NoC-DEVS Simulator

Hoda Ahmadinejad[1], Fatemeh Refan[2], Hessam S. Sarjoughian[3]

[1]Computer Architecture Research Group, ECE Department, University of Tehran, Tehran 14399, Iran
h.ahmadinejad@ece.ut.ac.ir,
[2]CAD Research Group, ECE Department, University of Tehran, Tehran 14399, Iran
refan@cad.ece.ut.ac.ir,
[3]School of Computing, Informatics, and Decision Systems Engineering, Computer Science and Engineering Faculty, Arizona State University, Tempe, Arizona, USA
ECE Department, University of Tehran, Tehran 14399, Iran
sarjoughian@asu.edu,

Keywords: DEVS, NoC, NoC-DEVS, Noxim, SystemC

Abstract
Study of Network-on-Chip (NoC) systems requires simulators capable of handling their unique characteristics. Toward this objective, a set of simulation models are developed based on NoC first principles and the DEVS framework. The components necessary to build simulation models for NoC are developed using Parallel DEVS and implemented in NoC-DEVS which extends the general-purpose DEVS-Suite simulator. An example mesh-based NoC model synthesized from processing elements, network interfaces, switches, and links is experimented with and analyzed. The same example is also studied in Noxim which is an extension of the SystemC simulator. The NoC-DEVS simulator is evaluated and compared against the Noxim simulator. The comparison focuses on their modeling capabilities and considers delay and throughput performance metrics as well as capabilities expected from advanced simulation tools. Related and future research directions are briefly discussed.

1 Introduction

Developing design specifications for System-on-Chip (SoC) is known to be challenging [1]. Similar to other complex networked systems, design solutions must account for operations and communications among different components. SoCs and in particular Network-on-Chip (NoC) systems [2] have some unique characteristics – e.g., their physical structures and interactions are constrained as compared with common computer network systems. Such distinct features have prompted development of new concepts, methods, and tools (e.g., simulators) to aid in the design and experimentation of Network-on-Chips.

Given the importance of early exploration of NoC design solutions, specialized modeling and simulation approaches and tools have been proposed. General-purpose counterparts are difficult to use. Indeed, there are numerous cases (e.g., computer network simulators) where specialized modeling concepts, theories, and methods have been shown to be very useful in the modeling and simulation life-cycle. It is also worth noting that real-world experimentation for complex systems, especially in the early stages of system development, remain limited and often impractical [3].

Recently, a few NoC simulators have been developed. Among formal models, Petri Net is used to model NoC performance characteristics under alternative communication schemes [3] [4]. These models are based on Colored Petri Net and Deterministic Stochastic Petri Net. Noxim [5] use concepts and constructs provided by SystemC [11] and programming languages.

This paper presents an NoC model developed based on NoC first-principles and the general-purpose Discrete Event System Specification (DEVS) modeling formalism [6]. The NoC model types are Processing Element (PE), Switch (SW), and Network Interface (NI). The Network Interface is a composite model as it has packetizing and de-packetizing model components. The logical structure and behavior of these models are derived from the NoC description. The NoC model is implemented in DEVS-Suite [7]. To experiment with the NoC models, an experimentation model is devised. Its responsibility is to define the conditions for conducting experiments. The NoC model can be externally controlled and observed via start and stop signals provided by the experimentation model.

The rest of paper is organized as follows: In Section 2 the basic concepts and characteristics of NoC are presented. In Section 3, related work and tools are briefly described. In Section 4, the approach and developed NoC simulation models are described. The experimentation testbed, results and a comparison with Noxim simulator is presented in Section 5. Finally Section 6 concludes the paper and highlights future work.

2 Background

As shown in Figure 1, Processing Elements, Network Interfaces, Switches, and Communication channels can be synthesized to create arbitrary mesh-structured NoCs. Processing elements (or cores) are the computational building blocks. They are the sources and destinations of data. In the NoC depicted in Figure 1, nine processing elements are uniformly connected to one another through a collection of network interfaces, switches, and communication links.

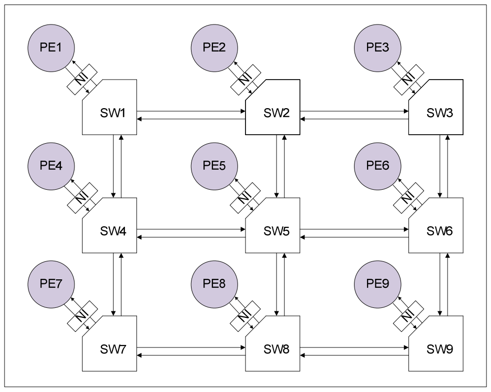

Figure 1: A typical mesh-based NoC architecture

A network interface, as the name suggests, is the gateway between a PE and a Switch. I.e. its main responsibility is to translate the PE's data into network data and vice versa using packetization, and depacketization schemes. Based on the NoC characteristics and PE requirements, the packet reordering and retransmission control need to be managed by NI [8].

A switch is responsible for routing data among PEs usually using other intermediary switches. It is comprised of input ports and buffers, router state, routing logic, allocators, and a crossbar. More detailed functionality of switches is highly dependent on the choice of other factors such as routing algorithm, flow control, and network topology. E.g. given the NoC in Figure 1, the switches may have three (e.g., Switch 1), four (e.g., Switch 2), or five (e.g., Switch 5) ports. Since the traverse of data in NoC is through switches and generally there are more than one switch in an NoC, the design of switches has a profound impact on the performance metrics such as latency, power consumption, and specially hardware cost [9].

The communication channels can be categorized in two types. Conventional type supports primitive communication – e.g., packets are sent and received between a PE and a switch. Buffering type uses ports for some NoC elements – e.g., switch-to-switch communication uses queuing.

Other elements and aspects of NoCs including topology, routing, and flow control also need to be modeled. The physical layout and connections between nodes (PEs, Switches, NIs) and channels (communication links) in the network are determined by network topology. For example, the connectivity pattern among PEs can significantly impact NoC performance measures. The total number of alternate paths between nodes directly affects how traffic spreads in relation to available network bandwidth. NoC can have one of several kinds of topologies (direct, indirect, regular, and irregular [9]). Given the NoC's topology, the routing algorithm determines the paths through the network that a message can traverse to reach its destination. It affects the latency, power consumption, and traffic distribution in addition to network topology. A routing algorithm, in addition to being deadlock free, should support features like contention minimization and hot spot avoidance, and at the same time does not require much hardware overhead in switches or additional fields in packets. Among routing algorithm categories, minimal vs. non-minimal and oblivious vs. adaptive routing can be named [9].

Flow control determines how resources are allocated to messages as they pass through the system. It allocates (and de-allocates) buffers and channel bandwidth to packets. Furthermore, it needs to determine the granularity of data to be traversed. Given time and area constraints, flow control is handled at the flit level [9]. The flow control affects the latency and throughput. Flit is a Flow Control unit, a part of a packet. Flits are usually defined in constant size in a NoC, and are commonly categorized into head, normal, and tail types. Head flit includes addressing information needed for routing a packet. Typically it is constructed from the header part of packet. Tail flits have an indicator in order to show that they are the last flit of packets. Normal flits are data flits which are sandwiched between head and tail flits.

Metrics such as network latency and accepted traffic are defined for evaluating NoC designs. The lower bound on average message latency is measured as zero-load latency which is the latency experienced by a packet when there are no other packets in the network. Another important metric is throughput which is calculated from latency, but provides more detailed information. The primary costs defined in NoCs are area and power. Area is mostly traded-off with design complexity, while total power consumption is affected by paths that packets traverse [9].

3 Related work

A number of NoC simulators have been proposed in literature. An earlier work uses SystemC and the ns-2 network simulator to efficiently simulate embedded systems [10]. A design is developed to accurately represent hardware and network interactions. Use of ns-2 makes the use of this kind of simulation modeling difficult for NoC. Most recently, Generalized Stochastic Petri Nets and Colored Petri Nets are used for NoC performance evaluations such as CPU memory throughput and average clock cycles required for establishing connections between a data source node and data sink source [3]. A basic model consisting of two processors, local RAM and dedicated serial ports compete for a share memory using a common communication structure has been developed. This Deterministic and Stochastic Petri Net (DSPN) model has 19 places and 20 transitions. It has been validated using an emulated FPGA-based testbed. To more accurately model this system, the basic model is extended with an additional 26 places and 28 transitions. Another model that has a regular mesh structure is also developed. The model which has 500 places and 600 transitions represents a system having a regular mesh-structure with eight clients and four nodes. The results reported are incomplete and hierarchical modeling is proposed as future work.

A more recent study uses Colored Petri Nets for modeling mesh-based and k-ary NoC [4]. The models capture detailed structure and dynamics of NoCs. The models similar to the DSPN models require very large number of places, arcs, and transitions and exhibit high visual complexity. In this study a 4×4 mesh is studied under different scenarios. Using a variety of supporting tools and considering diverse operational settings, it is observed that the model is deadlock free. Performance metrics (percent switch loading) are discussed. Study of alternative network communication and role of switch fabric types are proposed.

Among NoC simulators, the Noxim simulator [5] has attracted interest. It is studied and examined in detail as part of this research. This mesh-based simulator is developed using the SystemC discrete event simulator [11]. Figure 2 depicts the model components needed for NoC. The processing element and switch are SystemC modules with send and receive methods. A reset signal and positive edge of clock are used by each method. The switch has a buffer monitor method to manage the information exchange with neighboring switches. All functionalities in Noxim occur at the positive edge of a clock signal and a reset signal for the start of simulation. A NoC model can be configured in terms of network size, switch channel depth, routing algorithm, traffic time, and packet injection distribution using a command line interface. The total number of received packets/flits, global average delay, throughput, min/max global delay, total energy, and per-communication delay/throughput/energy can be measured.

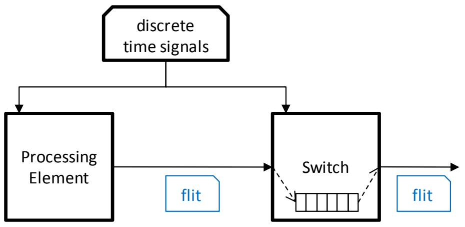

Figure 2: Model Components for Noxim

4 Model development

To model NoC, a bottom-up approach is chosen; i.e. we use the real physical NoC architecture (shown in Figure 1) and the coupled DEVS IO system level of abstractions [6]. Structural and behavioral abstractions representing the core functionality of the NoC are developed. The elements of the NoC for a segment of the NoC is shown in Figure 3 The operations for every model component can have their own independent timing as conceptualized with clocks. In the following, each of these models is detailed with emphasis on their dynamics with timing specification.

4.1 Processing element

As mentioned before, processing element (PE) is responsible for generating and consuming data. It is a simple atomic model which injects data to the network and receives data from its counterparts. It does not consider application-specific operations. Only the hardware aspect is taken into account since the goal is to accurately account for traffic generation mechanism. The work presented in [12] also uses the PE as traffic generator. In fact it collects execution traces of all PEs and then uses the results to model their behavior with a reduced set of instructions. The model has two basic phases (passive and busy). In the busy state the model generates and sends out data every *sendOutInterval.* Three patterns are used to generate *sendOutInterval* value. The time distribution of the generated data is Poisson, Gaussian, or Random. The size of the generated data can be set. The I/O specification of this model can be found in Section 5.1.

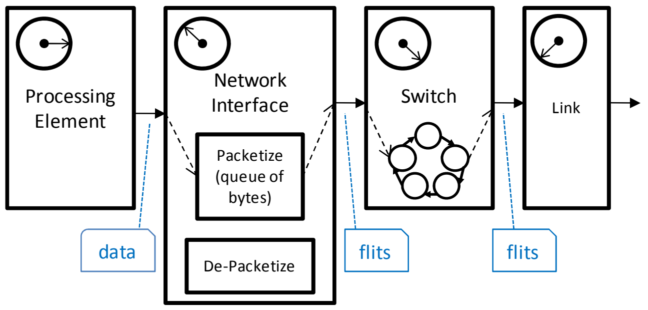

Figure 3: A conceptual snippet of a NoC model

4.2 Network interface

A Network Interface (NI) does both packetizing and de-packetizing. These two atomic models are defined as shown in Figure 4. NI receives data from PE and network via “fromPE” and “fromSW”, respectively. The dynamics of the packetize model is controlled through “statusFromSW” port; the de-packetize model is uncontrolled (i.e., data can be sent to PE without any limitation). The packetize model can send data only by receiving notification of free slots in NoC switch incoming buffer.

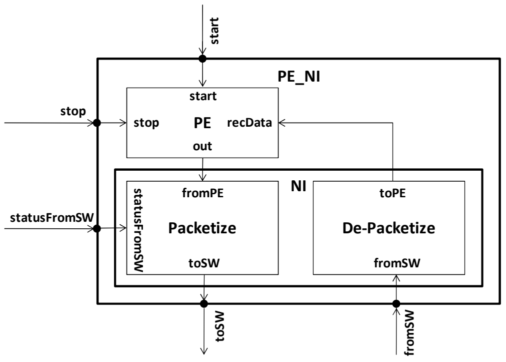

Figure 4: PE_NI and NI coupled model structures

4.2.1 Packetize

The basic functionality of this atomic model is to receive data from PE and convert the data to flits. The model states and transitions are shown in Figure 5. The model starts in "passive" phase. It can receive input via both input ports in this phase. While receiving data through "statusFromSW" port, the model sets the state variable "outQStatus" according to the obtained value which can be either "ok" or "nok". On the other hand, data received on port "fromPE", is split into 8-bits (1 byte) data and then inserted in a queue. Now if the number of elements in the queue is greater than or equal to the (user-defined) packet size in bytes (e.g. when the packet size is 64 bit the size of the queue must be greater than or equals to 8), the external transition changes the model phase to "packetize". Otherwise the queue size changes with no change in phase.

In the "packetize" phase, a packet is converted to flits. The flit is a complex data type containing the source PE id, the destination PE id, flit sequence number, and the flit type fields. As before, data can be received from both input ports. However, external events occurrence only changes the "outQStatus" and the queue size state variables and not the state itself. After *packetingTime*, an internal transition will occur. Depending on "outQStatus" value, two cases exist:

- "nok": the phase of the model changes to "wait4OK" and it remains there for infinity unless it receives an "ok" value on the "statusFromSW" port.
- "ok": the model's phase changes into "sendOutFlit".

During the *packetingTime* the model splits a packet into flits, which means filling an array of flits by the packet information. The type of the first, the last and other elements of the array is set to head, tail and normal respectively. The source id of the flits is set according to the corresponding PE id which is defined in the model. The destination id of the packet is set randomly, which means the traffic spatial distribution is random. And finally the sequence number of the flits is set using a counter which counts the number of packets sent out till now.

In the "sendOutFlit" phase, the model reacts to the reception of data on the "fromPE" port. While receiving data on "statusFromSW" port, depending on its content the model's phase may remain unchanged or changed to "wait4OK". After *senidngTime* units of time four internal transitions are possible:

- Number of sent flits is smaller than the maximum number of flits within a packet and also the "outQStatus" is "ok". the model remains in the "sendOutFlit" phase.
- The same as the previous scenario except for the "outQStatus" is "nok". The model enters the "wait4OK" phase and until the phase becomes "outQStatus".
- Number of the sent flits is greater than the maximum number of flits in a packet and the size of the queue holding the incoming data is greater than the size of a packet. In this case the phase changes to "packetize".
- The same as the previous case however the size of the queue is smaller than the size of a packet in bytes. So the phase of the model will change to "passive".

If the model receives data on the "statusFromSW" port while it is in "wait4OK" and the content is "ok", it will go to the "sendOutFlit" phase and starts sending out flits. Since flits are stored in an array in the "packetize" phase the order of flits and consequently the packets are preserved. The model generates flits at the end of the "sendOutFlit" phase.

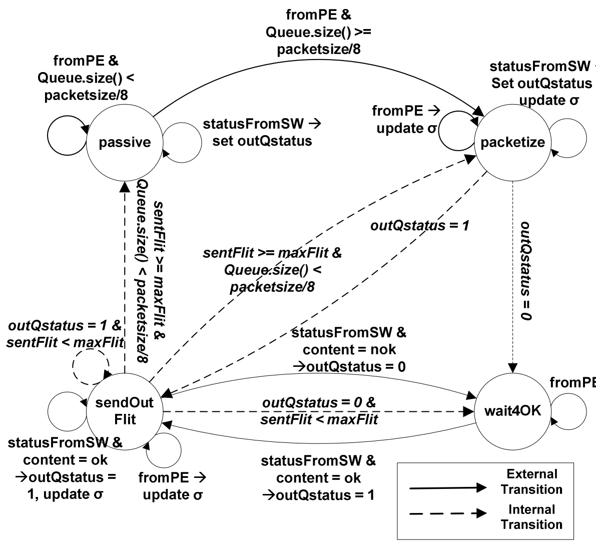
fromPE & Queue.size() >= packetsize/8
fromPE & Queue.size() < packetsize/8
statusFromSW → Set outQstatus & update σ
fromPE → update σ
passive
statusFromSW → set outQstatus
packetize
sentFlit >= maxFlit & Queue.size() < packetsize/8
outQstatus = 1
sentFlit >= maxFlit & Queue.size() < packetsize/8
outQstatus = 0
outQstatus = 1 & sentFlit < maxFlit
statusFromSW & content = nok →outQstatus = 0
fromPE
sendOut Flit
outQstatus = 0 & sentFlit < maxFlit
wait4OK
statusFromSW & content = ok →outQstatus = 1, update σ
fromPE → update σ
statusFromSW & content = ok →outQstatus = 1
External Transition
Internal Transition

Figure 5: Packetize atomic model

4.2.2 De-Packetize

This atomic model receives data from switch, collects them in packets and partitions them to data size units before sending them to a designated PE. We assumed that the PE only receives data without processing them. Accordingly the PE doesn't need input queues and there is no need for interrupt. The model states and details of the transition, output, and time advance functions of this are similar to the Packetize atomic model are excluded due to lack of space.

4.3 Switch

All communications in NoC are handled with this model. It receives incoming flits which are then routed according to a predefined network topology and a routing strategy. In order to avoid losing incoming flits, each input port has a dedicated queue. The flits stored in these queues are checked based on a round robin scheduling policy; i.e.

starting from an initial input port, each input port is checked (this is conceptualized in Figure 6). When there exists un-routed flits, a complete set of flits starting from head to tail flit is chosen. As the tail flit is routed (sent), the queue of next input port is checked for un-routed flits, and continues in the same way. After reading the first flit (the head flit), the ID of destination processing element is obtained from flit. The ID is used as the input for the routing algorithm which determines to which destination port this set of flits is to be sent. Finally, the head flit, and its consequent normal flits are sent to the determined destination port. This process terminates, when the tail flit is sent.

Since the available buffer for each input queue is limited, before sending a flit to the next switch, the sender switch should ensure that the next switch has enough capacity at least for one flit. Therefore, each switch informs its neighboring switches, whenever one of its input queues are full. On the other hand, as one flit of the full queue is routed, it again informs its neighbors about the freed buffer. This scheme guarantees that no flits will be lost. Deadlock between switches can occur the buffer size is small or the network is highly congested. It can be avoided using the Chandy/Misra/Bryant algorithm [13].

The NoC switch atomic model has two sets of input and output ports ("IN$_i$" and "OUT$_i$" given the *n* neighboring switches defined as $0 \leq i < n$). The "extST$_i$" and "intST$_i$" input and output ports are responsible for getting available buffer information of neighboring switches and announcing the available buffer information of the input queues.

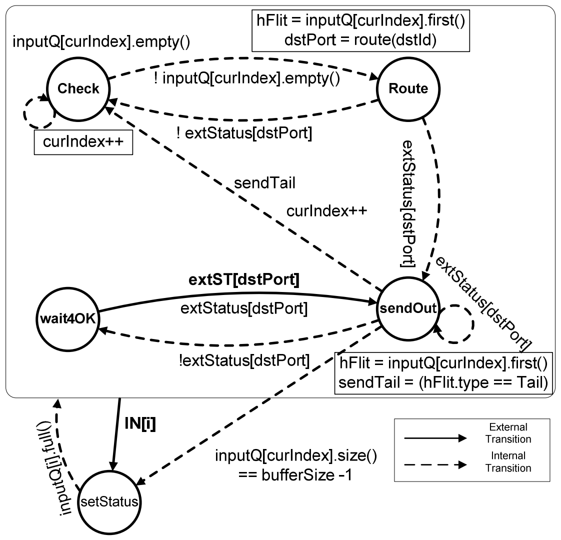

Figure 6: NoC switch atomic model

The phases and transitions for this atomic model are shown in Figure 6. The model starts in the "check" phase in which each input queue is checked for incoming flits periodically (through internal transition with period of *checkTime* to "check" phase). The "currentIndex" holds the index of currently examined input queue and "inputQ$_i$" for *i* in [0, "portNum"-1] represents the input queue associated to each input port ("IN$_i$"). If "inputQ$_{\text{currentIndex}}$" is not empty, the phase of the switch changes to "route" phase using its internal transition at the end of *routeTime* time.

In the "route" phase, the first flit of the "inputQ$_{\text{currentIndex}}$" is read and "dstID" field is extracted. The routing algorithm is implemented as a "routeTable", which is an array of integers of size "totalID" (the total number of processing elements in the whole NoC). The "routeTable$_i$" is equal to the selected output port for the flits of packet when the destination processing element ID of packet is equal to *i*. When the index of destination port ("routeTable$_{\text{dstID}}$") stored in "dstPort", the phase is determined and the "extStatus$_{\text{dstPort}}$" is checked to determine whether it is possible to send flits through this port according to the received status from destination switch. If it is true, the model changes phase to "sendOut" after *outTime* time. Else, the next phase is set to "check", after *checkTime*, through an internal transition. Note that each neighboring switch announces its status (full or not full) using "ok"/"nok" through its "intST" output ports. These statuses are stored in "extStatus" variable, an array of Boolean of size "portNum". The "extStatus$_i$" of switch is true if the "inputQ" of "IN$_j$" port of the neighboring switch connected to the "OUT$_i$" port of this switch is not full and vice versa.

The model remains in "sendOut" phase, using internal transitions with the delay of *outTime* until the tail flit is read from the "inputQ$_{\text{currentIndex}}$" and is sent to the "dstPort", or the "extStatus$_{\text{dstPort}}$" becomes false (i.e., the receiving switch buffer becomes full). In the former and latter cases, the model changes phase using internal transitions to "check" and "wait4Ok" with the delay of *checkTime* and infinity, respectively. Note that the Boolean "sendTail" variable is set whenever the tail flit is reached to determine that the next phase is "check". This variable is added due to the fact that the tail flit is sent to "dstPort" during the output function at the end of internal transition, and therefore the flit is not available to check whether it is tail or not. In the transition from the "sendOut" phase to the "check" phase, the "currentIndex" variable is increased.

The "wait4OK" phase has no internal transition. This phase may change due to external transition function (i.e., an "ok" is received on the "extST$_{\text{dstPort}}$" input port).

Finally, the "setStatus" phase is responsible for setting the values of "intST" output ports if there is any change in the status of "inputQs". This phase disturbs all other phases when a status set is needed. I.e., the current phase, sigma, and the index of the changed status queue are stored and an internal transition to the "setStatus" phase occurs in zero time. After setting the new status, the model transits back to its previous phase and sigma. This phase change is necessary for producing status outputs on "intST" output ports in two cases: (i) when the "inputQ" becomes full after receiving a flit, and (ii) when a flit is en-queued from "inputQ" which had been full. In the first case, where the phase change occurs due to an external transition, may happen in each phase except for the "setStatus". The latter case occurs in the "sendOut" phase when a flit from a full "inputQ" is de-queued and sent to "dstPort".

In order to prevent loosing concurrent incoming, the external transition function has two steps. First, all incoming messages are saved in their corresponding state variables and then a phase change happens if needed. When receiving data on the "INi" input ports, a change in the internal status of "inputQ" state variable can occur. This is saved by changing the content of "intStatusConsistent" and "intStatus" state variables. The "intStatusConsistent" is an array of Boolean of size "portNum". The true value of the "$\text{intStatusConsistent}_i$" shows that the status of the "inputQ_i" is consistent with the announced status through the "intST_i" port and the "intStatus_i" variable, and vice versa. The "intStatus" variable is an array of Boolean of size "portNum". The true value of "$\text{intStatusConsistent}_i$" shows that the "$\text{inputQ}_i$" is not full, and vice versa. These inconsistencies are saved in the first step and are handled in the second step by changing phase to "setStatus".

Only two phases of "setStatus" and "sendOut" can produce output. The "setStatus" produces "ok" or "nok" message for the "$\text{intST}_{\text{curAddedQ}}$" output port while in the "sendOut" phase a flit is removed from the top of $\text{inputQ}_{\text{currentIndex}}$ and send on the "$\text{OUT}_{\text{dstPort}}$" output port.

4.4 Unidirectional Link

The remaining model needed is a unidirectional link. This component is responsible for transferring flits between switches. It represents the communication channel with delay in a real digital system. So it requires a period of time (as defined by user) for flits to be transmitted between switches. Two switches are connected via a pair of unidirectional links (a bidirectional coupled link model).

4.5 Data collector

This model collects and analyzes the data generated by PEs data and consumed by other components. It has input ports "toNI_i" and "fromNI_i" which gather data entering and leaving every NI. The data is used to track network load. The model also has two output ports: "stop" and "start". The signals generated on these ports control the operation of the network. The number of the generated packets is tracked and average global, minimum, maximum, and maximum global delays as well as global throughput are computed. The inputs of the model are inserted in an array of *PacketInfo* which is a complex data type. This type has the following fields: sequence number, source id, destination id, start time, and stop time. Start time holds the time a packet leaves its source PE and stop time holds the time a packet enters its destination PE. The remaining details are left out due to lack of space.

5 Experimentation setup and results

A 3×3 NoC model is developed and simulated in the DEVS-Suite simulator [7]. Figure 7 shows a segment of the NoC spanning from Data Collector to Switch 3. The coupling details between switches are shown in Figure 3. Experimental results are collected, evaluated, and compared to the results obtained for the NoC model developed in Noxim simulator.

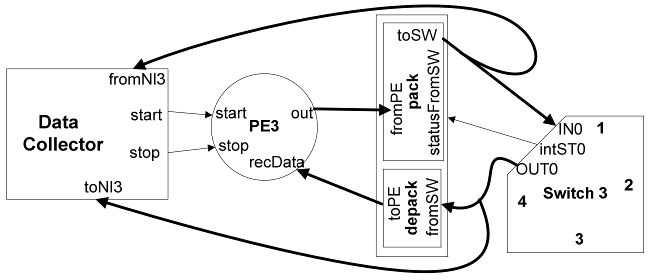

Figure 7: A NoC model module

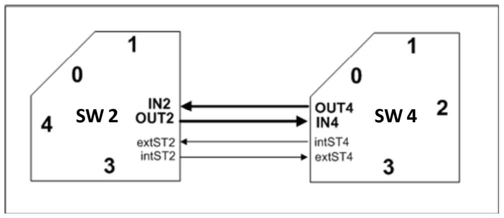

Figure 8: Bi-directional **c**ouplings between switches

5.1 Experimental setup

The execution process of NoC model can be divided into initialization, simulation, and result report parts. Since the simulation and report result are already described in Section 4, below a detailed description of the initialization part is provided. In the initialization part, the configuration parameters are read from an input file and the NoC model is constructed. The input file includes simulation setup, NoC size, routing table, and interconnection pattern; each starting by a special character. The order and syntax for determining each part is as follows:

Simulation setup: simulation timing and data size information are determined using

- simulation time (simTime): The time between start and stop commands,
- data size information, including size of packet (packSize), size of flit (flitSize), and the capacity of input queue buffers (inputQ) of switches,

- NI timing information, including packetizing time (pPackTime), depacketizing time (dPackTime), flit send time from de-packetizing model (dSendTime), flit send time in the packetizing model (pSendTime), and
- Switch timing parameters including the delay between checking two consequent input queues (sCheckTime), the switch routing table access time (sRouteTime), and send time for each flit (sSendTime).

NoC size: the input file determines the number of PEs (PENum) and switches (SWNum) in NoC model.

Routing table: the routing table for each switch is specified. It includes SWNum×PENum triples, each determining one entry of the routing table for each switch. The triples are of the form (Switch ID (curID), PE destination ID (dstID), Destination port (dstPort)). If a packet arrives in curID intending to go to dstID, it will go to the dstPort.

Interconnection pattern: this part determines the way switches are connect to one another or to NIs; there are SWNum entries each has a switch id (swID), swID input/output ports (pNum), and to which element (unconnected, PE through NI, or another switch) each of these ports are connected to. A 2×2 NoC model in DEVS-Suite is shown in Figure 9. The 3×3 NoC model has the same structure pattern as the 2×2 NoC model.

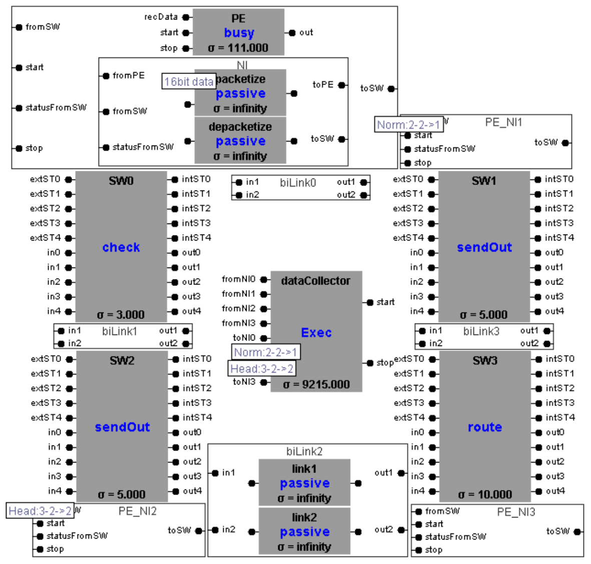

Figure 9: A 2×2 NoC model in DEVS-Suite Animation View

After reading all the parameters from the input file, the corresponding NoC structure is created by calling the constructors of NoC elements and performing the defined couplings. Thereafter, the simulation starts and data collector sends the start message to the processing elements. After a specified time period a stop message is sent to the processing elements. Finally, the simulation continues until all the sent packets are delivered to their destination (i.e., PEs) and the report of packet logging information and the overall NoC performance metrics are complete.

5.2 Results

The simulation experiment results are used as a means of validating the NoC-DEVS model abstractions and their dynamics. The intent has not been to show Noxim and NoC-DEVS produce results that are very close to each other. Instead, Noxim is used as a baseline for gaining confidence in the NoC-DEVS model specifications. Table 1 shows the parameters used for running the NoC-DEVS simulation. To the extent feasible, the same values are configured C++ in Noxim. Here, the comparative results of global average delay are reported (see Figure 10). Other data is excluded due to lack of space. By increasing the ratio of packet to flit size, congestion in NoC increases and therefore global average delay can also increase. Both simulations show the same type of behavior. The difference between the results can be attributed to several factors. Two basic factors are timing and different levels of model abstraction. NoC-DEVS defines more components to more closely represent NoCs (see Figures 2 and 3). In NoC-DEVS every transition function is defined in terms of time period it takes to execute some operation. Noxim uses simpler abstractions and acts as a functional simulator (uses execution cycle instead of a time variable). The implementation of NoC switch in Noxim uses reservation table for output ports; i.e. if an output port is not used for sending a packet and there is a packet to be sent through this port, the process of routing flits will be started even though another output port is busy sending another set of flits. This increases the global average delay in Noxim. The random distributions and the rate of feeding packets to the system are not the same for Noxim and NoC-DEVS.

Table 1: Configuration parameters of the model

Configuration Parameters	
sendOutInterval	random
packetingTime	10
sendingTime (packetize)	5
depacketingTime	12
sendingTime (depacketize)	5
route Time	10
check time	3
sendingTime (switch)	5
link delay	2
buffer size	1000
execution time	10000
Topology	mesh 3×3
Routing	X-Y

In addition to the trend of results, some other important modeling and simulation aspects should be considered when comparing the two NoC simulators. NoC-DEVS is designed using the formal modeling approach. Noxim is faster than NoC-DEVS. In the experiments conducted in this research, Noxim simulation outperforms NoC-DEVS by 1 to 2 orders of magnitude. This can be attributed to the C++ implementations of SystemC and use of simple model abstractions. NoC-DEVS simulation execution time can be reduced by flattening hierarchical models and/or use of distributed simulation platforms [6] [12]. Noxim lack of support for flexible modeling of time is important for detailed design tradeoff studies. NoC-DEVS provides multiple visualizations which can be useful throughout modeling and simulation lifecycle and also for educational purposes.

There are other differences between these two simulators. NoC-DEVS supports all NoC topologies while the Noxim supports only mesh-based topologies. Noxim provides different spatial distribution patterns while NoC-DEVS distributes packets randomly. NoC-DEVS supports static and distributed routing algorithms while Noxim has several conventional routing algorithms. Noxim supports total and local energy measurements. The NoC-DEVS model can be extended to support such capabilities. It is worth noting that DEVS supports cellular automata and variable structure modeling. DEVS models implements in Java and C++ can be executed with ease in parallel and distributed modes. It is also noted that NoC-DEVS models more closely represent NoC physical structure as compared with Petri-net approaches (see Section 2).

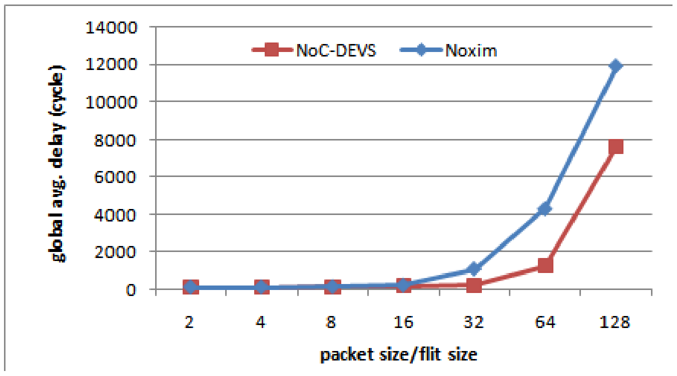

Figure 10: Global average delay

6 Conclusion

In this paper a discrete event model for Network-on-Chip is proposed and developed. The NoC-DEVS model is implemented in the DEVS-Suite simulator. The NoC-DEVS model is configurable and directly accounts for timing for the NoC Processing Element, Network Interface, Switch, and Link components. Some of the common metrics for evaluating digital systems are included in the model; the other metrics can also be incorporated in a straightforward manner. Important configuration parameters such as adaptive and localized routing algorithms and some essential performance metrics such as power and energy consumption are proposed for future work. With these capabilities, the simulator can support detailed, yet flexible NoC modeling and simulation.

References

[1] R. Tamhankar, S. Murali, S. Stergiou, A. Pullini, F. Angiolini, L. Benini, and G. D. Micheli, “Timing Error Tolerant Network-on-Chip Design Methodology,” *IEEE Transactions on Computer Aided Design*, vol. 26, no. 7, pp. 1297–1310, 2007.

[2] L. Benini and G. De Micheli, “Networks on Chips: A New SoC Paradigm,” *Computer*, vol. 35, no. 1, pp.70-78, 2002.

[3] H. Blume, T. von Sydow, D. Becker, and T. G. Noll, “Application of deterministic and stochastic Petri-Nets for performance modeling of NoC architectures,” *Journal of System Architecture*, vol 53, no. 8, pp. 466–476, 2007.

[4] H. Bazzaz, M. Sirjani, R. Khosravi, S. Taheri, “Modeling networking issues of network-on-chip: a coloured petri nets approach”, *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, March, 2009.

[5] Noxim – NoC Simulator, 2007, http://noxim.sourceforge.net/

[6] B. Zeigler, H. Praehofer, and T.G. Kim, *“Theory of Modeling and Simulation”*, 2nd Ed., Academic Press, New York, 2000.

[7] DEVS-Suite, 2009, “DEVS-Suite Simulator”, Ari*zona Center for Integrative M&*S, http://devs-suitesim.sourceforge.net/

[8] E. Salminen, A. Kulmala, and T. Hamalainen, "On network-on-chip comparison," *Euromicro DSD*, 2007, pp. 503–510.

[9] N. Enright-Jerger and L-S Peh, *“On-Chip Networks”*, Synthesis Lecture in Computer Architecture, Editor: Mark Hill, Morgan Claypool, 2009.

[10] F. Fummi, P. Gallo, S. Martini, G. Perbellini, M. Poncino, and F. Ricciato, “A timing-accurate modeling and simulation environment for networked embedded systems,” *Proceedings of the 42nd Design Automation Conference*, pp. 42–47, 2003.

[11] Open SystemC Initiative, SystemC Version 2.0, User’s Guide, www.systemc.org, 2001.

[12] S. Mahadevan et al. “A network traffic generator model for fast network on-chip simulation,” *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE’05)*, vol. II, pp. 780–785, March 2005.

[13] R. Fujimoto, *Parallel and Distributed Simulation Systems*, Wiley Interscience, 2000.

[14] O. Dalle, B. Zeigler, G. Wainer, “Extending DEVS to support multiple occurrence in component-based simulation,” *Proceedings of the 40th Winter Simulation Conference*, pp. 933-941, Dec., 2008.