allows composition of automaton. This would lend itself well to studying two or more algorithms working together. Say a leader was needed to run another algorithm, which is usually the case in any complex distributed system. One could first run a leader election algorithm, and then the algorithms that require a leader. Rather than weaving the two pieces of code together into one algorithm, it would make more sense to have one element of a composed pair run leader election and then hand control over to the other process through a *leader* output action. The other process would be told who the leader was and would thus be able to run the dependent algorithms. This allows the retention of modularity and a more involved study of the best way to have several algorithms work together.

## 5.3 Summary

This paper described three distributed algorithms–the Hirshberg-Sinclair leader election algorithm, the Peterson leader election algorithm, and the Dijkstra shortest paths algorithms. Each algorithm was presented formally using the I/O automaton model, and the formal description was explained in detail. In addition, the I/O automaton description was translated into Spectrum code, and examples were given of the intuitive feel that the simulation could provide the user. The simulator can also be a powerful tool for doing a more complete study of the algorithms.

All three algorithms lent themselves quite well to the I/O automaton model. While the benefits of writing the algorithms as I/O automata may be somewhat different than writing Spectrum code to simulate execution of the algorithms, the two modes of description complemented one another well. If the ultimate goal is to simulate an algorithm on Spectrum, it is still beneficial to write the higher level I/O automaton description. This preparation generally results in tighter, more modular Spectrum code since one is not mired down by the details of the language. Similarly, if one wishes only to study the algorithm at the higher level, it is still useful to perform the simulation in order gain a clearer intuitive understanding and debug possible problems.

Included as well are notes on experience with Spectrum, which is still a relatively new system. Overall it was found to be powerful, and yet surprisingly straightforward to use. Many enhancements are already planned for the next implementation of the system. Certain other additions were discussed that would make the system more convenient to work with, as well as more flexible as a tool.

And finally, the descriptions of algorithms discussed in this paper are only a starting point. The model opens up an entire field of formal studies that can be carried out on distributed algorithms, without becoming bogged down in the logistics of the method of study, and losing touch with the intuitive workings of the algorithms themselves. The ultimate advantage of these methods of describing distributed algorithms will be increased confidence in distributed systems that are actually put into use.

# Appendix A

# Hirshberg-Sinclair Code

## A.1 Minimal Spectrum Code

```
%
% data equates
%
DATA mf tuple(chan:automaton_id, id:integer, to_go:integer)
DATA mb tuple(chan:automaton_id, id:integer)

%
% action signature
%
ACTION initially()
ACTION forward_left mf
ACTION forward_right mf
ACTION kill_left mb
ACTION kill_right mb
ACTION ok_left mb
ACTION ok_right mb
```

```
AUTOMATON PROC
   %
   % automaton state information
   %
   STATE tuple(id:integer, status:string,
               distance_left:integer, distance_right:integer,
               pending_fl:set(mf), pending_fr:set(mf),
               pending_kl:set(mb), pending_kr:set(mb),
               pending_ol:set(mb), pending_or:set(mb)
               tf:mf, tb:mb)


   %
   % start state
   %
   INPUT initially
       EFF      % set 'id' to be unique random number
                assign(s.id,int_plus(int_times(int_random(0,7),100),
                                     str_to_int(name(self()))))
                assign(s.status,"waiting")
                % set dist 1st message to be sent out either direction.
                assign(s.distance_left,1)
                assign(s.distance_right,1)
                % assign component of message
                assign(s.tf.chan,set_random(in(self())))
                assign(s.tf.id,s.id)
                assign(s.tf.to_go,1)
                % send message to neighbors
                assign(s.pending_fl,set_single(s.tf))
                assign(s.tf.chan,set_random(out(self())))
                assign(s.pending_fr,set_single(s.tf))
                % initialize other pending message sets to empty
                set_init(s.pending_kl)
                set_init(s.pending_kr)
                set_init(s.pending_ol)
                set_init(s.pending_or)
```

```
%
% input actions
%

   %
   % message on its way out from the initiating proc
   %
   INPUT forward_left WHERE eq(self(),a.chan)
        EFF     % message has made it all the way around the ring
                %   -> elect self to be the leader
                ifthenelse(eq(a.id, s.id),
                        assign(s.status,"elected")
                % message still on its way out and its id > proc's id
                %   -> kill proc
                %   -> propagate same message to left-hand neighbor
                ifthenelse(bool_and(less(s.id,a.id),less(1,a.to_go)),
                        {assign(s.status,"dead")
                        assign(s.tf.chan,set_random(in(self())))
                        assign(s.tf.to_go,int_minus(a.to_go,1))
                        assign(s.tf.id,a.id)
                        set_insert(s.pending_fl,s.tf)},
                % message needs to be turned back and its id > proc's id
                %   -> kill proc
                %   -> send message to initiating proc saying no proc's with
                %      higher id's than its own were encountered.
                ifthenelse(bool_and(less(s.id,a.id),eq(1,a.to_go)),
                        {assign(s.status,"dead")
                        assign(s.tb.chan,set_random(out(self())))
                        assign(s.tb.id,a.id)
                        set_insert(s.pending_or,s.tb)},
                % proc's id > msg id
                %   -> tell initiating proc to kill itself.
                {assign(s.tb.chan,set_random(out(self())))
                assign(s.tb.id,a.id)
                set_insert(s.pending_kr,s.tb)})))
```

```
INPUT forward_right WHERE eq(self(),a.chan)
     EFF     ifthenelse(eq(a.id, s.id),
                     assign(s.status,"elected"),
             ifthenelse(bool_and(less(s.id,a.id),less(1,a.to_go)),
                     {assign(s.status,"dead")
                     assign(s.tf.chan,set_random(out(self())))
                     assign(s.tf.to_go,int_minus(a.to_go,1))
                     assign(s.tf.id,a.id)
                     set_insert(s.pending_fr,s.tf)},
             ifthenelse(bool_and(less(s.id,a.id),eq(1,a.to_go)),
                     {assign(s.status,"dead")
                     assign(s.tb.chan,set_random(in(self())))
                     assign(s.tb.id,a.id)
                     set_insert(s.pending_ol,s.tb)},
             {assign(s.tb.chan,set_random(in(self())))
             assign(s.tb.id,a.id)
             set_insert(s.pending_kl,s.tb)})))
```

```
%
% message on way back to initiating proc, telling it to kill itself.
%
INPUT kill_left WHERE eq(self(),a.chan)
     EFF     % message reached initiating proc
             %   -> kill proc
             ifthenelse(eq(s.id,a.id),
                assign(s.status,"dead"),
             % message net yet to initiating proc
             %   -> propagate same message
             {assign(s.tb.chan,set_random(in(self())))
             assign(s.tb.id,a.id)
             set_insert(s.pending_kl,s.tb)})


INPUT kill_right WHERE eq(self(),a.chan)
     EFF     ifthenelse(eq(s.id,a.id),
                assign(s.status,"dead"),
             {assign(s.tb.chan,set_random(out(self())))
             assign(s.tb.id,a.id)
             set_insert(s.pending_kr,s.tb)})
```

```
%
% message on way back to initiating proc, telling it ok so far.
%
INPUT ok_left WHERE eq(self(),a.chan)
     EFF     % message reached initiating proc and proc still alive
             %   -> send message twice as far out in same direction
             ifthenelse(bool_and(eq(s.id,a.id),eq(s.status,"waiting")),
                {assign(s.distance_right,int_times(s.distance_right,2))
                assign(s.tf.chan,set_random(out(self())))
                assign(s.tf.id,s.id)
                assign(s.tf.to_go,s.distance_right)
                set_insert(s.pending_fr,s.tf)},
             % message not yet to initiating proc
             %   -> propagate same message
             ifthen(bool_not(eq(s.id,a.id)),
                {assign(s.tb.chan,set_random(in(self())))
                assign(s.tb.id,a.id)
                set_insert(s.pending_ol,s.tb)}))

INPUT ok_right WHERE eq(self(),a.chan)
     EFF     ifthenelse(bool_and(eq(s.id,a.id),eq(s.status,"waiting")),
                {assign(s.distance_left,int_times(s.distance_left,2))
                assign(s.tf.chan,set_random(in(self())))
                assign(s.tf.id,s.id)
                assign(s.tf.to_go,s.distance_left)
                set_insert(s.pending_fl,s.tf)},
             ifthen(bool_not(eq(s.id,a.id)),
                {assign(s.tb.chan,set_random(out(self())))
                assign(s.tb.id,a.id)
                set_insert(s.pending_or,s.tb)}))
```

```
%
% output actions
%

   %
   % pending set not empty
   %   -> select message from set at random
   %   -> output selected message
   %

   CLASS
      OUTPUT forward_left
        PRE     bool_not(set_empty(s.pending_fl))
        SEL     assign(a,set_random(s.pending_fl))
        EFF     set_delete(s.pending_fl,a)

   CLASS
      OUTPUT forward_right
        PRE     bool_not(set_empty(s.pending_fr))
        SEL     assign(a,set_random(s.pending_fr))
        EFF     set_delete(s.pending_fr,a)

   CLASS
      OUTPUT kill_left
        PRE     bool_not(set_empty(s.pending_kl))
        SEL     assign(a,set_random(s.pending_kl))
        EFF     set_delete(s.pending_kl,a)

   CLASS
      OUTPUT kill_right
        PRE     bool_not(set_empty(s.pending_kr))
        SEL     assign(a,set_random(s.pending_kr))
        EFF     set_delete(s.pending_kr,a)

   CLASS
      OUTPUT ok_left
        PRE     bool_not(set_empty(s.pending_ol))
        SEL     assign(a,set_random(s.pending_ol))
        EFF     set_delete(s.pending_ol,a)
   CLASS
      OUTPUT ok_right
        PRE     bool_not(set_empty(s.pending_or))
        SEL     assign(a,set_random(s.pending_or))
        EFF     set_delete(s.pending_or,a)
```

## A.2 Code with Additions for Simulation Purposes

```
%
% equates & action sig
%   remain unchanged
%

AUTOMATON PROC
   STATE tuple(id:integer, status:string,
               distance_left:integer, distance_right:integer,
               pending_fl:set(mf), pending_fr:set(mf),
               pending_kl:set(mb), pending_kr:set(mb),
               pending_ol:set(mb), pending_or:set(mb)
               tf:mf, tb:mb,
               %
               % 2 vars added to state for simulation purposes:
               %   max, holds most significant digit of highest id yet seen
               %   stat_int, 4(green) <-> status=''waiting''
               %             0(black) <-> status=''dead''
               %             3(red)   <-> status=''elected''
               %
               max:integer, stat_int:integer)
```

```
%
% maintain simulation variable, stat_int, at proper value
%
MAINTAIN
      ifthenelse(eq(s.status,"waiting"),assign(s.stat_int,4),
      ifthenelse(eq(s.status,"dead"),assign(s.stat_int,0),
                                     assign(s.stat_int,3)))

INPUT initially
     EFF     %
             % most sig digit of proc's id chosen s.t. it is greater
             % than or equal to 0, but less than 7-reserved for leader.
             %
             assign(s.id,int_plus(int_times(int_random(0,6),100),
                                  str_to_int(name(self()))))
             %
             % max so far assigned to be this most sig digit
             %
             assign(s.max,int_div(s.id,100))
             assign(s.status,"waiting")
             assign(s.distance_left,1)
             assign(s.distance_right,1)
             assign(s.tf.chan,set_random(in(self())))
             assign(s.tf.id,s.id)
             assign(s.tf.to_go,1)
             assign(s.pending_fl,set_single(s.tf))
             assign(s.tf.chan,set_random(out(self())))
             assign(s.pending_fr,set_single(s.tf))
             set_init(s.pending_kl)
             set_init(s.pending_kr)
             set_init(s.pending_ol)
             set_init(s.pending_or)
```

```
INPUT forward_left WHERE eq(self(),a.chan)
     EFF     ifthenelse(eq(a.id, s.id),
                     assign(s.status,"elected"),
             ifthenelse(bool_and(less(s.id,a.id),less(1,a.to_go)),
                     {assign(s.status,"dead")
                     %
                     % higher id seen
                     %   -> update value of max
                     %
                     ifthen(less(s.max,int_div(a.id,100)),
                             assign(s.max,int_div(a.id,100)))
                     assign(s.tf.chan,set_random(in(self())))
                     assign(s.tf.to_go,int_minus(a.to_go,1))
                     assign(s.tf.id,a.id)
                     set_insert(s.pending_fl,s.tf)},
             ifthenelse(bool_and(less(s.id,a.id),eq(1,a.to_go)),
                     {assign(s.status,"dead")
                     %
                     % higher id seen
                     %   -> update value of max
                     %
                     ifthen(less(s.max,int_div(a.id,100)),
                             assign(s.max,int_div(a.id,100)))
                     assign(s.tb.chan,set_random(out(self())))
                     assign(s.tb.id,a.id)
                     set_insert(s.pending_or,s.tb)},
             {assign(s.tb.chan,set_random(out(self())))
             assign(s.tb.id,a.id)
             set_insert(s.pending_kr,s.tb)})))
```

```
    INPUT forward_right WHERE eq(self(),a.chan)
         EFF      ifthenelse(eq(a.id, s.id),
                          assign(s.status,"elected"),
                  ifthenelse(bool_and(less(s.id,a.id),less(1,a.to_go)),
                          {assign(s.status,"dead")
                          %
                          % higher id seen
                          %    -> update value of max
                          %
                          ifthen(less(s.max,int_div(a.id,100)),
                                  assign(s.max,int_div(a.id,100)))
                          assign(s.tf.chan,set_random(out(self())))
                          assign(s.tf.to_go,int_minus(a.to_go,1))
                          assign(s.tf.id,a.id)
                          set_insert(s.pending_fr,s.tf)},
                  ifthenelse(bool_and(less(s.id,a.id),eq(1,a.to_go)),
                          {assign(s.status,"dead")
                          %
                          % higher id seen
                          %    -> update value of max
                          %
                          ifthen(less(s.max,int_div(a.id,100)),
                                  assign(s.max,int_div(a.id,100)))
                          assign(s.tb.chan,set_random(in(self())))
                          assign(s.tb.id,a.id)
                          set_insert(s.pending_ol,s.tb)},
                  {assign(s.tb.chan,set_random(in(self())))
                  assign(s.tb.id,a.id)
                  set_insert(s.pending_kl,s.tb)})))

%
% rest of input actions and all output actions
%   remain unchanged
%
```

```
%
% exact same code as for proc automaton type, with one exception:
%   most significant digit of its id = 7, while most significant digit
%   of proc id's < 7.  max is initialized to 7, and since leader will
%   never see a value higher than 8, there is no need to update max's val.
% this is so that the message from the leader can be seen clearly being
%   sent further and further around the ring.
%
AUTOMATON LEADER
   STATE tuple(id:integer, status:string,
               distance_left:integer, distance_right:integer,
               pending_fl:set(mf), pending_fr:set(mf),
               pending_kl:set(mb), pending_kr:set(mb),
               pending_ol:set(mb), pending_or:set(mb)
               tf:mf, tb:mb, max:integer, stat_int:integer)

   MAINTAIN
         ifthenelse(eq(s.status,"waiting"),assign(s.stat_int,4),
         ifthenelse(eq(s.status,"dead"),assign(s.stat_int,0),
                                        assign(s.stat_int,3)))

   INPUT initially
        EFF    %
               % assign id s.t. most sig digit = 7
               %
               assign(s.id,int_plus(700,str_to_int(name(self()))))
               %
               % initialize max to be 7.
               %
               assign(s.status,"waiting")
               assign(s.distance_left,1)
               assign(s.distance_right,1)
               assign(s.tf.chan,set_random(in(self())))
               assign(s.tf.id,s.id)
               assign(s.tf.to_go,1)
               assign(s.pending_fl,set_single(s.tf))
               assign(s.tf.chan,set_random(out(self())))
               assign(s.pending_fr,set_single(s.tf))
```

```
INPUT forward_left WHERE eq(self(),a.chan)
     EFF     ifthenelse(eq(a.id, s.id),
                     assign(s.status,"elected"),
             ifthenelse(bool_and(less(s.id,a.id),less(1,a.to_go)),
                     {assign(s.status,"dead")
                     assign(s.tf.chan,set_random(in(self())))
                     assign(s.tf.to_go,int_minus(a.to_go,1))
                     assign(s.tf.id,a.id)
                     set_insert(s.pending_fl,s.tf)},
             ifthenelse(bool_and(less(s.id,a.id),eq(1,a.to_go)),
                     {assign(s.status,"dead")
                     assign(s.tb.chan,set_random(out(self())))
                     assign(s.tb.id,a.id)
                     set_insert(s.pending_or,s.tb)},
             {assign(s.tb.chan,set_random(out(self())))
             assign(s.tb.id,a.id)
             set_insert(s.pending_kr,s.tb)})))

INPUT forward_right WHERE eq(self(),a.chan)
     EFF     ifthenelse(eq(a.id, s.id),
                     assign(s.status,"elected"),
             ifthenelse(bool_and(less(s.id,a.id),less(1,a.to_go)),
                     {assign(s.status,"dead")
                     assign(s.tf.chan,set_random(out(self())))
                     assign(s.tf.to_go,int_minus(a.to_go,1))
                     assign(s.tf.id,a.id)
                     set_insert(s.pending_fr,s.tf)},
             ifthenelse(bool_and(less(s.id,a.id),eq(1,a.to_go)),
                     {assign(s.status,"dead")
                     assign(s.tb.chan,set_random(in(self())))
                     assign(s.tb.id,a.id)
                     set_insert(s.pending_ol,s.tb)},
             {assign(s.tb.chan,set_random(in(self())))
             assign(s.tb.id,a.id)
             set_insert(s.pending_kl,s.tb)})))
```

```
INPUT kill_left WHERE eq(self(),a.chan)
     EFF     ifthenelse(eq(s.id,a.id),
                assign(s.status,"dead"),
             {assign(s.tb.chan,set_random(in(self())))
             assign(s.tb.id,a.id)
             set_insert(s.pending_kl,s.tb)})


INPUT kill_right WHERE eq(self(),a.chan)
     EFF     ifthenelse(eq(s.id,a.id),
                assign(s.status,"dead"),
             {assign(s.tb.chan,set_random(out(self())))
             assign(s.tb.id,a.id)
             set_insert(s.pending_kr,s.tb)})

INPUT ok_left WHERE eq(self(),a.chan)
     EFF     ifthenelse(bool_and(eq(s.id,a.id),eq(s.status,"waiting")),
                {assign(s.distance_right,int_times(s.distance_right,2))
                assign(s.tf.chan,set_random(out(self())))
                assign(s.tf.id,s.id)
                assign(s.tf.to_go,s.distance_right)
                set_insert(s.pending_fr,s.tf)},
             ifthen(bool_not(eq(s.id,a.id)),
                {assign(s.tb.chan,set_random(in(self())))
                assign(s.tb.id,a.id)
                set_insert(s.pending_ol,s.tb)}))

INPUT ok_right WHERE eq(self(),a.chan)
     EFF     ifthenelse(bool_and(eq(s.id,a.id),eq(s.status,"waiting")),
                {assign(s.distance_left,int_times(s.distance_left,2))
                assign(s.tf.chan,set_random(in(self())))
                assign(s.tf.id,s.id)
                assign(s.tf.to_go,s.distance_left)
                set_insert(s.pending_fl,s.tf)},
             ifthen(bool_not(eq(s.id,a.id)),
                {assign(s.tb.chan,set_random(out(self())))
                assign(s.tb.id,a.id)
                set_insert(s.pending_or,s.tb)}))
```

```
CLASS
   OUTPUT forward_left
     PRE      bool_not(set_empty(s.pending_fl))
     SEL      assign(a,set_random(s.pending_fl))
     EFF      set_delete(s.pending_fl,a)

CLASS
   OUTPUT forward_right
     PRE      bool_not(set_empty(s.pending_fr))
     SEL      assign(a,set_random(s.pending_fr))
     EFF      set_delete(s.pending_fr,a)

CLASS
   OUTPUT kill_left
     PRE      bool_not(set_empty(s.pending_kl))
     SEL      assign(a,set_random(s.pending_kl))
     EFF      set_delete(s.pending_kl,a)

CLASS
   OUTPUT kill_right
     PRE      bool_not(set_empty(s.pending_kr))
     SEL      assign(a,set_random(s.pending_kr))
     EFF      set_delete(s.pending_kr,a)

CLASS
   OUTPUT ok_left
     PRE      bool_not(set_empty(s.pending_ol))
     SEL      assign(a,set_random(s.pending_ol))
     EFF      set_delete(s.pending_ol,a)

CLASS
   OUTPUT ok_right
     PRE      bool_not(set_empty(s.pending_or))
     SEL      assign(a,set_random(s.pending_or))
     EFF      set_delete(s.pending_or,a)
```

## A.3 High Level I/O Automaton Code

```
Action Signature

   AUTOMATON PROC
        Input: forward_left((chan:automaton_id,to_go:integer,id:integer))
               forward_right((chan:automaton_id,to_go:integer,id:integer))
               kill_left((chan:automaton_id,id:integer))
               kill_right((chan:automaton_id,id:integer))
               ok_left((chan:automaton_id,id:integer))
               ok_right((chan:automaton_id,id:integer))

        Output: forward_left((chan:automaton_id,to_go:integer,id:integer))
                forward_right((chan:automaton_id,to_go:integer,id:integer))
                kill_left((chan;automaton_id,id:integer))
                kill_right((chan:automaton_id,id:integer))
                ok_left((chan:automaton_id,id:integer))
                ok_right((chan:automaton_id,id:integer))

AUTOMATON PROC

The state s of each PROC contains the following information and
initial values:

   id:integer = unique random integer
        Automaton with highiest id is elected the leader.
   status:string = ''waiting''
        Becomes ''relay'' iff. comes to know of proc with higher id.
        Becomes ''elected'' iff. realises that its id is highest in the ring.
   distance_left:integer = 1
   distance_right:integer = 1
        Distance out from proc that previous message was sent in each
        direction, where immediate neighbor is at a distance of 1.
   pending_fl:set((automaton_id,integer,integer)) = {(in(self()),id,1)}
   pending_fr:set((automaton_id,integer,integer)) = {(out(self()),id,1)}
   pending_kl:set((automaton_id,integer)) = { }
   pending_kr:set((automaton_id,integer)) = { }
   pending_ol:set((automaton_id,integer)) = { }
   pending_or:set((automaton_id,integer)) = { }
        Sets of messages waiting to go out.
```

```
%
% input actions
%

%
% message on its way out from the initiating proc
%
forward_left((self(),to_go,id))
effect: % message has made it all the way around the ring
        %   -> elect self to be the leader
        if id=s'.id then
           s.status=''elected''
        % message still on its way out and its id > proc's id
        %   -> kill proc
        %   -> propagate same message to left-hand neighbor
        elseif s'.id<id and to_go>1 then
           s.status=''dead''
           s.pending_fl=s'.pending_fl U {(in(self()),to_go-1,id)}
        % message needs to be turned back and its id > proc's id
        %   -> kill proc
        %   -> send message to initiating proc saying no proc's with
        %      higher id's than its own were encountered.
        elseif s'.id<id and to_go=1 then
           s.status=''dead''
           s.pending_or=s'.pending_or U {(out(self()),id)}
        % proc's id > msg id
        %   -> tell initiating proc to kill itself.
        else
           s.pending_kr=s'.pending_kr U {(out(self()),id)}

forward_right((self(),to_go,id))
effect: if id=s'.id then
           s.status=''elected''
        elseif s'.id<id and to_go>1 then
           s.status=''dead''
           s.pending_fr=s'.pending_fr U {(out(self()),to_go-1,id)}
        elseif s'.id<id and to_go=1 then
           s.status=''dead''
           s.pending_ol=s'.pending_ol U {(in(self()),id)}
        else
           s.pending_kl=s'.pending_kl U {(in(self()),id)}
```

```
%
% message on way back to initiating proc, telling it to kill itself.
%
kill_left((self(),id))
effect: % message reached initiating proc
        %   -> kill proc
        if s'.id=id then
           s.status=''dead''
        % message net yet to initiating proc
        %   -> propagate same message
        else
           s.pending_kl=s'.pending_kl U {(out(self()),id)}

kill_right((self(),id))
effect: if s'.id=id then
           s.status=''dead''
        else
           s.pending_kr=s'.pending_kr U {(out(self()),id)}

%
% message on way back to initiating proc, telling it ok so far.
%
ok_left((self(),id))
effect: % message reached initiating proc and proc still alive
        %   -> send message twice as far out in same direction
        if s'.id=id and s'.status=''waiting'' then
           s.distance_left=s'.distance_left*2
           s.pending_fr=s'.pending_fr U {(out(self()),s.distance_left,s.id)}
        % message not yet to initiating proc
        %   -> propagate same message
        elseif ~(s'.id=s.id) then
           s.pending_ol=s'.pending_ol U {(out(self()),id)}

ok_right((self(),id))
effect: if s'.id=id and s'.status=''waiting'' then
           s.distance_right=s'.distance_right*2
           s.pending_fl=s'.pending_fl U {(in(self()),s.distance_right,s.id)}
        elseif ~(s'.id=s.id) then
           s.pending_or=s'.pending_or U {(in(self()),id)}
```

```
%
% output actions
%

%
% pending set not empty
%    -> select message from set at random
%    -> output selected message
%

forward_left(x)
precondition: x is an element of s'.pending_fl
effect: s.pending_fl=s'.pending_fl-x

forward_right(x)
precondition: x is an element of s'.pending_fr
effect: s.pending_fr=s'.pending_fr-x

kill_left(x)
precondition: x is an element of s'.pending_kl
effect: s.pending_kl=s'.pending_kl-x

kill_right(x)
precondition: x is an element of s'.pending_kr
effect: s.pending_kr=s'.pending_kr-x

ok_left(x)
precondition: x is an element of s'.pending_ol
effect: s.pending_ol=s'.pending_ol-x

ok_right(x)
precondition: x is an element of s'.pending_or
effect: s.pending_or=s'.pending_or-x
```

# Appendix B

# Peterson Code

## B.1 Minimal Spectrum Code

```
%
% data equates
%
DATA message tuple(phase:integer, id:integer, chan:automaton_id)

%
% action signature
%
ACTION initially()
ACTION send_high message
ACTION send_low message
ACTION check_status()
ACTION do_wh()
ACTION do_wl()
```

```
AUTOMATON PROC
   %
   % automaton state information
   %
   STATE tuple(status:string, phase:integer, id:integer,
              received:integer, high:integer, low:integer,
              to_sl:set(message), to_sh:set(message),
              random:integer, temp:message, child:automaton_id,
              waiting_high:set(message), waiting_low:set(message),
              check_waiting:string)

   %
   % start state
   %
   INPUT initially
        EFF     assign(s.status, "active")
                assign(s.phase, 0)
                assign(s.random,int_random(0,8))
                assign(s.id, int_plus(int_times(s.random,100),
                                      str_to_int(name(self()))))
                assign(s.received,0)
                assign(s.child,set_random(out(self())))
                set_init(s.to_sl)
                assign(s.temp.chan,s.child)
                assign(s.temp.phase,s.phase)
                assign(s.temp.id,s.id)
                assign(s.to_sh,set_single(s.temp))
                set_init(s.waiting_high)
                set_init(s.waiting_low)
                assign(s.check_waiting,"no")
```

```
%
% input actions
%

%
% message from next active counterclockwise neighbor
%
INPUT send_high WHERE eq(self(),a.chan)
     EFF     % processor relay
             %   -> propagate msg in tact to clockwise neighbor
             ifthenelse(eq(s.status,"relay"),
                     {assign(s.temp.chan,s.child)
                     assign(s.temp.phase,a.phase)
                     assign(s.temp.id,a.id)
                     set_insert(s.to_sh,s.temp)},
             % proc id = msg id
             %   -> elect self to be the leader
             ifthenelse(eq(a.id,s.id),
                     assign(s.status,"elected"),
             % proc phase = msg phase
             %   -> save msg id in s.high for later comparison
             %   -> increment received counter
             %   -> do send_low output to clockwise neighbor
             ifthenelse(eq(a.phase,s.phase),
                     {assign(s.high,a.id)
                     assign(s.received,int_plus(s.received,1))
                     assign(s.temp.chan,s.child)
                     assign(s.temp.phase,a.phase)
                     assign(s.temp.id,a.id)
                     set_insert(s.to_sl,s.temp)},
             % proc phase < msg phase, but proc still active
             %   -> buffer message until it can be handled
             ifthen(eq(s.status,"active"),
                     {assign(s.temp.chan,self())
                     assign(s.temp.phase,a.phase)
                     assign(s.temp.id,a.id)
                     set_insert(s.waiting_high,s.temp)}))))
```

```
%
% message from next after next active counterclockwise neighbor
%
INPUT send_low WHERE eq(self(),a.chan)
     EFF     % processor relay
             %   -> propagate msg in tact to clockwise neighbor
             ifthenelse(eq(s.status,"relay"),
                     {assign(s.temp.chan,s.child)
                     assign(s.temp.phase,a.phase)
                     assign(s.temp.id,a.id)
                     set_insert(s.to_sl,s.temp)},
             % proc phase = msg phase
             %   -> save msg id in s.low for later comparison
             %   -> increment received counter
             ifthenelse(eq(a.phase,s.phase),
                     {assign(s.low,a.id)
                     assign(s.received,int_plus(s.received,1))},
             % proc phase < msg phase, but proc still active
             %   -> buffer message until it can be handled
             ifthen(eq(s.status,"active"),
                     {assign(s.temp.chan,self())
                     assign(s.temp.phase,a.phase)
                     assign(s.temp.id,a.id)
                     set_insert(s.waiting_low,s.temp)})))
```

```
%
% internal actions
%

   %
   % send_low and send_high received for current phase
   %
   CLASS
      OUTPUT check_status
         PRE    eq(s.received,2)
         EFF    % reinit received counter to zero for next phase
                assign(s.received,0)
                % check buffer of send_high msg's after checking status
                assign(s.check_waiting,"high")
                % next active neighbor id > max(other two id's)
                %   -> remain active
                %   -> adopt id of next active neighbor as own id
                %   -> increment phase counter
                %   -> initiate send_high msg for current proc phase
                ifthenelse(bool_and(greater(s.high,s.id),
                                    greater(s.high,s.low)),
                        {assign(s.id,s.high)
                        assign(s.phase,int_plus(s.phase,1))
                        assign(s.temp.chan,s.child)
                        assign(s.temp.phase,s.phase)
                        assign(s.temp.id,s.id)
                        set_insert(s.to_sh,s.temp)},
                % otherwise
                %   -> proc becomes a relay
                assign(s.status,"relay"))
```

```
%
% check buffer of send_high msg's
%
CLASS
   OUTPUT do_wh
     PRE     eq(s.check_waiting,"high")
     EFF     % check buffer of send_low msg's after send_high msg's
             assign(s.check_waiting,"low")
             % buffer not empty
             %   -> select msg in set with lowest phase counter
             %   -> delete this msg from set
             %   -> handle msg as if just received from send_high
             ifthen(bool_not(set_empty(s.waiting_high)),
                     {assign(s.temp,set_minimum(s.waiting_high))
                     set_delete(s.waiting_high,s.temp)
                     ifthenelse(eq(s.status,"relay"),
                             {assign(s.temp.chan,s.child)
                             set_insert(s.to_sh,s.temp)},
                     ifthenelse(eq(s.temp.id,s.id),
                             assign(s.status,"elected"),
                     ifthenelse(eq(s.temp.phase,s.phase),
                             {assign(s.high,s.temp.id)
                             assign(s.received,int_plus(s.received,1))
                             assign(s.temp.chan,s.child)
                             set_insert(s.to_sl,s.temp)},
                     ifthen(eq(s.status,"active"),
                             set_insert(s.waiting_high,s.temp)))))})
```

```
%
% check buffer of send_low msg's
%
CLASS
   OUTPUT do_wl
     PRE      eq(s.check_waiting,"low")
     EFF      % next either check status, or wait for input
     assign(s.check_waiting,"no")
             % buffer not empty
             %    -> select msg in set with lowest phase counter
             %    -> delete this msg from set
             %    -> handle msg as if just received from send_low,
             %          except if proc is relay,
             %          check send_high buffer again,
             %          since it will not be enabled by check_status
             ifthen(bool_not(set_empty(s.waiting_low)),
                     {assign(s.temp,set_minimum(s.waiting_low))
                     set_delete(s.waiting_low,s.temp)
                     ifthenelse(eq(s.status,"relay"),
                             {assign(s.temp.chan,s.child)
                             set_insert(s.to_sl,s.temp)},
                     ifthenelse(eq(s.temp.phase,s.phase),
                             {assign(s.low,s.temp.id)
                             assign(s.received,int_plus(s.received,1))},
                     ifthen(eq(s.status,"active"),
                             set_insert(s.waiting_low,s.temp))))})
```

```
%
% output actions
%

%
% pending set not empty
%   -> select message from set at random
%   -> output selected message
%

   CLASS
      OUTPUT send_high
         PRE bool_not(set_empty(s.to_sh))
         SEL assign(a,set_random(s.to_sh))
         EFF set_delete(s.to_sh,a)

   CLASS
      OUTPUT send_low
         PRE bool_not(set_empty(s.to_sl))
         SEL assign(a,set_random(s.to_sl))
         EFF set_delete(s.to_sl,a)
```

## B.2 Code with Additions for Simulation Purposes

```
%
% equate & action sigs
%   remain unchanged
%

AUTOMATON PROC
   STATE tuple(status:string, phase:integer, id:integer,
               received:integer, high:integer, low:integer,
               to_sl:set(message), to_sh:set(message)
               random:integer, temp:message, child:automaton_id,
               waiting_high:set(message), waiting_low:set(message),
               check_waiting:string,
               %
               % 1 var added to state for simulation purposes:
               %   stat_int, 7(purple) <-> status=''active''
               %             0(black)  <-> status=''relay''
               %             4(green)  <-> status=''elected''
               stat_int:integer)

   %
   % maintain simulation variable, stat_int, at proper value
   %
   MAINTAIN
        ifthenelse(eq(s.status,"active"),assign(s.stat_int,7),
        ifthenelse(eq(s.status,"relay"), assign(s.stat_int,0),
                                          assign(s.stat_int,4)))

   %
   % all input, internal, and output actions
   %   remain unchanged
   %
```

## B.3 High Level I/O Automaton Code

```
Action Signatures

   AUTOMATON PROC
        Input: send_high((automaton_id, integer, integer))
               send_low((automaton_id, integer, integer))
        Internal: check_status()
                  do_wh()
                  do_wl()
        Output: send_high((automaton_id, integer, integer))
                send_low((automaton_id, integer, integer))
```

AUTOMATON PROC

The state s of each PROC contains the following information and initial values:

```
status:string="active"
     "relay" iff.  there is not possibility that it will be
     chosen as the leader.  "elected" iff. process has been elected
     as the leader.
phase:integer=0
     Acts as a method of synchronizing the messages.  If process
     further back in the ring has already taken 5 steps, while another
     process has only taken 2, the messages that the second process
     receives still need to be handled in order.  The phase ensures
     that this is done.
id:integer=unique random number
     Current id, used to make comparisons and elect leader.
received:integer=0
     Number of messages received with current phase.
high:integer=undefined
     Id of next active process at given phase.
low:integer=undefined
     Id of next-next active process at given phase.
to_sh:set((automaton_id,integer,integer))={(out(self()),phase,id)}
     Messages waiting to go a distance of one out from current
     process.
to_sl:set((automaton_id,integer,integer))={}
     Messages waiting to go a distance of two out from current
     process.
waiting_high:set((automaton_id,integer,integer))={}
     Messages received from send_high action with a higher phase than
     the process was ready to handle at the time of receipt.
waiting_low:set((automaton_id, integer, integer))={}
     Messages received from send_low action with a higher phase than
     the process was ready to handle at the time of receipt.
check_waiting:string="no"
     "no"   -> No need to check waiting queues.
     "high" -> Check waiting_high.
     "low"  -> Check waiting_low.
```

```
%
% input actions
%
%
% message from next active counterclockwise neighbor
%
send_high((self(),t,id))
effect: % processor relay
        %   -> propagate msg in tact to clockwise neighbor
        if s'.status="relay" then
           s.to_sh=s'.to_sh U {(out(self()),t,id)}
           % proc id = msg id
           %   -> elect self to be the leader
        elseif id=s'.id then
           s.status="elected"
        % proc phase = msg phase
           %   -> save msg id in s.high for later comparison
           %   -> increment received counter
           %   -> do send_low output to clockwise neighbor
        elseif t=s.phase then
           s.high=id
           s.received=s'.received+1
           s.to_sl=s'.to_sl U {(out(self()),t,id)}
        % proc phase < msg phase, but proc still active
        %   -> buffer message until it can be handled
        elseif s.status="active" then
           s.waiting_high=s'.waiting_high U {(self(),t,id)}
%
% message from next after next active counterclockwise neighbor
%
send_low((self(),t,id))
effect: % processor relay
        %   -> propagate msg in tact to clockwise neighbor
        if s'.status="relay" then
           s.to_sl=s'.to_sl U {(out(self()),t,id)}
        % proc phase = msg phase
        %   -> save msg id in s.low for later comparison
        %   -> increment received counter
        elseif t=s.phase then
           s.low=id
           s.received=s'.received+1
        % proc phase < msg phase, but proc still active
        %   -> buffer message until it can be handled
        elseif s.status="active" then
           s.waiting_low=s'.waiting_low U {(self(),t,id)}
```

```
%
% internal actions
%

%
% send_low and send_high received for current phase
%
check_status()
precondition:   s'.received=2 and s'.status="active"
effect: % reinit received counter to zero for next phase
        s.received=0
        % check buffer of send_high msg's after checking status
        s.check_waiting="high"
        % next active neighbor id > max(other two id's)
        %    -> remain active
        %    -> adopt id of next active neighbor as own id
        %    -> increment phase counter
        %    -> initiate send_high msg for current proc phase
        if (s'.high > max(s'.id, s'.low)) then
           s.id=s'.high
           s.phase=s'.phase+1
           s.to_sh=s'.to_sh U {(out(self()),s.phase,s.id)}
        % otherwise
        %    -> proc becomes a relay
        else
           s.status="relay"
```

```
%
% check buffer of send_high msg's
%
do_sh()
precondition:   s'.check_waiting="high"
effect: % check buffer of send_low msg's after send_high msg's
        s.check_waiting="low"
        % buffer contains msg whose phase = current proc phase
        %   -> delete this msg from set
        %   -> handle msg as if just received from send_high,
        %         knowing that id phase = proc phase
        if there exists an x element of s.waiting_high s.t.
              x.phase=s.phase then
           s.waiting_high=s'.waiting_high-x
        if s'.status="relay" then
           s.to_sh=s'.to_sh U {(out(self()),x.phase,x.id)}
        elseif x.id=s'.id then
           s.status="elected"
        else
           s.high=x.id
           s.received=s'.received+1
           s.to_sl=s'.to_sl U {(out(self()),x.phase,x.id)}
%
% check buffer for send_low msg's
%
do_sl()
precondition:   s'.check_waiting="low"
effect: % next either check status, or wait for input
        s.check_waiting="no"
        % buffer contains msg whose phase = current proc phase
        %   -> delete this msg from set
        %   -> handle msg as if just received from send_low,
        %         knowing that id phase = proc phase,
        %         except if proc is relay,
        %         check send_high buffer again,
        %         since it will not be enabled by check_status
        if there exists an x element of s.waiting_low s.t.
              x.phase=s.phase then
           s.waiting_low=s'.waiting_low-x
        if s'.status="relay" then
           s.to_sl=s'.to_sl U {(out(self()),x.phaseXS,x.id)}
           s.check_waiting="high"
        else
           s.low=x.id
           s.received=s'.received+1
```

```
%
% output actions
%

%
% pending set not empty
%    -> select message from set at random
%    -> output selected message
%

send_high(x)
precondition:   x element of s'.to_sh
effect: s.to_sh=s'.to_sh-x

send_low(x)
precondition:   x element of s'.to_sl
effect: s.to_sl=s'.to_sl-x
```

# Appendix C

# Dijkstra Code

## C.1 Minimal Spectrum Code

```
%
% data equates
%
DATA mp tuple(chan:automaton_id, from:automaton_id)
DATA mf tuple(chan:automaton_id)
DATA mb tuple(chan:automaton_id, new:boolean)
DATA ma tuple(chan:automaton_id, from:automaton_id, new_int:integer)

%
% action signature
%
ACTION initially()
ACTION probe mp
ACTION forward mf
ACTION back mb
ACTION make_pending()
ACTION ack ma
ACTION terminate()
```

```
AUTOMATON SOURCE
   %
   % automaton state information
   %
   INPUT initially
        EFF     assign(s.status,"waiting")
                assign(s.new, false)
                assign(s.distance,1)
                assign(s.pending, out(self()))
                assign(s.returned,0)

   %
   % input actions
   %

   %
   % response to forward output
   %
   INPUT back WHERE eq(a.chan, self())
        EFF     % set s.new to true if at least one new node found so far
                assign(s.new, bool_or(s.new, a.new))
                % increment response counter
                assign(s.returned, int_plus(s.returned,1))

   %
   % response to probe output
   %
   INPUT ack WHERE eq(a.chan, self())
        EFF     % each child of source will be added to tree at this iteration
                assign(s.new, true)
                % increment response counter
                assign(s.returned, int_plus(s.returned,1))
```

```
%
% internal actions
%

   %
   % responses collected to all msgs sent out and new node found
   %
   CLASS
      OUTPUT make_pending
        PRE      bool_and(eq(s.returned, set_size(out(self()))),
                          eq(s.new,true))
        EFF      % increment distance var, and reinit other values
                 assign(s.returned,0)
                 assign(s.distance, int_plus(s.distance,1))
                 assign(s.new,false)
                 assign(s.pending,out(self()))

   %
   % responses collected to all msgs sent out, no new nodes found
   %
   CLASS
      OUTPUT terminate
        PRE      bool_and(eq(s.returned, set_size(out(self()))),
                 bool_and(eq(s.new,false),
                          eq(s.status,"waiting")))
        EFF      % execution completed
                 assign(s.status,"done")
```

```
%
% output actions
%

   %
   % msgs waiting to go out, first level of tree already in place
   %
   CLASS
      OUTPUT forward
        PRE     bool_and(bool_not(set_empty(s.pending)),
                         greater(s.distance,1))
        SEL     assign(a.chan,set_random(s.pending))
        EFF     set_delete(s.pending,a.chan)

   %
   % msgs waiting to go out, first level of tree being determined
   %
   CLASS
      OUTPUT probe
        PRE     bool_and(bool_not(set_empty(s.pending)),
                         eq(s.distance,1))
        SEL     assign(a.chan,set_random(s.pending))
                assign(a.from,self())
        EFF     set_delete(s.pending,a.chan)
```

```
AUTOMATON PROC
   %
   % automaton state information
   %
   STATE tuple(new:boolean, distance:integer, children:set(automaton_id),
               pending:set(automaton_id), returned:integer,
               parent:automaton_id, to_ack:set(automaton_id),
               back_ok:boolean, root:automaton_id)

   %
   % start state
   %
   INPUT initially
        EFF     assign(s.new,false)
                assign(s.distance,0)
                assign(s.children,out(self()))
                assign(s.root, set_random(all_of_type("SOURCE")))
                ifthen(set_el(s.children,s.root),set_delete(s.children,s.root))
                assign(s.back_ok,false)
                set_init(s.pending)
                assign(s.returned,0)
                set_init(s.to_ack)
```

```
%
% input actions
%

   %
   % probe from parent candidate
   %
   INPUT probe WHERE eq(a.chan, self())
        EFF % insert in set of probes to be acknowledged
            set_insert(s.to_ack,a.from)

   %
   % message from parent
   %
   INPUT forward WHERE eq(a.chan, self())
        EFF     % sent msgs out to all children
                assign(s.pending,s.children)
                % since forward received, back not yet sent, back_ok=true.
                assign(s.back_ok,true)

   %
   % acknowledgment of probe by child candidate
   %
   INPUT ack WHERE eq(a.chan, self())
        EFF     % positive acknowledgment
                %   -> remember new node found
                %   -> increment response counter
                ifthenelse(eq(a.new_int,0),
                        {assign(s.new,true)
                        assign(s.returned,int_plus(s.returned,1))},
                % negative acknowledgment
                %   -> delete child from proc's set of children
                set_delete(s.children,a.from))

   %
   % response from forward output
   %
   INPUT back WHERE eq(a.chan, self())
        EFF     % set s.new to true if at least one new node found so far
                assign(s.new, bool_or(s.new,a.new))
                % increment response counter
                assign(s.returned, int_plus(s.returned,1))
```

```
%
% output actions
%

   %
   % msgs waiting to go out, level of tree being determined=1+level of node
   %
   CLASS
      OUTPUT probe
        PRE     bool_and(bool_not(set_empty(s.pending)),
                          eq(s.distance,1))
        SEL     assign(a.chan, set_random(s.pending))
                assign(a.from,self())
        EFF     set_delete(s.pending,a.chan)

   %
   % msgs waiting to go out, level of tree being determined>1+level of node
   %
   CLASS
      OUTPUT forward
        PRE     bool_and(bool_not(set_empty(s.pending)),
                          greater(s.distance,1))
        SEL     assign(a.chan, set_random(s.pending))
        EFF     set_delete(s.pending,a.chan)
```

```
%
% probes not yet responded to
%
CLASS
   OUTPUT ack
     PRE      bool_not(set_empty(s.to_ack))
     SEL      assign(a.chan, set_random(s.to_ack))
              assign(a.from, self())
              assign(a.new_int, s.distance)
     EFF      set_delete(s.to_ack, a.chan)
              % first probe being acknowledged
              %   -> increment distance counter
              %   -> adopt sender of probe as parent
              ifthen(eq(s.distance,0),
                      {assign(s.distance,1)
                      assign(s.parent,a.chan)})

%
% forward input received, msgs sent to children, all responses collected
%
CLASS
   OUTPUT back
     PRE      bool_and(eq(s.returned, set_size(s.children)),s.back_ok)
     SEL      assign(a.chan, s.parent)
              assign(a.new, s.new)
     EFF      % increment distance counter and reinit other vars
              assign(s.returned,0)
              assign(s.distance, int_plus(s.distance,1))
              assign(s.new,false)
              assign(s.back_ok,false)
```

## C.2 Code with Additions for Simulation Purposes

```
%
% level variable needs to be sent as part of the message when a
% probe is output so that the receiving process knows to make its
% own level one greater, when appropriate
%
DATA mp tuple(chan:automaton_id, from:automaton_id, level:integer)
DATA mf tuple(chan:automaton_id)
DATA mb tuple(chan:automaton_id, new:boolean)
DATA ma tuple(chan:automaton_id, from:automaton_id, new_int:integer)

%
% action sig
%   remains unchanged
%

AUTOMATON SOURCE
   STATE tuple(status:string, new:boolean, distance:integer,
               pending:set(automaton_id), returned:integer
               %
               % 2 vars added to state for simulation purposes:
               %   in_tree, 0(black)  <-> node not yet in shortest paths tree
               %            7(purple) <-> once node add to shortest paths tree
               %   level, holds the level of the node in the tree mod 8
               %
               in_tree:integer, level:integer)

   INPUT initially
        EFF     assign(s.status,"waiting")
                assign(s.new, false)
                assign(s.distance,1)
                assign(s.pending, out(self()))
                assign(s.returned,0)
                %
                % source start off in the tree from the beginning,
                % as the root, which is level 0.
                %
                assign(s.level,0)
                assign(s.in_tree,7)
```

```
%
% all input and internal actions
%   remain unchanged
%

   CLASS
      OUTPUT probe
        PRE     bool_and(bool_not(set_empty(s.pending)),
                         eq(s.distance,1))
        SEL     assign(a.chan,set_random(s.pending))
                assign(a.from,self())
                %
                % if node is adopted as parent of recipient,
                %   recipient will know to make its own level equal 1.
                %
                assign(a.level,1)
        EFF     set_delete(s.pending,a.chan)

%
% output forward action
%   remains unchanged
%
```

```
AUTOMATON PROC

   STATE tuple(new:boolean, distance:integer, children:set(automaton_id),
               pending:set(automaton_id), returned:integer,
               parent:automaton_id, to_ack:set(automaton_id),
               back_ok:boolean, root:automaton_id,
               %
               % 2 vars added to state for simulation purposes:
               %   in_tree, 0(black)  <-> node not yet in shortest paths tree
               %            7(purple) <-> once node add to shortest paths tree
               %   level, holds the level of the node in the tree mod 8
               %
               level:integer, in_tree:integer)

   %
   % use distance variable to determine whether or not node is yet in tree.
   %   if node has been added to tree, distance variable will have been
   %   incremented.  if distance is still 0, then node must not yet be in tree.
   %
   MAINTAIN
        ifthenelse(eq(s.distance,0), assign(s.in_tree,0),
                                     assign(s.in_tree,7))

%
% start state
%   remains unchanged
%

   INPUT probe WHERE eq(a.chan, self())
        EFF     set_insert(s.to_ack,a.from)
                %
                % if node does not yet have a partent,
                %   it will be added to tree at level carried in msg.
                %
                ifthen(eq(s.distance,0), assign(s.level,a.level))
        EFF     assign(s.new, bool_or(s.new,a.new))
                assign(s.returned, int_plus(s.returned,1))

%
% rest of input actions
%   remain unchanged
%
```

```
   CLASS
      OUTPUT probe
        PRE      bool_and(bool_not(set_empty(s.pending)),
                          eq(s.distance,1))
        SEL      assign(a.chan, set_random(s.pending))
                 assign(a.from,self())
                 %
                 % if node is adopted as parent of recipient,
                 %   recipient will know to make its own level equal a.level
                 %
                 ifthenelse(eq(s.level,7),
                         assign(a.level,0),
                         assign(a.level, int_plus(s.level,1)))
        EFF      set_delete(s.pending,a.chan)

   CLASS
      OUTPUT ack
        PRE      bool_not(set_empty(s.to_ack))
        SEL      assign(a.chan, set_random(s.to_ack))
                 assign(a.from, self())
                 assign(a.new_int, s.distance)
        EFF      set_delete(s.to_ack, a.chan)
                 ifthen(eq(s.distance,0),
                         {assign(s.distance,1)
                         assign(s.parent,a.chan)
                         %
                         % color edges in the shortest paths tree purple
                         %
                         edge_val(s.parent,self(),7)})

%
% rest of output actions
%   remain unchanged
%
```

## C.3 High Level I/O Automaton Code

```
Action Signature

   AUTOMATON SOURCE
        Input: ack(automaton_id, automaton_id, integer)
               back(automaton_id, boolean)
        Internal: make_pending()
                  terminate()
        Output: probe(automaton_id, automaton_id)
                forward(automaton_id)

   AUTOMATON PROC
        Input: probe(automaton_id, automaton_id)
               forward(automaton_id)
               ack(automaton_id, automaton_id, integer)
               back(automaton_id, boolean)
        Output: probe(automaton_id, automaton_id)
                forward(automaton_id)
                ack(automaton_id, automaton_id, integer)
                back(automaton_id, boolean)


The state s of the SOURCE contains the following information and
initial values:

   status:string="waiting"
        "done" iff. the simulation has been completed.
   new?:boolean=false
        True iff. at least one new node yet added to shortest paths
        tree at current level.
   distance:integer=1
        Level of tree, rooted at SOURCE, to which next set of messages
        must travel.
   pending:set(automaton_id)=children(self())
        Automaton id's of nodes to which messages are waiting to go
        out.
   returned:integer=0
        Number of responses received to messages sent out at current
        level.
```

```
%
% input actions
%

   %
   % response to probe output
   %
   ack(self(),f,n)
   effect: % each child of source will be added to tree at this iteration
           s.new?=true
           % increment response counter
           s.returned=s'.returned+1

   %
   % response to forward output
   %
   back(self(),n)
   effect: % set s.new? to be true if at least one new node found so far
           s.new?=s'.new? or n
           % increment response counter
           s.returned=s'.returned+1
```

```
%
% internal actions
%

    %
    % responses collected to all msgs sent out and new node found
    %
    make_pending()
    precondition: s'.returned=size(children(self()))
                  s'.new?=true
    effect: s.returned=0
            s.distance=s'.distance+1
            s.new?=false
            s.pending=children(self())

    %
    % responses collected to all msgs sent out, no new nodes found
    %
    terminate()
    precondition: s'.returned=size(children(self()))
                  s'.new?=false
                  s'.status="waiting"
    effect: s.status="done"
```

```
%
% output actions
%

   %
   % msgs waiting to go out, first level of tree being determinod
   %
   probe(x,f)
   precondition: x element of s'.pending
                 s'.distance=1
                 f=self()
   effect: s.pending=s'.pending-x

   %
   % msgs waiting to go out, first level of tree already in place
   %
   forward(x)
   precondition: x element of s'.pending
                 s'.distance>1
   effect: s.pending=s'.pending-x
```

AUTOMATON PROC

The state s of each PROC contains the following information and initial values:

```
new?:boolean=false
     True iff. at least one new node yet added to shortest paths
     tree at current level.
distance:integer=0
     Distance from node, out from SOURCE, which current set of
     messages need to travel.
children:set(automaton_id)=children(self())
     Initially set of all children of node in configuration other
     than the SOURCE and finally all children of node in the
     shortest paths tree.
pending:set(automaton_id)={}
     Automaton id's of nodes to which messages are waiting to go
     out.
returned:integer=0
     Number of responses received to messages sent out at current
     level.
parent:automaton_id=undefined
     Set to automaton id of parent of node in shortest paths tree,
     once known.
to_ack:set(automaton_id)={}
     Responses to probes waiting to go back.
back_ok:boolean=false
     Used when process has no children of its own in order to know
     when to respond to a forward message.  true iff. forward input
     received and back output not yet sent.
```

```
%
% input actions
%

   %
   % probe from parent candidate
   %
   probe(self(),f)
   effect: % insert in set of probes to be acknowledged
           s.to_ack=s'.to_ack U f

   %
   % message from parent
   %
   forward(self())
   effect: % send msgs out to all children
           s.pending=s'.children
           % since forward received, back not yet sent, back_ok=true
           s.back_ok=true

   %
   % acknowledgment of probe by child candidate
   %
   ack(self(),f,n)
   effect: % positive acknowledgment
           %   -> remember new node found
           %   -> increment response counter
           if n=0 then
              s.new=true
              s.returned=s'.returned+1
           % negative acknowledgment
           %   -> delete child from proc's set of children
           else
              s.children=s'.children-f

   %
   % response from forward output
   %
   back(self(),n)
   effect: % set s.new to be true if at least one new node found so far
           s.returned=s'.returned+1
           % increment reponse counter
           s.new?=s'.new? or n
```

```
%
% output actions
%

   %
   % msgs waiting to go out, level of tree being determined=1+level of node
   %
   probe(x,f)
   precondition: x element of s'.pending
                 s'.distance=1
                 f=self()
   effect: s.pending=s'.pending-x

   %
   % msgs waiting to go out, level of tree being determined>1+level of node
   %
   forward(x)
   precondition: x element of s'.pending
                 s'.distance>1
   effect: s.pending=s'.pending-x
```

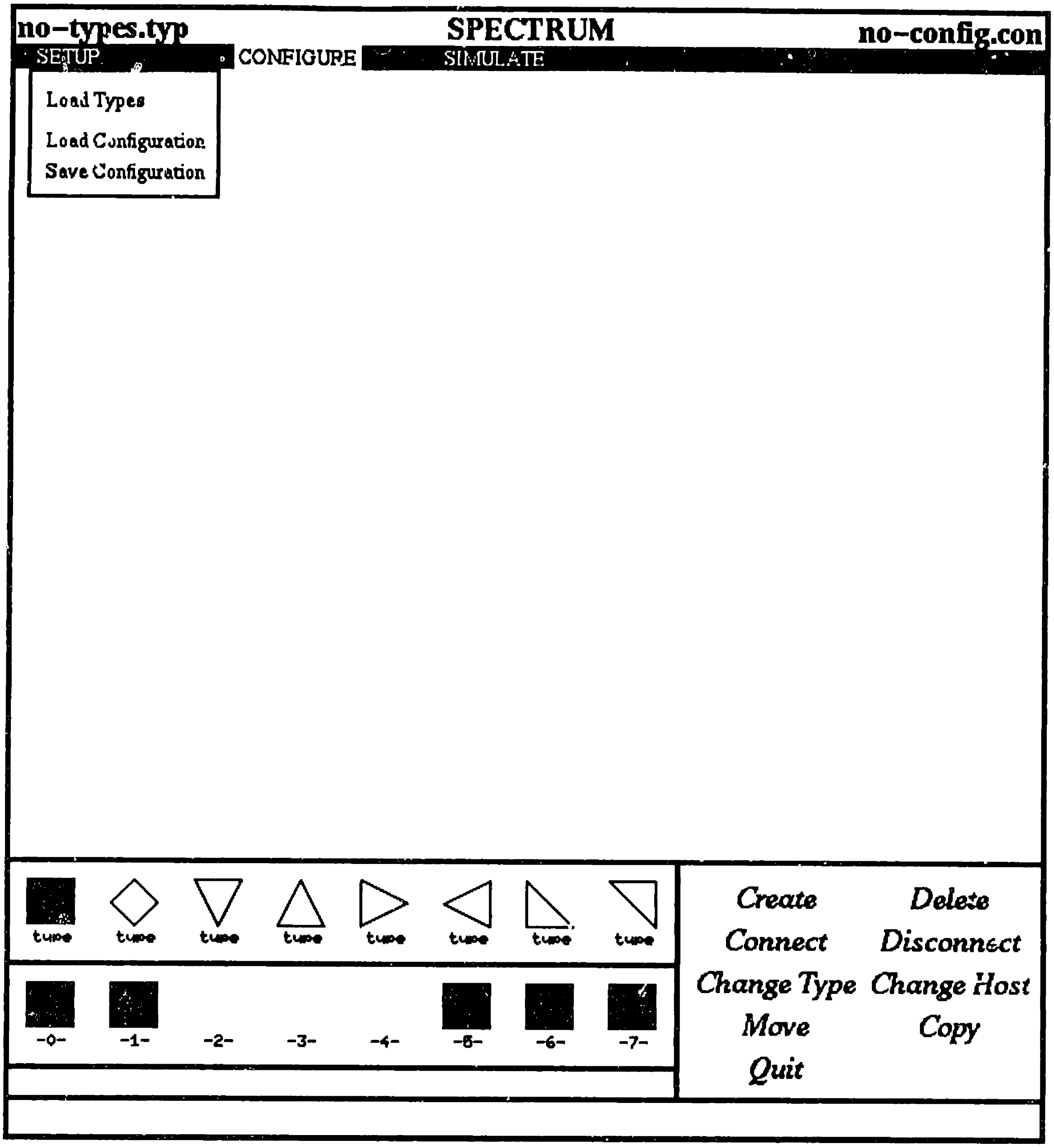
```
%
% probes not yet responded to
%
ack(x,f,n)
precondition: x element of s'.to_ack
              f=self()
              n=s'.distance
effect: s.to_ack=s'.to_ack-x
        % first probe being acknowledged
        %   -> increment distance counter
        %   -> adopt sender of probe as parent
        if s'.distance=0 then
           s.distance=1
           s.parent=x

%
% forward input received, msgs sent to children, all responses collected
%
back(x,n)
precondition: s'.returned=size(s'.children)
              x=s'.parent
              n=s'.new?
              s'.back_ok
effect: % increment distance counter and reinit other vars
        s.returned=0
        s.distance=s'.distance+1
        s.new?=false
        s.back_ok=false
```

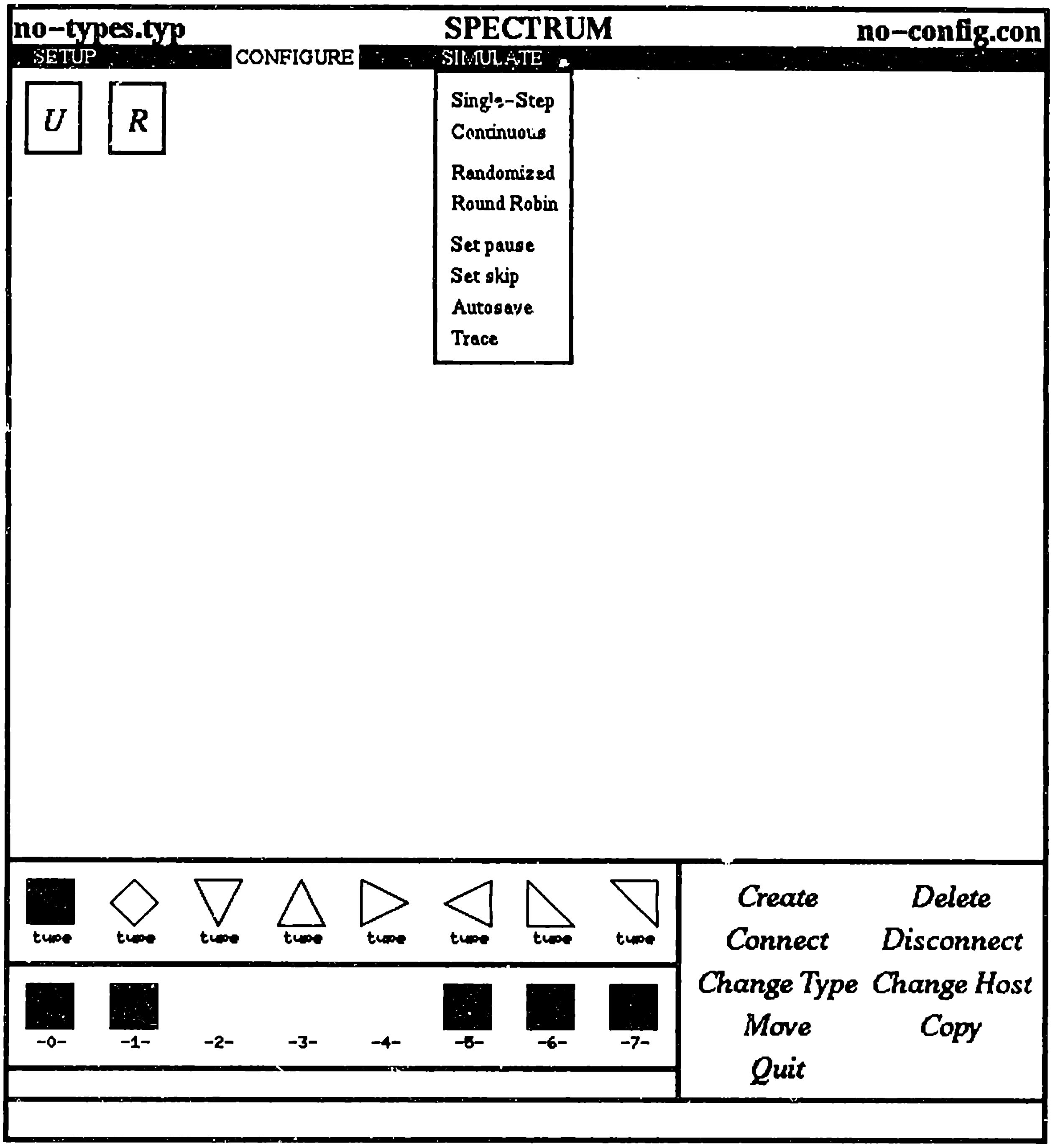
# Appendix D

# Spectrum Display

## D.1 SETUP Menu Down



## D.2 SIMULATE Menu Down



# Bibliography

[Awe] Baruch Awerbuch. Distributed shortest paths algorithms. Extended Abstract.

[DS80] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, August 1980.

[FLS87] A. Fekete, N. Lynch, and L. Shrira. A modular proof of correctness for a network synchronizer. In *Proceedings of the 2nd International Workshop on Distributed Algorithms*, July 1987.

[Gola] Kenneth J. Goldman. The spectrum programming language. Unpublished Draft.

[Golb] Kenneth J. Goldman. The spectrum user interface. Unpublished Draft.

[HS80] D. Hirschberg and J. Sinclair. Decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, November 1980.

[LG88] Nancy A. Lynch and Kenneth J. Goldman. Distributed algorithms. MIT Laboratory for Computer Science, Fall 1988. Lecture Notes for 6.852.

[LT86] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proccedings of 6th ACM Symposium of Principles of Distributed Computing*, 1986.

[Pet82] G. L. Peterson. An $o(nlogn)$ unidirectional distributed algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, October 1982.

## ACKNOWLEDGMENTS

I would like to thank Professor Lynch and Kenneth Goldman for their valuable time and helpful advice. Professor Lynch consistantly made time in her busy schedule to discuss progress on this work and answer questions. Ken was also very willingly available and excellent at explaining points of confusion. He will make a superb professor. And to everyone who encouraged me along the way, thank you. I needed it.