Network-on-Chip (NoC) Simulation with ALRT-DEVS: Supplementary Material and Experiments

Soroosh Gholami
Hessam S. Sarjoughian

ASUCIDSE-CSE-2013-001

Arizona Center for Integrative Modeling & Simulation
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, Arizona, USA

November 2013

1 Introduction

This document is provided as collection of supplementary material for our research on real-time M&S of Network-on-Chip. In Section 2, we provided the Booksim simulator configuration for the experiments done in our research with DEVS-Suite real-time simulation engine. Also, here we performed several experiments to validate the NoC simulator. In Section 3, several DEVS-Suite visualizations of the NoC models are depicted. Section 4 contains all ALRT-DEVS models of NoC components (input/output port, crossbar, packetizer, depacketizer, link, and processing element). Finally, Section 5 provides an insight to the operation of the simulator and our real-time simulation protocol.

2 BookSim Configuration and Comparison with DEVS-Suite

In this section, we conducted several experiments to demonstrate the capabilities of DEVS-Suite in NoC simulation. Consequently, for limited number of NoC configurations, we compared the experimentation results of BookSim simulator against that of DEVS-Suite. Of course, experiments are done in logical times since BookSim does not support real-time simulation of NoC.

The input configuration file for the BookSim simulator is shown in Figure 1. This file holds all the information that BookSim needs to simulate the NoC, such as topology, size, routing algorithm, allocation policy, flow control method, etc. We modify several of these parameters to construct various NoC configurations.

As for the DEVS model, the ALRT-DEVS specification of NoC in the paper, provides the same platform to formulate various configurations of NoC. Identical to the specification in the paper, the NoC models here incorporate on/off flow control method, X-Y routing algorithm, and round-robin allocation policy. Table 1 summarizes NoC configurations for the experiments conducted in this section. While these variables remain unchanged throughout these experiments, the size of the network and injection rate of processors are modified to analyze their impact on NoC measures such as flit latency.

In the first experiment, we analyzed the impact of flit generation rate (for each PE) on average flit latency. Flit generation rate was increased from 0.1 (flit/cycle) up to 0.2 (flit/cycle) and the results are reported in Table 2.

Figure 2 is a statistical analysis on the five different runs of experiment one. Of course, one of the most important reasons in these variations is the inherent randomness in packet generation and destination setting. Also, virtual channels and ports are serviced based on round robin policy which may contribute to more variation in times of congestion.

Next is changing the size of the network and measuring the same variables. What we expect is to see increasing flit latency as the network becomes larger in size. We conducted this experiments with 4 different sizes from 9 to 36 nodes and the results are reported in Table 3.

```
1.  num_vcs    =   2;            //Number of Virtual Channels
2.  vc_buf_size = 8;             //Number of buffers for each channel
3.  wait_for_tail_credit = 1;
4.
5.  vc_allocator = separable_input_first;     //Virtual channel allocation policy
6.  sw_allocator = separable_input_first;     //Switch allocation policy
7.
8.  credit_delay   =   1;     //One cycle for credits to reach the next switch
9.  routing_delay =   1;      //One cycle for routing
10. vc_alloc_delay =  1;      //One cycle for vc allocation
11. sw_alloc_delay =  1;      //One cycle for switch allocation
12. st_final_delay =  2;      //One cycle for switch traversal
13.
14. input_speedup =   1;         //Input/output ratio in crossbar
15. output_speedup =  1;
16. interval_speedup = 1;
17.
18. sim_type   =   latency;      //Let all packets drain
19. warmup_period =   0;
20. sample_period =   1000;
21. sim_count =   5;             //Perform each simulation 5 times
22.
23. topology   =   mesh;
24. k = 3;                   //Number of nodes in each dimension
25. n = 2;                   //Number of dimentions
26.
27. routing_function  =   dor;       //Dimension-order-routing
28. const_flits_per_packet = 1;      //Flits per packet
29. use_read_write =  0;
30. traffic = uniform;
31. injection_rate =  0.1;           //One flit per 10 cycles
```

Figure 1: External transition and output functions for the simulated relay in mixed simulation scheme

Table 1: Configuration variables of NoC

Variable	Value (cycles)	Variable	Value (cycles)
numOfVirtualChannels	2	routingMax	1
maxInputBufferSize	8	routingMin	1
numberOfPorts	4	vcAllocationMax	1
numOfFlits	1	vcAllocationMin	1
linkTraversalMin	10	swAllocationMax	1
linkTraversalMax	10	swAllocationMin	1
statusTransmitterMax	1	swTraversalMax	1
statusTransmitterMin	1	swTraversalMin	1

Table 2: The impact of flit generation rate on average flit latency in a 9-node mesh NoC

generation rate (flit/cycle)	average flit latency with DEVS-Suite (cycles)	average flit latency with Booksim (cycles)	difference	difference %
0.10	15.506	15.40	+0.106	0.68%
0.13	15.548	15.70	−0.152	0.97%
0.15	15.840	15.98	−0.14	0.88%
0.18	16.484	16.8	−0.316	1.88%
0.2	16.782	17.44	−0.342	1.96%

Table 3: The impact of network size on average flit latency

Network size	average flit latency with DEVS-Suite (cycles)	average flit latency with Booksim (cycles)	difference	difference %
3×3	15.506	15.40	+0.106	0.68%
4×4	19.542	19.36	+0.182	0.93%
5×5	23.714	23.82	−0.106	0.45%
6×6	28.408	28.86	−0.452	1.59%

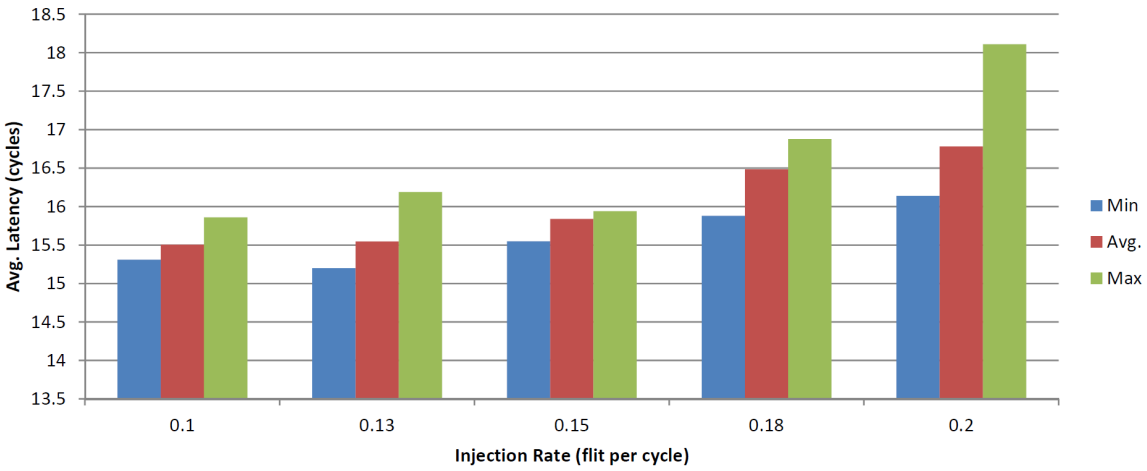


Figure 2: The difference between maximum, minimum, and average latency for the first experiment

Again, Figure 3 demonstrates the variation in maximum, minimum, and average latency when the size of the network is scaled in the second experiment. Similar to the previous variation chart, the likelihood of having large variations increases when the traffic becomes denser which is the consequence of scaling the network size.

From these experiments, we conclude that the DEVS-based simulation of NoC is reasonably close to the BookSim simulation and manifests the same behavior. The two simulators act very similar in times of regular traffic but the difference may grow in moments of congestion.

3 NoC-DEVS Visualizations

This section contains several visualization of NoC coupled models in DEVS-Suite. Since these models are large and their interconnections are so complex, it is even difficult to show a complete (all internal components visible) NoC of two nodes. However, in DEVS-Suite it is possible to visualize some coupled models as black boxes so that their internal elements are not drawn.

Figure 4 depicts a simulation of 4 nodes of NoC, in which all components are black boxes. For this 2→2 network, we have 4 PEs, switches, network interfaces (NI), and buses. This network has two virtual channels as apparent from the number of bus input/output ports.

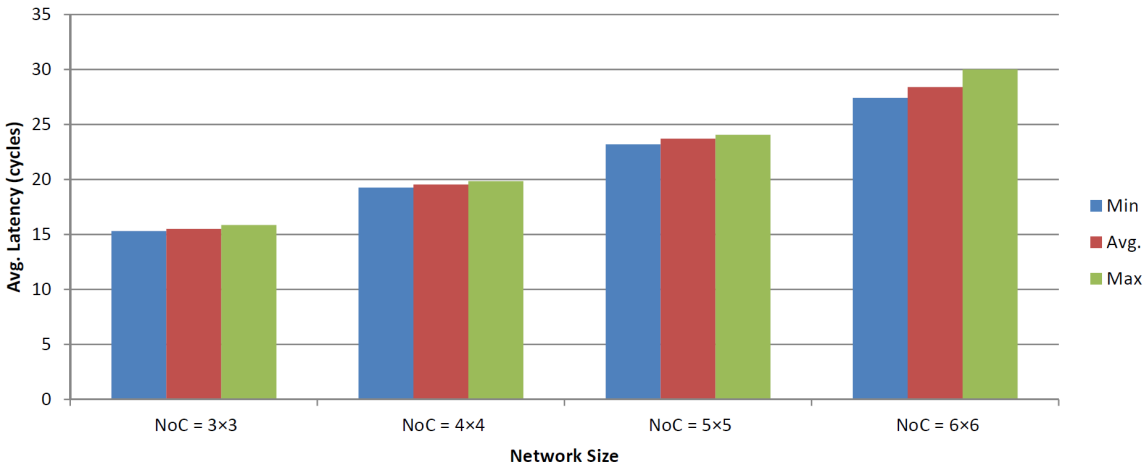


Figure 3: The difference between maximum, minimum, and average latency for experiment No. 2

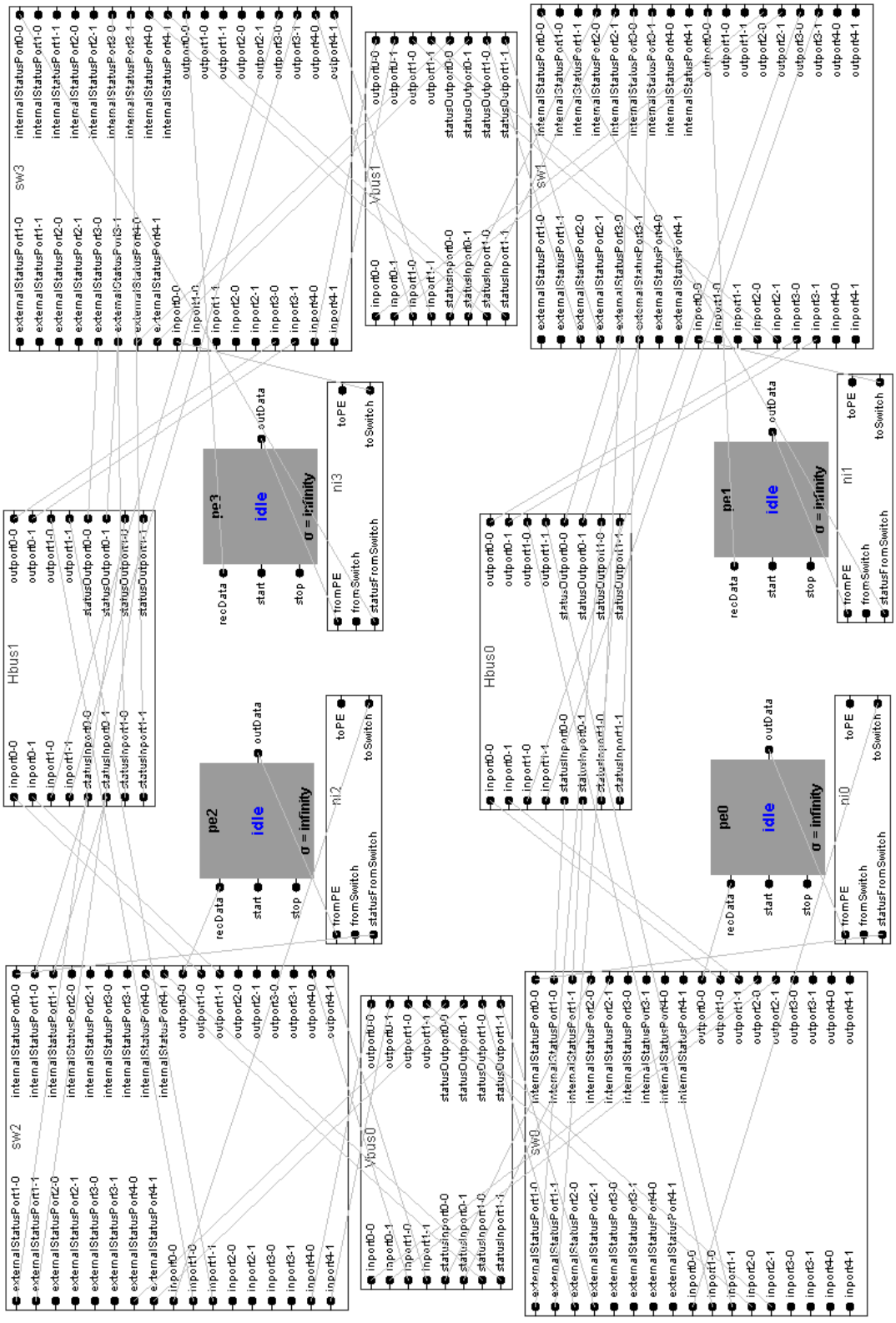


Figure 4: Hierarchical component view of a 2x2 mesh NoC

Figure 5, shows the internal architecture of a single switch. In order to keep the figure simple, we picked a switch with one virtual channel and four ports. The number of input/output ports doubles if virtual channels are two. In this figure, 8 input ports handle data arriving on 4 data links and 4 status links. Furthermore, a 4-4 crossbar handles the connection between input and output ports.

4 ALRT-DEVS models for NoC

This section contains the ALRT-DEVS specifications of several NoC components: output port, depacketizer, crossbar, and link.

4.1 Switch

Switch is the most important component in real-time modeling and simulation of NoC in which all the routing decisions are made. This includes deciding about the priority of packets over each other and choosing the best paths considering network congestion, link capacity, and packet deadline. Simplicity and light-weight communication patterns are the two important properties of routing algorithms that should be considered when choosing one. Aside from time-bounded messages, in-order packet delivery could also be important in some NoCs. A switch has several internal components for input ports, output ports, and a crossbar to connect these input ports to output ones. Flow control mechanism (here on/off method) is also implemented in this component which is further illustrated in subcomponents. From the internal components of switch input port is specified below.

4.1.1 Input Port

The input port component is responsible for receiving incoming flits from the neighboring switches or processing elements. Each input port contains status arrays and flit queues equal to the number of the virtual channels of that network. Each status array holds status information about the head flit of its corresponding queue. In addition, routing and allocating operations are done via actions in this component instead of embedding them in separate router/allocator components. Flow control mechanism is also implemented in this component. Input queues send *off* signals to the upstream switch whenever the input queue has passed the upper margin. The *on* signal is later sent when the number of waiting flits drop below the lower margin. Below, ALRT-DEVS model of this atomic component is specified.

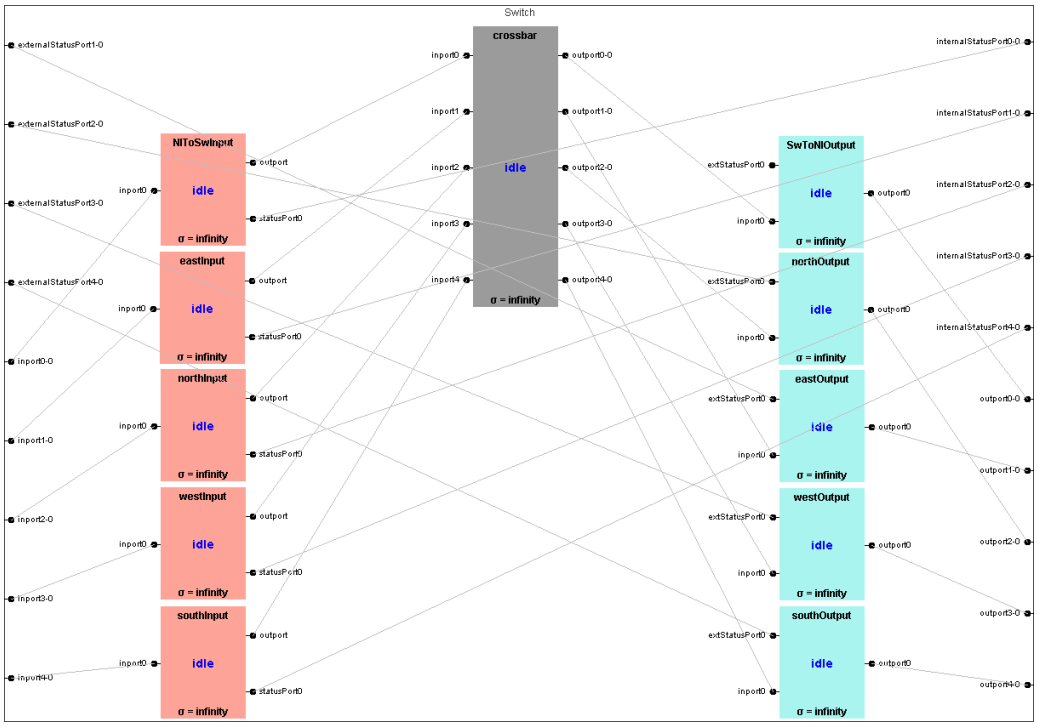


Figure 5: Flit-level design of switch component

Table 4: Variable description for input port

Variable	**Type**	**Description**
N_{VCs}	Integer	number of virtual channels in the network
N_{Ports}	Integer	number of ports for each switch
statusArray[N_{VCs}]	status array of the size of virtual channels. Each cell contains information such as: *output port, destination VC, stage*, and *size*	holds the status of each virtual channel in the input port. *Output Port* and destination VC hold the destination port and VC, respectively. Stage specifies the current stage of the head flit (*R, VA, SA,* and *ST*). The current length of the queue is held in *size* variable
statusInconsistent	boolean variable	*True* if the internal status of an input port is inconsistent with the value on *statusPorts*
inQs[N_{VCs}]	array of flits	array of input queues for each virtual channel
round	integer	specifies the turn for switch allocation among VCs

$$S = \overbrace{\{Active, Idle\}}^{\text{phase}} \times \overbrace{\sigma}^{\text{sigma}} \times \overbrace{\{0,1\}^*}^{\text{inQs}[N_{VCs}]} \times \overbrace{\{0,1\}^*}^{\text{outFlit}} \times \overbrace{\mathbb{N}}^{\text{round}} \times \overbrace{vcStatus}^{\text{statusArray}[N_{VCs}]} \times \overbrace{\{true, false\}}^{\text{statusInconsistent}} \quad (1)$$

$$X = \big\{\big(inport[0..N_{VCs}], \{0,1\}^*\big)\big\} \quad (2)$$

$$Y = \big\{\big(outport, \{0,1\}^*\big), \big(statusPort[0..N_{VCs}], \{Ok, Nok\}\big)\big\} \quad (3)$$

$$A = \big\{statusTransmitter, router, vcAllocation, swAllocation, swTraversal\big\} \quad (4)$$

The definition of state, in addition to phase and sigma, contains input queues, head flit (under service), turn (which virtual channel to service), status of input queues (for flow control purposes), and the *statusInconsistent* boolean variable. All these variables are described in Table 8. We specified two types of output ports for this component: one for sending out flits and one for status signals.

$$\delta_{ext}\big(phase, \sigma, inputQs, outFlit, round, statusArray, statusInconsistent, e, (inport[i], X)\big) = (\text{“}Active\text{”}, \delta t, inputQs[i].X, outFlit, round, statusArray) \quad (5)$$

$$\delta_{int}\big(\text{“}Active\text{”}, \sigma, inputQs, outFlit, round, statusArray, statusInconsistent\big) = \begin{cases} (\text{“}Active\text{”}, \delta t, ..., round = i, ...) & [inputQs\ is\ nonempty] \\ (\text{“}Idle\text{”}, \infty, ..., round + 1, ...) & [O.W.] \end{cases} \quad (6)$$

$$\lambda(\text{“}Active\text{”}, \sigma, inputQs, outFlit, round, statusArray, false) = \big(outport, outFlit\big) \quad (7)$$

$$\lambda(\text{“}Active\text{”}, \sigma, inputQs, outFlit, round, statusArray, true) = \big(statusPort[i], maxSize - statusArray[i].size\big) \quad [for\ 0 \le i < N_{VCs}] \quad (8)$$

$$\psi(\text{“}Active\text{”}, \sigma, inputQs, outFlit, round, statusArray, statusInconsistent) = \{statusTransmitter, router, vcAllocation, swAllocation, swTraversal\} \quad (9)$$

$$\psi(\text{“}Idle\text{”}, \sigma, inputQs, outFlit, round, statusArray, statusInconsistent) = \{\} \quad (10)$$

The only external event is the receipt of a new flit which adds this newly received flit to the queue. The only internal event results in sending one flit to the crossbar and passing

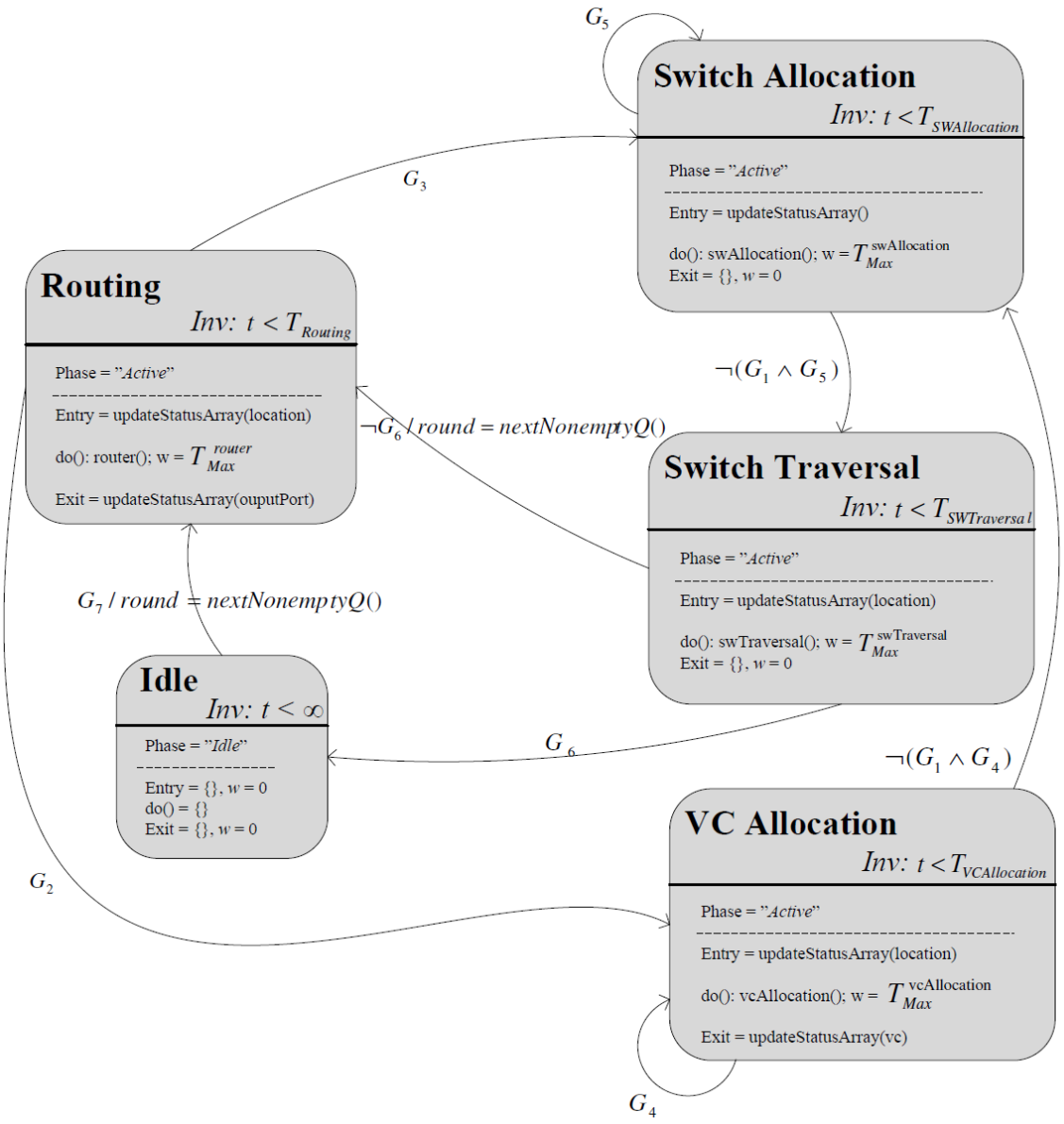
Table 5: Variable description for output port

Variable	Type	Description
downstreamStatus	boolean variable	holds the status of the downstream virtual channel (*false* for full)
flits[N_{VCs}]	array of flits	array holding output flits (one per virtual channel because of single output buffer architecture)
targetVC	integer	specifies the target virtual channel in the downstream node

the turn to the next input queue based on round robin scheme. In case of empty queues the component goes into *idle* phase. Also, in case of inconsistency between internal and external status (*statusInconsistent* changes to true) the new status signal is sent to neighboring switches. After specifying the component using DEVS, we use real-time Statecharts to complete the ALRT-DEVS model. The Statecharts diagram for this component is divided into two in Figures 6 and 7. The guard for each transition is specified under each figure. Guards with lower numbers have priority over guards with higher numbers. Therefore, if $guard_i$ and $guard_j$ are both satisfied and $i < j$, the first transition (guarded by i) is taken. With this in mind, it is obvious from Figure 7 that status transmission has the highest priority among all actions in this model of NoC. Thus, whenever *statusInconsistent* is true, the model changes its location to *Status Transmission* to synchronize internal and external status signals. Routing, allocations, and traversal actions are executed in sequence based on the guards set on transitions in Figure 6.

4.1.2 Output Port

Sending out ready flits is done via output ports. Similar to input ports, they include ready flit queues and status arrays. This component has only one action for sending out flits. We designed this model to be single buffered, therefore, single buffers replace ready queues. Single buffered output ports can only store one flit for every virtual channel. This is different from the model of input port in which each virtual channel possesses a queue for input flits. Output ports receive flow control signals from the downstream switch and block/allow sending of flits based on the on/off signal. This mechanism along with other features of this component are specified below.



$G_1 = [statusInconsistent\ is\ true]$

$G_2 = [statusArray[round].outputPort\ is\ set \wedge N_{VCs} > 1]$

$G_3 = [statusArray[round].outputPort\ is\ set \wedge N_{VCs} == 1]$

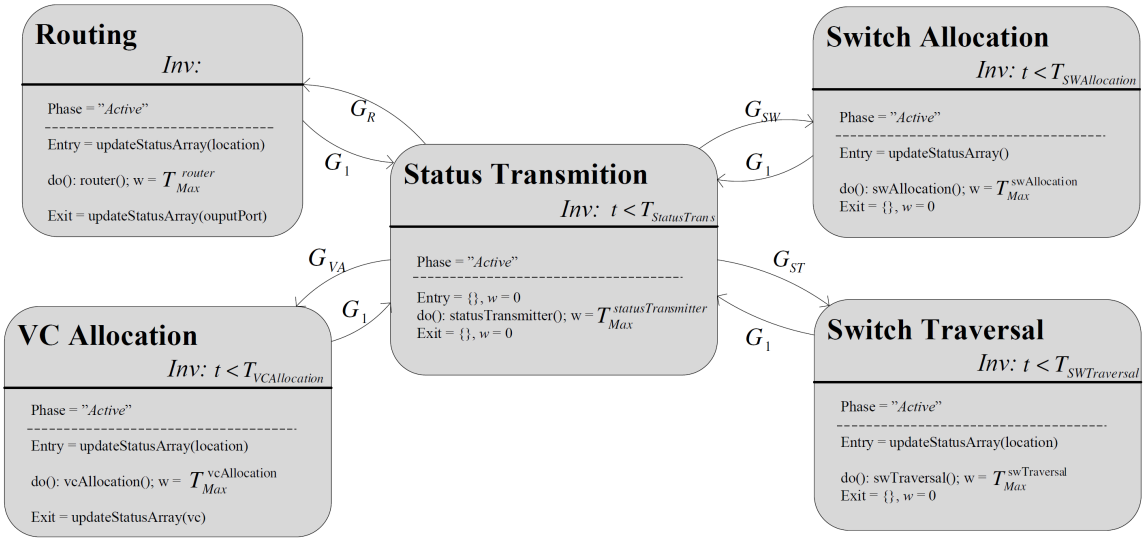
$G_4 = [statusArray[round].vc\ is\ not\ set]$

$G_5 = [swAllocation\ unsuccessful]$

$G_6 = [all\ inQs\ are\ empty]$

$G_7 = Event_{External}[\neg G_6]$

Figure 6: Statecharts diagram for input port - Part 1



$G_1 = [statusInconsistent\ is\ true]$
$G_R = [statusArray[round].stage = \text{“}R\text{”}]$
$G_{VA} = [statusArray[round].stage = \text{“}VA\text{”}]$
$G_{SW} = [statusArray[round].stage = \text{“}SW\text{”}]$
$G_{ST} = [statusArray[round].stage = \text{“}ST\text{”}]$

Figure 7: Statecharts diagram for input port - Part 2

$$S = \overbrace{\{Active, Idle\}}^{\text{phase}} \times \overbrace{\sigma}^{\text{sigma}} \times \overbrace{\{0,1\}^*}^{\text{flits}[N_{VCs}]} \times \overbrace{\{0,1\}^*}^{\text{outFlit}} \times \overbrace{\mathbb{N}}^{\text{round}} \times \overbrace{\mathbb{N}}^{\text{targetVC}} \times \overbrace{\{Ok, Nok\}^{N_{VCs}}}^{\text{downstreamStatus}} \tag{11}$$

$$X = \big\{\big(inport[0..N_{VCs}], \{0,1\}^*\big), \big(extStatusPort[0..N_{VCs}], \{Ok, Nok\}\big)\big\} \tag{12}$$

$$Y = \big\{\big(outport[0..N_{VCs}], \{0,1\}^*\big)\big\} \tag{13}$$

$$A = \big\{linkTraversal\big\} \tag{14}$$

The DEVS model of this component is close to Input Port. This component receives the status of downstream node through *extStatusPort*s and stores them in *downstreamStatus* state variable. The only action of this component is *linkTraversal* which transmits a flit on the link. Transmission is canceled if the status of the downstream node is *Nok*. The formulation brought below illustrates the behavior.

$$\begin{aligned}\delta_{ext}\big(&phase, \sigma, flits, outFlit, round, targetVC, downstreamStatus, e, (inport[i], X)\big)\\ &= (\text{“}Active\text{”}, \delta t, flits[i] = X, outFlit, round, targetVC, downstreamStatus)\end{aligned} \tag{15}$$

$$\begin{aligned}\delta_{ext}\big(&phase, \sigma, flits, outFlit, round, targetVC, downstreamStatus, e, (extStatusPort[i], X)\big)\\ &= (phase, \delta t, flits, outFlit, round, targetVC, downstreamStatus[i] = X)\end{aligned} \tag{16}$$

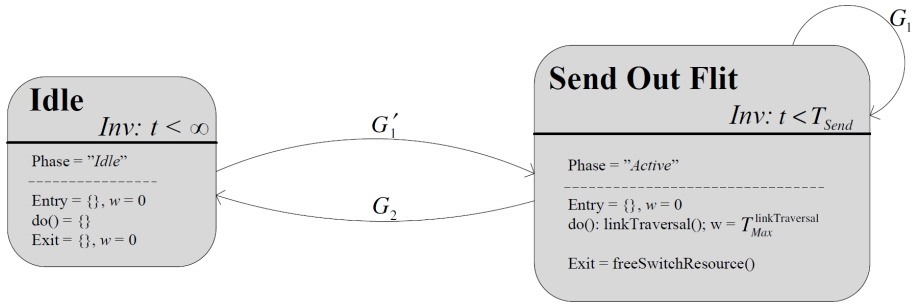
$$\delta_{int}\big(\text{“}Active\text{”}, \sigma, flits, outFlit, round, targetVC, downstreamStatus\big) = \begin{cases}(\text{“}Idle\text{”}, \infty, flits, outFlit, round, targetVC, downstreamStatus) & \big[flits\ is\ empty\big]\\ (\text{“}Active\text{”}, \delta t, flits, outFlit, round+1, targetVC, downstreamStatus) & \big[O.W.\big]\end{cases} \tag{17}$$

$$\begin{aligned}\lambda(\text{“}Active\text{”}, \sigma, flits, outFlit, round, targetVC, downstreamStatus)\\ = \big(outport[targetVC], outFlit\big)\end{aligned} \tag{18}$$

$$\psi(\text{“}Active\text{”}, \sigma, flits, outFlit, round, targetVC, downstreamStatus) = \{linkTraversal\} \tag{19}$$

$$\psi(\text{“}Idle\text{”}, \sigma, flits, outFlit, round, targetVC, downstreamStatus) = \{\} \tag{20}$$

Based on the formal model above, an internal event is raised in time of sending one flit out, the output function sends one flit on the line, and the receipt of a new packet from crossbar or a new downstream status signal invokes the external transition function. State variables are clarified in Table 5. Figure 8 depicts a fairly simple Statecharts diagram for this component along with the definition of guards.



$G_1 = [flits\ is\ nonempty]$
$G_1' = Event_{External}[\,]$
$G_2 = [flits\ is\ empty]$

Figure 8: Statechart diagram for output port

Table 6: Variable description for crossbar

Variable	Type	Description
config$[N_{Ports}][N_{Ports}]$	two-dimensional integer array	specifies the configuration of the crossbar by setting the used virtual channel number to port-to-port connections
outFlits$[N_{Ports}]$	array of flits	array of outgoing flits for each port
flits$[N_{Ports}]$	array of flits	list of incoming flits into the crossbar

4.1.3 Crossbar

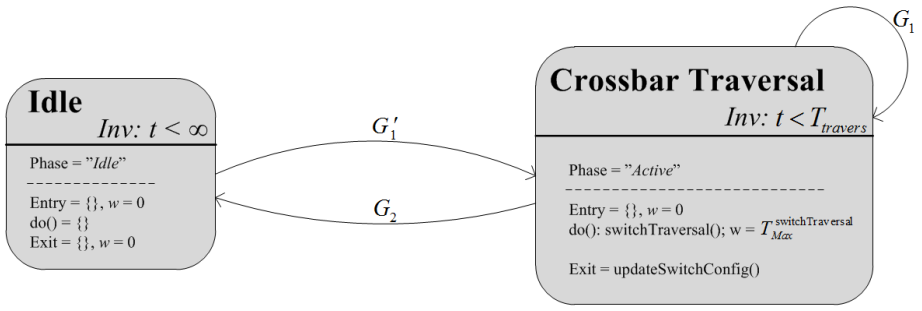
This component, as described earlier, connects input ports to output ports. After the routing phase in which the outgoing port is chosen, the input port allocates a virtual channel and a path in the crossbar to the outgoing port. The allocation process dynamically configures the crossbar to set a path from the requester input port to the destined output port. Based on the internal architecture of the crossbar switch, it can support multiple connections or only one. Below is the ALRT-DEVS model of this component.

$$S = \overbrace{\{Active, Idle\}}^{\text{phase}} \times \overbrace{\sigma}^{\text{sigma}} \times \overbrace{\{0,1\}^*}^{\text{flits}[N_{Ports}]} \times \overbrace{\{0,1\}^*}^{\text{outFlits}[N_{Ports}]} \times \overbrace{\{Int\}^*}^{\text{config}[N_{Ports}][N_{Ports}]} \quad (21)$$

$$X = \{(inport[0..N_{Ports}], \{0,1\}^*)\} \quad (22)$$

$$Y = \{(outport[0..N_{VCs} \times N_{Ports}], \{0,1\}^*)\} \quad (23)$$

$$A = \{switchTraversal\} \quad (24)$$



$G_1 = [flits\ is\ nonempty]$
$G_1' = Event_{External}[\,]$
$G_2 = [flits\ is\ empty]$

Figure 9: Statechart diagram for crossbar switch

$$\delta_{ext}\big(phase, \sigma, flits, outFlits, config, e, (inport[i], X)\big) = (\text{“}Active\text{”}, \delta t, flits[i] = X, outFlits, config) \quad (25)$$

$$\delta_{int}\big(phase, \sigma, flits, outFlits, config\big) = \begin{cases} (\text{“}Idle\text{”}, \infty, flits, outFlits, config) & \big[flits\ is\ empty\big] \\ (\text{“}Active\text{”}, \delta t, flits, outFlits, config) & \big[O.W.\big] \end{cases} \quad (26)$$

$$\lambda(phase, \sigma, flits, outFlits, config) = for\ all\ i\ and\ j, \big(outport\big[j \times N_{VCs} + config[i][j]\big], outFlits[i]\big) \quad (27)$$

$$\psi(\text{“}Active\text{”}, \sigma, flits, outFlits, config) = \{switchTraversal\} \quad (28)$$

$$\psi(\text{“}Idle\text{”}, \sigma, flits, outFlits, config) = \{\} \quad (29)$$

The DEVS model for crossbar component is straightforward. The only action executed for each flit is *switchTraversal* which transfers the flit from one side of the switch to the other. The configuration of the crossbar at every instance of time is kept in a two-dimensional array representing input ports as rows and output ports as columns. Each cell of this two-dimensional array keeps the destination virtual channel from which the flit is outputted. The Statecharts diagram for this component is a two-state model shown in Figure 9.

Table 7: Variable description for packetizer

Variable	Type	Description
outFlitQ	queue of flits	packetized flits, ready to be sent out
inData	stream of data	data received from PE to be packetized
outQStatus	boolean	holds the status of switch input queue

4.2 Network Interface

This component is the interface between PE and Switch. It is responsible for packetizing and depacketizing. In a real-time model of NoC, these two actions should be time-bounded as well. Variable length data is an obstacle for predictability and consequently for real-time modeling given arbitrary hardware and software realizations. We modeled Network interface as a coupled model in our specification of NoC. The inner components are: packetizer and depacketizer. The packetizer component is specified below.

4.2.1 Packetizer

This component packetizes the data streams it receives from the adjacent processing element. It also sets flit header and footer while each flit is created and sent to the neighboring switch. The packetizer component must be cautious not to overwhelm the switch, therefore, it receives flow control signals from the switch. When switch input queue is full, no new flit is packetized and sent forward. ALRT-DEVS specification of this component is as follows.

$$S = \overbrace{\{Active, Idle\}}^{\text{phase}} \times \overbrace{\sigma}^{\text{sigma}} \times \overbrace{\{0,1\}^*}^{\text{outFlitQ}} \times \overbrace{\{0,1\}^*}^{\text{inData}} \times \overbrace{\{0,1\}}^{\text{outQStatus}} \tag{30}$$

$$X = \big\{(fromPE, \{0,1\}^*), (statusFromSW, \{Ok, Nok\})\big\} \tag{31}$$

$$Y = \big\{(toSW, \{0,1\}^*)\big\} \tag{32}$$

$$A = \big\{packetize, sendFlitToSwitch\big\} \tag{33}$$

$$\delta_{ext}\big(\text{“}Idle\text{”}, \sigma, outFlitQ, inData, outQStatus, e, (fromPE, X)\big) = (\text{“}Active\text{”}, \delta t, outFlitQ, inData.X) \tag{34}$$

$$\delta_{ext}\big(\text{“}Active\text{”}, \sigma, outFlitQ, inData.X, outQStatus, e, (fromPE, X)\big) = (\text{“}Active\text{”}, \sigma - e, inData.X, outData) \tag{35}$$

Table 8: Variable description for depacketizer

Variable	Type	Description
incomingFlitQueue	queue of flits	flits received from the switch
outgoingDataBuffer	stream of data	depacketized data ready to be sent to the PE

$$\delta_{int}(\text{“}Active\text{”}, \sigma, outFlitQ, inData, outQStatus) = \begin{cases} (\text{“}Idle\text{”}, \infty, ...) & \big[(both\ outFlitQ\ and\ inData\ are\ empty\big] \\ (\text{“}Active\text{”}, \delta t, ...) & \big[O.W.\big] \end{cases} \tag{36}$$

$$\lambda(\text{“}Active\text{”}, \sigma, outFlitQ, inData, outQStatus) = \big(toSW, outFlitQ.remove()\big) \tag{37}$$

$$\psi(\text{“}Active\text{”}, \sigma, outFlitQ, inData, outQStatus) = \{packetize, sendFlitToSwitch\} \tag{38}$$

$$\psi(\text{“}Idle\text{”}, \sigma, outFlitQ, inData, outQStatus) = \{\} \tag{39}$$

As specified in Table 7, *outQStatus* holds the status of the switch. This variable is changed if an external event is received from *statusFromSW* port. This status variable, ensures that switch's input queue does not overflow. The Statecharts model of this component in Figure 10 demonstrates this behavior. The model waits infinitely in the *Waiting* location until it receives an *OK* signal from the switch.

4.2.2 Depacketizer

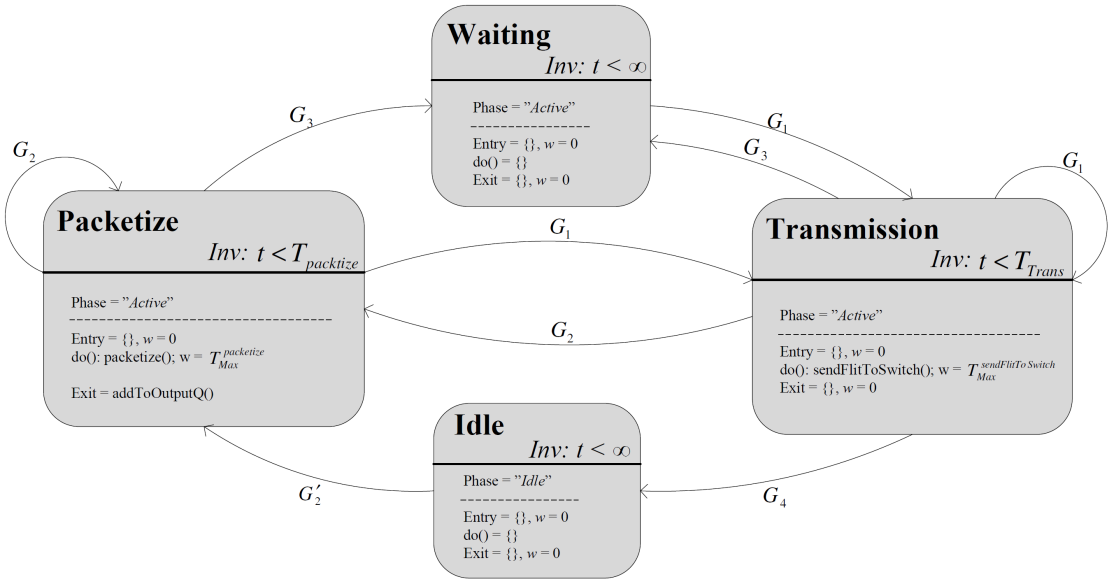
Flits that reach their destination must pass the depacketizer component to reach the processing element. This component is responsible for transforming flits into meaningful data streams for the PE. In this specification of NoC, we did not extend the flow control algorithm to this component. In other words, we assumed the processing element is always ready to receive data stream from the depacketizer. Therefore, at any instance of time, the switch can send flits to the depacketizer and this component transforms them into streams of data and sends them to the intended processing element. This is clarified in the specification of this component.

$$S = \overbrace{\{Active, Idle\}}^{\text{phase}} \times \overbrace{\sigma}^{\text{sigma}} \times \overbrace{\{0,1\}^*}^{\text{outgoingDataBuffer}} \times \overbrace{\{0,1\}^*}^{\text{incomingFlitQueue}} \tag{40}$$

$$X = \big\{\big(fromSW, \{0,1\}^*\big)\big\} \tag{41}$$

$$Y = \big\{\big(toPE, \{0,1\}^*\big)\big\} \tag{42}$$

$$A = \big\{Depacketize, SendFlitToPE\big\} \tag{43}$$



$G_1 = [outFlitQ\ is\ nonempty]$

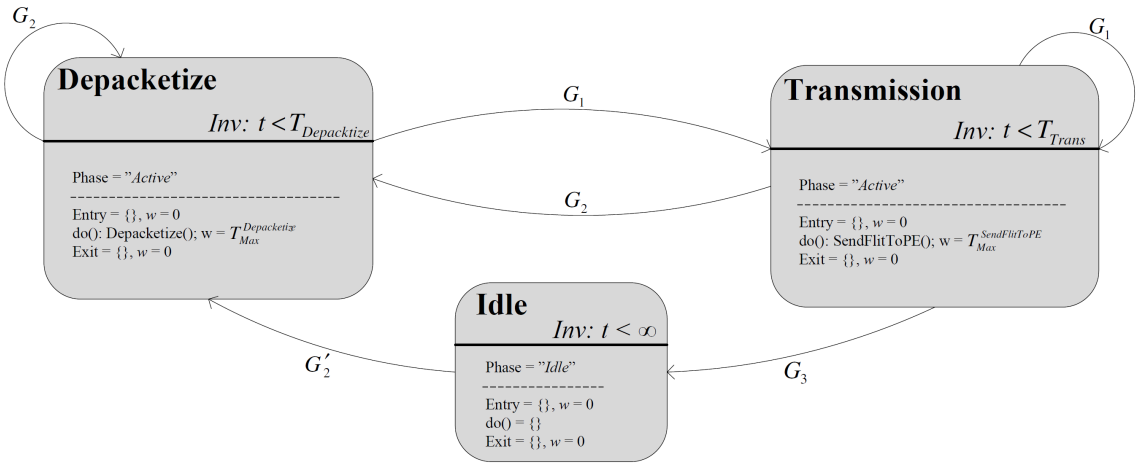
$G_2 = [inData\ is\ nonempty]$

$G_2' = Event_{External}[G_2]$

$G_3 = [outFlitQ\ is\ nonempty \wedge outQStatus = \text{“}Nok\text{”}]$

$G_4 = [inData\ is\ empty \wedge outFlitQ\ is\ empty]$

Figure 10: Statecharts diagram for packetizer



$G_1 = [outgoingDataBuffer\ is\ nonempty]$

$G_2 = [incomingFlitQueue\ is\ nonempty]$

$G_2' = Event_{External}[G_2]$

$G_3 = [incomingFlitQueue\ is\ empty \wedge outgoingDataBuffer\ is\ empty]$

Figure 11: Real-time Statechart for depacketizer

The behavior of this component is simple and only illustrated using Statecharts in Figure 11. Since this component does not implement flow control mechanism, no *Waiting* location is assumed in its ALRT-DEVS model.

4.3 Processing Element

Processing Element is responsible for processing data. The idea of NoC is to connect these separated cores to each other enabling them to complete a task cooperatively. Some high-level modeling schemes may use simple data generator representing PEs. Our focus is also on the network, not on the modeling of PEs; however, we modeled processing elements to have various frequencies and clock signals to gain heterogeneity in NoC M&S. The explanation for the model of this component is given below.

$$S = \overbrace{\{Active,\ Idle\}}^{\text{phase}} \times \overbrace{\sigma}^{\text{sigma}} \times \overbrace{\{0,1\}^*}^{\text{incomingDataBuffer}} \times \overbrace{\{0,1\}^*}^{\text{outgoingDataBuffer}} \tag{44}$$

$$X = \big\{(inport, \{0,1\}^*), (start, 1), (stop, 1)\big\} \tag{45}$$

$$Y = \big\{(outport, \{0,1\}^*)\big\} \tag{46}$$

$$A = \big\{ProcessData,\ GenerateData\big\} \tag{47}$$

Figure 12 presents the Statecharts diagram for processing element. Processing elements generate and process data through *GenerateData* and *ProcessData* actions, respectively. This component goes to *Idle* state if no data is waiting to be sent out or processed.

4.4 Link

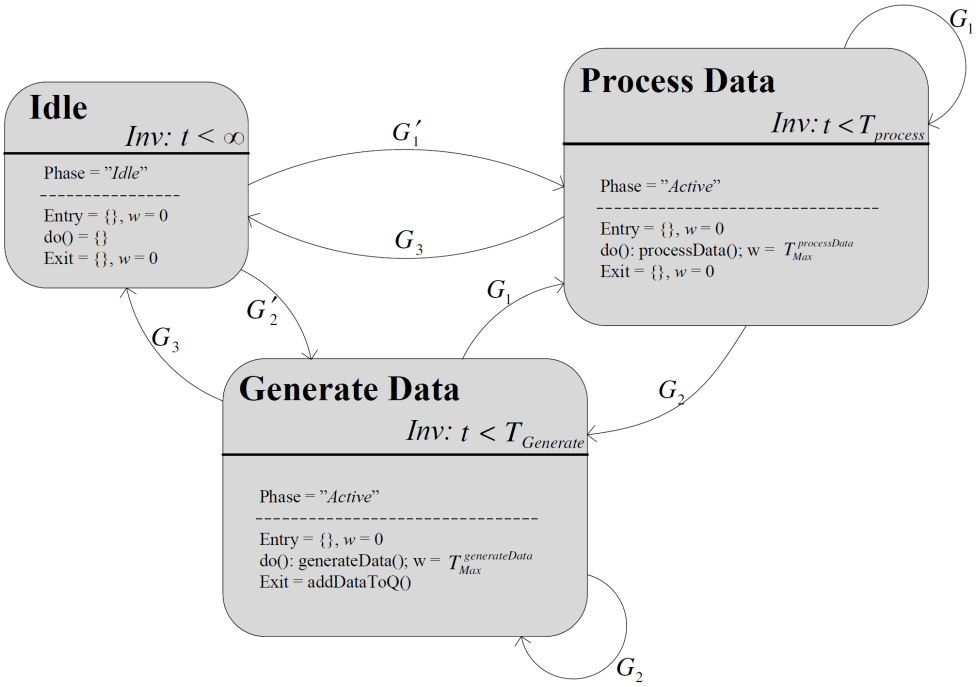
Link acts as a connector between components of the system. NoC may incorporate synchronous/asynchronous and parallel/serial models of link. Either of these links require different ALRT-DEVS specifications. A link is modeled as simple as possible in the specification below. We gather several links in one Bus coupled model to handle the communication between switches.

$$S = \overbrace{\{Active, Idle\}}^{\text{phase}} \times \overbrace{\sigma}^{\text{sigma}} \times \overbrace{\{0,1\}^*}^{\text{travellingFlit}} \qquad \big(where\ \ \sigma \in (0, \infty]\big) \tag{48}$$

$$X = \{(linkIn, \{0,1\}^*)\} \tag{49}$$

$$Y = \{(linkOut, \{0,1\}^*)\} \tag{50}$$

$$A = \{deliverFlit\} \tag{51}$$



$G_1 = [incomingDataBuffer\ is\ nonempty]$

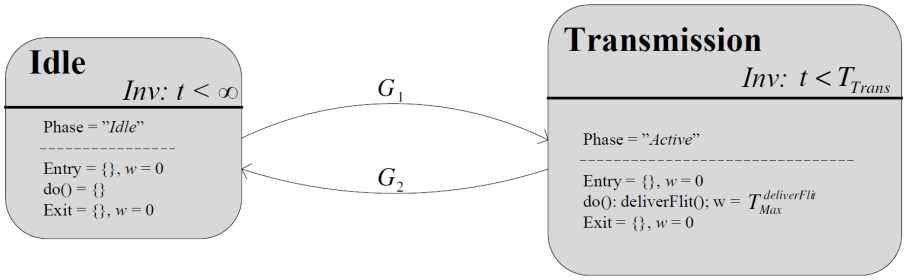
$G'_1 = Event_{External}[G_1]$

$G_2 = [outgoingDataBuffer\ is\ nonempty]$

$G'_2 = Event_{External}[G_2]$

$G_3 = [incomingDataBuffer\ is\ empty \wedge outgoingDataBuffer\ is\ empty]$

Figure 12: Real-time Statecharts for PE



$G_1 = Event_{External}[\,]$
$G_2 = [travellingFlit\ is\ NULL]$

Figure 13: Statechart diagram for link

$$\delta_{ext}\big(\text{“}Idle\text{”}, \sigma, travellingFlit, e, (linkIn, X)\big) = (\text{“}Active\text{”}, \delta t, X) \tag{52}$$
$$\delta_{int}(\text{“}Active\text{”}, \sigma, travellingFlit) = (\text{“}Idle\text{”}, \infty, NULL) \tag{53}$$
$$\lambda(\text{“}Active\text{”}, \sigma, travellingFlit) = \big(linkOut, travellingFlit\big) \tag{54}$$
$$\psi(\text{“}Active\text{”}, \sigma, travellingFlit) = \{deliverFlit\} \tag{55}$$
$$\psi(\text{“}Idle\text{”}, \sigma, travellingFlit) = \{\} \tag{56}$$

Figure 13 completes the ALRT-DEVS model with a Statecharts diagram. A link is capable of transferring only one flit (or signal) at a time. Therefore, a single virtual channel is modeled by a link. A set of links form a physical link which is a coupled model called *Bus*. These coupled models connect neighbor switches to each other.

5 Simulator Design and Execution Protocol

The real-time simulation protocol must adopt the time-constrained action-based features of ALRT-DEVS. First, we should introduce ALRT-DEVS syntax to atomic models implemented in DEVS-Suite [1]. Therefore, we extended atomic models so that they support actions, locations, transitions, and the activity mapping function. Also, atomic simulator classes were extended to support action execution, deadline handling, and physical clock synchronization. In order to simulate all models concurrently (as opposed to P-DEVS which does it sequentially), each atomic simulator is Java thread.

An insight to the structure of the *model* layer of the simulator (responsible for the entire M&S process) is presented here. In addition, we put code snippets from the real-time atomic simulator and the Input Port atomic model to illustrate DEVS-Suite's simulation protocol.

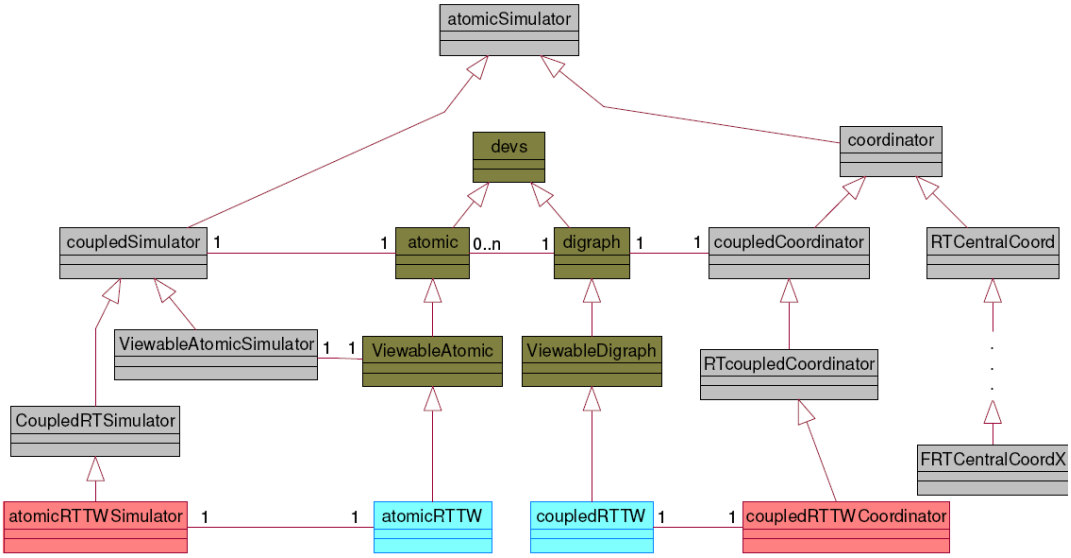


Figure 14: Class diagram for real-time simulation in DEVS-Suite

The UML diagram shown in Figure 14 contains the core classes in the DEVS-Suite simulator. The one-to-one relationship between model and simulator classes shows atomic and coupled models are executed by their respective simulators. The `FRTCentralCoordX` which acts as the central simulator class for logical-time simulations is also responsible for starting and ending real-time simulations. In addition, this class still executes all non-real-time components if they exist in the model. Atomic simulators implement `Runnable` to execute in parallel. The real-time atomic and coupled models are inherited from `ViewableAtomic` and `ViewableDigraph` classes in case their executions are to be visualized.

The execution of models proceeds according to the simulation protocol specifically designed for real-time execution. This protocol is exemplified with a sequence diagram which shows the simulation of a system of one coupled and two atomic models in Figure 15. This sequence diagram includes two atomic models, one for a generator (`gen`) and another for a consumer (`cons`). As noted in the previous paragraph, central simulator class (`FRTCentralCoordX`) launches the simulation by starting atomic models' threads. From that point forward, the central simulator is not needed anymore and terminates. In this example, the generator periodically creates output and the consumer waits to receive the generated message as external event. In the simulation loop first outputs are generated by the `computeInputOutput` method and then the messages are sent via `sendMessages` method. Then, `putMessages` method adds the messages to the input queue of the consumer model. Next, `WrapDeltFunc` method is invoked which handles the events (if any) via `deltint` or `deltext` method.

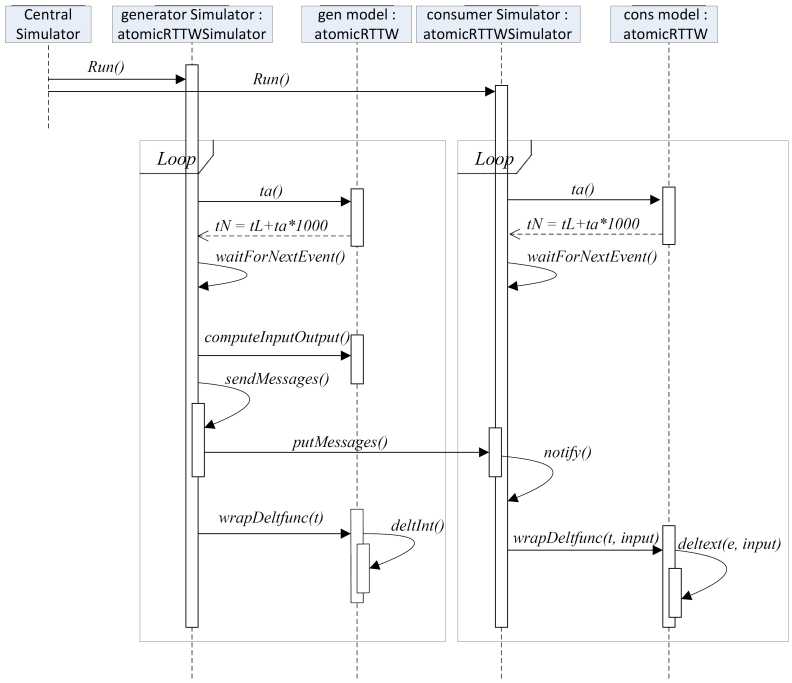


Figure 15: Sequence diagram presenting DEVS-Suite real-time execution

5.1 Model Execution

The generator and consumer models are implemented in the DEVS-Suite simulator using the newly introduced real-time model and simulator classes. The execution of these models is defined in the simulation protocol which has three steps: 1) Model is realized by instantiating corresponding model and simulator classes, 2) Central simulator class starts the simulation, and 3) In every cycle, it first asks all models for the time of their next events (`tN`) and raises events for those whose tw_l is the smallest. After one or more events are raised, all simulators are called upon to exchange messages and handle their internal/external events. Finally, the simulation advances to the next cycle.

This was a short description of how modeling and simulation is handled in DEVS-Suite. Model and simulator classes along with the simulation protocol should all be extended to support ALRT-DEVS model specification and their real-time execution. Figure 16 is the main loop of the real-time atomic simulator and the sequence of operations for carrying out one cycle of the simulation. In lines 3-9 the time to sleep is determined and the model is put to a *waiting* state. Deadline violation is checked at line 13 and is handled later in `wrapDeltfunc`. At lines 19-23, an internal event is handled and lines 25-26 handle external events. The details of internal and external event handling were covered in the sequence diagram presented in Figure 15. At the end of the cycle, the next event is recognized and time variables (`tN` and `tL`) are given their new values. Figures 17 and 18 show the realization of two actions for the Input Port atomic model as a part of the switch coupled model. In Figure 17, the input port sends an on/off signal to the upstream switch to start/stop sending more flits. The code snippet in Figure 18 shows the implementation of the virtual channel allocation. In both of these actions, the constructors have the lower-bound, upper-bound, and the name of the action respectively.

5.2 Multi-threading

In order to perform the simulation in real-time, parallel execution was introduced to the current simulation protocol of the DEVS-Suite simulator. In logical-time simulation, the central simulator (`FRTCentralCoordX`) is responsible for the entire simulation. The simulation loop takes all events, chooses the nearest and after advancing to that time instance, requests all components inside the model to submit outputs and handle possible external/internal events. In this scheme we have only one thread, models are executed sequentially, and time is synchronized throughout the model. In real-time execution with central execution, similar procedure is followed with a difference that instead of jumping to the next event, the central class uses a timer (measured in real-time) to sleep until the nearest event. In the real-time protocol, atomic simulator classes implement `Runnable` interface to be executed in parallel. No central simulator exists and the control is delegated to the atomic simulators. Each of these atomic simulators has its own simulation loop, which interacts with the model. This is in contrast to invoking all models when a single event

```
1.  while(true){
2.      deadlineViolation = false;
3.      while((currentTime=timeInMillis()) < getTN()-1){
4.          timeToSleep = (long)(getTN() - timeInMillis() - 1);
5.          if (timeToSleep < DevsInterface.INFINITY){
6.              timer = new simTimer(this, timeToSleep);
7.              Elapsed = false;
8.          }
9.          waitForNextEvent();
10.         if (inputReady) break;
11.     }
12.     if(timeInMillis() > (long)getTN()) deadlineViolation = true;
13.     if(timeInMillis() >= (long)getTN() - 1) Elapsed = true;
14.
15.     if(Elapsed){
16.         computeInputOutput(getTN());
17.         sendMessages();
18.         this.wrapDeltfunc(getTN());
19.     }
20.     else if(inputReady)
21.         this.wrapDeltfuncExternal(getTN());
22.     if(timer != null) timer.interrupt();
23.     Elapsed = false;
24.     tL = tN;
25.     tN = tL + myModel.ta()*1000;
26.     iter++;
27. }
```

Figure 16: Atomic Simulator main loop

```
1.  class statusTransmitter extends FixedTimeActionClass{
2.      public statusTransmitter(){
3.          super(.01, .01, "status transmitter");
4.      }
5.
6.      public void action(){
7.          for(int i = 0 ; i < numOfVCs ; i++){
8.              if((vcStatus[i] == true) &&
9.                      (vcs[i].size() >= Config.offThreashold)){
10.                 inconsistentStatusIndex.add(i);
11.                 vcStatus[i] = false;
12.             }
13.             else if((vcStatus[i] == false) &&
14.                     (vcs[i].size() <= Config.onThreashold)){
15.                 inconsistentStatusIndex.add(i);
16.                 vcStatus[i] = true;
17. }}}}
```

Figure 17: Status transmitter action for the *input port* atomic model

```
1.  class vcAllocation extends FixedTimeActionClass{
2.      public vcAllocation() {
3.          super(.01, .01, "VC Allocation Action");
4.      }
5.
6.      public void action() {
7.          int vc = parentSwitch.vcAllocator(vcs[round].getOutputPort());
8.          if(vc != -1){
9.              vcs[round].setTargetVC(vc);
10.             vcs[round].poll().setOutputVC(vc);
11.             vcs[round].setStatus("SA");
12. }}}
```

Figure 18: Virtual channel allocation action for the *input port* atomic model

occurs. In real-time simulation, a model is executed only when an internal or external event occurs. Using this method of simulation, models are not always in synchrony, which makes it suitable for asynchronous execution. The coupled coordinator classes, however, remain mostly unchanged and have the responsibility of sending/receiving messages from inner/outer components of the model. Coupled models are executed as one thread (sequentially) and their functions are synchronized to avoid race conditions. In general, coupled simulators act as utility classes with methods (for sending messages out or sending them to the inner models) accessed frequently by atomic/coupled simulators for communication purposes.

We suggest the reader to see our publication on ALRT-DEVS [3] and the RT-DEVS-Suite code [2] for more details.

References

[1] ACIMS. DEVS-Suite simulator, version 2.1.0. http://devs-suitesim.sourceforge.net/, 2009.

[2] S. Gholami and H.S. Sarjoughian. RT-DEVS NoC modeling with simulation support in DEVS-Suite. Technical Report TR-ASUCIDSE-CSE-2012-001, Arizona State University, http://devs-suitesim.sourceforge.net/, 2012.

[3] S. Gholami and H.S. Sarjoughian. Real-time network-on-chip simulation modeling. In *SIMUTools, Desenzano, Italy*, pages 103–112. ICST, 2012.