Agent Based Modeling of a 3 Dimensional Network on Chip

Master of Science in Technology Thesis
University of Turku
Department of Information Technology
Embedded Computer Systems Laboratory
2014
Manjusha Kasu

Supervisors:
Ph.D. (Tech.) Tomi Westerlund
Adj.Prof. Juha Plosila

TURUN YLIOPISTO
Informaatioteknologian laitos

MANJUSHA KASU: Agent Based Modeling of a 3 Dimensional Network on Chip

Diplomityö, 60 s., 0 liites.
Tietokonejärjestelmät
Tammikuu 2014

Tässä tutkielmassa tarkastellaan verkkopiirin (Network on Chip, (NoC)) agenttipohjaisen mallin suunnittelemisen toteutettavuutta käyttäen moniagenttipohjaista MultiAgent Simulator of Neighborhoods (MASON) suunnittelu- ja simulointiympäristöä, joka on toteutettu Java-ohjelmointikielellä. Esitetyllä agenttipohjaisella NoC-mallilla on mahdollista monitoroida pakettien reititysprosessia verkossa.
Tämän tutkielman pääasiallisena kohteena on kolmiulotteisen agenttipohjaisen NoC-mallin suunnittelu ja toteutus. Käytimme MASON 3D -luokkia kolmiulotteisen agenttipohjaisen NoC-mallin luomiseen. Malliin valittu topologia on 4x4 3D-silmukkaverkkotopologia. Kommunikaatio NoC-mallin prosessointielementtien välillä tapahtuu XYZ-reititysalgoritmia käyttäen. Paketit, joita tässä työssä kutsutaan agenteiksi, kykenevät itsenäisesti liikkumaan verkossa noudattaen valittua reititysalgoritmia. Malli mahdollistaa pakettien dynaamisen seurannan NoC-mallissa. Tämän lisäksi toteutimme älykkäitä agentteja, jotka liikkuu verkossa satunnaisesti ja seuraa verkossa tapahtuvaa liikennettä hyödyntäen reitittimien keräämää tietoa.

Asiasanat: Agent, Agent-Based Modeling, Multi-Agent Simulator of Neighborhoods (MASON), Network on Chip (NoC)

UNIVERSITY OF TURKU
Department of Information Technology

MANJUSHA KASU: Agent Based Modeling of a 3 Dimensional Network on Chip

Master of Science in Technology Thesis, 60 p., 0 app. p.
Embedded Computer Systems Laboratory
January 2014

In this thesis, we examine the feasibility of designing a Network on Chip (NoC) agent based model using a multi agent simulation toolkit MASON (Multi-Agent Simulator of Neighborhoods). MASON is a Java based modeling and simulation toolkit for fast discrete-event multiagent simulations. The proposed NoC agent based model allows us to monitor routing process of packets in the network. Packets used in this model are agents that follows routing algorithm to navigate themselves from source to destination in the network.
The main focus of the thesis is on the design and implementation of a 3D NoC agent based model. We used MASON 3D classes to create 3D NoC agent based model. The topology chosen for the model is a 4x4 3D mesh network topology. The 3D NoC agent based model consists of the following elements: agents who follow XYZ-routing algorithm, intelligent agents who move randomly in the network observing its operation using the information gathered by network nodes. All the agents are autonomous and act according to specific rules. We can dynamically monitor the behavior of agents and the routing process of packets.

Keywords: Agent, Agent-Based Modeling, Multi-Agent Simulator of Neighborhoods (MASON), Network on Chip (NoC)

Contents

List of Figures

List of Acronyms

2D Two Dimensional

3D Three Dimensional

ABM Agent Based Modeling

ABMS Agent Based Modeling and Simulation

AI Artificial Intelligence

API Application Protocol Interface

ASIC Application Specific Integrated Circuit

GUI Graphical User Interface

MAS Multi Agent Simulation

MASON Multi-Agent Simulator of Neighborhood

NI Network Interface

NoC Network on Chip

PE Processing Element

SoC System on Chip

Chapter 1

Introduction

It is worthless to build a system without knowing how it reacts in complex situations. We should be able to simulate the functionality before building any complex system. In this thesis, we use agent based modeling technique in modeling complex systems, more precisely a Network on Chip (NoC) [4]. Agent Based Modeling and Simulation (ABMS) can be defined as a set of techniques, rules and tools for implementing computation models for complex adaptive systems [1]. It comprises of many interactive components and also agents which are capable of doing complex adaptive tasks. ABMS allows users to create complex systems using agents. Agent Based Modeling tools provide a good support when designing a system model by using Multi Agent Systems. ABMS has roots in the field of Multi Agent Simulation (MAS), robotics, Artificial Intelligence (AI), game theory, computational sociology and evolutionary programming.

The movie “World War Z” stands as a real life example for using Intelligent agents. The zombies in this movie are intelligent agents programmed with AI. The agents (zombies) can interact with each other and also with the environment (in this case 3D buildings and a helicopter) to perform some actions. The agents act with an objective and have a set of rules to achieve. These agents are designed and simulated using Alice.

In this thesis, we introduce the concept of using intelligent smart agents in modeling NoC architecture. We create a NoC agent based model which can run on MASON

multi agent simulation toolkit. NoC agent based model presented in this thesis, is a new approach for developing the routing algorithms and dynamically monitoring the routing process in graphical manner.

Initially, we create a 2D NoC agent based model and XY-routing algorithm [53] using GeoMason [50] classes. Having been succeeded in modeling a 2D NoC, we created a more complex architecture of a 3D NoC agent based model.

We design a 3D NoC agent based model and XYZ-routing algorithm using MASON3D class library. This model consist of self directive agent which can navigate along the network autonomously. There are agents which are developed to follow the XYZ-routing algorithm. We create a 2D and 3D mesh network environments with our complex logic, as there are no predefined supporting classes in the MASON.

We use MASON and GeoMason to create the agent based models. GeoMason is an extension library for MASON that allows us to create models by using geospatial data. MASON models can be checkpointed and migrated to other machines. A MASON model is duplicable, that is, simulation results on different machines will remain same when the simulation is carried out using the same parameters. Using Java language in modeling makes a model more flexible to run in heterogeneous computer environments.

This thesis is organized in such a way that first chapters introduce the basic theoretical concepts required to understand the actual thesis work. Next chapters discusses the MASON simulation toolkit, 2D and 3D NoC models, and finally conclusion and future work. Chapter-wise details are discussed below.

Chapter 2 describes the basic concepts of NoC. It gives a brief overview on types of existing network topologies, switching techniques and routing algorithms.

Chapter 3 discusses Multi agent systems, agent properties, agent based modeling basics and simulation environments. It also presents the existing categories of simulation environments and compares several existing multi agent simulators. It discusses about different multi agent simulation environments such as Behavior Based Simulators, Com-

piled Multiagent Simulators, Interpreted Multiagent Simulators. This chapter, justifies the idea of choosing MASON multi agent simulation toolkit for this model.

Chapter 4 introduces users to the MASON multi agent simulation toolkit. It has a brief description about the MASON architecture, model layer and visualization layer of the MASON multi agent simulation toolkit.

Chapter 5 presents the implementation of 2D NoC agent based model using GeoMason classes. This chapter contains the architecture of the 2D NoC model, agents used, routing algorithm implemented, schedule and accomplishments. It also consists of conclusion and evolution of 3D NoC model.

Chapter 6 is about 3D NoC agent based model. This chapter explains the Model architecture of 3D NoC agent based model, hierarchical levels, agents used in the model, implemented routing algorithm , class description and schedule.

Chapter 7 is the final chapter which has conclusion and future work of the thesis.

Chapter 2

Networks on Chip

Researchers have introduced Network on Chips (NoCs) in order to scale down the concepts of large scale networks to the level of System on Chips (SoCs). The communication paradigm that is used in NoCs is a packet based [2, 3], for example, 2D mesh NoC is shown in Fig. 2.1. This figure illustrates a NoC architecture with mesh topology, consisting of processing elements (*PEs*) connected together with the wires through routers. In this case, a *PE* (also called as *node*) can be a microprocessor, an application specific integrated circuit (ASIC) block or a memory, or a combination of components connected together [2]. A *network interface* (NI), present at each *PE*, is used to packetize the data generated by *PE*. NI is connected to a router which is responsible for routing the packets to specified destination address. There are various NoC architectures that have been proposed till date. NoC architectures can be differentiated from one another with their characteristics. Characteristics of NoCs are explained in detail in the following sections.

2.1 Network Topologies

In NoC, physical organization of a interconnection network is called network topology. The topology of NoC describes how switches, nodes and links are connected to one another in a network. The NoC topologies are of three types, direct, indirect and irregular

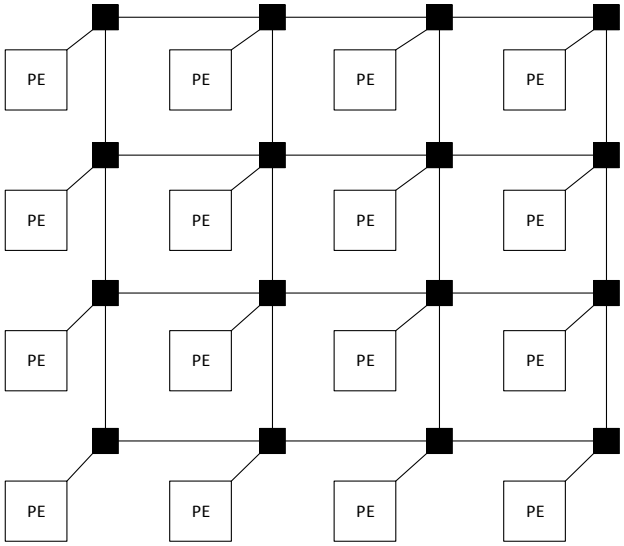
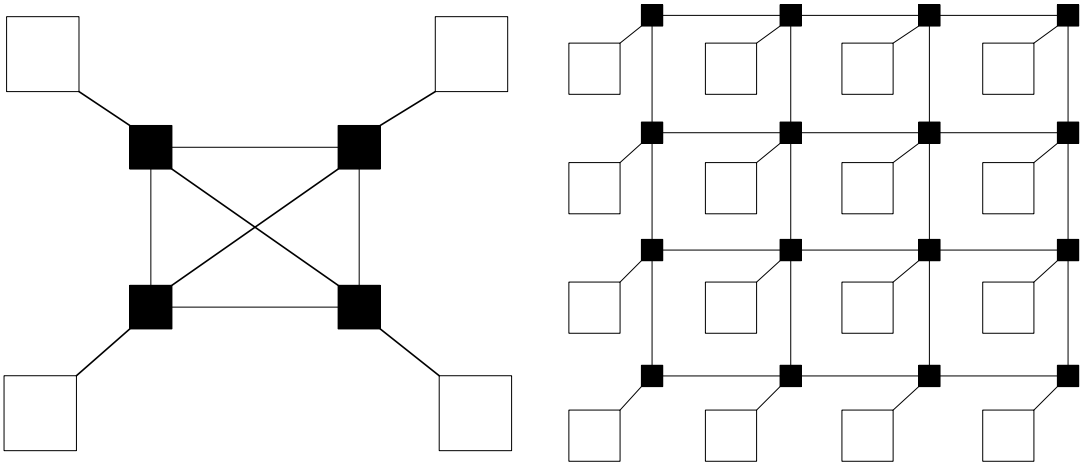


Figure 2.1: 2D mesh NoC [2].

networks.

2.1.1 Direct Networks

Direct network topologies are those in which each node in the network is connected to a subset of other nodes with a direct point-to-point link. Adjacent nodes in the network are termed as *neighboring* nodes. A Node can be a computational element, a memory, or a NI block which acts as router. A router is connected through links (called as channels) to the routers of the other neighboring nodes. In direct networks, when the number of nodes increases the communication bandwidth also increases. The main trade-off in the direct networks is between cost and connectivity. Performance increases with higher connectivity, but implementing router and link connection consumes more energy and results in higher area costs. It is better to avoid fully connected direct networks as shown in Fig. 2.2a, in which each node is connected directly to every other node. Hence, more common implementations of direct networks require messages to pass through several



(a) Point-to-point network topology.

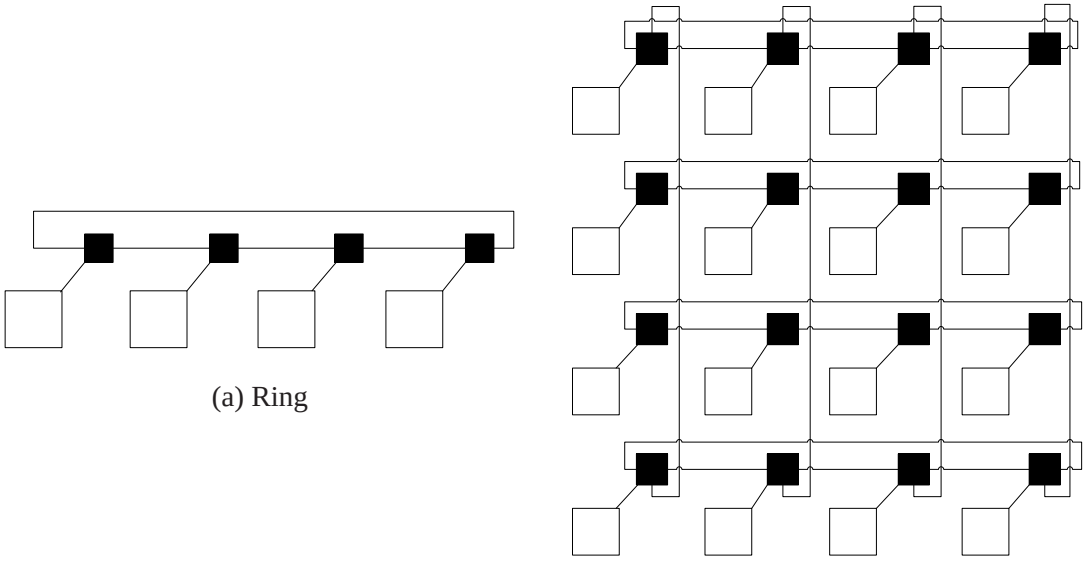
(b) mesh.

Figure 2.2: Network topologies: (a). point-to-point (b). mesh.

intermediate nodes before reaching the destination.

Many of the direct network topologies are orthogonally implemented, meaning that nodes in the network are arranged in an n-dimensional orthogonal space and the links to the nodes produce displacement in the single direction. Routing for these networks is simple and can be efficiently implemented in hardware. There are several types of orthogonal network topologies such as n-dimensional mesh, hypercube, torus, folded torus and octagon. Mesh topology is most popularly used in NoC architectures among the above mentioned topologies. The reason for this is that its physical design is easy as all the links in the network are of same length.

An example of a 2D mesh topology is shown in Fig. 2.2b. In this 2D network, every node is connected to four adjacent nodes other than those at the edges. We can easily add and remove the number of nodes in the network, which on the whole affects the networks area. The torus topology is also named as *k*-ary *n*-cube network which means a *n*-dimensional grid consisting of *k*-nodes in each dimension. A *k*-ary 1-cube (1D torus) is



(a) Ring

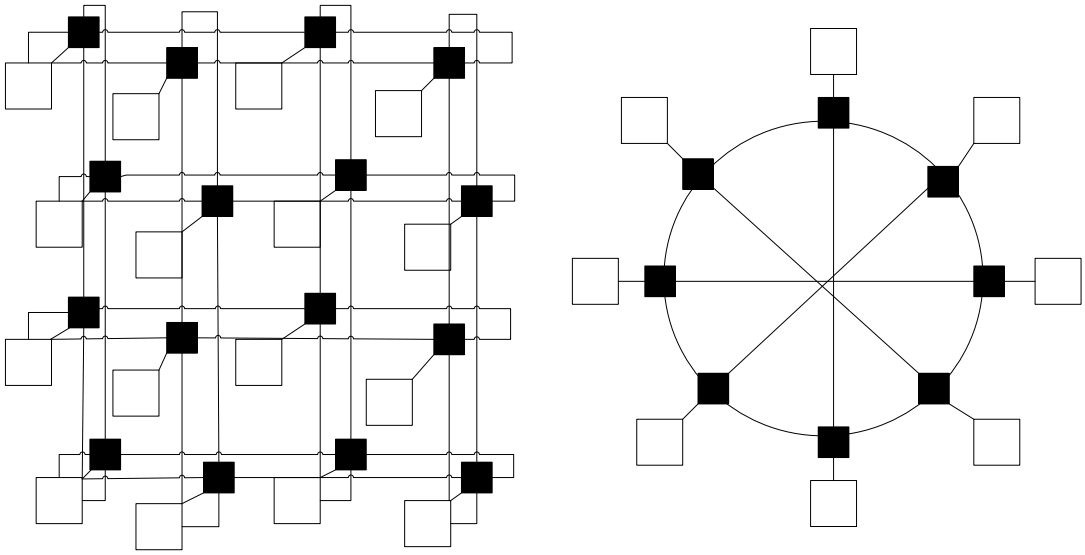
(b) Torus

Figure 2.3: Network topologies (a). Ring (b). Torus.

a ring network which has *k* nodes. An example of 4-ary 1-cube is shown in Fig. 2.3a. The main drawback of this network is its limited scalability. When the number of the nodes in the network increases, its performance decreases.

A 4-ary 2-cube (4x4 2D) torus is shown in Fig. 2.3b. This topology is similar to that of a 2D mesh topology except that the wrap-around channels are used to connect the nodes at the edges to the switches at the opposite edge. Each node in this network is connected to four nodes. The long connections in this network lead to delays in message transfer. This drawback is rectified by using a folding torus network topology which is shown in Fig. 2.4a. In this folded torus topology all the links have same size. A *k*-ary *n*-cube where $k = 2$ is an *n*-dimensional cube which is generally referred as a hypercube [2]. Performance can be increased, for example, by extending the torus or by adding bypass links. Adding bypass links will, however, result in higher area costs.

Finally, an octagon topology is another example of direct networks. It has 8 nodes



(a) Folded Torus

(b) Octagon

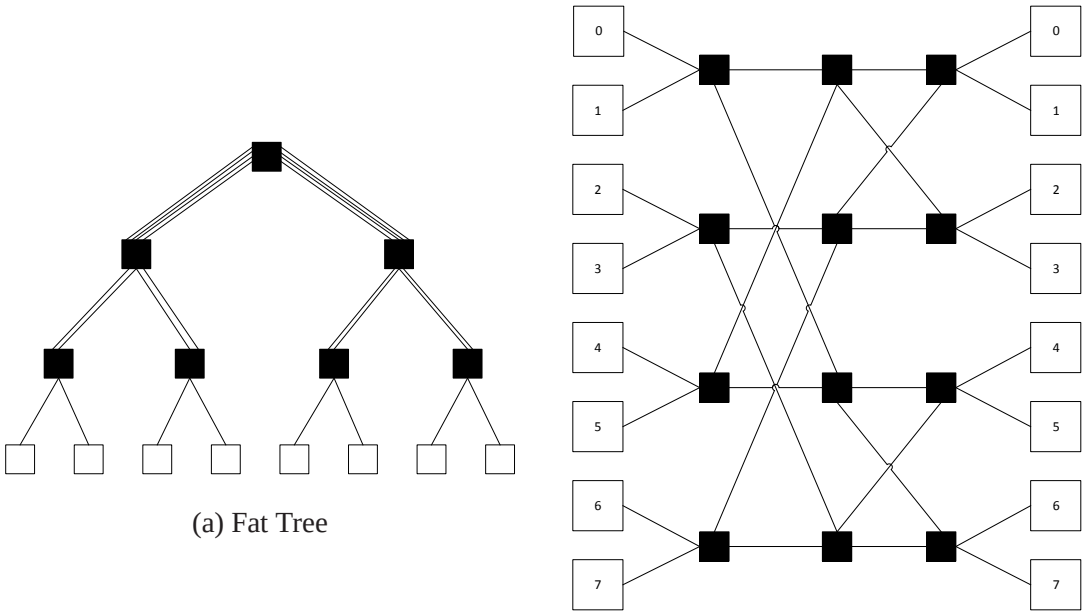
Figure 2.4: Network topologies (a). Folded Torus (b). Octagon.

and 12 links. Messages transferred between any two nodes need to hop at most two times. Nostrum [4], Proteo [5], SOCBUS [6] and Octagon [7] NoC architectures are examples of direct networks.

2.1.2 Indirect Networks

In indirect networks, the NI of each node is connected to an external switch and these switches are connected to other switches using point-to-point links. Crossbar can be stated as one of the simplest indirect networks. In crossbar, each PE is connected to other PEs by traversing a single switch. Researchers have proposed cost efficient partial crossbar networks, that are smaller and more energy efficient with respect to full crossbar networks [8]. The main drawbacks of a crossbar network is its scalability.

Some examples of multi-stage indirect networks are shown in Fig. 2.5. Figure 2.5a is an example of the fat-tree network topology [9, 10]. In a fat-tree topology, all nodes



(a) Fat Tree

(b) Butterfly

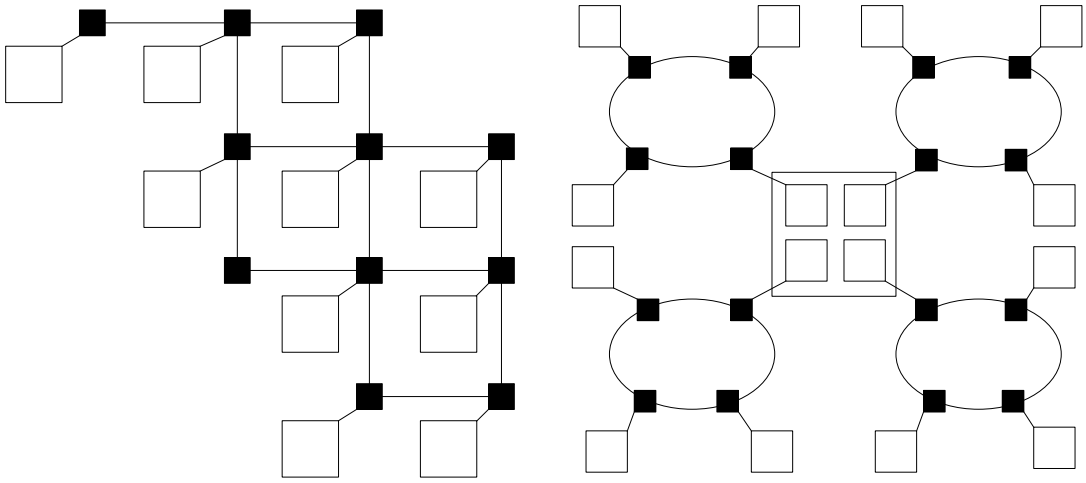
Figure 2.5: Network topologies (a). Fat Tree (b). Butterfly.

are only connected to the leaves of the tree. As the number of links at the root increases, more bandwidth is allocated on channels with high traffic.

Figure 2.5b illustrates a 2-ary, 3-fly butterfly topology. A butterfly network can be termed as a blocking multi-stage network. It means that when a contention occurs in the network the data may be temporarily dropped or blocked.

Another type of indirect network topology is Clos. It is an expensive non-blocking network topology because it consists of several full crossbar networks, but, due to its large bandwidth, it can support high performance.

Benes network is another indirect networks, which requires a controller to rearrange paths in order to establish a connection. Benes is an example of rearrangeable networks. An example of NoC architecture with indirect network topology is SPIN [11]



(a) Optimized Mesh (b) Hybrid

Figure 2.6: Network topologies (a). Optimized Mesh (b). Hybrid.

2.1.3 Irregular Networks

Irregular network topologies are also called as ad-hoc networks. These topologies are a mix of direct, indirect and shared bus topologies. The main objectives of this topology are: to reduce the distance between nodes when compared with that of direct and indirect network topologies, and to increase available bandwidth than that of a shared buses. Figure 2.6a gives an example of irregular network called optimized (also called as reduced) mesh network which eliminates all unnecessary links and routes.

There is another kind of irregular network which is named as cluster-based hybrid network. In cluster-based hybrid network, each cluster is a combination of any of the networks (direct, indirect or shared bus based network). Figure 2.6b illustrates an example of the cluster-based hybrid network topology which is a combination of ring and mesh network topologies. Examples of NoC architecture which follow irregular topologies are Æthereal [12] and Xpipes [13].

2.2 Switching Techniques

The NoC switching technique (strategy) regulates the data flow via routers in the network. Messages sent from PEs (nodes) are split into multiple data *packets*. A packet is again partitioned into several *flits* (flow control units). A flit is made of one or more *phits* (physical units). A particular NoC architecture has its own message, packet, flit and phit sizes. These affect some factors such as cost, power and performance of NoCs. There are mostly two ways to transport flits in an NoC, they are circuit switching and packet switching.

Circuit switching

In circuit switching, a path is reserved between source and destination before starting data transmission. Some routers and links together form a path in the network. When a path (circuit) is reserved, the sender sends a complete message to the receiver. A message header flit travels along the network from source to destination and reserves all the links on its way. If the header flit reaches its destination without encountering any conflict, it sends a positive acknowledgement to the sender. Sender starts data transfer only after receiving positive acknowledgement from header flit. If the path is reserved for any other transmission, header flit sends negative acknowledgement to a sender. The Path is reserved until the completion of data transfer. Circuit switching involves start up waiting time, and also results in low latency data transfer because the bandwidth of the full circuit is reserved for a particular data transfer. The two main virtual circuit switching schemes in use are: allocating one buffer per virtual link and allocating one buffer per link. MANGO [14] NoC architecture uses variant version of allocating one buffer per virtual link scheme. Nostrum and Æthereal are examples of NoC architectures that use allocating one buffer per link scheme.

Packet switching

In packet switching, packets are sent from sender to receiver without establishing any path. Packets make their own independent way from source to destination, during the retransmission. Packets follow different routes during the transmission and also have different delays. Packet switching does not have any start up waiting time. The main drawback in packet switching is that it has variable delays during transmission due to the traffic in the network. Several packets arrive at a router and attempts to use the link, only one packet can access the link each time and the other packets should wait for the link to become free. There are three packet switching schemes such as: store and forward, virtual cut through and wormhole switching.

2.3 Routing algorithms

Routing algorithms are necessary to route packets or circuits efficiently from source to destination. Routing schemes are categorized into several groups such as: static or dynamic routing, minimal or non-minimal routing and distributed or source routing.

Static and dynamic routing

NoC router can take decisions which are either static (also named as oblivious or deterministic) or dynamic (also named as adaptive). Static routing uses fixed routes in order to transfer data from source to destination. Static routing does not require any complex router logic, which makes its implementation easier. Routing decisions in static routing are made without considering the traffic at routers and links, and current state of network. Static routing allows the packets to be split among various paths between a source and destination, in a predefined manner. If static routing uses only one path to transfer packets, they are transferred in order. Since the packets reach a target in the order of transmission, there is no need to reorder them at destination, that is no bits are added to packets at NI in

order to identify the correct packet sequence. Examples of static routing algorithms are as follows: dimension order routing [15], XY [16], surrounding XY [17], pseudo-adaptive XY [16], turn model (negative-first, west-first, north-last) [18], ALOAS [19], valiant's random [15], topology adaptive [20], source [22], destination tag [15] and random walk routing [21].

Dynamic routing (also called as adaptive) will consider the current state of the network and it is also aware of the load on routers and links when taking routing decisions. Depending on the traffic conditions in the network, the path between a source and destination changes dynamically. In dynamic routing, there is a need for additional resources for monitoring the traffic in the network and to change the routing paths dynamically. However, dynamic routing distributes traffic in a network and make use of alternate paths, when certain paths are congested. Static routing is used when there are predictable traffic conditions in a network, whereas dynamic routing is preferred when traffic in the network is unpredictable and irregular. Examples of dynamic routing algorithms are as follows: minimal adaptive [15], fully adaptive [15], congestion look ahead [23], IVAL [18], odd-even [24], hot potato [26] and slack time aware routing [25].

Distributed and source routing

Further classification of static and dynamic routing schemes depends on factors such as: where the routing information is saved and where routing decisions are made. In distributed routing, router is responsible for making the routing decisions. In this routing, destination address is carried by the packet. Router either looks up for a destination address or executes a hardware function. Router implements a function whose input is packet's destination address and output is routing decision.

In source routing, each node's (PEs) NI has precomputed routing tables. When a data packet is sent by source node, the router (NI) at the source, reads packet destination address and looks up for the routing information in a table. This routing information is

attached to the packets header. When packet reaches any router, the routing information present in the routing header of packet is extracted by router. Source routing does not need any destination address in a packet and also table or functions are not needed to calculate route.

Minimal and non-minimal routing

Routing schemes are also classified as minimal and non-minimal distance routing. A routing is said to be minimal if the path between the source and destination nodes is of shortest possible length. In minimal routing, source does not send any packets if the shortest path is busy and not available.

Non-minimal routing does not follow any constraints, instead it uses longer path if the shorter path is not available. This results in many alternative paths, thus helping in avoiding congestion in the network. Drawback with this non-minimal routing is that it consumes more power.

Chapter 3

Agent Based Modeling and Simulation

Interdependencies in the systems are getting complex day by day, and therefore we need more precise modeling methodologies. Modeling complex systems can be done using agents. Modeling a system with agents is named as Agent Based Modeling (ABM). Using ABM, it is possible to simulate the functionality of large scale models involving autonomous agents, in order to evaluate the performance of the system as a whole. ABM also allows us to visualize the results of simulation in a graphical manner. These results are generally utilized to evaluate and draw conclusions from the behavior of the system, which in turn validates the ABM's specification [27].

Agent based approach has the following characteristics [28]:

1. It is capable of providing a natural environment for study of huge complex systems.
2. While developing geospatial models, agent based approach is more flexible when compared to other approaches.
3. Capable of capturing the transient behavior of system.

An agent based model comprises of three major elements. They are agents, relationships among agents and methods for their interaction, and the environment under which

agents operate [1]. Detailed explanation to these elements can be found in section 3.2. The task of the model developer is to analyze and decide on the number of needed agents, the required relationships between agents and the type of environment to create that particular agent based model. The developer should be able to model, identify and program the above elements. To run a model, there is a need for a computational engine in order to simulate the agent's behaviors and their respective relationships.

3.1 Multi Agent Systems

ABM has become a more powerful approach in many fields. It provides an efficient way to model complex systems using agents whose behavior can be different from one another. These kind of systems where there are multiple agents which interact and act autonomously are called Multi Agent Systems (MAS). MAS [29] is a collection of unrelated and dissimilar interacting agents. As it is not possible for a single agent to handle several complex tasks, there is a need for using multiple individual agents in a single system. These agents interact with each other in order to resolve the existing obstacles. The ability of an agent can be analyzed by the collaboration [30] it does with other agents to solve a particular problem when encountered. At the time of designing a multiagent system, we need to consider the behaviors of all the agents in the environment. Agents in the multiagent systems need not be application specific. They can also be generic agents with their own behaviors.

Before designing an agent based model, the model developer should create a design specification of what he/she needs to implement. This design specification should include the following information about the model.

1. How many agents should the model contain?

2. What are the responsibilities of each individual agent?

3. How they should interact with the environment and fellow agents?

4. What kind of data an agent needs and from where it should retrieve the data from?

5. What kind of environment that model should contain?

6. What kind of data an environment needs to have about an agent?

7. What are the other individual components this model should contain?

This is the essential information which we should know before designing a model. Once the specification is created, environment, individual components and agents should be created according to the specification. Once the agent based model is created, it should be simulated using agent based modeling toolkits which are described in more detail in Section 3.3.

Agent Based Modeling and Simulation (ABMS) has roots in the field of MAS, robotics, Artificial Intelligence (AI), game theory, computational sociology and evolutionary programming. ABMS can be defined as a set of techniques, rules and tools for implementing computation models for complex adaptive systems [1]. It comprises of many interactive components and also agents which are capable of doing complex adaptive tasks. ABM tools provide a good support when designing a system model by using MAS system. There is a need for Agent based modeling toolkits in order to run an agent based model. Running an agent based model means, to have all the components and simulating agents which execute their interactions and behaviors in the environment.

3.2 Agent

It is worthless to build a system without knowing how it reacts in complex situations. We should test the functionality before building any complex systems. Agents can be used to test the functionality of a system. An Agent is described as an independent component

which is capable of making intelligent decisions to complex situations [31]. Agents are usually more active rather than being passive [32]. Each agent has its own set of attributes and methods as illustrated in Fig.3.1. Method of an agent defines its behavior. Each agent has its own behavior. Agents attributes can be both static and dynamic. Static attributes cannot be changed, whereas dynamic attributes can be changed during the simulation. An agent can communicate with another agent and to its environment. Agents are capable of learning from the environment. Some of the features of an agent are discussed below:

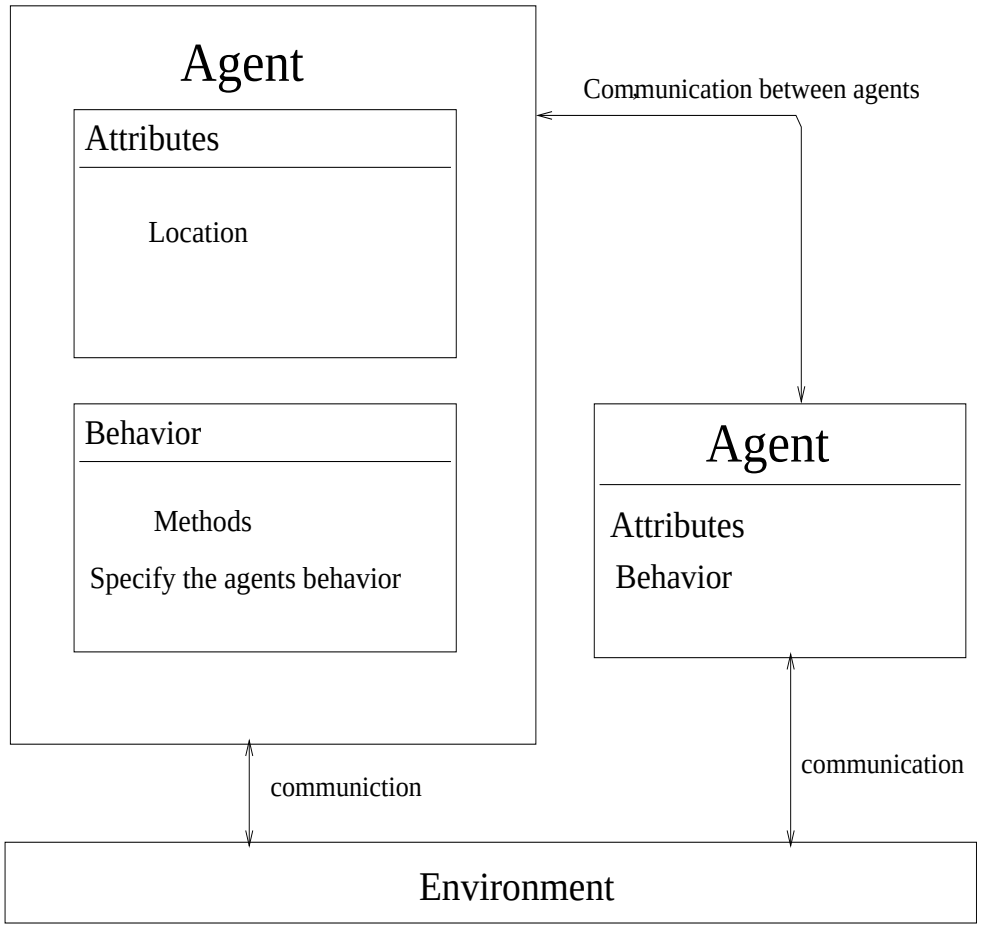


Figure 3.1: Agents interaction with environment.

Autonomous

While simulating a model, agents can act according to their own decision without any external interference. Agents act autonomously and perform spontaneous actions according to the changing situations.

Goal Oriented

An agent can be goal oriented. It can be designed to follow a specific algorithm in order to accomplish certain goals.

Reusable

Agents are designed to be reusable. An agent once created can also be used in any other environment.

Interactive

Agents can communicate with other agents and also can interact with the environment. Environment is the medium where agents and other components of the model are present. By exchanging data with environment an agent can know the state of other components existing in the same environment. There is a possibility to have several agents in an environment, in this case each agent has its own behavior. The environment in which the agents act may not be reliable.

For instance, consider an environment which has some paths, intersections and an agent that moves along the paths. Here an agent should have the knowledge of paths and intersections and should be able to navigate among them. This can only be possible if the agent can access the required information from the environment. The agent should be capable of taking independent decision to know whether it is in the correct path or not. When the agent is at intersection of two paths the agent should choose one of the

paths based on the predefined instructions. We design the agent in such a way that it will be able to take the decision and act accordingly in all the cases. If we design the agent to take a random path at intersections, then the agent will do so whenever it encounters an intersection. It is also possible to design an agent with complex behavior which can handle complicated tasks. This is how the agents fulfill the interactive property.

Self Directive

In some situations there is a need for two agents to communicate with each other. For example, if there is a moving and stable agent in the environment, the moving agent should detect the stable agent when they are close to each other, and vice versa. Sometimes, the agents may need one another's coordinates in order to identify each other. This task will be accomplished only if both the agents can communicate with each other and share data among them such as coordinates and state of the agent. An agent can be self directive, as in this case the moving agent directs itself based on the information it retrieves from the environment.

Apart from the above mentioned properties, there are few more characteristics of agents which are stated below:

1. Agents are capable of interacting and influencing other agents while having independent behavior.

2. Agents are adaptive to their environments, which means that agents must be designed to function properly even in unpredictable environments.

3. Agents should be trained to accomplish a particular task regardless of the given environment. If we train agents in this way, it will be possible to reuse the same agent in different environments for similar functionality.

4. Agents can be heterogeneous; meaning that each agent can have its own property

and behavior and therefore we cannot expect two communicating agents to be behaving in the same way.

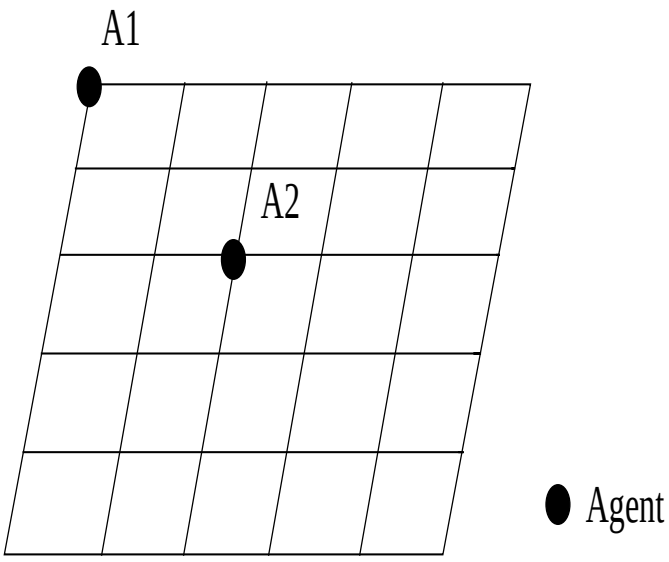


Figure 3.2: Agents and environment.

Figure.3.2 illustrates the grid environment. Where A1 and A2 are moving and stable agents, respectively. A1 and A2 can interact with the grid directly. Both agents have different methods which specify their behaviors. A1 chooses a random path when it comes across an intersection, whereas A2 is stable at particular location. A1 moves along the grid and randomly changes its direction at each node, A2 should take a note whether or not A1 has passed by. In this particular case, A1 should know whether it has arrived at A2's location, which means that A1 should interact with environment and check whether A2 is present at current position or not. Here A2 should also have the information about A1 to determine whether A1 is passing through it or not. This can be achieved only if we design the agents in such a way that they interact with each other when necessary. The above example illustrates the adaptability property of agents. This model comes under multiagent systems because agents in this model have different properties and behaviors.

3.3 Simulation Environments

There are many simulation environments [33] which can be used for multiagent simulations. Simulation environments can also be called as simulators. We will specify them as simulators throughout this section for convenience. Simulators are categorized into three main types. They are behavior based simulators, compiled multiagent simulators and interpreted multiagent simulators.

3.3.1 Behavior Based Simulators

Teambots [34] simulator belongs to this category. It is a light weight Java based robotics simulation environment. Teambots provides graphical display, minimal robot and physical sensor facilities and a simple schedule procedure. It is mostly used for behavior based robotic applications. This simulator has the distinction between the agents which drive the objects (robotic software) and objects in the world (robots). Teambots can port the robotic agent behaviors to the real robots with the help of real-robot application protocol interfaces (APIs).

In research paper [35], it was presented that implementing non-robotic multiagent simulations with an existing Teambots simulator involved modifying and adding significant amount of new functionality. This has added serious overhead to their codebase which slowed their experiment and has introduced many bugs. Owing to this limitation, it cannot be used for all kinds of applications.

3.3.2 Compiled Multiagent Simulators

There are several Compiled multiagent simulators, for example, SWARM [36], RePast [37, 38], Ascape [39, 40] and Multi-Agent Simulator of Neighborhoods (MASON) [41, 35, 42].

The earlier version's of SWARM applications have been written in Objective-C and

Tcl/Tk in order to schedule. It can now run the applications written in Java which uses special libraries to communicate with Objective-C. The usage of unusual language in the simulator has become a challenging endeavor for the developers to further extend and maintain SWARM.

After analyzing the complications in SWARM simulator, RePast has envisioned to re-implement most of SWARM's ideas either in Java or .NET. In recent years, it has become a greatly used multiagent simulator. RePast distribution is now widely used for modeling applications such as neural networks, GIS, charts, graphs and social network modeling.

On the other hand, an Ascape multiagent simulator is a rule oriented framework written entirely in Java. Agents in these applications have specific rule based behaviors and they are scheduled according to certain environmental conditions. These agents are grouped in huge structures which are then scheduled in an order specified by user (and/or scheduler). This framework is efficient and only suitable for model development in few cases. It also imposes constraints on the whole schedule, particularity in cases where there is a need for arbitrary agent handling.

MASON simulator provides graphical visualization, event monitoring, generation of various forms of media and event ordering. These functionalities can also be provided by the above discussed three multiagent simulators. Apart from these similarities, MASON also have some major differences: MASON provides 3D fields, both 2D and 3D field visualizations in 3D and more efficient and flexible 2D visualization. MASON is faster in visualizing models when compared to other simulator toolkits. MASON is also capable of providing duplicable results when necessary, where as most of the simulators are not. MASON models can be separated from visualization dynamically. The above mentioned frameworks run "headless" (i.e. a model can run without visualization) and it is not possible to migrate a headless process to another machine during mid-run. Visualization is much less in headless processes. Generally, many simulators are designed for creating single-shot models. That is, in these kind of models, a developer will construct and run a

model once and then analyze the results. In MASON, it is possible to run the models as many times as possible without any complications. The models can also be check-pointed and migrated to other machines, and it is possible to get the duplicable results.

3.3.3 Interpreted Multiagent Simulators

In Interpreted Multiagent Simulators, a user can manipulate the existing world by using interpreted programming language such as Logo. Instead of making minor changes to the model before compile time, a user can directly modify the code at runtime to examine the effects. This eliminates several compile and run cycles for the entire model. This is design strategy behind StarLogo [43, 44] and NetLogo [45]. These simulators have same basic functionality like in SWARM. Unlike in SWARM, the language used by developers to implement the model in these simulators is a modified version of Logo. Breve [46, 47] is also a similar simulator which can handle 2D and 3D worlds with which the user can manipulate control objects with the help of ODE physics engine and a proprietary language named as Steve. These simulators have many benefits such as instant feedback on change of code and encourages mid-run. The benefits of Breve are: it wraps the powerful physics environment with easy to learn library. According to the theory, it is observed that interpreted language model can dynamically add and remove visualization and can be ported easily in mid run between any two platforms. The main disadvantage of these simulators is that it can be slower for complex models. The schedules built with these simulators are mainly constrained by their respective graphical interfaces. Rapidly building models using this language make them an attractive proposition for model developers. But, at the same time this would lead to long runtimes for complex simulations.

Considering all the above simulators and their pros and cons, we have come to the conclusion that we will use MASON (agent based modeling toolkit) for this thesis to model and simulate a Network-on-chip. In Chap. 4, we will give an overview on MASON and its architectural design.

Chapter 4

Multi-Agent Simulator of Neighborhoods

MASON [41, 35, 42] is an open source, domain independent and multiagent simulator which supports various agents in a single system. It is an easily extensible, fast, flexible and discrete-event simulator [35]. Simulation core and visualization libraries of MASON are completely written in Java. Unlike in other simulators any experienced Java user can easily add or modify some features which are not domain-specific. Most of the simulation toolkits are designed for small tasks. The main advantage in choosing MASON is that it is meant for huge and complex multiagent simulation tasks involving several simulation runs. It has a good simulation speed and also documentation is sufficient enough when compared to the other toolkits. There is a vast amount of research on multiagent simulations as they are currently used in applications such as robotics, security and defense systems, and unmanned aerial and underwater vehicles. Using MASON we can easily create multiagent simulation models and can run several of them in parallel on back-end machines. MASON's properties, architecture and architectural layers are discussed in the following sections.

4.1 Architecture

MASON is comprised of two parts, model and visualization [42]. Both of them can be in either 2D or 3D. Model and visualization are independent until and unless we choose to have model objects display themselves. It is also possible to run a model without visualization. It can support up to a million agents when we are not using any visualization. Running a model with or without visualization can be controlled by the designer. MASON models can be entirely serializable to disk. This means that the files can be checkpointed and can be resumed later. MASON models are completely encapsulated; We can run two models independently in the same process without interrupting each other. In order to overcome Java's reputation of being slow, some Sun (subsidiary of Oracle corporation) classes are replaced by MASON equivalents which perform more accurately. One such class is MASON's random number generator which is supposedly high in quality when compared to the one in Java. Usage of Java language in modeling makes the model more flexible to run in heterogeneous computer environments. MASON models are also duplicable, that is, simulation results on different machines will remain the same when the simulation is carried out using the same parameters.

When choosing MASON as a simulation toolkit for creating models, we need to consider the following aspects :

1. MASON is not intended to achieve parallelization of an individual simulation. It cannot be used when there is a need to distribute a single simulation among multiple processors.

2. MASON core is meant to be simple, and hence it does not provide any built-in features specific to robotics simulators or other social agents.

3. MASON toolkit is not designed for creating memory efficient models. It can only be moderately memory efficient.

Apart from the above mentioned limitations, MASON is good in certain aspects such as model detachment from visualization, check-pointing, speed, portability and strong visualizations. These features are commonly considered as assets among the simulation community.

Java language is chosen to build MASON toolkit in order to take the advantage of its strict math and type definitions, portability and object serialization. These properties of Java will help MASON models to achieve duplicability and also to checkpoint the simulations.

MASON toolkit has three layers: utility layer, the model layer and the visualization layer. Firstly, the utility layer has classes that are used for many purposes. These classes include data structures, a random-number generator, which are more powerful than those available in Java distribution, movie and snapshot generating facilities and various Graphical User Interface (GUI) widgets. Secondly, the model layer consists of classes such as schedule utilities, a discrete event schedule, and a collection of fields that include objects and combine them with locations. This code alone will allow the developer to create basic simulations which can be run from the command line. Finally, the visualization layer enables GUI based visualization and manipulation of the created model. Model and visualization layers are further explained in Section 4.2 and Section 4.3 respectively. The pictorial representation of these two layers and their classes are shown in Fig.4.1.

4.2 Model Layer

Model layer in MASON does not have any dependencies on Visualization layer and can be easily detached from it. MASON's model is enclosed within an instance of user-defined subclass of a class called `SimState`. This instance consists of a `MersenneTwister` random number generator, a discrete event `Schedule` and zero or more fields [35].

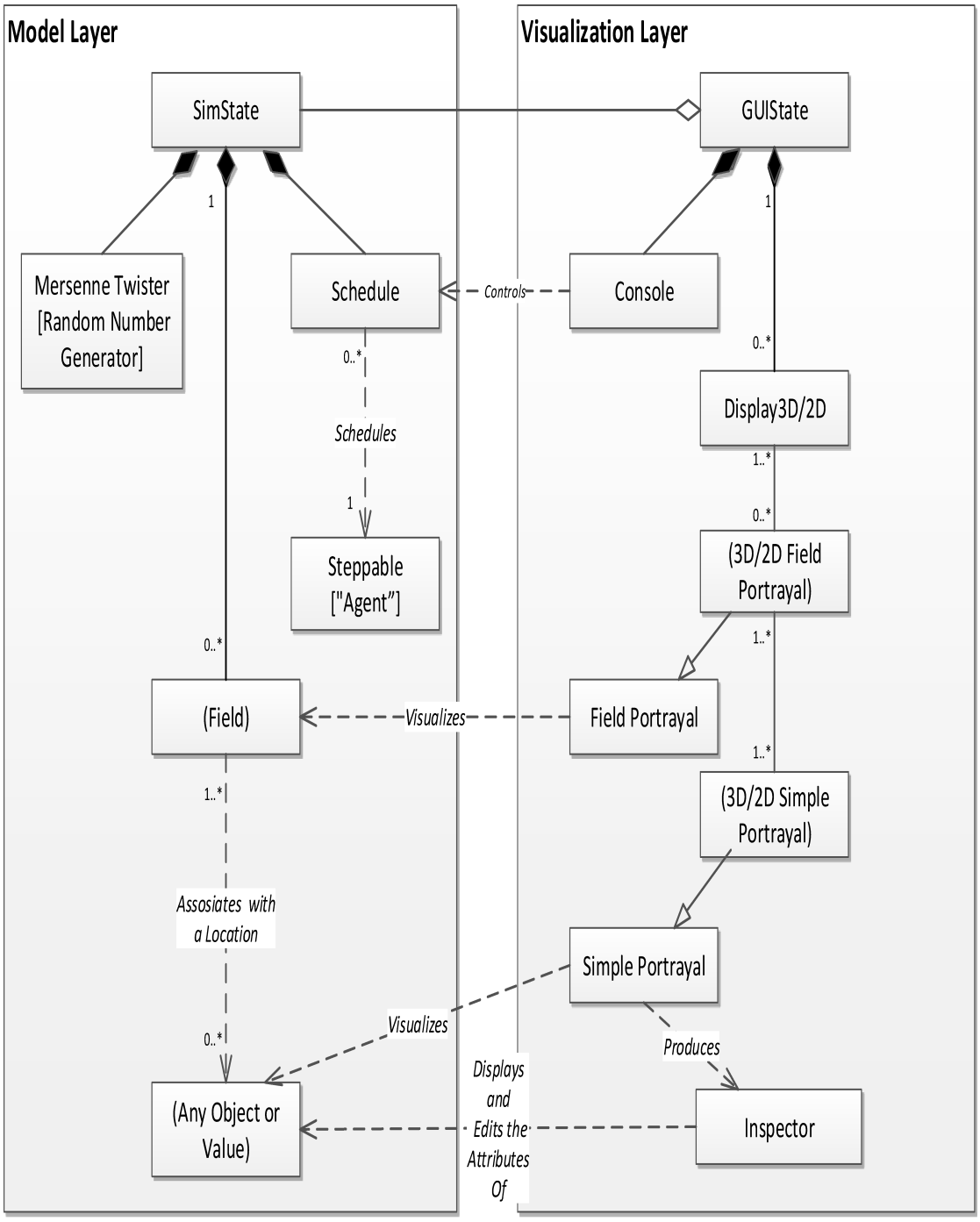


Figure 4.1: Simplified UML diagram which consists of basic classes present in model and visualization layers. Labels in parenthesis indicate the sets from which several classes are available [35].

Agents and the Schedule

In the context of MASON, agents are computational units that can be scheduled to implement a certain action and to manipulate the environment. MASON directly schedules an agent to be *stepped* (or called) at some particular time rather than scheduling other events to be sent to an agent. Agents created in MASON implements the `Steppable` interface as illustrated in Fig.4.1. Anonymous wrapper class allows agents to schedule several times to accomplish different functions. MASON is capable of scheduling steppable objects at any point of time in the near future. Additionally, a schedule can be split into various *orderings* which subdivide the given time step: at any given time the scheduled agents which are in earlier ordering will be stepped prior to those of scheduled agents in the later ordering. MASON has several `Steppable` wrappers of which some can group agents together, perform them in parallel on separate threads and iterate them. Agents can be scheduled to run asyncronously with the schedule in their own thread. This thread can loop indefinitely, run continously until completion or run till the `Schedule` arrives at the next time step.

Fields

MASON's fields associate arbitrary values or objects to locations in some abstract space. Some of these fields are a bit more than wrappers for simple 2D/3D arrays, whereas other fields cater to sparse relationships. An object can be present in different fields at the same time and the same object may also exist multiple times in some fields. The usage of these fields in an application can be optional. There is a possibility for a user to design and add his or her own fields. MASON is providing fields for the following [35]:

1. The edges of Networks (graphs) which may be either directed or undirected and optionally weighted or labeled.

2. 2D and 3D bounded arrays of integers, objects or doubles, which can be toroidal

or bounded as per user requirements, and can also be with triangular, hexagonal or square layouts.

3. 2D and 3D sparsely populated object grids. These grids can be toroidal, bounded or unbounded. These may also be with triangular, hexagonal or square layouts.

4. 2D and 3D sparse continuous space, which can be toroidal, bounded or unbounded.

When a model is run without any visualization, MASON has a primary top level simulation loop. MASON initiates either by creating a entirely new `SimState` or by loading it from Java serialized checkpoint file. MASON then starts the successive loop. Firstly, it checks for the remaining agents in the `Schedule` to step. If there are no agents, or if the maximum time limit in the `Schedule` has been exceeded, MASON quits the loop, completes the `SimState` and halts. Secondly, the `Schedule` increases its time to be equivalent to the minimum time step for the agents to `Schedule` and then steps all agents whose `Schedule` starts at that time step. If checkpoint of the model is needed, that can be done at this point of time. Check-pointing can be done in three steps as stated below.

1. A request is sent to all asynchronous agents to pause their threads,

2. Entire model checkpoint is written out.

3. All threads related to asynchronous agents are resumed, and the loop proceeds further.

Agents can access the `SimState` to full extent and can manipulate its `Schedule`, random number generator, and fields. MASON has few restrictions on the actions of the agents and it does not provide any simple protocols for designing an agent.

4.3 Visualization Layer

`GUIState` is wrapped around `SimState` and acts as a gatekeeper. Visualization layer objects can examine the objects of Model layer only after the approval from `GUIState`. When running a model with GUI, `GUIState` class is responsible for attaching or detaching the visualization with `SimState` and also to checkpoint the `SimState` to or from disk. As some of the objects in the visualization group requires to be scheduled (particularly, windows must be refreshed to take the effect of new changes in the model), these elements can "schedule" by themselves along with the `GUIState` to be restored when the fundamental `Schedule` is pulsed but not scheduled on the actual `Schedule` itself. All of this supports the visualization layer being able to separate from the actual model.

Besides the `SimState`, `GUIState` can also manage zero or more displays, GUI windows which are capable of providing both 2D and 3D views of primary fields. Displays act upon by taking hold of zero or more *field portrayals*, correlating each one with a distinct field in a model. Every field portrayal is bound to draw a field on the screen and to act in response to user requests to inspect the fields features. *Field portrayals* accomplish by correlating *simple portrayals* with specific objects or values which are stored in the fields. Simple portrayals may also, on demand, call inspectors of fundamental objects. Inspectors are nothing but GUI panels which grant user to modify and inspect parameters of objects. User can create his/her own inspectors or can access the available basic ones.

The GUI window is a top level controller for the `GUIState`, which is commonly the given console. `Consoles` fundamental functionality is to authorize user to play, pause, stop , step the schedule.

Apart from the above mentioned basic functionality the console also provides the following GUI functionalities:

1. Allows user to save and load models which are checkpointed

2. Hides and shows the displays as per user requirements

3. Helps user in viewing the inspectors

4. Permits user to load additional simulations (each of which has its own individual `SimStates`, `GUIStates` and `Consoles`).

To run a model without visualization is less complex when compared to a model with visualization, not only because of including display but based on the following explanation. A fundamental model runs on its very own thread which is independent from the GUI's main thread and both threads need to access the model data. For this reason, GUI thread and the model thread enters into synchronization procedure which assures that only one of these threads have access model data at any given point of time. The whole process is further explained in next few sentences. When `GUIState` is built, it in turn constructs a `Console`, several displays, and the fundamental `SimState`. In the `Console` a user can initiate the simulation by pressing "play" which allows the `Console` to start the `SimState` and then creates the model thread. The model thread will then enter into the loop and then quits the loop either when the `Console` asks to shut down or if there are no more scheduled agents. In the loop which was mentioned in the above statement, the `GUIState` completes some pre-scheduled items, then the schedule progresses to the minimum time step required for the agents to start scheduling and then steps all the agents at that point of time. After this, post-scheduled items are completed and then the model thread suspends and give the GUI thread access to model. While model thread is in waiting state, the GUI thread completes redrawing the displays, checks for the users requests for model inspection if any, and checkponts the model. When a user press "Stop" from the `Console`, the thread is requested for a shut down and the `Console` completes the `SimState`. When user closes the `Console`, the `SimState`, displays and `GUIStates` are destroyed.

4.3.1 Dependencies of MASON

The following are required, in order to keep MASON simulation toolkit up and running: JDK (higher version than 1.3), JFreeChat, JCommon, JMF, iText and Java3D (to run 3D models).

Chapter 5

Basic 2D Network on Chip model

In this chapter, we present a basic 2D Network on Chip (NoC) agent based model using the classes in GeoMason [48, 49] Library. The 2D NoC model consists of a self routing agent moving along the 2D mesh network and following XY-routing algorithm. The architecture of this model is further explained in the following section.

5.1 Architecture

The 2D NoC agent based model is designed using GeoMason classes such as GeomVectorField [51] and GeomPlanarGraph [52]. This model consists of an Environment, a NodeElement and a Packet agent which are described below.

Environment

The Environment chosen for this 2D NoC agent based model is 4x4 2D mesh network. This environment is created by using a LineString class and a node at intersection of the each LineString. Environment consists of NodeElement and Packets.

NodeElement

The NodeElement is the place where the Packets are created and scheduled. A NodeEle-

ment is created and scheduled from a “Noc.java” class.

Packet

A Packet is a self routing agent. Packets are created at the NodeElement class. A Packet follows the XY-routing algorithm to travel from a source to a destination. XY- routing algorithm used in the model is explained below:

Algorithm 1 XY Routing Algorithm

```
1: if (Dx > Cx)
2:     Move the Packet to East;
3: else if (Dx < Cx) then
4:     Move the Packet West;
5: else if (Dx = Cx) then
6:     if (Dy > Cy)
7:         Move the Packet North;
8:     else if (Dy < Cy) then
9:         Move the Packet South;
10:    else
11:        Packet reached its destination address (Dx, Dy);
12:    end if
13: end if
```

Algorithm 1 is as follows: Current address of a Packet is (Cx, Cy) and destination address is (Dx, Dy). Firstly, the X-coordinate of the current address (Cx) is compared to the X-coordinate of the destination address (Dx). Packet moves towards East when $Dx > Cx$, towards West when $Dx < Cx$, and if $Dx = Cx$ it has already aligned in

the destination address's X-coordinate (Dx). Finally, Cy is compared with Dy. When $Dy > Cy$ the Packet moves towards North, when $Dy < Cy$ it moves towards South and when $Dy = Cy$ the packet stops moving further as it has reached its final destination. When all the packets have reached their destinations the schedule ends.

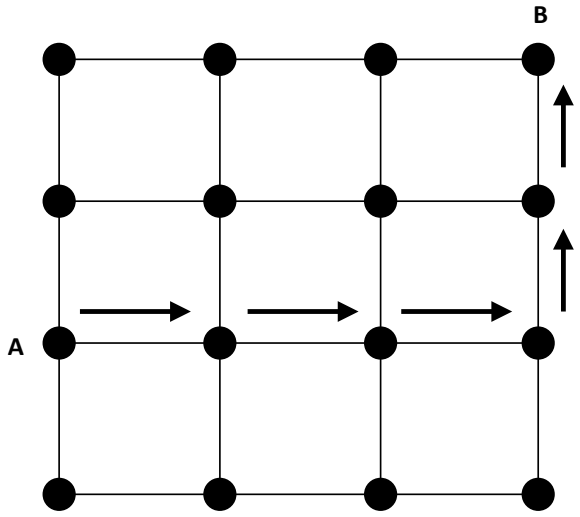


Figure 5.1: 2D Mesh Network with XY-routing

In Fig.5.1, we can see a 2D mesh Network with "A" as a current address of the Packet and "B" as a destination address. According to the XY-routing algorithm, A Packet from "A" will traverse through the marked route in Fig.5.1 to reach "B". Packet travels in X-direction (horizontally) until it reaches the correct column and then it travels in Y-direction (vertically) till it reach the destination address.

5.1.1 Running the Model

When the model is scheduled, the windows looks like as shown in Fig.5.2. In the right window, we can see an "html" page displaying the description of model. Window on the left is the actual schedule, where agents traverse in the environment. The window on the

right side has three buttons: play, pause and stop. With the help of these buttons, we can play, pause and stop the schedule. When we click on play button, the schedule starts. When schedule begins, packets start moving from their respective nodes. Green color spheres represent nodes in the network. White color spheres are Packets. NodeElements can be seen as red color spheres. When all the packets reach their destination the schedule stops.

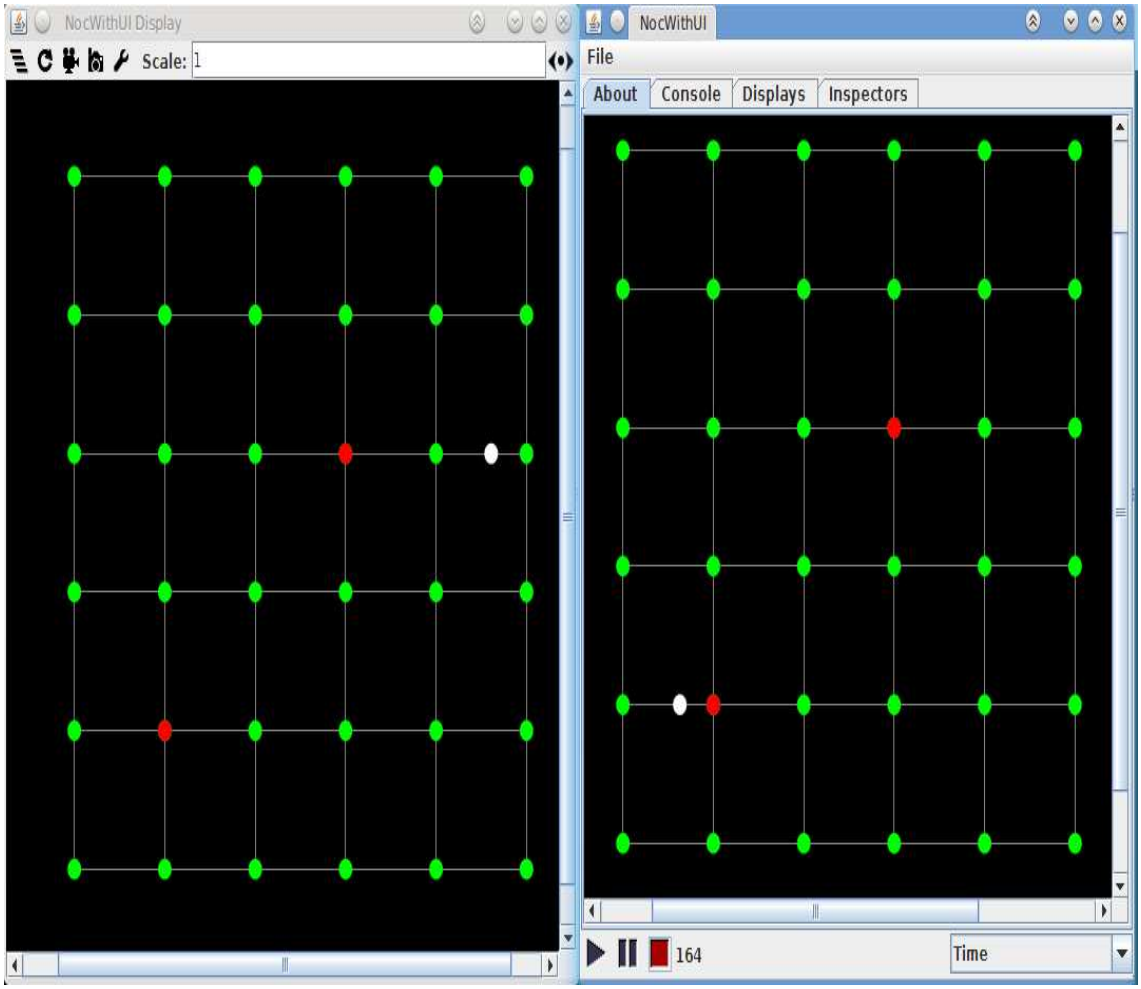


Figure 5.2: Scheduling 2D NoC

5.2 Procedure

Designing a 2D NoC agent based model using GeoMason classes involves the following steps.

1. Create a package named $noc2d$ in app folder of the tool. Create all the classes in $noc2d$ package.

2. Create a class `Noc2D`. In `Noc2D` the environment is created using Java classes.

3. Create a `NodeElement` class for creating Packets.

4. Design a self routing agent which moves along the 2D network and follows XY-Routing algorithm. Self routing agent gets the location information from the `Noc2D` class in order to navigate between the nodes. This information can be retrieved by creating and using the objects of the `Noc2D` class.

5. Create a class `Noc2DWithUI` which extend the `GUIState` class. `GUIState` class enables visualization to MASON models.

6. When we run `Noc2DWithUI`, all the other classes starts their schedule and runs until "$exit$" command is encountered.

7. Run model using command "$java\ sim.app.noc2d.Noc2DWithUI$".

After modeling a 2D NoC, the next step is to create a 3D NoC agent based model. We use MASON 3D classes in this model because the support provided by GeoMason classes is not sufficient to create our 3D NoC model. We introduce our 3D 4x4 mesh NoC agent based model in the next chapter.

Chapter 6

Agent Based 3D Network on Chip Model

A 3D Network on Chip (NoC) agent based model is a new approach to develop routing algorithms and dynamically monitoring the routing process in a graphical form. In this 3D NoC agent based model, we use a 3D mesh network topology. The model has self directive intelligent agents which are capable of navigating themselves in the network. The architecture of this model will be explained in the following sections.

6.1 Model Architecture

The 3D NoC agent based model consists of an Environment and agents such as Packet and RandomAgent. Other elements in the environment include Mesh network, NodeUnit, Node and Path. There are three levels of hierarchy in this model. The levels of hierarchy are represented in Fig. 6.1.

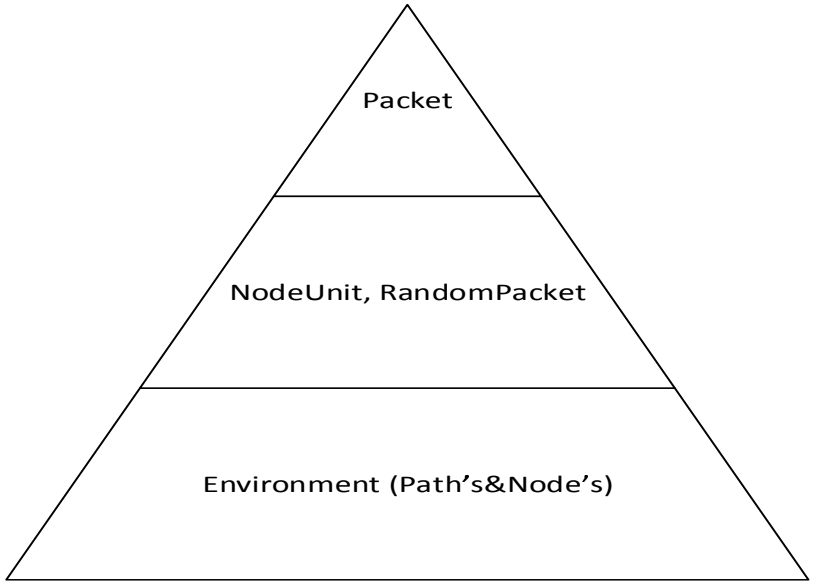


Figure 6.1: Levels of hierarchy

In 3D NoC agent based model, Nodes and Paths are collectively called as Environment. It serves as a medium for the other elements in the network to be present. Nodes and Paths belongs to the first level of the hierarchy and are build at the beginning. NodeUnit and RandomAgent are created in the second level of hierarchy. At the final level, packets are created and scheduled.

6.1.1 Environment

Environment chosen for this NoC agent based model is 4x4 3D mesh network. Nodes and Paths collectively form an environment. Environment is a specific region of the window. All the elements and agents of the network should be present within that region. Agents should only move in that designated region.

Node

Node is the intersection point of Paths in the network. Node's feature is to keep track of the number of the packets passing by at each time. If the Packet count exceeds certain limit specified by a user, Node class displays a predefined message that there is more traffic at current node location. This information will help a user to develop efficient adaptive routing algorithms.

Nodes are present in a 3-Dimensional (3D) space. Node is a 3D point created at a location (X, Y, Z). A location is of type $Double3D$ [54] which is a predefined class in the MASON class library.

Algorithm 2 Creating Nodes in 3-Dimensional space

```
for (x = 0; x <= 3; x+ = 1) do
    for (y = 0; y <= 3; y+ = 1) do
        for (z = 0; z <= 3; z+ = 1) do
            Scale x,y,z coordinates according to the size of the window.
            Create a node using scaled coordinate values.
            Schedule the node once all elements of the network are created.
        end for
    end for
end for
```

Algorithm 2, is used to create the Nodes in this model. X,Y and Z values are of integer type, since using double values in for loops will lead to complex calculations. In the inner most for loop, scale the X,Y,Z coordinates according to the window size. After the creation of all nodes, they are scheduled.

Path

The Path is a line which connects two nodes in the network. A 4x4 3D mesh network is formed with the combination of Paths and Nodes. Nodes are created as explained previously. All created Node objects are stored in a Bag. Bag is a predefined class from $sim.util$ package in MASON. Bag maintains a simple array $(objs)$ of objects and the number of objects $(numObjs)$ in the array [55].

Mesh network

To create a 3D 4x4 mesh network, we have used three separate algorithms. These three algorithms are required to create paths along Z-axis, X-axis and Y-axis which form a 3D 4x4 mesh network. The logic for these three algorithms is given below.

Algorithm 3 Algorithm used to create edges between nodes along the Z-axis

1: **for** $(i = 0; i <= 62; i++)$ **do** ▷ All nodes are stored in a array of type Bag.
2: $from \leftarrow objs[i]$ ▷ The node object objs[i] is retried from the bag of nodes.
3: $to \leftarrow objs[i+1]$ ▷ The node object objs[i+1] is retried from the bag of nodes.
4: Draw an edge between $from$ and to nodes. ▷ An edge is drawn from objs[i] to objs[i+1].
5: **if** $(i >= 2)$ **then** ▷ Avoid necessary edges to create 3D mesh network
6: **switch** i
7: **case** 2: **case** 6: **case** 10: **case** 14: **case** 18: **case** 22: **case** 26: **case** 30: **case** 34: **case** 38: **case** 42: **case** 46: **case** 50: **case** 54: **case** 58: $i++$; break;
8: **case** Default: break;
9: **end if**
10: **end for**

Algorithm 4 Algorithm used to create edges along the X-axis

1: **for** $(k = 0; k <= 47; k++)$ **do**
2: $from \leftarrow objs[k]$
3: $to \leftarrow objs[k + 16]$
4: Draw an edge between $from$ and to nodes. ▷ An edge is drawn from objs[k] to objs[k+16].
5: **end for**

Algorithm 5 Algorithm used to create edges between nodes along the Y-axis

1: **for** $(j = 0; j <= 59; j++)$ **do**
2: $from \leftarrow objs[j]$
3: $to \leftarrow objs[j + 4]$
4: Draw an edge between $from$ and to nodes. ▷ An edge is drawn from objs[j] to objs[j+4].
5: **if** $(j >= 8)$ **then** ▷ Avoid unnecessary edges to create 3D mesh network
6: **switch** j
7: **case** 11: **case** 27: **case** 43: $j+ = 4$; break;
8: **case** Default: break;
9: **end if**
10: **end for**

After creating the 3D 4x4 mesh network, we need to create all the other elements of the network. Each element is explained in detail below.

6.1.2 NodeUnit

NodeUnit is one of the main elements of the network. While creating NodeUnits, a user should pass the location coordinates as an argument to method `setNodeUnit`. Then the NodeUnit is created at the specified location. NodeUnit class is responsible for creating and scheduling packets at a particular time. While creating, packets are aware of its destination address to which they need to navigate. Packets direct themselves in the network to reach the specified destination address. Number of packets to be scheduled at NodeUnit is provided by a user in local variable of the NodeUnit class. All the packets scheduled form a single NodeUnit will have same destination location. User can create any number of NodeUnits by using `setNodeUnit` method in class `Noc3D`.

6.1.3 RandomAgent

RandomAgent is a random self routing agent. A RandomAgent checks the packets at any random node in the network. RandomAgent's characteristic is to detect any unpredictable packet traffic in the network.

Algorithm 6 RandomAgent Logic

```
double[] n =  0.0, 30.0, 60.0, 90.0;          ▷ Array of possible coordinate values.
n[random.nextInt(n.length)];                  ▷ Choosing a random number from array n.
Choose random numbers for x,y,z coordinates.
Set RandomAgent location using these random numbers.
```

Whenever RandomAgent encounters a node, it counts the number of packets at that

node and displays the packet count, location and current time of the schedule. We stored the possible values for X, Y and Z coordinates in an array. From this array, we retrieved random values to set RandomAgent's position in the network. Logic used for RandomAgent is given in Alg. 6:

6.1.4 Packet

Packet is designed as a self routing agent in the 3D NoC agent based model. When schedule starts, all the elements of the network are created and then scheduled. Packets are scheduled by NodeUnit at a particular time specified by user. The Packet agent navigates itself in the 3D mesh network. The Packet follows XYZ-routing algorithm to reach the specified destination. In 3D NoC model, each NodeUnit is identified by a location (X, Y, Z). When a Packet arrives to any NodeUnit then the Packet's address (Cx, Cy, Cz) is updated to that of current NodeUnit's address. When creating a Packet, the destination NodeUnit address (Dx, Dy, Dz) is passed as an argument to the object of the Packet class. The Packet is aware of its destination address before the time of scheduling. When the packet is created and scheduled from the NodeUnit, it then follows a XYZ-routing algorithm to reach the destination. The implemented XYZ-routing algorithm in 3D NoC agent based model is shown in Alg. 7.

In the Alg. 7, d is the distance between any two nodes in the network. Consider, two nodes as $(x1, y1, z1)$ and $(x2, y2, z2)$. Distance between these two nodes is d which is calculated using the formula $\sqrt{(\Delta X)^2 + (\Delta Y)^2 + (\Delta Z)^2}$. The formula is expanded as $\sqrt{(x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2}$. This distance value is required for the Packet to navigate from one node to another. When a Packet is scheduled, in each step the Packet moves and updates its current address to new address. Each time the address is updated either by adding or subtracting the distance value from the appropriate coordinate as per the algorithm. When a Packet reaches the destination then it stops moving and removed

from the schedule.

Algorithm 7 XYZ Routing Algorithm

```
1: if (Dx > Cx)
2:     Move the Packet to address (Cx+d, Cy, Cz);
3: else if (Dx < Cx) then
4:     Move the Packet to address (Cx-d, Cy, Cz);
5: else if (Dx = Cx) then
6:     if (Dy > Cy)
7:         Move the Packet to address (Cx, Cy+d, Cz);
8:     else if (Dy < Cy) then
9:         Move the Packet to address (Cx, Cy-d, Cz);
10:    else if (Dx = Cx)&&(Dy = Cy) then
11:        if (Dz > Cz)
12:            Move the Packet to address (Cx, Cy, Cz+d);
13:        else if (Dz < Cz) then
14:            Move the Packet to address (Cx, Cy, Cz-d);
15:        else
16:            Packet reached its destination address (Dx, Dy, Dz);
17:        end if
18:    end if
19: end if
```

Detailed explanation of the Alg. 7 is as follows: The Current address of Packet is (Cx, Cy, Cz) and destination address is (Dx, Dy, Dz). Firstly, X-coordinate of the current address (Cx) is compared with that of destination address (Dx). Packet moves to

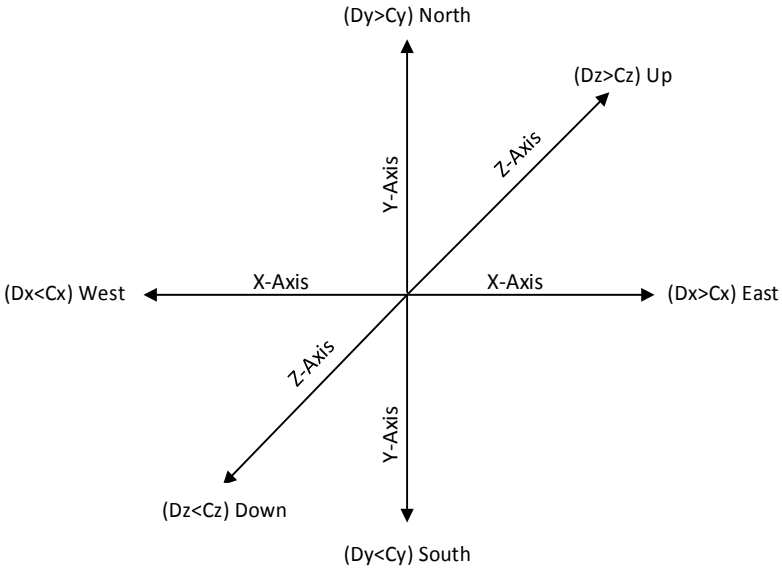


Figure 6.2: Directions with respect to conditional statements

East if $Dx > Cx$, West if $Dx < Cx$ and if $Dx = Cx$ it has already aligned in the destination address X-coordinate (Dx). Secondly, Cy is compared with Dy. if $Dy > Cy$ the Packet moves to North, if $Dy < Cy$ it moves to South, and if $Dy = Cy$, it has arrived to the corresponding Y-coordinate (Dy). Finally, Cz and Dz are compared with each other. If $Dz > Cz$, then Packet moves up. If $Dz < Cz$, then it moves downwards and if $Dz = Cz$, the packet stops moving further as it has reached its final destination.

6.2 Class Description

The package for the 3D NoC agent based model and its classes are explained in detail along with UML representation in this section.

Figure.6.3 illustrates the group of classes in 3D NoC agent based model. We created all the classes in a single package named `noc3d`. The contents of the noc3d package

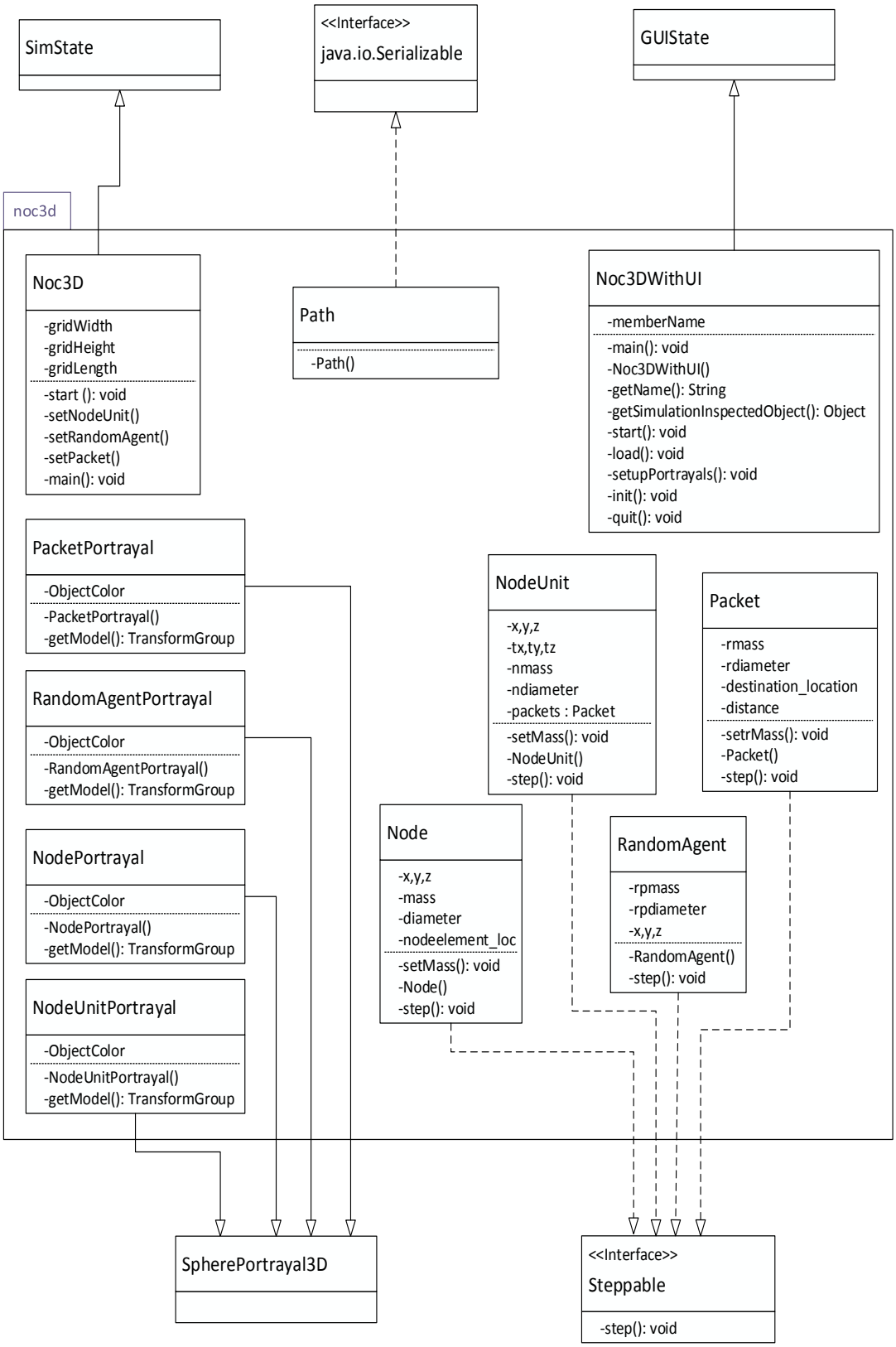


Figure 6.3: UML class diagram for the 3D NoC agent based model

includes the following classes: `NodeUnitPortrayal`, `PacketPortrayal`, `NodePortrayal`, `RandomAgentPortrayal`, `Node`, `Packet`, `RandomAgent`, `Noc3D`, `NodeUnit` and `Noc3DWithUI`. These classes are either implemented or inherited from predefined MASON interfaces and classes. Each class has three fields such as the name of the class, parameters and methods in the class. The dotted lines in the diagram represent that the class implements an interface. Solid lines represent the inheritance property of subclass. A subclass is derived from a superclass using extends keyword. Subclass inherits all the protected and public members of the superclass, no matter in which package that subclass is in.

In this model, we have subclasses which are derived from the predefined MASON classes. The `Node`, `NodeUnit`, `RandomAgent` and `Packet` classes implements the `Steppable` interface. The `Steppable` interface allows the elements to perform their respective functionality in each step. The `Noc3D` class is inherited from the `SimState` class which helps in scheduling the model. `RandomAgentPortrayal`, `NodeUnitPortrayal`, `PacketPortrayal`, `NodePortrayal` classes implements the `SpherePortrayal3D` interface. This interface influences the physical appearance of elements as a sphere. The `Noc3DWithUI` class is a subclass which is inherited from the superclass `GUIState` which is responsible for visualization. The `Path` class implements *java.io.serializable* interface. It allows the `Path` class to be serializable.

6.3 Schedule

3D NoC agent based model is scheduled using MASON multi agent simulation toolkit discussed in Chap. 4. `SimState` class, from MASON predefined classes, is responsible for the scheduling the model in this toolkit. `Noc3D` class is inherited from `SimState` class. `GUIState` class provides the functionality for visualization. `Noc3DWithUI` class is the subclass which is inherited from superclass `GUIState`. This model can also

run without any visualization. User can decide whether he wants to run the model with or without visualization. We can also visualize any of the elements separately using the schedule. For example, it is possible to visualize only nodes without any paths in the schedule. This can be done by choosing the elements which we want to visualize from the list of elements before we start the schedule.

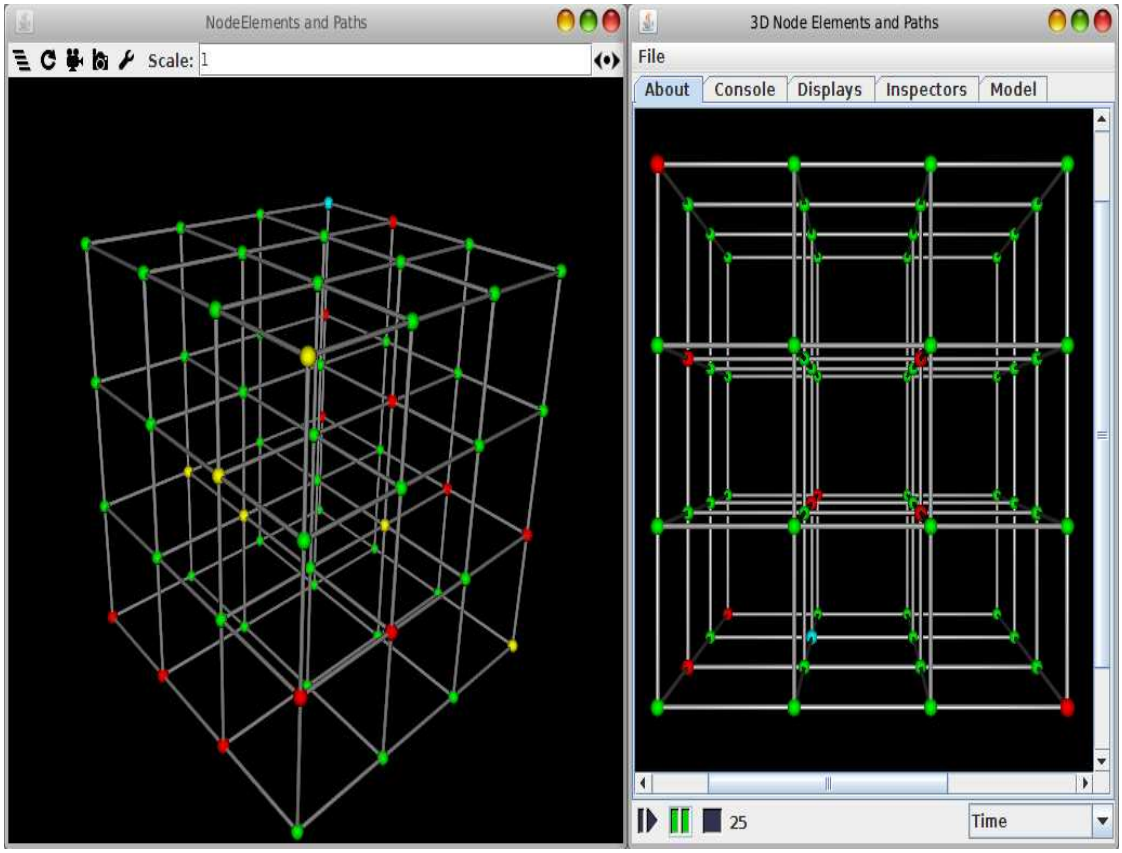


Figure 6.4: Scheduling 3D NoC

As we can see in Fig.6.4, two windows will appear when we run the noc3d model either from command-line or with the Eclipse tool. The window on the right side has three buttons, with which one can play, pause and stop the schedule. When we click on play button the schedule starts. It is also possible to start the schedule from certain time

by specifying time in the time field before starting the schedule. When schedule begins, the packets and randomagent starts moving from their respective nodes. When all the packets reach their respective destination the packets are removed from the schedule and the schedule ends. In the left window, we can visualize the movement of packets and randomagent in the network. We can also monitor the packets following XYZ-routing algorithm and the number of the packets at each node is displayed at console.

The yellow spheres are NodeUnit's from where packets are scheduled. Red spheres are packets. Nodes are displayed in green color. Cyan color sphere is a RandomAgent. Paths are in gray color.

6.4 Procedure

Designing a 3D NoC agent based model with MASON agent based modeling toolkit involves the following steps:

1. Download MASON simulation library. Make sure all the dependencies of MASON Java and Java3D are installed in your computer.

2. Create a package named $noc3d$ in app folder of the MASON tool.Create all the classes in $noc3d$ package.

3. Create a class `Noc3D`. In `Noc3D` the environment is created using Java3D classes.

4. Create a node which can keep count of all the packets and gives the count of packets at each instance of time. Include the functionality to identify the packet traffic. When packets count exceeds certain limit a node gives information about when and where the traffic occurred, and the number of the packets present at a node. For this we should access the packet information from environment.

5. When an agent needs information of other agent, it can be accessed from `Noc3D` class. The access of information is carried out by creating objects to `Noc3D` class.

6. Implement XYZ- routing algorithm to create self routing packets.

7. Create a randomagent which visits random nodes in the network and give the count of packets at that node.

8. Create a class `Noc3DWithUI` which extend the `GUIState` class. This class enables visualization to MASON models.

9. When we run `Noc3DWithUI`, all the other classes starts their schedule and runs until "$exit$" command is encountered.

10. Run model using command "$java\ sim.app.noc3d.Noc3DWithUI$".

Chapter 7

Conclusion and Future Work

Introducing the concept of intelligent smart agents and agent based modeling in NoC architectures helps in detecting many real time problems. One such problem is excess packet traffic. We created a basic 2D NoC agent based model and a 3D NoC agent based model. A 3D NoC agent based model can monitor the number of packets at each node.

Initially, we have created a 2D NoC agent based model using GeoMason classes. We have created the network and we designed packets which follow XY-routing algorithm to reach the destination location in the network.

In the 3D NoC agent based model, we designed an environment for the 3D 4x4 mesh network and the basic XYZ routing algorithm. A user can make use of the mesh topology network of the 3D NoC agent based model and develop more complex adaptive routing algorithms. Users should consider the MASON tool for creating agent based models, as MASON can provide duplicable results, that is, simulation results on different machines will remain same when the simulation is carried out using the same parameters. Most of the other simulators are not capable of providing duplicable results. MASON is faster in visualizing models when compared to other simulators. If needed, the models in MASON can be check-pointed and migrated to other machines.

7.1 Future Work

We can implement more complex models using smart agents. Smart agents approach is an evolving concept in the world of artificial intelligence. This could help us to detect, prevent and solve many real time problems in embedded systems. Some problems involved in embedded systems such as excess packet traffic can be prevented by analyzing the systems using smart agents.

In the 3D NoC agent based model we are using scheduling times for the packets to start their scheduling. This can be further modified such that packets start automatically without specifying any scheduling time. In addition, paths are created by reading two nodes each time from an array and drawing a path between them. It is worth designing a set of classes which support the environment creation. Thus far, this model is restricted to basic functionality. Current model can be further extended by developing more complex routing algorithms.

References

[1] Macal, C. M., & North, M. J. (2010). Tutorial on agent-based modelling and simulation. Journal of Simulation, 4(3).

[2] Pasricha, S., & Dutt, N. (2008). On-chip communication architectures: system on chip interconnect. Morgan Kaufmann.

[3] Duato, J. (2003). Interconnection networks an engineering approach. Morgan Kaufmann.

[4] Kumar, S., Jantsch, A., Soininen, J. P., Forsell, M., Millberg, M., Oberg, J., ... & Hemani, A. (2002). A network on chip architecture and design methodology. In VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium. IEEE.

[5] Sigüenza-Tortosa, D., Ahonen, T., & Nurmi, J. (2004). Issues in the development of a practical NoC: the Proteo concept. Integration, the VLSI Journal.

[6] Wolkotte, P. T., Smit, G. J., Rauwerda, G. K., & Smit, L. T. (2005, April). An energy-efficient reconfigurable circuit-switched network-on-chip. In Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International. IEEE.

[7] Karim, F., Nguyen, A., & Dey, S. (2002). An interconnect architecture for networking systems on chips. Micro, IEEE.

[8] Pasricha, S., Dutt, N.,& Ben-Romdhane, M. (2006, January). Constraint-driven bus matrix synthesis for MPSoC. In Proceedings of the 2006 Asia and South Pacific Design Automation Conference. IEEE Press.

[9] Guerrier, P., & Greiner, A. (2000, January). A generic architecture for on-chip packet-switched interconnections. In Proceedings of the conference on Design, automation and test in Europe. ACM.

[10] Leiserson, C. E. (1985). Fat-trees: universal networks for hardware-efficient supercomputing. Computers, IEEE Transactions.

[11] Adriahantenaina, A., Charlery, H., Greiner, A., Mortiez, L., & Zeferino, C. A. (2003). SPIN: a scalable, packet switched, on-chip micro-network. In Design, Automation and Test in Europe Conference and Exhibition, 2003. IEEE.

[12] Goossens, K., Dielissen, J., & Radulescu, A. (2005). Æthereal network on chip: concepts, architectures, and implementations. Design & Test of Computers, IEEE.

[13] Dall'Osso, M., Biccari, G., Giovannini, L., Bertozzi, D., & Benini, L. (2012, September). Xpipes: a latency insensitive parameterized network-on-chip architecture for multi-processor SoCs. In Computer Design (ICCD), 2012 IEEE 30th International Conference. IEEE.

[14] Bjerregaard, T., & Sparsø, J. (2005). The MANGO clockless network-on-chip: Concepts and implementation (Doctoral dissertation, Technical University of Denmark-Danmarks Tekniske Universitet, Department of Information TechnologyInstitut for Informationsteknologi).

[15] Dally, W. J., & Towles, B. P. (2004). Principles and practices of interconnection networks. Morgan Kaufmann.

[16] Dehyadgari, M., Nickray, M., Afzali-Kusha, A., & Navabi, Z. (2005, December). Evaluation of pseudo adaptive XY routing using an object oriented model for NoC. In Microelectronics, 2005. ICM 2005. The 17th International Conference. IEEE.

[17] Bobda, C., Ahmadinia, A., Majer, M., Teich, J., Fekete, S., & van der Veen, J. (2005, August). Dynoc: A dynamic infrastructure for communication in dynamically reconfugurable devices. In Field Programmable Logic and Applications, 2005. International Conference. IEEE.

[18] Kariniemi, H., & Nurmi, J. (2005). Arbitration and routing schemes for on-chip packet networks. In Interconnect-centric design for advanced SoC and NoC. Springer US.

[19] Kim, K., Lee, S. J., Lee, K., & Yoo, H. J. (2005, May). An arbitration look-ahead scheme for reducing end-to-end latency in networks on chip. In Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium. IEEE.

[20] Bartic, T. A., Mignolet, J. Y., Nollet, V., Marescaux, T., Verkest, D., Vernalde, S., & Lauwereins, R. (2005). Topology adaptive network-on-chip design and implementation. IEE Proceedings-Computers and Digital Techniques, 152(4).

[21] Pirretti, M., Link, G. M., Brooks, R. R., Vijaykrishnan, N., Kandemir, M., & Irwin, M. J. (2004, February). Fault tolerant algorithms for network-on-chip interconnect. In VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium. IEEE.

[22] Dally, W. J., & Towles, B. (2001). Route packets, not wires: On-chip interconnection networks. In Design Automation Conference, 2001. Proceedings. IEEE.

[23] Kim, J., Park, D., Theocharides, T., Vijaykrishnan, N., & Das, C. R. (2005, June). A low latency router supporting adaptivity for on-chip interconnects. In Proceedings of the 42nd annual Design Automation Conference. ACM.

[24] Hu, J., & Marculescu, R. (2004, June). DyAD: smart routing for networks-on-chip. In Proceedings of the 41st annual Design Automation Conference. ACM.

[25] Andreasson, D., & Kumar, S. (2005, May). Slack-time aware routing in NoC systems. In Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium. IEEE.

[26] Feige, U., & Raghavan, P. (1992, October). Exact analysis of hot-potato routing. In Foundations of Computer Science, 1992. Proceedings., 33rd Annual Symposium. IEEE.

[27] Barbosa, J., & Leitão, P. (2011, July). Simulation of multi-agent manufacturing systems using agent-based modelling platforms. In Industrial Informatics (INDIN), 2011 9th IEEE International Conference. IEEE.

[28] Castle, C. J., & Crooks, A. T. (2006). Principles and concepts of agent-based modelling for developing geospatial simulations.

[29] Siebers, P. O., & Aickelin, U. (2008). Introduction to multi-agent simulation. (2008).

[30] Dunin-Keplicz, B., & Verbrugge, R. (2011). Teamwork in multi-agent systems: A formal approach (Vol. 21). John Wiley & Sons.

[31] Bonabeau, E. (2002). Agent-based modeling: Methods and techniques for simulating human systems. Proceedings of the National Academy of Sciences of the United States of America, 99(Suppl 3).

[32] Charles, M. M., & Michael, J. N. (2006). Tutorial on agent-based modeling and simulation part 2: how to model with agents. In Proceedings of the 38th Winter Simulation Conference, Monterey, CA .

[33] Allan, R. J. (2010). Survey of agent based modelling and simulation tools. Science & Technology Facilities Council.

[34] Balch,T.1997. Teambot Simulation Environment “http://www.cs.cmu.edu/ trb/TeamBots/”. Cited on 20th January 2014.

[35] Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., & Balan, G. (2005). Mason: A multiagent simulation environment. Simulation, 81(7).

[36] Johnson, P., & Lancaster, A. (1999). Swarm user guide. Lawrence, KS: University of Kansas.

[37] Collier, N. ”Repast: An agent based modelling toolkit for java.” (2001).

[38] Repast “http://repast.sourceforge.net/“. Cited on 20th January 2014.

[39] Parker, M. T. (2001). What is Ascape and why should you care. Journal of Artificial Societies and Social Simulation, 4(1).

[40] Ascape “http://jasss.soc.surrey.ac.uk/4/1/5.html”. Cited on 20th January 2014.

[41] MASON “http://cs.gmu.edu/∼eclab/projects/mason/”. Cited on 20th January 2014.

[42] Luke, S. (2011). Multiagent Simulation And the MASON Library. George Mason University.

[43] Resnick, M. (1994). Turtles, termites, and traffic jams: Explorations in massively parallel microworlds. Mit Press.

[44] StarLogo ”http://education.mit.edu/starlogo/“. Cited on 20th January 2014.

[45] Wilensky, U. (1999). NetLogo. ”http://ccl.northwestern.edu/netlogo/“. Cited on 20th January 2014.

[46] Klein, J. (2003). Breve: a 3d environment for the simulation of decentralized systems and artificial life. In Proceedings of the eighth international conference on Artificial life.

[47] Breve “http://www.spiderland.org/breve/”. Cited on 20th January 2014.

[48] GEOMASON “http://cs.gmu.edu/∼eclab/projects/mason/extensions/geomason/”. Cited on 20th January 2014.

[49] Sullivan, K., Coletti, M., & Luke, S. (2010). GeoMason: GeoSpatial support for MASON. Department of Computer Science, George Mason University, Technical Report Series.

[50] Coletti, M. (2013, January). The GeoMason Cookbook. “http://cs.gmu.edu/∼eclab/projects/mason/extensions/geomason/geomason.pdf”. Cited on 28th January 2014.

[51] GMU “http://cs.gmu.edu/ eclab/projects/mason/extensions/geomason/classdocs/sim/field/geo/GeomVectorField.html”. Cited on 26th December 2013.

[52] GMU “http://cs.gmu.edu/ eclab/projects/mason/extensions/geomason/classdocs/sim/util/geo/GeomPlanarGraph.html”. Cited on 26th December 2013.

[53] Zhang, W., Hou, L., Wang, J., Geng, S., & Wu, W. (2009, May). Comparison research between xy and odd-even routing algorithm of a 2-dimension 3x3 mesh topology network-on-chip. In Intelligent Systems, 2009. GCIS’09. WRI Global Congress. IEEE.

[54] GMU “http://cs.gmu.edu/∼eclab/projects/mason/docs/classdocs/sim/util/Double3D.html”. Cited on 10th December 2013.

[55] GMU “http://cs.gmu.edu/∼eclab/projects/mason/docs/classdocs/index.html”, Cited on 10th December 2013.