# Real-time Network-on-Chip Simulation Modeling

Soroosh Gholami[a]

a. School of Electrical and Computer Engineering
University of Tehran
Tehran, Iran
Soroosh.Gholami@ut.ac.ir

Hessam S. Sarjoughian[a,b]

b. School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, Arizona, U.S.A.
Hessam.Sarjoughian@asu.edu

## ABSTRACT

We present a Network on Chip (NoC) model with basic support for execution in constrained real-time. Actions for the processing element, switch, network interface, and channel components of NoC are specified in RT-DEVS, an extension of the DEVS formalism for real-time modeling. A desirable simulator must execute the actions defined in each NoC component within finite time periods. Execution of components' actions is supported by introducing a new capability to the DEVS-Suite simulator such that actions can be executed in real-time. The extended simulator can be used to develop, simulate, and evaluate the class of NoC designs that the underlying computing platform can support. NoC simulation can be used to obtain measurements such as system throughput and latency metrics under different communication patterns. This work offers a basis for future research where a NoC simulation can be embedded in a physical environment and thus enable NoC application designs and experimentations.

## Categories and Subject Descriptors

I.6.5 [**Simulation and Modeling**]: Model Development; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Interconnection architectures*

## Keywords

DEVS-Suite Simulator, Network on Chip, Real-Time DEVS Modeling

## 1. INTRODUCTION

Systems on Chips (SoC) are designed by putting several interconnected cores on a 2-D or 3-D chip. These cores may communicate with one another using shared buses available in the chip. The alternative to this approach is Network on Chip (NoC) in which communication between two cores is treated as series of packets sent and received using an underlying network. NoC is our target system for real-time modeling and simulation in this paper.

Modeling and simulation approaches are commonly used for prediction of the behavior or performance of dynamical systems. Real-time systems are usually too complex for analytical solutions; therefore, simulation methods are used as a substitution. However, simulation of a system in logical-time with its surrounding environment is difficult. This is when real-time simulation is useful. For example, systems communicating with a human operator, such as a flight simulator, requires real-time simulation. In such a situation, the simulation of the model must be carried out as closely as possible to physical time.

Several NoC simulation platforms have been designed and implemented in the recent years. Simulators with low-level abstractions (Register Transfer Level) [3][15][4] provide an accurate representation of the system for a wide range of performance metrics from energy consumption to packet latency. However, because of accounting for low-level interactions among components, simulation execution time is often prolonged. FPGA-based simulators [21] try to alleviate this problem by hardware execution. High-level abstractions [9] provide better execution time but less accurate results.

In this paper, NoC platform is first modeled using RT-DEVS formalism [13]. In this phase, RT-DEVS capabilities for modeling a complex system of this sort are described. The complexity of this phase is also dependent to the level of detail included in the model which is discussed in sections to follow. The model generated in this phase is then executed in real-time to provide a real-time simulation of NoC. The simulation is carried out with the DEVS-Suite tool which is extended to support real-time execution of RT-DEVS models. There are many obstacles with detailed real-time simulation of a hardware systems such as NoCs that are further discussed throughout this paper.

This article is organized as follows. Previous work on both real-time modeling/simulation and NoC modeling is discussed in Section 2. In section 3, we overview the RT-DEVS formalism and our approach for modeling NoC system. Section 4 is an introduction to DEVS-Suite tool and modifications made for the purpose of this research. Experiments and results are provided in Section 5 and Section 6 concludes this work.

## 2. PREVIOUS WORK

DEVS [22] is a modeling and simulation framework which is capable of expressing hierarchical, discrete event models and logical time simulation. In the field of modeling and simulation of hardware components with DEVS, single-cycle, multi-cycle, and pipeline MIPS32 processors were modeled in [5]. Furthermore, an approach for modeling hardware components using DEVS and HLA simulation was proposed in [18]. In addition, NoC modeling and simulation using DEVS for both regular and irregular topologies have been developed in [1]. Several extensions of DEVS such as RT-DEVS [13] and I-DEVS [16] have been developed.

Modeling of embedded systems with timing constraints is the main purpose of using RT-DEVS for M&S purposes. In [10], RT-DEVS formalism was incorporated for designing embedded control systems for temporal analysis with UPPAAL tool. This formalism is also used for modeling a safety critical system (Railroad crossing control) in [20]. In addition to RT-DEVS, a DEVS compatible extension to support hardware-in-loop and real time simulation of a model with imprecise timing was suggested in [17]. As for simulation environment of RT-DEVS, in [6] a simulation framework was proposed in which the model described in RT-DEVS formalism is simulated while communicating with its environment. This framework requires control over low level aspects of the system such as Interrupt Service Routines (ISR) which we want to avoid in order to provide a general purpose framework for real-time NoC system level simulation.

## 3. RT-DEVS MODELING APPROACH

In this section, we first introduce the RT-DEVS formalism and then discuss our approach for modeling the NoC platform by going over each component individually.

### 3.1 Real-Time DEVS Formalism

RT-DEVS is an extension of the DEVS formalism which supports real-time system modeling [20]. Real-time DEVS models may be simulated in physical time. Similar to DEVS, it consists of two classes of models: atomic models and coupled models. An atomic model in RT-DEVS formalism is defined as follows:

$$\text{RTAtomic} = \langle X, S, Y, \delta_{ext}, \delta_{int}, \lambda, ti, \psi, A\rangle$$

$X, S, Y$: are input events, sequential states, and output events respectively.
$\delta_{ext} : Q \times X \to S$, external transition function
(where Q is the total set of: $Q = \{(s, e) | s \in S, 0 \leq e \leq ti(s)|_{max}\}$).
$\delta_{int} : S \to S$, internal transition function.
$\lambda : S \to Y$, output function.
$ti : A \to \mathbb{R}^{+}_{0,\infty} \times \mathbb{R}^{+}_{0,\infty}$, time interval function.
(in which the first and second real numbers specify the lower bound and the upper bound of the execution of an action)
$\psi : S \to A$, activity mapping function.
$A$: set of actions.

Compared to (Parallel) DEVS formalism [7], in addition to a set of actions, time advance function is replaced with a "time interval" function, which provides a window based representation of time for the execution of actions specified for each state. Also, activity mapping function is added which assigns one or more actions to each state of the model [22]. These changes support specifying real-time system models which may be simulated in physical (wall-clock) time. Since time is not captured in the coupled model specification, the coupled model of RT-DEVS is similar to that of DEVS formalism.

### 3.2 Primary and Secondary State Variables

Next states with hold-in times for Parallel DEVS models are mainly defined using the *phase* and *sigma* state variables. These so-called *primary* state variables form the basis of how a model responds to external and internal events. Other state variables are generally needed as the dynamics of a model grows in complexity and scale (refer to a switch model developed for NoC-DEVS [1]). Thus, the state of an atomic model is described by a tuple ($S = phase \times \sigma \times v_1 \times \cdots \times v_n$) that has one or more state variables in addition to the phase and sigma state variables. The secondary state variables can play a significant role in specifying RT-DEVS models. Specifying state to action mappings — $\psi : S \to A$ (see RT-DEVS formal specification) can be undertaken systematically. They can help in formulating state to action mappings such that multiple actions can be defined according to specific ordering of actions while satisfying real-time simulation execution constraints. The secondary state concept also aids not only in state to action mappings within each of the internal and external transition functions but also external to internal transitions or vice versa.

### 3.3 State-to-Action Mappings

A finite number of actions can be specified in terms of (*phase*, $\Delta t$). The time window for all the actions is specified as $\Delta t = t_{max} - t_{min}$ given $t_{action_i} : A \to t_{min} \times t_{max}, 0 < t_{max} - t_{min} < \infty, t_{min} > 0$. For every given *phase*, $action_1, \cdots, action_p$ can be assigned to it. Each $action_i$ has a time window $\delta t_j \in \{\delta t_1, \cdots, \delta t_k\}, k \leq p, \Sigma_{j=1}^{j=k} \delta t_j \leq \Delta t$. These actions are sequentially executed to completion in some order subject to the availability of physical time for $\Delta t$. The total physical time for execution of all the actions is expected to be within $\Delta t$. This requires having time specifications for every $action_i$. However, one or more actions may not be executed if the available physical time is less than $\Delta t$. Each pair of (*phase*, $\Delta t$) specify a distinct location with a set of actions mapped to it. Transitions from one location to another are triggered by the secondary state variables. In general one or more secondary state variables act as guard to allow one or more actions to be invoked. An invoked action is completed iff its relative execution time in physical time is within its designated $\delta t_{min}$ and $\delta t_{max}$. Otherwise, the invoked action must be terminated. In other words, *partial time* constraints $\delta t_j, 0 < j \leq k$ for all actions $action_i, 0 < i, \leq p$ must be defined given the *total time* constraint $\Delta t$ defined for every phase.

### 3.4 NoC Modeling with RT-DEVS

All components of NoC should be modeled using the RT-DEVS formalism described in section 3.1. Then these atomic models are coupled together to create a novel abstraction of the NoC system. Our objective is to develop a generic model which is configurable to various scheduling/routing algorithms and topologies. Using this approach our model would

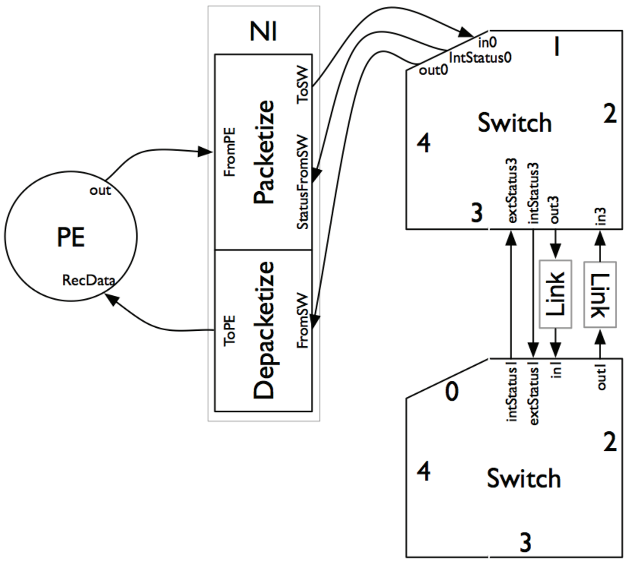


**Figure 1: NoC model components**

not be application dependent and with the right configuration – via model parametrization and/or extending generic models to include application-specific structures and behaviors – can be applicable to various kinds of workloads. In other words, generic models may be systematically extended to be domain-specific models. As illustrated in Figure 1, NoC consists of four main parts: Processing Element, Network Interface, Link, and Switch [1]. In addition, topology, buffer, flit, routing algorithm, and arbiter are other parts of NoC, which should be modeled as a part of the modeling phase.

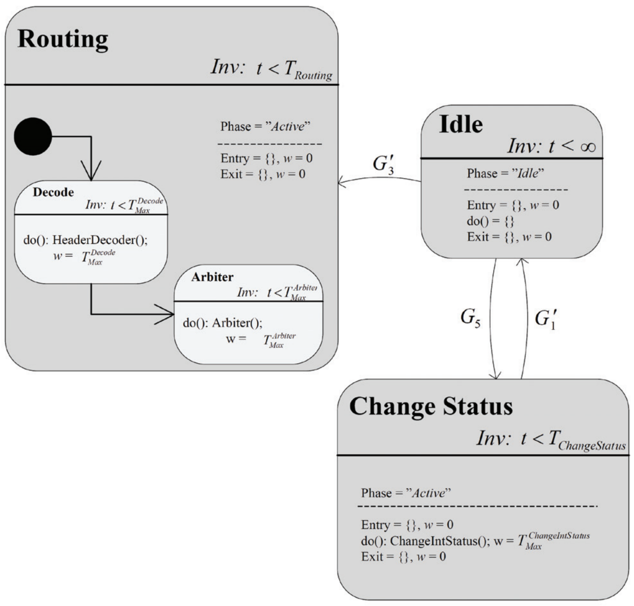
Processing Element is responsible for processing data. The idea of SoC is to connect these separated cores to each other enabling them to complete a task cooperatively. In order to concentrate on the networking aspects of NoC platforms, we consider details such as connectivity, protocol, buffering, routing, and delays for network model components. Therefore, excluding all SoC related concerns such as PE modeling, task mapping, memory-chip communication, and in general the flow of a job in the system from the mapping phase until completion. As a result, our model of PE is a simple data generator with extra specifications for operating frequency and performance.

Network Interface is the interface between PE and Switch. It is responsible for packetizing and depacketizing. This part of the system could be implemented in both hardware and software.

Switch is a key component in NoC in which all the routing decisions are made. The routing algorithm is responsible for choosing the best path for the flits traveling around the chip. 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 algorithm that should be considered.

$States = \{Active, Idle\}$
$Actions = \{HeaderDecoder, Arbiter, ChangeIntStatus\}$

$T_{Routing} = T_{Max}^{Decode} + T_{Max}^{Arbiter}$
$T_{ChangeStatus} = T_{Max}^{ChangeIntStatus}$



$G_1 = [statusInconsistent = true]$
$G'_1 = [Event_{External} \wedge G_1]$
$G_3 = [\exists I \in inBufs : I \text{ is nonempty}]$
$G'_3 = [Event_{External} \wedge G_3]$
$G_5 = [\forall I, O \,.\, I \in inBufs, O \in outBufs : I, O \text{ are empty}]$

**Figure 2: Switch model**

Link acts as a connector between components of the system. NoC may incorporate one of synchronous or asynchronous models of link. Also, both Serial and Parallel links may be used in NoCs, which are completely different in modeling. In our model of NoC, asynchronous parallel links are used. The topology of NoC is described using the input/output ports available in the DEVS formalism. Also, Buffer could be a simple first-in-first-out queue, which has a basic model in DEVS formalism. Clock component is responsible for synchronizing all or several (dependent to architecture) PEs in the chip. It is also possible to exclude this component from the chip which is the method used in this work. In this case, each PE has an internal clock, which transforms the chip to a heterogenous one because of the various operating frequencies and imperfect synchronization among clocks. In such systems, the use of buffers between PEs are inevitable.

Given the RT-DEVS specification of NoC elements, we model the details of those elements (definition of actions, priorities, etc.) using Real-time Statechart [12]. The reason of using this notation is that some of these details are not captured in their RT-DEVS specification. To illustrate our complementary modeling approach, a part of the switch model is shown in Figure 2 and described below. This model is further illustrated in Appendix A and the complete model can be found in [11].

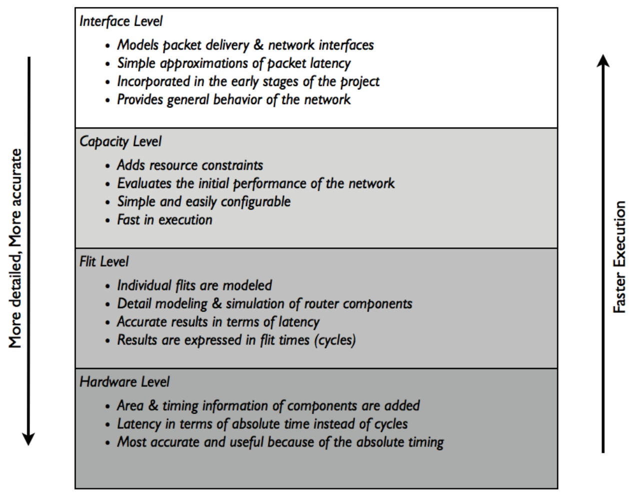
Banerjee et. al. [2] provided a detailed internal structure of NoC components in which switch contained five major elements: Virtual Channel, Header Decoder, Crossbar, Arbiter, and Link Controllers. We modeled these operations for these elements in 4 actions. First, the incoming flits from input ports are routed by the *HeaderDecoder*. Then, the *Arbiter* action allocates the appropriate output virtual channel for the flit. Finally, the packet is sent out by the *SendOutFlit* action. For flow control purposes a *ChangeIntStatus* action notifies neighboring switches of the status of its buffers. In this simple switch model, there are five locations in which the first two actions (HeaderDecoder and Arbiter) are mapped to *Routing* location, the *SendOutFlit* action is mapped to *Transmission* location, the *ChangeIntStatus* action is mapped to the *Change Status* location, and two other locations represent waiting and idle phases. The *Idle* location is mapped to *Idle* phase and the rest of the locations are mapped to the *Active* phase. The locations *Routing*, *Change Status*, and *Idle* are shown in Figure 2.

In Figure 2, in *Idle* location, $\Delta t$ is set to *INFINITY*. This means that the model stays in this phase without executing an action until an external event reactivates the model. Furthermore, *InBuf* and *OutBuf* represent input and output buffers of the switch. Since we use the single-buffer output NoC model, each port possesses an input buffer queue and a single output buffer. These input and output buffers are considered as secondary state variables which are used in conditional transitions (guarded transitions) between locations. The priority of transitions are specified by their number. Transition with lower number has priority over a transition with a higher number. In this part of the system three transitions are specified. $G_5$ is the guard on transition from *Change Status* to *Idle*. This guard checks whether input and output buffers are all empty. If true, the switch component has no task to do and transits to the idle phase. For a switch to transit to the *Routing* location ($G'_3$), two conditions must be met. Obviously, there should be a non-empty input queue with a packet ready to be routed through the network. However, an external event must occur for such a packet to enter one of the input queues. Therefore, occurrence of an external event ($Event_{External}$) is considered as a precondition for this guard. Similar to $G'_3$, transition from *Idle* to *Change Status* using $G'_1$ has the precondition of the occurrence of an external event. In addition, the status inconsistency flag (*statusInconsistent*) must be true for this guard to be satisfied. This flag is true whenever there is an inconsistency between the status of input buffers of the switch and the external status which is transmitted previously.

It is important to note that the above is a simple part of the switch model and thus intended to shed light on some of the basic model specification elements of our approach. The complete model contains five locations and a few other secondary state variables that are involved in the functionality of switch component. Details for the switch model is provided in Appendix A and [11].

## 3.5 NoC Simulation Model

RT-DEVS offers basic artifacts that are important for modeling the real-time properties of NoCs (see Section 3.1. However, there are still obstacles in the way of real-time simulation. First, even though the formalism defines physical time for actions, how the DEVS simulator protocol can specifically handle minimum and maximum times in physical time are not provided. Normally in a DEVS-based simulation, this is not a problem because the simulation is being carried out in logical time. However, in a RT-DEVS-based simulation, this could result in missing deadlines if there are too many actions or events given the executing simulation platform limitations. Second, actions in a state may have priorities and sequence of execution and activity mapping function (in charge of mapping actions to states) does not specify order or priority of execution among actions mapped to a specific state.



Interface Level
• Models packet delivery & network interfaces
• Simple approximations of packet latency
• Incorporated in the early stages of the project
• Provides general behavior of the network
Capacity Level
• Adds resource constraints
• Evaluates the initial performance of the network
• Simple and easily configurable
• Fast in execution
Flit Level
• Individual flits are modeled
• Detail modeling & simulation of router components
• Accurate results in terms of latency
• Results are expressed in flit times (cycles)
Hardware Level
• Area & timing information of components are added
• Latency in terms of absolute time instead of cycles
• Most accurate and useful because of the absolute timing
More detailed, More accurate
Faster Execution

**Figure 3: Simulation level hierarchy**

The similarity of the real-time simulation to the real system is dependent on the modeling phase and the level of detail included. Simulation models specified at the Register Transfer Level (RTL) are close to their physical realizations; however, the physical time required for simulation is greater as compared with high level models assuming the use of a simulator and comparable simulation scenarios. As illustrated later, in order to reach real-time simulation we eliminate many details from each component of the system. For example, a processing element possesses internal buffers, memories, and ALUs, which are all summarized in a single PE model in our modeling approach. Figure 3 shows different levels of network on chip simulations [8].

Our objective in terms of level of simulation is better described by the Flit level. Since we are concerned with individual flits, buffering, routing, and accurate latency analysis, flit level simulation suits better than the others. The upper two layers are not accurate enough and our real-time simulation constraint eliminates the possibility of using the hardware level simulation because of its lengthy execution time.

## 4. DEVS-SUITE SIMULATION TOOL

DEVS-Suite is a simulation environment for developing and simulating hierarchical and component based models. This tool combines the capabilities of DEVSJAVA and DEVS

Tracking Environment [14]. This simulator supports execution of models according to the parallel DEVS formalism. It provides monitoring capabilities (such as time trajectories) and automated data collection. The simulator clock can be set to be faster, equal, or greater than the physical clock of the computing platform that executes the models. However, while it is possible to execute models in wall clock time using a real time factor control provided in the tool, there is no guarantee for meeting the deadlines for simulations that have high computational requirements. In such scenarios, the simulator cannot be synchronized with wall-clock. Therefore, DEVS-Suite is generally considered as a logical-time simulator. In order to reduce mismatch between simulation execution speed and simulation data visualization, a real-time synchronization was introduced. However, arbitrary synchronization accuracy cannot be guaranteed.

In addition, DEVS-Suite does not provide direct support for developing RT-DEVS models. For example, in DEVS-Suite every operation (action) in the external transition function can be defined to have its own time advance, but the simulator protocol does not facilitate orchestrating handling of multiple operations with external or internal transition functions (see Section 4.1). Therefore, support for RT-DEVS formalism (model specification and simulator protocol) is required. Next, a brief overview of the DEVS-Suite simulator is provided.

## 4.1 DEVS-Suite Simulation Engine

DEVS-Suite tool uses MFVC (Model-Facade-View-Controller) design pattern in which a facade layer is added to the Model layer to provide selected functionality to other layers. In this scheme, the View and the Controller layers are in charge of communication with the user by receiving inputs and visualizing simulation updates sent from the Facade layer. Controller is responsible for transforming user inputs to Model layer requests and call appropriate procedures. In the end, Model layer is the core of the simulation in which atomic and coupled models communicate to carry out the simulation and provide output results to the View layer [19]. In this simulator, models are implemented as *atomic* or *coupled* with *atomicSimulator* and *coupledCoordinator* being responsible for executing them. Detailed information regarding this tool can be found in [14].

## 4.2 RT-DEVS Simulation Platform for DEVS-Suite

Modifications to the DEVS-Suite starts with the simulation protocol. A complete simulation protocol with double-struck edged rectangles representing separate processes is shown in Figure 4. Since the flowcharts for those processes are simple and similar to those of the DEVS-Suite's protocol, they are excluded here.

As illustrated in Figure 4, the simulation cycle starts after an event occurs or the model is initialized with $T_{max}$ to have a finite value. The event could be triggered due to an action not being completed within its maximum allowed period, receipt of external events, or completion of an action. In the first decision block, input message bag is checked. In case of the bag not being empty, an external event has occurred which is handled by external transition function.

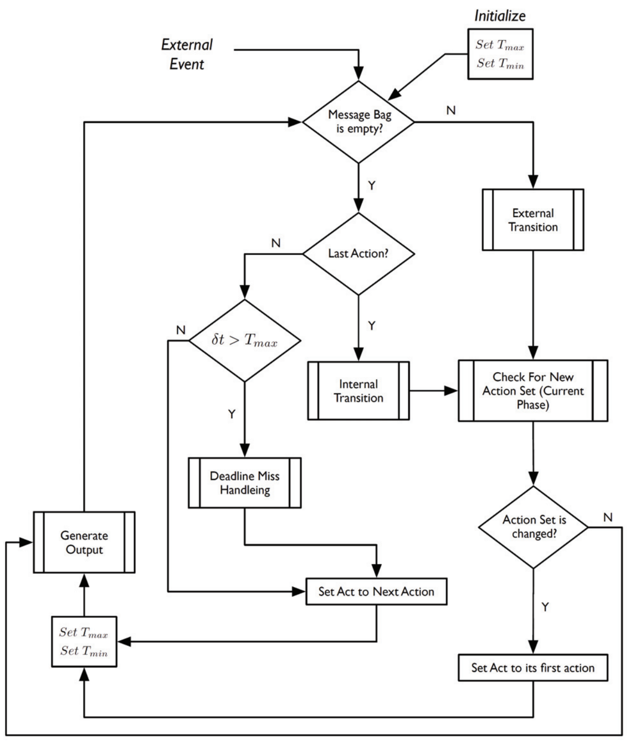


**Figure 4: Simulation protocol for RT-DEVS simulation platform**

Otherwise, the event could either be the completion of an action or the missing of an action's deadline. These two events are distinguished by comparing $\delta t$ with the maximum allowed execution time for the action ($T_{max}$). If a deadline miss is occurred, deadline miss handling is done and then the next action is picked for execution. Before checking for deadline violation, another decision block checks whether the executed action was the last one in its state. If the action is the last one, internal transition function is called and then the action set is changed. The first action in this set is chosen for execution and the cycle continues.

In the Parallel DEVS simulation protocol, completion of internal events cannot trigger external transition function. External events can interrupt external and internal transition functions. The consecutive execution of these two has only two combinations: external to internal and internal to internal. The execution period defined for external and internal transition functions are relative to the simulator's clock.

A finite number of actions are independently assigned to internal and external transition functions. Actions are not exclusive to internal or external transition functions. The triggering and execution of the actions is handled by the simulator's real-time clock. The absolute event times (i.e., $t_{min}$ and $t_{max}$) for every action is specified in its atomic model. The model's time periods (i.e., $\delta_{t,min}$ and $\delta_{t,max}$) are relative with respect to the simulator's real-time clock variables $t_{last}$, $t_{current}$ and $t_{next}$. A model's start and end time instances for an action are synchronized with the simulator's clock. The time instance at which an action is completed or terminated is determined by the simulator's clock. That is, if an action cannot be completed by the action's $t_{max}$ (i.e., $t_{max} > t_{next}$), the action is terminated and the simulator may enter its next execution cycle. This is because hardware resource availability is assumed to be bounded. If an action is completed within $t_{min}$ and $t_{max}$ measured against the simulator's real-time clock, the simulator enter its next cycle or the simulation is terminated if no more actions are scheduled. It should be noted that the start and end time instances of an action specified in the model are not changed by the simulator.

Confluent function is not supported in this flowchart. It is hypothesized by some [16] that since two events do not happen at the same time and a single processor simulator is capable of handling only one event at each time, the confluent function is not needed for real-time formalisms. However, we believe that the occurrence of two or more events at a time instance is dependent on time granularity and the level of the abstraction in modeling. Low-granular time causes multiple events to be perceived by the model at one time instance, although they may have occurred in different times. In addition, high-level complex models have a higher probability of receiving multiple events at the same time. Thus, for infinite granularity, the simulation protocol sketched in Figure 4 works fine; otherwise, this would result in missing output events if these multiple events are not manually checked and handled.

Execution of model can be achieved in three modes – i.e., as-fast-as-possible, real-time, or scaled real-time. As noted above, we are interested in real-time execution where advances in the simulation clock occur in synchrony with advances in a wall (or a physical) clock. The simulation clock (logical clock) is the physical time within the simulator. The wall-clock represents the time during the execution of a model that is equal to the rate at which the physical time progresses. In a real-time simulation, the simulation clock is synchronized with the wall-clock (such as the clock provided by hardware) at certain points. This enables the simulation to be a realistic representation of the physical system as perceived by human operators.

Synchronization between wall-clock and logical time is necessary for a real-time simulator. Thus, simulation clock is advanced after the advance of wall-clock. In other words, the simulation engine must wait for a time advance in wall-clock and then increment logical time in order to reach real-time execution. A real-time simulation (as opposed to faster- or slower-than real-time) of this sort equips us with useful controls, which neither real system nor logical-time simulation are capable of providing. A hardware system is only able to raise interrupts and handle them at certain times (clock edges) and so as the logical-time simulation. However, real-time simulation (because of its synchrony with wall-clock and communication with the environment) gives us the ability to raise and handle events at any time instance, which is most useful in reactive systems.

## 5. EXPERIMENTS & RESULTS

In this section, sample NoC system is modeled with the RT-DEVS formalism and is then executed in the extended DEVS-Suite simulator. For this purpose, first of all, the components are modeled with the approach presented in this paper. The specification for the Switch component is provided in Appendix A. In order to be able to execute the RT-DEVS models with the properties described in Section 4, we modified the DEVS-Suite tool to support RT-DEVS models and the concepts of locations, guard, and transitions. This model helps to demonstrate the details of the modeling approach and the capability of the modified DEVS-Suite tool to handle actions and their timing requirements.

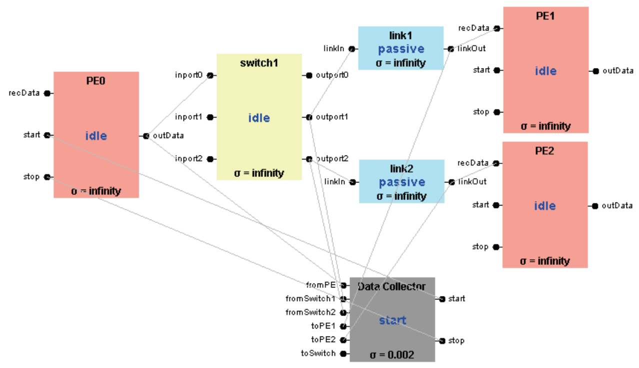


**Figure 5: RT-DEVS NoC sample model**

### 5.1 A Basic NoC Model

A sample RT-DEVS NoC model shown in Figure 5 contains three processing elements, a switch and two links. A data collector component is also modeled and implemented to gather results from model execution cycle.

The PE model is a model with the simple functionality of receiving incoming flits and generating outputs. The switch model (refer to Appendix A for a detailed description) receives flits from *PE0* and routes them to either *PE1* or *PE2* based on their destination addresses. As shown in Figure 5, the *Data Collector* component starts/ends the simulation by sending a start/stop signal to *PE0* which acts as a producer while other processors (*PE1* and *PE2*) act as consumers. In addition to starting and ending the simulation, the *Data Collector* component gathers timing data from the model components. The timing analysis compares the simulation clock with the wall-clock with the simulator's maximum accuracy set at $10^{-4}$ second). Performance measurements (i.e., average flit latency and average queueing times) are calculated.

### 5.2 Model Configuration

In the set of experiments conducted on this model the actions for each component are configured in terms of time windows within which actions are to be executed. Since our software simulation has time granularity limitations, we used linear scaling in order to carry out the simulation and test the functionality of the NoC model and its parts. Following the model for switch described in Section 3.4, the switch model implemented for the purpose of this experiment contains the same actions and locations. The model for PE, has a single action (*Processing*) which generates output (a set of flits) with a random period between its $T_{min}$ and $T_{max}$. Finally, *Transmission* action for link delivers the flits some time between its respective time window. The

**Table 1: Performance results for default configuration (PT: Physical Time, ST: Simulation Time)**

| | Avg. Point ($sec$) | $D_{Max}$ ($sec$) | $D_{Min}$ ($sec$) |
|---|---|---|---|
| Flit Latency (ST) | 0.18370 | 0.00189 | 0.00242 |
| Flit Latency (PT) | 0.20078 | 0.00257 | 0.00197 |
| Queueing Time (ST) | 0.18023 | 0.00185 | 0.00236 |
| Queueing Time (PT) | 0.19627 | 0.00254 | 0.00219 |
| Link Latency (ST) | 0.00347 | 0.00006 | 0.00006 |
| Link Latency (PT) | 0.00451 | 0.00021 | 0.00020 |

default timing configuration is specified below (expressed in seconds).

- PE – Processing: $[.8, .9]$
- Link – Transmission: $[.002, .005]$
- Switch – HeaderDecoder: $[.005, .01]$
- Switch – Arbiter: $[.005, .01]$
- Switch – SendOutFlit: $[.02, .03]$

In this configuration, the processing element generates 8 flits in each cycle which gives us a packet generation rate between 9 and 10. The difference of this action with other actions is that other actions work on individual flits instead of a group of them. Keep in mind that these values are not meant to represent a real NoC system. The sample model and its configuration is used to verify the simulator's correctness (executing actions within their allowed time windows) and to show NoC model generates expected dynamics.

## 5.3 Results & Analysis

First, we executed the model with the default configuration. Each execution has a warm up period of 30 cycles (enough for such system) and then a period of 30 cycles for gathering data. The average results of 5 runs are presented in the first column of Table 1 (all numbers are in seconds). The second column ($D_{Max}$) provides the deviation with the maximum value among all five runs and the third column ($D_{Min}$) the deviation with the minimum of all five results. As shown in this table, the difference between the simulation clock (rows marked with ST) and the real-time (rows marked with PT) clock is small for each performance measure. All actions are executed within their time windows. Link latency should be in the time window of $[.002, .005]$ as in the default configuration. The results show that none of the *Transmission* action times have violated their deadlines. Showing the same thing about other actions is not as straightforward as the *Transmission* action. Below is a formulation showing the actions inside the switch are executed according to their specified timings. The demonstration uses the knowledge that at each point there are 8 packets in the switch's input buffer.

Single packet latency: $L_{ZeroLoad} = .0075 + .0075 + .015$
First packet latency: $L_1 = L_{ZeroLoad}$
Second packet latency: $L_2 = L_{ZeroLoad} + L_1$

$$\vdots$$

Eighth packet latency: $L_8 = L_{ZeroLoad} + L_1 + L_2 + \cdots + L_7$

**Table 2: Queueing time based on packet injection**

| $ti$(Processing) ($packet/sec$) | Injection Rate ($flit/sec$) | Average QT ($sec$) |
|---|---|---|
| [.8, .9] | 9.41 | 0.198808392 |
| [.32, .35] | 23.88 | 0.206464698 |
| [.28, .31] | 27.12 | 1.832193793 |
| [.25, .28] | 30.19 | 3.442061815 |
| [.20, .23] | 37.21 | 6.254637832 |
| [.15, .18] | 69.57 | 8.749146720 |
| [.08, .1] | 88.89 | 12.59097263 |

Therefore, average packet latency is:

$$\frac{L_1 + L_2 + \cdots + L_8}{8} = \frac{.04 \times (1 + 2 + \cdots + 8)}{8} = .1775$$

Comparing this value (reached analytically) to the values in Table 1 (reached practically) shows that actions in switch have also met their deadlines on average.

Increasing the packet injection rate, by changing the time interval of the *Processing* action of producer PE, results in longer queueing times in switch. We tested the model with 7 different injection rates (packets per second) and the results in terms of average queueing times (in seconds) are shown in Table 2. The queueing time does not experience a radical change between the first two points. The reason is that switch's service rate is still higher that PE's injection rate. However, the queueing time is rapidly increased as the injection rate goes higher. In this situation the rate of packet injection is higher than service rate. Therefore, flits traveling through the network experience long queueing times. The simulation experiments were conducted on a Macintosh platform with 2.4GHz Intel Core 2 Duo processor and 2GB of memory.

## 6. CONCLUSION

In this paper, we provided a novel model for NoC system using RT-DEVS formalism. We contributed to the modeling approach by introducing the concept of primary/secondary state variables and Real-time Statecharts which resulted in a more systematic and simpler modeling approach as demonstrated in Appendix A. In addition, DEVS-Suite simulation environment is extended to support execution of RT-DEVS models in physical time.

We concluded that RT-DEVS formalism is insufficient for modeling the kinds of details and flexibility that are needed for NoC. Our work introduces detailed timing specification for actions and their priorities over each other under various time constraints. The resulting modeling elements and method offer a rigorous approach for specifying real-time software systems such as NoC. Real-time Statechart support for real-time specification made this modeling approach the best choice for our purpose. In addition, Real-time Statechart can be converted to Timed-Automata [12] which in turn can be used for model checking purposes. This enables us to use the simulation model for verification purposes as well.

It is important to emphasize that the possibility of real-time simulation for a given model is dependent on the underlying computing platform. Although our generic modeling approach is capable of modeling NoC systems, it is impractical to simulate every NoC with arbitrary scale and complexity using existing computation platforms. For example, communication speed in some NoC systems can be several magnitude faster than what may be possible in simulation. In this situation, assuming NoC as a system operates linearly, simulation time can be scaled. However, scaling for a real-time simulation is not straightforward especially considering embedding NoCs in some physical environment. There are several other areas requiring research. One is distributed simulation where a systematic approach is taken to migrate models for execution on a distributed simulation platform. The ability of simulating a system in real time needs is directly related to how much details is to be modeled. Exclusion of details from the target system can lend to real-time simulation. Obviously, there can be a substantial difference between a real system and its high-level simulated counterpart. Currently, determining the right amount of detail to be included in the model is the responsibility of the modeler.

## 7. REFERENCES

[1] H. Ahmadinejad, F. Refan, and H. Sarjoughian. NoC simulation modeling in DEVS-Suite. In *Spring Simulation Conference, Orlando, FL*, pages 134–139. ACM Press, 2011.

[2] N. Banerjee, P. Vellanki, and K. Chatha. A power and performance model for network-on-chip architectures. In *Proceedings of the conference on Design, automation and test in Europe-Volume 2*, pages 1250–1255. IEEE Computer Society, 2004.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39:1–7, Aug. 2011.

[4] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The M5 simulator: Modeling networked systems. *Micro, IEEE*, 26(4):52–60, 2006.

[5] Y. Chen and H. Sarjoughian. A component-based simulator for MIPS32 processors. *Simulation*, 86(5-6):271–290, 2010.

[6] S. Cho and T. Kim. Real time simulation framework for RT-DEVS models. *Transactions of the Society for Computer Simulation International*, 18(4):203–215, 2001.

[7] A. Chow and B. Zeigler. Parallel DEVS: A parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th conference on Winter simulation*, pages 716–722. Society for Computer Simulation International, 1994.

[8] W. Dally and B. Towles. *Principles and practices of interconnection networks*. Morgan Kaufmann, 2004.

[9] F. Fazzino, M. Palesi, and D. Patti. Noxim: Network-on-chip simulator. *URL: http://sourceforge.net/projects/noxim [24.06.2008]*.

[10] A. Furfaro and L. Nigro. Embedded control systems design based on RT-DEVS and temporal analysis using UPPAAL. In *Computer Science and Information Technology, 2008. IMCSIT 2008. International Multiconference on*, pages 601–608. IEEE, 2008.

[11] S. Gholami and H. 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.

[12] H. Giese and S. Burmester. Real-time statechart semantics. Technical Report TR-RI-03-239, University of Paderborn, 2003.

[13] J. Hong, H. Song, T. Kim, and K. Park. A real-time discrete event system specification formalism for seamless real-time software development. *Discrete Event Dynamic Systems*, 7(4):355–375, 1997.

[14] S. Kim, H. Sarjoughian, and V. Elamvazhuthi. DEVS-Suite: a component-based simulation tool for rapid experimentation and evaluation. In *Spring Simulation Multi-conference, San Diego, CA, USA*, 2009.

[15] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[16] M. Moallemi and G. Wainer. Designing an interface for real-time and embedded DEVS. In *Proceedings of the 2010 Spring Simulation Multiconference*, SpringSim '10, pages 137:1–137:8, New York, NY, USA, 2010. ACM.

[17] M. Moallemi and G. Wainer. I-DEVS: imprecise real-time and embedded DEVS modeling. In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, TMS-DEVS '11, pages 95–102, San Diego, CA, USA, 2011.

[18] A. Saghir, T. Pearce, and G. Wainer. Modeling computer hardware platforms using devs and hla simulation. In *2004 Summer Simulation Conference*, San Jose, California, July 2004.

[19] H. Sarjoughian. Component-based simulators: DEVS-Suite concepts, techniques, and operations. *http://devs-suitesim.sourceforge.net/*, Feburary 2009.

[20] H. S. Song and T. G. Kim. Application of real-time DEVS to analysis of safety-critical embedded control systems: Railroad crossing control example. *Simulation*, 81:119–136, February 2005.

[21] D. Wang, N. E. Jerger, and J. G. Steffan. Dart: a programmable architecture for noc simulation on fpgas. In *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '11, pages 145–152, New York, NY, USA, 2011. ACM.

[22] B. P. Zeigler, T. G. Kim, and H. Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2nd edition, 2000.

## APPENDIX

This appendix is provided as a complementary section for the modeling approach described in this paper. As mentioned earlier in the paper, the specifications for the NoC components can be found in [11].

## A. SWITCH RT-DEVS SPECIFICATION

In this section, we provide a summary of the RT-DEVS specification for NoC switch component. The model described here, is more complex than the model partially described in section 3.4. Our modeling approach for NoC components is clarified in sections A.1 to A.7 by covering the important aspects of the switch component. In RT-DEVS specification approach, the functionality of the target system is presented in a much more systematic way via actions. We contribute to this approach by introducing primary/secondary state variables and the concepts of locations and transitions. The switch model presented below is a good example for demonstrating the suitability of this approach for real-time modeling and simulation.

### A.1 Introduction to NoC Switch

This component is responsible for routing and forwarding flits received from network interfaces or switches to other switch or NIs. The model described in this section uses deterministic table based routing scheme. Since the complete model is lengthy, the following sections provide a selection of features for illustrating the switch model.

### A.2 State, Input, Output

States ($S$), input ports ($X$) and output ports ($Y$) of switch are described below.

$$S = \overbrace{\{Active, Idle\}}^{\text{phase}} \times \overbrace{\sigma}^{\text{sigma}} \times \overbrace{\{0,1\}^*}^{\text{inBufs}} \times \overbrace{\{0,1\}^*}^{\text{outBufs}} \times \overbrace{\{0,1\}^*}^{\text{extStatus}}$$
$$\times \underbrace{\{0,1\}^*}_{\text{intStatus}} \times \underbrace{\{true, false\}}_{\text{statusInconsistent}}$$

$$X = \big\{\big(in[0..portNum-1], \{0,1\}^*\big), \big(extST[0..portNum-1], \{Ok, Nok\}\big)\big\}$$

$$Y = \big\{\big(out[0..portNum-1], \{0,1\}^*\big), \big(intST[0..portNum-1], \{Ok, Nok\}\big)\big\}$$

The simplicity of this modeling approach causes the state model to be much simpler than DEVS specification [1]. Here, the switch is either active or idle and various functionalities are provided by each location and the actions mapped to it (described in sections to follow). Each switch possesses several input (*InBufs*) and output (*OutBufs*) buffers for storing incoming and routed flits. However, these buffers have finite capacity. Thus, internal status stores the status of each input buffer. Whenever one of the model's input buffers is full the respective status is changed into "Nok". This value is changed back to "Ok" whenever the buffer becomes non-full. The *statusInconsistent* state variable is set to true whenever an input buffer has changed its status but it has not been reflected in *intST* output ports yet. The external status keeps the status of neighboring nodes so that the switch knows when it is safe to send out the flits stored in output buffers.

### A.3 External Transition Function

External transitions handle the events received via input ports of the component. In switch, there are two types of external events: arrival of a flit or status change in a neighboring node. Below one example of each are expressed in RT-DEVS approach.

$$\delta_{ext}(phase, \sigma, inBufs, outBufs, extStatus, intStatus, statusInconsistent, e, (in[m], X))$$
$$= (phase, \sigma - e, inBufs[m].add(X), outBufs, extStatus, intStatus, statusInconsistent) \quad \text{[if inbuf[m] is not full]}$$
$$= (phase, \sigma - e, inBufs[m].add(X), outBufs, extStatus, intStatus, true) \quad \text{[if inbuf[m] is full]}$$

$$\delta_{ext}(phase, \sigma, inBufs, outBufs, extStatus, intStatus, statusInconsistent, e, (extST[m], X))$$
$$= (phase, \sigma - e, inBufs, outBufs, extStatus[m] = X, intStatus, statusInconsistent)$$

The first event, shows switch behavior after the arrival a flit. The flit is added to the respective input queue and the model continues with the previous action it was executing. However, the status of input queues must always be checked after an insertion. In case of a full input buffer, the model must change its *statusInconsistent* state variable to modify the value of its *intST* ports later. The second event occurs when the status of a neighboring node is changed. This event is handled by changing the external status variable to the right value.

### A.4 Internal Transition Function

Internal transitions are scheduled by the generation of an output. They change the state of the model based on its current state. In RT-DEVS, internal transitions occur whenever all actions associated with that state are executed. Switch has a simple and straightforward specification for internal transitions. Since there are only two phases for switch (described in A.2), there are only two internal transitions that are described below.

$$\delta_{int}(\text{``}Active\text{''}, \sigma, inBufs, outBufs, extStatus, intStatus, statusInconsistent)$$
$$= (\text{``}Idle\text{''}, \infty, inBufs, outBufs, extStatus, intStatus, statusInconsistent) \quad \text{[If all } InBufs \text{ and } OutBufs \text{ are empty]}$$

$$\delta_{int}(\text{``}Idle\text{''}, \sigma, inBufs, outBufs, extStatus, intStatus, statusInconsistent)$$
$$= (\text{``}Active\text{''}, \Delta t_{loc}, inBufs, outBufs, extStatus, intStatus, statusInconsistent) \quad \text{[If at least one of } InBufs \text{ is not empty]}$$

In the first internal transition, the model changes its phase into *Idle* whenever it does not have any flit to route. The second one demonstrates the opposite situation. In addition to the phase of the model, the sigma of the model is affected by the internal transitions. Obviously, the sigma is set to infinity for the *Idle* phase. As for the second transition, the sigma is set based on the model location chosen for this phase. The locations of the switch and how they are chosen are described in section A.7. As it is obvious in the specification above, internal transitions are concerned with primary state variables (refer to 3.2) and they do not affect secondary state variables. So, internal transitions change the primary

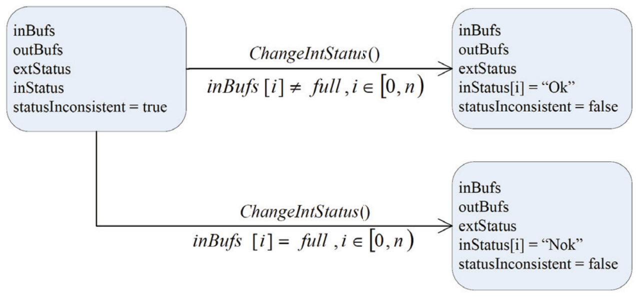

inBufs
outBufs
extStatus
inStatus
statusInconsistent = true
ChangeIntStatus()
$inBufs\,[i] \neq full, i \in [0,n)$
inBufs
outBufs
extStatus
inStatus[i] = "Ok"
statusInconsistent = false
ChangeIntStatus()
$inBufs\,[i] = full, i \in [0,n)$
inBufs
outBufs
extStatus
inStatus[i] = "Nok"
statusInconsistent = false

**Figure 6: Effect of ChangeIntStatus action on secondary state variables**

state variables and then the location of the model is decided based on secondary state variables.

## A.5 Actions, Activity Mapping Function, Time Intervals

RT-DEVS specifies a set of actions for each model but provides no accurate definition for them. In this work, an action is described by its output and effect on secondary state variables. In contrast to DEVS which creates an output after each internal transition, our approach does that after the execution of each action. In order to accurately define the effect of each action on the secondary states variables we employ Real-time Statechart modeling as described below.

In order to show the effect of actions using Real-time Statechart, each possible evaluation of secondary state variables forms a location. Based on previous knowledge, each transition in Real-time Statechart may contain an action which is executed whenever the transition occurs. The action which its effect is being defined is placed as the action of transition between two locations. A detailed definition of action can even split the action into sub-actions which only modify one state variable at a time. Below, is the list of actions for switch, their definition using Real-time Statechart, and the time intervals associated with them.

$A = \{HeaderDecoder, ChangeIntStatus, Arbiter,$
$\quad SendOutFlit\}$

The definition of the *ChangeIntStatus* action is specified in Figure 6. The *ChangeIntStatus* action checks input buffers ($i \in [0, n), n = portNum$) and changes the internal status of input buffers to "Ok" if they contain empty cells and to "Nok" otherwise. In addition, after modifying all *intStatus* values, it changes the value of *statusInconsistent* to false since the internal status and buffer status values are synchronized. Other state variables are left untouched. Other actions are also defined with the same approach described above. However, we move on to the next topic because of lack of space.

Time intervals for the actions specified above are described by a time window.

$$ti(Action) = [T_{min}, T_{max}]$$

Time windows for actions in a NoC system are set based on the hardware specification of the target system.

## A.6 Output Function

As mentioned before, in our approach, each action generates an output after completion. Therefore, the output function should be defined over all action for the switch model. These definitions are as follows.

–$\lambda(HeaderDecoder) = \emptyset$
–$\lambda(ChangeIntStatus) = (intST[i], intStatus[i])$ [for all $i \in [0, NumOfBufs)$]
–$\lambda(Arbiter) = \emptyset$
–$\lambda(SendOutFlit) = (out[i], outBufs[i].head)$ [if outBufs[i] is not empty, for all $i \in [0, NumOfBufs)$]

In switch model, *Arbiter* and *HeaderDecoder* actions do not generate output. The *ChangeIntStatus* action generates "Ok" or "Nok" signals on *intST* ports and *SendOutFlit* sends one flit for each output port if their respective output buffers are not empty.

## A.7 Locations, Transitions, Guards

We add two extra locations to the switch model presented in section 3.4. First, *Waiting* location which waits infinity until the neighboring node is ready to receive or other input/output buffers have pending flits.

–Location *Waiting*
–Actions: $\emptyset$
–Guard: $[\exists i \in \mathbb{N} \Rightarrow outBufs[i]\ is\ nonempty \wedge extStatus[i] = \text{“}Nok\text{”}]$

This location is mapped to *Active* phase and is activated whenever there exists an stalled outgoing flit because of the status of a neighboring switch. The other location added to the switch model is *Transmission*. This location is responsible for sending ready flits out to the next node. This location is defined below.

–Location *Transmission*
–Actions: $\{SendOutFlit\}$
–Guard: $[\exists O \in outBufs : O\ is\ nonempty]$

Transition to *Transmission* location is enabled whenever there exists an output port which contains a ready packet. Based on the numbering provided on the guards to each location, transition to *Transmission* is second in priority after *Change Status*.