# The Evolution of a Distributed Dataflow Processing Model using Ada

Scott James
Management Communications and Control, Inc.
Suite 220
2000 N. 14th Street
Arlington VA 22201
james@mcci-arl-va.com

**Abstract**

This paper presents the stages of design for a dataflow program. A sequence of models is presented in increasing order of complexity, demonstrating the values and shortcomings of each. Techniques and challenges mapping the dataflow paradigm to the Ada concurrency model and distribution annex are described.

## 1 Definitions and Intent

A *dataflow graph* [4] can be thought of as plumbing. Information flows as water through the piping (*queues*) is stored and released from various sinks, fixtures, heaters and pumps (*nodes*) until finally released from the system. [1]

A bit more precisely, a dataflow graph is defined as a directed acyclic graph with data passing through the directed edges, or queues, and dynamics at the nodes. Each queue is connected to exactly two nodes: an upstream node attached to its *tail* and a downstream node attached to its *head* (data flows downstream). Each node possesses a set of *input queues* entering it, and a set of *output queues* leaving it, either set possibly being empty. Each queue contains a *threshold* and a *buffer size* and receives its data in a first-in-first-out fashion from its upstream node. Queues are said to be *linked* to nodes at *ports*, thus there are both *output ports* and *input ports*.

When a queue accumulates a threshold of data, it is said to have *reached threshold*. When every upstream queue of a node has reached threshold, the node is ready to *fire*. Upon firing, a node *reads* and *consumes* data from each of its upstream queues, processes this data, and writes results to its downstream queues. All nodes can function concurrently, and conceptually begin processing as soon as they reach a fire condition.

[1] This paradigm is used in many endeavors, specific examples include circuit modeling and signal processing applications.



Nodes with no upstream queues are said to be *sources* and generate data autonomously. Nodes with no downstream queues are said to be *sinks* and can be thought to discard or display the data. Completely disconnected nodes are of no interest to the model.

Our goal is to model this behavior, providing a system with no deadlocks and minimal bottlenecks to dataflow. For purposes of analysis, the node processing time will be loosely considered *significant* compared to the data transfer time, thus we wish to minimize any delays on data transfer due to node processing.

## 2 Nodes with Queues

We begin by organizing the dataflow graph into concurrent units, each consisting of a node and all upstream queues attached to the node (Figure 1). Each unit accumulates data in its input queues until all thresholds are met, at which time the unit fires and produces results onto its downstream unit(s). The no-deadlock condition is met as upstream units will only call downstream units and the graph is acyclic.

```
protected type nodes_with_queues (input: ports,
                                  output: ports) is

   procedure append (token: tokens;
                     onto_input_port: ports);

   procedure link (output_queue: ports;
                   to_unit: access_to_nodes_with_queues;
                   on_input_port: ports);
   -- ....
private
   buffer_array: buffer_arrays (1 .. input);
   -- ...
end nodes_with_queues;
```

Figure 1: Nodes with Queues

However, a serious bottleneck exists: while a downstream unit is processing, any upstream unit will be blocked from attempting to pass data to the downstream unit. As a result, any attempts to write to this upstream unit will also be blocked, and so forth, cascading upstream.

## 3 Separate Queue and Node Objects

A preferable solution is to allow writing to the data queues while nodes are processing, freeing upstream nodes to continue processing. To accomplish this we will break the interdependence of the data queues and processing nodes by separating the node and its upstream queues (and thus all queues and nodes) into different units (Figures 2 and 3).

```
type tips is (head, tail);
protected type queues is
   procedure initialize (with_threshold: thresholds);
   procedure append (token: tokens);
   procedure consume (amount: amounts);
   procedure link (at_tip: tips;
                   to_node: access_to_nodes;
                   on_port: ports);
   procedure unlink (at_tip: tips);
   entry read (amount: amounts;
               token: out tokens);
   --...
private
   buffer: buffers;
   -- ...
end queues;
```

Figure 2: Queues

```
type directions is (input, output);
task type nodes (input, output: ports) is
   procedure append (token: tokens;
                     onto_input_port: ports);

   procedure link (in_direction: directions;
                   on_port: ports;
                   to_queue: access_to_queues);

   procedure unlink (in_direction: directions;
                     on_port: ports);
end nodes;

task body nodes is
   port_array: port_arrays;
   queue: array (directions,ports) of access_to_queues:
   --...
begin
   loop
      -- ..
      queue(input,port).read (amount, token);
      -- .. process
      queue(output,port).append (token);
      -- ..
   end loop;
end nodes;
```

Figure 3: Nodes

In this model, the queue will remain a **protected** type but the node will be converted into a **task** waiting on an **entry** point in the queue until data is available to read. Again, deadlock is impossible as upstream units only call downstream units, and the graph is acyclic. Furthermore, while a downstream node is processing, a queue may be receiving data from its upstream node, thus releasing the bottleneck of the previous model.

## 4 Mailbox

One nuisance with the previous model is that for multiple input queues, a node must wait upon multiple entry points. [2] To tidy this interface we will introduce another object: a mailbox. The mailbox in this sense is simply a repository for upstream queues to place messages that data is ready for downstream nodes.

Consider the case where a node possesses only a single input queue. The desired flow of handshaking between an upstream queue and a downstream node might take place as in Figure 4.

| | |
|---|---|
| | queue leaves ready message |
| node awakens | |
| node clears data-ready | |
| node fires | |
| node reads and consumes data | |
| | queue reaches threshold |
| | queue indicates data ready |
| node completes processing | |
| node sees data-ready | |
| node clears data-ready | |
| node refires | |
| node reads and consumes data | |
| node sees no data-ready | |
| node goes to sleep | |

Figure 4: Upstream Handshaking

To clarify this description we create three states:

**Sleeping:** Node asleep; no queue data ready

**Firing:** Node awake; no queue data ready

**Fireable:** Node awake or asleep; queue data ready

The state transition diagram appears in Figure 5.

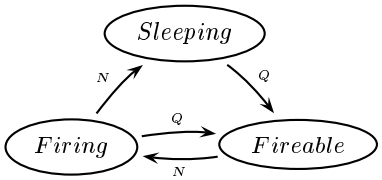


Figure 5: Firing States

Figure 8 shows the node task waiting on an **entry** point in a mailbox (Figure 6). The no-deadlock condition is preserved as there are no cycles between the respective entities (Figure 9): mailboxes call nothing, queues call mailboxes, and nodes call both queues and mailboxes. Downstream nodes can be blocked from an upstream queue while a node upstream to the queue is writing data, but this wait is considered acceptable, as the queue is simply acting as a protective wrapper around the queue data buffer.

With multiple input queues the semantics of the states become:

[2] Discussion on the lack of Ada support for multiple entry points can be found in [1].

**Sleeping:** Node asleep; some input queue not at threshold

**Firing:** Node awake; some input queue not at threshold

**Fireable:** Node awake or asleep; all input queues at threshold

Otherwise, the transition diagram is as in Figure 5.

```
type node_statuses is (sleeping,
                       firing,
                       linking,
                       fireable,
                       linkable);

protected type mailboxes (input, output: ports) is
   -- graph manager access
   procedure link (in_direction: directions;
                   on_port: ports;
                   to_queue: access_to_queues);

   procedure unlink (in_direction: directions;
                     on_port: ports);
   -- queue access
   procedure indicate_ready (on_port: ports);
   -- node access
   entry process (port_array: out port_arrays;
                  status: out node_statuses);
   -- ...
private
   port_array: port_arrays;
   notify_the_node: boolean:=false;
   -- ...
end mailboxes;
```

Figure 6: Mailbox

```
protected body mailboxes is
   -- ...
   entry process (port_array: out port_arrays;
                  status: out node_statuses)
   when notify_the_node is
   begin
      --...
   end process;
   -- ...
end mailboxes;
```

Figure 7: Mailbox body

## 5 Linking/Unlinking

We now wish to add the ability for queues to be unlinked and relinked to nodes during dataflow processing. Linking/unlinking commands will be issued by an external *graph manager*, submitted to the mailboxes, received by the nodes, and finally sent from the nodes to the queues. The linking states are:

**Sleeping:** Node asleep; no un/link requests

**Linking:** Node processing links; no un/link requests

**Linkable:** Node awake or asleep; un/link requests waiting

The state diagram appears in Figure 10.

When a mailbox has no input queues for a particular input port, the port will said to be *unlinked*. When a port has

```
task type nodes is
   entry initialize (with_mailbox: access_to_mailboxes);
   -- ...
end nodes;

task body nodes is
   mailbox: access_to_mailboxes;
   status: statuses;
   port_array: port_arrays;
   -- ...
begin
   accept initialize (with_mailbox: access_to_mailboxes) do
      mailbox:=with_mailbox;
   end initialize;
   loop
      mailbox.process (port_array, status);
      -- ...
      case status is
         when sleeping =>
            null;
         when firing =>
            --...
         when linking =>
            -- ...
      end case;
   end loop;
   -- ...
end nodes;
```

Figure 8: Node Task

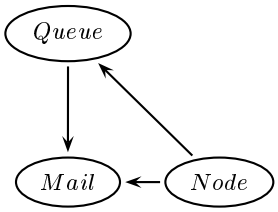


Figure 9: Node-Queue-Mailbox Interaction

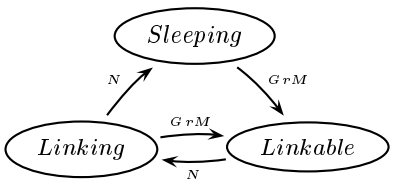


Figure 10: Linking States

a queue which is ready to be linked the port is said to be *linkable*. once the node receives the link request from the mailbox, the port state transitions from linkable to *linked*. Similarly, when a port has a queue which is ready to be unlinked from a linked port, it is said to be *unlinkable* until the node recognizes the request. The relevant diagram is shown in Figure 11. A queue will store its own connection statuses for its head and tail but need only store the *linked* and *unlinked* states.

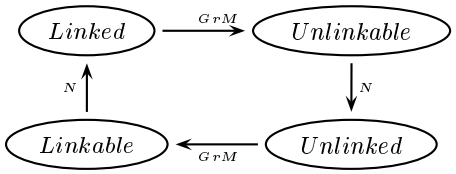


Figure 11: Port Perspective of Linking/Unlinking

Synchronization of these linking calls is an issue. A graph manager may, for instance, wish to disconnect one queue from one node and connect it to another. Before reconnecting, it wants to be assured that the first node is disconnected or else submitting a link command to another node could result in conflict. To alleviate this conflict, an **entry** call is provided by the queue type and guarded until the queue status changes to unlinked (Figure 12).

```
type connection_statuses is (unlinked,
                             linked);

protected type queues is
   -- ...
   entry confirm (at_tip: tips;
                  status: connection_statuses);
   --...
private
   buffer: buffers;
   connection: connections;
   -- ...
end queues;

protected body queues is
   -- ...
   entry confirm (at_tip: tips;
                  status: connection_statuses)
   when connection (head).status_changed or else
    connection (tail).status_changed is
   begin
      for tip in tips loop
         connection (tip).status_changed:=false;
      end loop;
      if connection (at_tip).status /= status then
         requeue confirm;
      end if;
   end confirm;
   -- ...
end queues;
```

Figure 12: Queue with Confirmation

A composite state transition diagram for firing and linking is shown in 13.

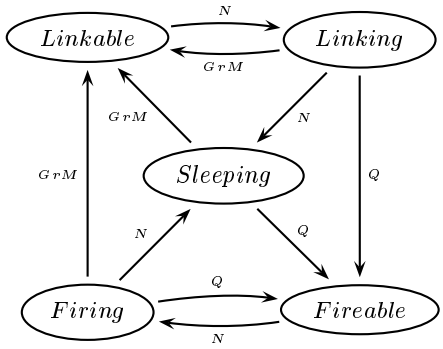


Figure 13: Mailbox Perspective of Linking and Firing

## 6 Distribution

To enable high performance on multiple processor platforms, we wish to distribute these concurrent objects across the hardware. For the purpose of distribution, mailboxes will always be associated with their corresponding node.

The immediate difficulty is that the standardized mechanism for Ada distribution is at the package level (via categorization pragmas) and not at the type level (specifically protected and task types). Of the two categorization pragmas which allow the package to be distributed, **remote_call_interface** (RCI) does not allow any task or protected types in its declaration and **shared_passive** forbids tasks and protected types with entry declarations entirely [3]. As a result, the concurrent objects must reside in the body of an RCI categorized package.

The consequence of this is that if we wish to treat the concurrent objects as data types we must hold a database of objects inside the RCI packages and, upon initialization, return handles to the actual objects. Additionally, for the objects to be assigned to partitions, a distributor package must exist which contains the necessary mappings to the various packages (Figure 14). Example packages are shown in Figures 15, 16,17, and 18.

One remaining point regarding distribution is to what extent asynchronous communication can be allowed between partitions. The fortunate answer is that most procedures with no returning values can be called asynchronously. The reason is that, with the exception of the append routine, every command issued between node and queue object without parameters is the last command sent before waiting for a response; e.g. once queues indicate they have reached threshold they wait until data has been read and consumed, once nodes indicate they have read the data they wait until more data is ready, etc... The graph manager will potentially have synchronization difficulties during linking/unlinking, but these are the same difficulties mentioned in Section 5 and are rectified using the same methods.

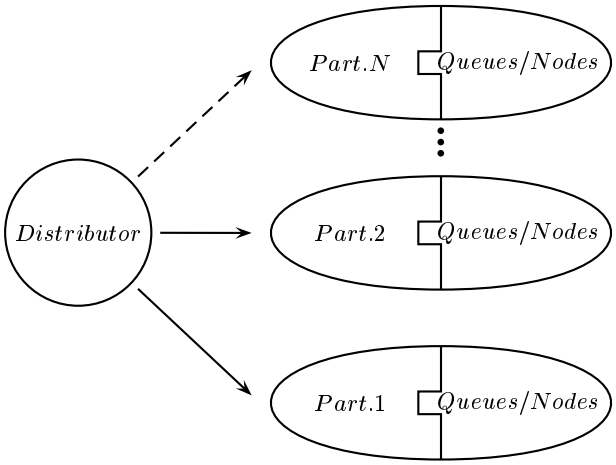


Figure 14: Partitioning Architecture

```
package distributor is
   -- queues
   procedure initialize (queue: out distributed_queues;
                         on_partition: partitions);

   procedure append (to_queue: distributed_queues;
                     data: tokens);

   procedure consume (from_queue: distributed_queues;
                      amount: amounts);

   function read (from_queue: distributed_queues;
                  amount: amounts)
                  return tokens;

   procedure link (queue: distributed_queues;
                   at_tip: tips;
                   to_node: distributed_nodes;
                   on_port: ports);

   procedure unlink (queue: distributed_queues;
                     at_tip: tips);

   procedure confirm (on_queue: distributed_queues;
                      at_tip: tips;
                      status: connection_statuses);
   -- nodes
   procedure initialize (node: out distributed_nodes;
                         on_partition: partitions);

   procedure indicate_ready (for_node: distributed_nodes;
                             in_direction: directions;
                             on_port: ports);

   procedure link (node: distributed_nodes;
                   in_direction: directions;
                   on_port: ports;
                   to_queue: distributed_queues);

   procedure unlink (node: distributed_nodes;
                     in_direction: directions;
                     from_port: ports);
end distributor;
```

Figure 15: Distributor

```
package body distributor is
   -- ...
   procedure initialize (queue: out distributed_queues;
                         on_partition: partitions) is
      partitioned_queue: partitioned_queues;
   begin
      case on_partition is
         when first =>
            partition1.initialize (partitioned_queue);
         when second =>
            partition2.initialize (partitioned_queue);
         -- ...
      end case;
      queue:=to_distributed (partitioned_queue);
   end initialize;
   -- ...
end distributor;
```

Figure 16: Distributor Body

```
package sample_partition is
   pragma remote_call_interface;
   -- queues
   procedure initialize (queue: out partitioned_queues);
   procedure link (queue: partitioned_queues;
                   at_tip: tips;
                   to_node: distributed_nodes;
                   on_port: ports;
   pragma asynchronous (link);

   function read (from_queue: partitioned_queues;
                  amount: amounts);
                  return tokens;

   procedure consume (from_queue: partitioned_queues;
                      amount: amounts);
   pragma asynchronous (consume);

   procedure append (to_queue: partitioned_queues;
                     data: tokens);
   -- pragma synchronous (append);

   procedure confirm (on_queue: partitioned_queues;
                      at_tip: tips;
                      status: connection_statuses);
   -- pragma synchronous (confirm);
   -- nodes
   procedure initialize (node: out partitioned_nodes);

   procedure indicate_ready (for_node: partitioned_nodes;
                             in_direction: directions;
                             on_port: ports);
   pragma asynchronous (indicate_ready);

   procedure link (node: partitioned_nodes;
                   in_direction: directions;
                   on_port: ports;
                   to_queue: distributed_queues);
   pragma asynchronous (link);

   procedure unlink (node: partitioned_nodes;
                     in_direction: directions;
                     from_port: ports);
   pragma asynchronous (unlink);
end sample_partition;
```

Figure 17: Example Partition

```
package body sample_partition is
   -- ..
   queue_array is array (partitioned_queues) of queues;
   -- ...
   procedure initialize (queue: out partitioned_queues) is
   begin
      -- ...
      get_next_available_queue (queue);
      queue_array(queue).initialize;
   end initialize;
   --...
end sample_partition;
```

Figure 18: Example Partition Body

## 7 Conclusions

A model for a simple dataflow program was presented and a methodology for an Ada implementation was shown. The Ada concurrency and distribution paradigms provided, for the most part, a natural fit.

One area where the implementation felt contrived was mentioned: the need for a distributor of object types to partitions. This is primarily an aesthetic issue and a debated one at that. The granularity of the Ada distribution mechanism is a complicated issue and distribution models based on types have been discussed [6].

It will be of interest to see how the Ada concurrency and distribution model evolves as more practitioners discover and utilize its features.

## References


[1] Burns, A., and Wellings, A. *Concurrency in Ada*. Cambridge University Press, 1995.

[2] Cohen, N. H. *Ada as a Second Language*. McGraw-Hill, 1996.

[3] Intermetrics. *Ada Reference Manual*, ISO/IEC 8652:1995 ed., 1995.

[4] Karp, R. M., and Miller, R. E. Properties of a model for parallel computations: determinacy, termination, queueing. *SIAM J.Appl.,Math* (1966).

[5] Levine, T. Deadlock control with Ada95. *Ada Letters XVIII*, 2 (March/April 1998).

[6] Nielsen, K. *Ada in Distributed Real-Time Systems*. McGraw-Hill, 1990.

[7] Pettit, R. G. Using Ada 95 for the design of distributed real-time systems. In *Tri-Ada'96 Conference* (December 1996), ACM SIGAda, pp. 49–55.

[8] Sih, G. C., and Lee, E. A. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems 4*, 2 (1993).