Hybrid Multiresolution Simulation & Model Checking: Network-On-Chip Systems

by

Soroosh Gholami

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved October 2017 by the
Graduate Supervisory Committee:

Hessam S. Sarjoughian, Chair
Georgios Fainekos
Umit Ogras
Aviral Shrivastava

ARIZONA STATE UNIVERSITY

December 2017

ABSTRACT

Designers employ a variety of modeling theories and methodologies to create functional models of discrete network systems. These dynamical models are evaluated using verification and validation techniques throughout incremental design stages. Models created for these systems should directly represent their growing complexity with respect to composition and heterogeneity. Similar to software engineering practices, incremental model design is required for complex system design. As a result, models at early increments are significantly simpler relative to real systems. While experimenting (verification or validation) on models at early increments are computationally less demanding, the results of these experiments are less trustworthy and less rewarding. At any increment of design, a set of tools and technique are required for controlling the complexity of models and experimentation.

A complex system such as Network-on-Chip (NoC) may benefit from incremental design stages. Current design methods for NoC rely on multiple models developed using various modeling frameworks. It is useful to develop frameworks that can formalize the relationships among these models. Fine-grain models are derived using their coarse-grain counterparts. Moreover, validation and verification capability at various design stages enabled through disciplined model conversion is very beneficial.

In this research, Multiresolution Modeling (MRM) is used for system level design of NoC. MRM aids in creating a family of models at different levels of scale and complexity with well-formed relationships. In addition, a variant of the Discrete Event System Specification (DEVS) formalism is proposed which supports model checking. Hierarchical models of Network-on-Chip components may be created at different resolutions while each model can be validated using discrete-event simulation and verified via state exploration. System property expressions are defined in the DEVS language and developed as *Transducers* which can be applied seamlessly for model checking and simulation purposes.

Multiresolution Modeling with verification and validation capabilities of this framework complement one another. MRM manages the scale and complexity of models which in turn can reduces V&V time and effort and conversely the V&V helps ensure correctness of models at multiple resolutions. This framework is realized through extending the DEVS-Suite simulator and its applicability demonstrated for exemplar NoC models.

This dissertation is dedicated to:

my parents, for their unconditional love and support

my brothers, for being the best mentors and role models

and my true friend whom I always regarded as family

The five of you are the reason for who I am today.

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to Dr. Hessam Sarjoughian for the guidance and help he generously provided me throughout these past few years. I am very thankful for the freedom I was given to explore various research directions and for his patience in times of disagreement. From the long list of qualities I hope to have acquired from him, above all were his integrity and ethics: qualities in very short supply these days.

I would also like to thank my committee members: Dr. Georgios Fainekos, Dr. Umit Ogras, and Dr. Aviral Shrivastava for their open mind, insightful advices, and constructive feedback which assisted me in the completion of this research. In addition to that, I give thanks to their exceptional one-on-one meetings which definitely improved the quality of this research leading to this point.

Furthermore, I must acknowledge members of the ACIMS, present and past, for their friendship, assistance, and support. I must specifically thank Abdurrahman Alshareef, William Boyd, Masudul Quraishi, Savitha Sundaramoorthi, and Chao Zhang, my labmates at ACIMS with whom I have great memories, for tolerating my presence in the lab in times of frustration or despair. Among past members, I express my thanks to Dr. Gary Godding for everything I learned from him in our collaborative project and my great appreciation toward Dr. Gary Mayer for his assistance toward finding the right career upon graduation. I wish all these individuals the best of luck and I hope our paths cross each other again soon.

Next, I must thank all my colleagues at PayPal especially my immediate manager Brian Bogard for his support and for the great times I had there as an intern. Everything I learned from my colleagues during this internship did nothing but teach me new things about myself which helped me better myself even more as a PhD student.

Also, I must thank all my close friends (they know who they are) who made my

PhD experience much more enjoyable and these years away from my loving family tolerable. One great benefit of my PhD experience was all the new friends I made here from each of whom I learned a lot. I wish all of them the best in their future endeavors and thank their indirect but great contributions to this dissertation.

Last but not least, I would not have been able to do this dissertation without the help of many others here at ASU, some of whom are unknown to me. That is why I would like to extend my appreciation toward the administrative staff especially Monica Dugan, Pamela Dunn, and Christina Sebring whom I troubled a lot but were always there to help me. Thank you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

Development of any complex discrete network system involves a design phase in which multiple models of the system are created and experimented with. Each model may focus on some aspect of the system and is specified at a certain abstraction level. The models that are created and the evaluations that are conducted all depend on the nature of the target system. As for discrete network systems, models and evaluations focus on the structure, functionality correctness, and QoS satisfaction. Designing network systems also requires multiple models to be created at different abstraction levels, each focusing on certain structural and behavioral aspects.

Systems on Chips (SoC), as a complex discrete network system, has several interconnected intellectual properties (IP) on a 2-D or 3-D chip. The IPs may communicate with one another using shared buses. Shared buses require complex scheduling or prioritized access as each bus can be only driven by one component at a time. An alternative to this approach is Network on Chip (NoC) (Hemani *et al.*, 2000) in which communication among IPs (cores) are treated as series of packets sent and received using an underlying network.

Development process of typical Integrated Chip (IC) designs can be categorized into three major phases. The process starts with the Electronic System Level (ESL) phase (Martin *et al.*, 2010). There are three steps within the ESL phase: 1) function design, 2) application-driven architectural design, and 3) platform-oriented architectural design (Martin *et al.*, 2010). In the first step, a functional model of the application is devised. The second step is dedicated to creating high-level description of the platform and verifying it with the functional application. Finally, in the third

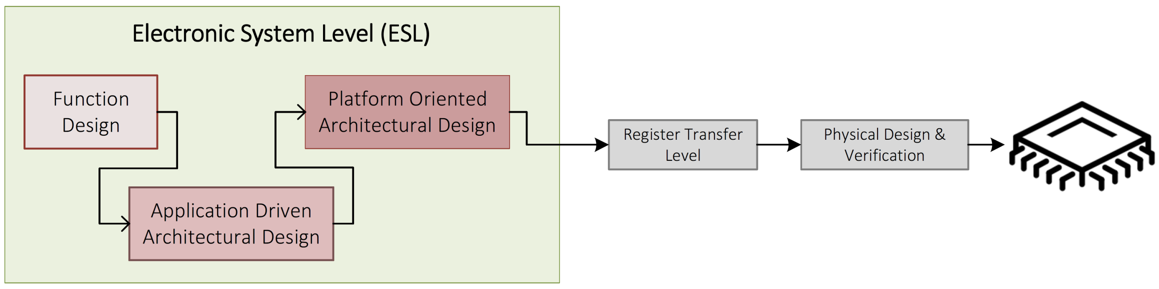

Electronic System Level (ESL)
Function Design
Platform Oriented Architectural Design
Application Driven Architectural Design
Register Transfer Level
Physical Design & Verification

Figure 1.1: 3-phase Chip Design Approach

step, low-level description of the platform is created and the architecture is fine-tuned (Bricaud, 2012). After the completion of the ESL phase, a Register Transfer Level (RTL) design phase will follow in which gate-level descriptions are created based on the resulting architecture from the ESL phase. Finally, in the physical design phase, the gate-level representation is converted into geometric representations. This three-phase approach toward chip design is demonstrated in Figure 1.1.

Similar to the ESL phase, the RTL and Physical Design phases introduce a number of steps to IC design. There is generally strict dependencies between these tasks. In order to design a computer chip from scratch, one cannot automate or bypass these tasks as each of them is responsible for key design decisions. However, a systematic transition from system-level to physical-level specifications is highly desired.

Currently, various tools in different levels of abstraction are used to design new chips or improve existing ones. High-level decisions (such as number of nodes and topology) can be made using coarse-grain models while low-level decisions (such as operating frequency, flow control policy, and hardware brands) should be taken using fine-grain models. Models are developed using two approaches: 1) formal specifications that are implementation-independent (not specific to a certain programming language) and 2) semi-formal specifications written in programming languages (Bricaud, 2012). In order to produce system design at the end of the Electronic System Level

(ESL) phase, the user may use a number of these specifications, languages, and tools.

The concept of abstraction levels for NoC is well-established. The NoC community has introduced four levels of abstraction. From the highest to lowest, abstractions are defined at Interface, Capacity, Flit, and Hardware levels (Dally and Towles, 2004). Hardware-level models have the most details while interface-level models have the least. Designers develop models at various (fine-grain to coarse-grain) granularities during the ESL phase. NoC models at each level are evaluated to ensure correctness and compatibility with their respective requirements using verification and validation methods.

Ideally, a desired design process for NoC should provide a systematic transition from coarse-grain models to fine-grain models while providing the designer with V&V capabilities. Models should be parameterized and easily modifiable. MRM and V&V should complement one another to facilitate a systematic design process. In this process, MRM assists the V&V process by providing a family of models at different resolutions each of which can be used for validation or verification of some properties. This way, verification or validation of some properties can be conducted in higher abstractions which are computationally less demanding and may hold for lower abstractions. On the other side of this equation, V&V assists the MRM process by ensuring correctness and therefore facilitating the creation of finer-grain model from coarse-grain counterparts.

In reality, the design process remains ad-hoc requiring designers to create numerous models at various levels of abstraction often using a number of tools/languages. Some these models are so different from others that there is no well-established way for deducting similarity relationships between them. Without established relationships, creating new models based on existing ones is a manual and ad-hoc process compromised by human error and incompatible modeling frameworks.

Multiresolution modeling is the candidate toward our goal for transitioning from high-level requirements to low-level design specifications of NoCs. Resolution is defined as the degree of detail used to model systems (Department of Defense, 1998). Fidelity of models can be varied in terms of NoC time, object, process, and space characteristics of interest. In Multiresolution Modeling (MRM), a single model or an integrated family of models describe the physical system/phenomena in various levels of fidelity (Davis and Bigelow, 1998). Potentially, these multiresolution models can be leveraged to validate and verify a system in different levels of detail. As opposed to ad-hoc modeling at various abstraction levels, MRM aim is to clearly define the relationship between models at various levels of detail. An MRM provides a set of predefined levels of abstractions within which the designer can create models.

The terms abstraction level and resolution may be used interchangeably. Highly abstracted models hold less details within their specifications and therefore they have lower resolutions. Similarly, models specified in low abstractions contain detailed specifications and can be referred to as high-resolution models. One problem standing in the way of NoC modeling is that the relationships among the Interface, Capacity, Flit, and Hardware abstraction levels are not formalized. Models at these levels of abstraction are distinct in some aspects such as functionality, timing, attributes, entities, and logical dependencies. Multi-Resolution Modeling can help manage specifying models at various levels of resolution and can define relationships between them based on the aspects that are changed moving from one resolution to another. Models at different resolutions are similar in some aspects and different in others. As mentioned, MRM attempts to categorize and form relationships based on the differences between models.

Using semi-formal specifications is very common in modeling and simulation of Network-on-Chips. SystemC (SystemC, 2017) is widely used for modeling and simulation of hardware and software architectures for System-on-Chips. One important aspect of semi-formal specifications is the integration of the modeling method and the simulation engine. Models are created directly in an executable programming language. This is different from formal approaches in which a modeling formalism is used to create the models. The formalism is implementation independent and can be later developed within any execution environment supporting that formalism. Separation of model and simulation is one of the main features of formal specifications. Establishing relationships between models at different resolutions using MRM and homomorphic mapping is one of the benefits of separating model and simulation. One can reuse verification or validation efforts applied to one model on another model with established relationships. This may not be as straightforward on semi-formal languages.

Researchers in various domains have made use of Modeling and Simulation (M&S) for analysis and design of very large, complex systems. These systems range from environmental systems (Antle *et al.*, 2001; Mayer *et al.*, 2006), to virtual environments for training purposes (Basdogan *et al.*, 2001; Norlin, 1995). Although validation through simulation is widespread, the coverage of simulation scenarios is incomplete. Therefore, it is possible to miss some scenarios in which some models do not behave as intended. In the case of verification, model checking may theoretically explore the entire state space of the model. Therefore, through model checking approach one can guarantee that the verification coverage is complete. A weakness of model checking is the amount of time it can take for exhaustively exploring a models state space. Complex models may possess an enormous state space. It may not be possible to verify these models using model checking due to the unreasonable amount of time it may take. For verifying large and complex models, one can reduce the size of state space by developing coarser grain models. Verification using model checking may be achievable for these coarse grain models.

Designers use both validation and verification techniques in order to achieve some degree of assurance that the model is an accurate representation of the reference model and its specifications. Model validation is based on some reference model which could be mental (expectations of the designer from the model) or physical (the real system which is cloned into a model). However, for verification, one focuses on whether the model accurately reflects the specifications of the system. Based on the nature of the models, the system they represent, and the type of input/outputs, one can use various techniques for V&V (Sargent, 2005; Whitner and Balci, 1989) such as stress testing (Myers *et al.*, 2011), Turing tests (Schruben, 1980), and induction (Stoy, 1977). In this research we focus on 1) simulation scenarios and 2) model checking as main methods for validation and verification of models at each resolution. A simulation scenario validates the operation of the model based on references such as the expectation of the modeler, previously gathered data, or another model (Sargent, 2005).

Analyses on a specification of models at each level of resolution provide a set of useful results for making design decisions. The analyses for each level of abstraction could be simulation or model checking. The results of these analyses are reported in various ways. For a network system, these analyses may be reported in average latency, throughput, data loss rate, and average queuing time.

1.1 Problem Description

Marculescu *et al.* have identified four major categories of open research and future challenges for NoC design: 1) application specification and modeling, 2) application optimization for communication, 3) communication architecture analysis and evaluation, and 4) NoC design validation and synthesis (Marculescu *et al.*, 2009). Our work in this research focuses on the categories 3 and 4. In the third category, performance models are used to recognize communication issues such as congestion points. This is possible through V&V. Upon reaching a certain level of confidence with the model, the designer may move forward with layout design. In the fourth category of research problems, the focus is on design validation and synthesis. Various methods of validation such as simulation, prototyping, and testing exist to ensure the designed NoC will operate as expected.

In this section, we describe the research problems we are trying to address. Our contributions relate to the following research solutions: 1) NoC multiresolution modeling and 2) NoC model validation and verification.

1.1.1 NoC Multiresolution Modeling

Currently, IC designers start with identifying requirements and developing system-level specifications of the desired system and then gradually add details to it toward a synthesizable specification (one which can be synthesized on physical hardware). During this process, the designer may use various modeling methods. In addition, evaluation of a design introduces the need for use of several tools/languages throughout the design process. Over the course of the design process, the modeler may end up with a set of disconnected models in various levels of abstraction, each evaluating a crucial aspect of the system under design. In the end, all design decisions made and evaluations performed on these models in various abstractions contribute to developing a synthesizable specification that is used for physical hardware implementation.

The design process for NoC is similar to the one described for integrated computer chips. Complexity of NoC design is amplified due to having software components (network software and application software) in addition to hardware components. The requirements of software impacts hardware design and vice versa. Each major component in NoC (whether a SW or HW component) can be modeled at different abstraction levels. Considering all components in a NoC, the number of models that can be created at various levels of abstraction is very large. Finding the right abstraction to model NoC in this large space so that it can be used for validation/verification is by itself a challenge.

There are various methods/tools with support for modeling the NoC at different abstraction levels. A collection of these tools is necessary for the designer to create accurate system-level models. To our knowledge, there exists no established modeling process that can support modeling NoC from system specification to low-level designs. Of course, one can use programming languages with a semi-formal structure (Bricaud, 2012); however, the design process would be difficult. There is less support for validation and verification of semi-formal specifications developed using programming languages. Also, programming languages are less extensible for specifying models in multiresolution compared to formal specifications.

A similar problem exists even for frameworks supported by formal modeling approaches such as DEVS-based approaches. Existing IOS (input-output system) DEVS-based modeling frameworks do not provide the basis for applying both verification and validation techniques on models. There are many simulation engines created for DEVS models; a few of them are robust and being used. In our view, we believe the user should be able to develop models at different levels of abstraction, identify and categorize the similarity relationships between these models, use the similarity relationships to identify simulation scenarios for validation purposes, and finally constrain the DEVS models for verification.

Figure 1.2 depicts a multidimensional space by which models of NoC can be categorized. The horizontal axis embodies the system specification hierarchy. This dimension captures the level by which the structure and behavior of models are specified. A model can be in one of the following five categories: input/output observation,

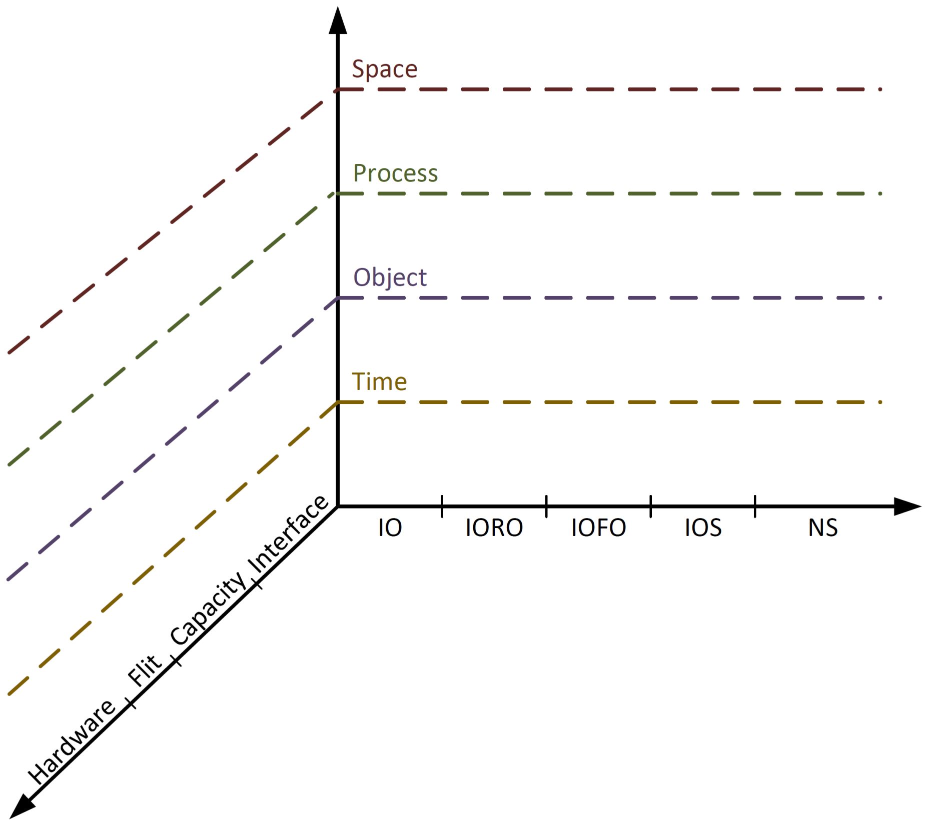


Figure 1.2: NoC Model Categorization Using 1) System Specification Hierarchy, 2) NoC Model Abstraction Levels, and 3) Multiresolution Dimensions

input/output relation observation, input/output function observation, input/output system, and network system (Zeigler *et al.*, 2000). The Input/Output Observation (IO) models only have input and output sets. IO Relation Observation (IORO) adds a set of relations between input and output sets. IO Function Observation (IOFO) replaces the relation set with a set of functions each from input segment ω to a unique output segment $\rho = f_i(\omega)$. IO System adds a state set, time advance, and replaces the function set with a state transition and output functions. Finally, multi-component network system (NS) is made up of a set of IO systems connected through ports (influencers and influencees) (Zeigler *et al.*, 2000). The second axis captures NoC abstraction level using the four categories introduced in (Dally and Towles, 2004) as interface, capacity, flit, and hardware. Finally, resolution of the model is captured in the third axis and is identified by dimensions of resolution. Davis and Bigelow identify four dimensions necessary to identify the resolution of a model: Time, Object, Process, and Space (Davis and Bigelow, 1998). We use Figure 1.2 as our reference for NoC multiresolution modeling.

Since these models belong to a family of multiresolution models, evaluation of models (in the form of validation or verification) at one level of detail may influence the evaluation steps at other levels of detail. However, using the current ad-hoc approach, the similarity relationships between models at different levels of detail are not leveraged and therefore the validation or verification efforts are independent of each other. This independence leaves a lot of room for redundancy and human error. The desired modeling approach should enable the designer to substitute models of a design with models of higher or lower resolutions without disrupting the V&V analyses (simulation and model checking). The new model, in higher resolution may be different with respect to resolution of time, the functionality, models/couplings, or even data.

The paragraph above demonstrates the problems we have with current methods and tools for system design. The relationships between models at various resolutions are not well established. In addition, we do not reuse the evaluation efforts applied to one model cannot in the evaluation of another model in a higher or lower level of resolution. In a complete design process of a system, the designer may end up with a number of models without proper relationships between them specified in one or several tools/languages.

1.1.2 NoC V&V

NoC specifications are usually validated using simulation. Designers specify various scenarios with which they can evaluate the operations of models. One can control the simulation by changing a configuration file containing all the simulation parameters supported by that tool. Few tools offer limited support for NoC model checking which are for properties such as deadlock (Taktak *et al.*, 2008; Verbeek and Schmaltz, 2010). In case the designer requires verifying some aspects of a model, he/she can use one of these tools; however, new models may be created solely for model checking. These models are different from simulation models, although they are both describing the same system/phenomenon. The designer needs to convert the simulation model to a verification model or create a new verification model while considering the simulation model as reference. There are many challenges associated model conversion. Automated model conversion is limited. Although one can convert individual models from one specification to another, universal and automated model conversion for different modeling formalisms cannot be guaranteed. Manual conversion is also challenging as it suffers from human error and is not straightforward for incompatible modeling formalisms. Although the validation and verification techniques remain separate efforts for evaluating the correctness of models, it is still desirable to suggest a single modeling approach for both validation and verification and reduce the need for creating new models or converting one to another.

What we see as important in the landscape of NoC modeling approach is MRM support for both simulation and model checking. We aim at this goal by suggesting a formal specification in which models at different resolutions can be both verified and validated. However, validation using a simulation model is different from model checking of a verification model. To manage the differences, we intend to put restrictions on some aspects of the validation model and extend it to support verification. When validation is targeted, the restrictions are ignored and the extensions are not used. For verification, on the other hand, those restrictions and extensions are taken into consideration. We also intend to use the simulation protocol as a part of the model checking engine in order to stay consistent with the simulation engine. In addition, Experimental Frame (EF) as the method of experimentation in DEVS is reused in the verification protocol for property checking. It is important to be able to apply simulation or model checking to the NoC as a whole or to selective components. Both simulation and model checking can be applied to selective components of NoC. In addition to this, models used for model checking can be easily reconfigured to control the size of the state space which is explored.

1.1.3 Realization in a DEVS-Suite Tool

Various tools and M&S simulation environments exist for network system modeling, validation, and verification. Most of these tools operate at a certain level of abstraction and have capabilities for a subset of validation or verification methods. While Sections 1.1.1 and 1.1.2 introduce research problems concerning multiresolution modeling and NoC validation/verification, they discuss the theoretical aspects. Another research problem would be the development of these in a tool. Therefore, another important element of this research is the realization of a tool with capabilities for MRM and V&V for DEVS-NoC.

Figure 1.3, depicts various steps in the modeling, realization, and tool execution. The use of the tool is depicted in steps 5, 6, 7, and 8. The entire process starts with defining target system requirements is step 1. Using these requirements, one develops conceptual models (in step 2) and experimental frames (EF) for the properties of the system (in step 3). Characteristics of the model developed impacts how experi-

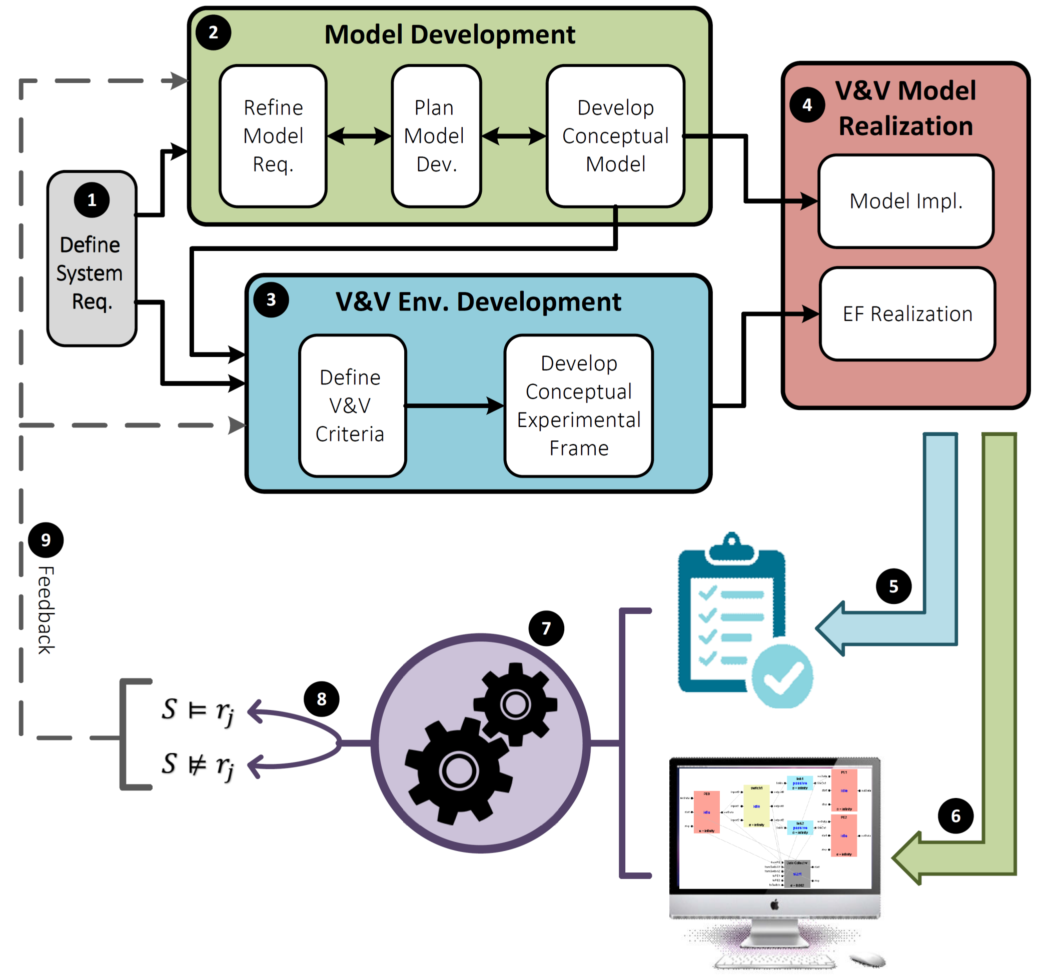


Figure 1.3: The Role of a Designated Tool in Model Development and V&V

mentation (in the form of simulation or model checking) is conducted; therefore, the conceptual model feeds into step 3 for EF development. During step 4, the model and EFs are implemented. In steps 5 and 6, these implementations are realized in the tool. The tools is responsible for executing the model (for either simulation or model checking) in step 7 and checking whether the model satisfies properties (deducted from requirements) in step 8. The result of this assessment is fed back into the model and V&V development in step 9.

As mentioned earlier, we incorporate DEVS for model and EF development. We

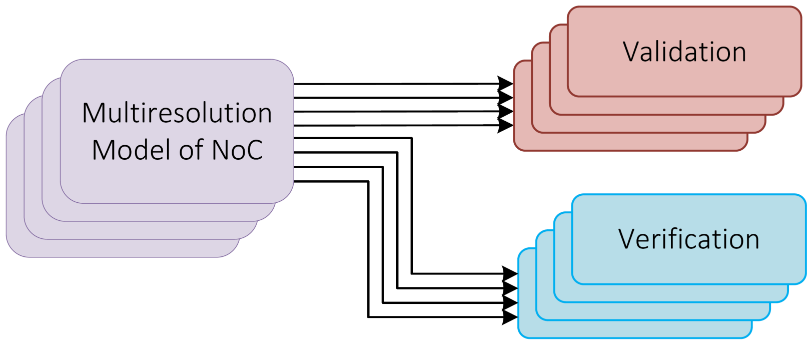


Figure 1.4: V&V in MRM-based for ESL Chip Design Phase

intend to extend DEVS-Suite (ACIMS, 2017) to support steps 5, 6, 7, and 8 shown in Figure 1.3

1.2 Dissertation Goals and Scope

The needs mentioned in Section 1.1 motivated us toward developing an approach and a prototype tool for NoC modeling and simulation based on MRM and V&V capabilities. We do not claim that this approach completely fills the gap between high-level requirements and RTL models. However, it provides a methodology by which one can create a family of related NoC models (using a formal modeling language) at various resolutions. These models can be validated using simulation scenarios or verified via model checking capabilities. We propose a unified approach toward property expression. Properties are expressed in DEVS language and developed as *Transducers* in the model. Transducers can be used in both simulation and model checking efforts for validation/verification of properties without any change.

Figure 1.4 depicts a high-level view of NoC ESL design phase. During the ESL phase, models of the NoC (at different resolutions) are developed and evaluated using validation or verification techniques. The results of these evaluations may be used again to refine the models. Multiresolution process starts with coarse-grain models of

NoC and gradually moves toward finer-grain models. We categorize models based on the categorization introduced in Figure 1.2. Based on the category of the model, one should apply a suitable set of evaluations. These evaluations may be in the form of simulation scenarios (validation) and model checking (verification). One the category of a model is known, simulation scenarios or verification properties of the models in the same category may be used to evaluate the model. This is our primary effort in MRM validation and verification.

Applying the same experiments to a model at a different resolution may not be as straightforward as above. Many aspects of the model may change in a fine-grain model such as the type of input data, the type of output data, and number of ports. A part of our effort in the MRM front is providing methods for reusing validation techniques in a family of multiresolution models.

Definition 1 Application Software: is the software, which is executed on the intellectual properties. It is a single or a collection of tasks. If there are more than one task with dependencies between them, they use the network for the purpose of communication. A multi-tasked decoder application is an example of application software

Definition 2 Network Software: is the communication software controlling the interactions among IPs in the network. This software makes low-level decisions on routing of flits and error handling. Components within a switch are examples of network software

Definition 3 Hardware: is a physical entity within the NoC with simple functionality. Examples of hardware components are MUX or DEMUX. We usually avoid specifying these components except for very fine-grain models

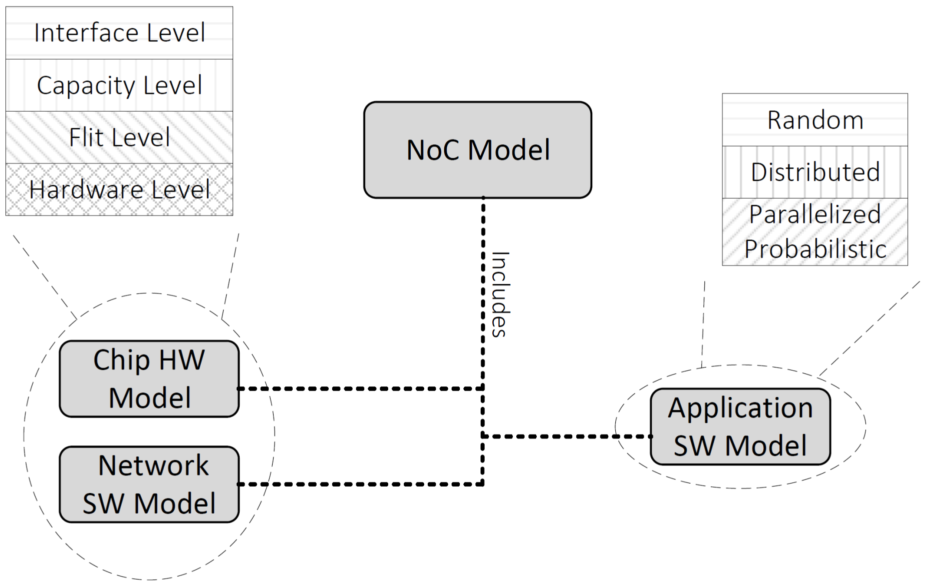


Figure 1.5: NoC Pieces and Their Abstraction Levels

We categorize NoC it into 3 pieces: pure hardware, network software, and application software. These three pieces are defined in **Definition 1** to **Definition 3**. Based on the level of abstraction, the designer devises a model of application software and another for the hardware and network software. Figure 1.5 categorizes NoC model into the aforementioned three pieces and presents the abstraction levels for each of them. For application software model, we consider three levels of abstraction: random, distributed, and parallelized probabilistic (Butler, 1994). As for network software and chip hardware, we consider four levels of abstraction: interface, capacity, flit, and hardware (Dally and Towles, 2004). One can create numerous models by matching various abstractions of application software with those of hardware models and network software models. Each combination is useful in for certain design evaluations. The abstraction levels and methods of design evaluation will be discussed in detail later in this proposal.

Discrete EVent System Specification (DEVS) is chosen as the main modeling language and DEVS-Suite (ACIMS, 2017; Kim *et al.*, 2009) as the modeling and execution tool. There are multiple reasons for choosing this pair (DEVS and DEVS-Suite) for this research. First, DEVS supports hierarchical and parallel modeling of systems which is particularly useful for complex systems. Second, a number of DEVS variants (with support for characteristics such as cellular, fuzzy, real-time) are proposed covering a wide range of systems. Third, while various formalisms are suggested on the modeling side, extended tool support for all these varieties of DEVS makes it a suitable choice for system modeling. Fourth, DEVS-Suite is an open-source, platform-independent tool for DEVS modeling and simulation. Finally, DEVS-Suite is enhanced with a variety of features such as animation, graphical trajectories, simulation view, and multiple addon libraries (for real-time M&S, activity modeling, EMF DEVS, and hardware modeling) makes DEVS-Suite an ideal choice as the modeling and execution tool.

A more detailed view of the verification and validation analyses at each level of resolution is depicted as a block diagram in Figure 1.6. The process starts with the modeling of NoC. This is the hardware, the network software, and the application software. The designer specifies these models in the DEVS modeling formalism and develops them in the DEVS-Suite simulator. Parallel to this, based on the M&S requirements, experimental frames (EF) are devised containing a set of experiments and measurement tools within. Similarly, based on the model checking requirements, a verification engine containing a set of properties to be checked are developed. The validation step uses the DEVS model and the experimental frame (EF) in order to test the model against requirements. As for the verification step, the DEVS model is extended (to constrained variant of DEVS) in order to make model checking possible. The state space of the resulting model is exhaustively traversed for verifying the properties specified earlier.

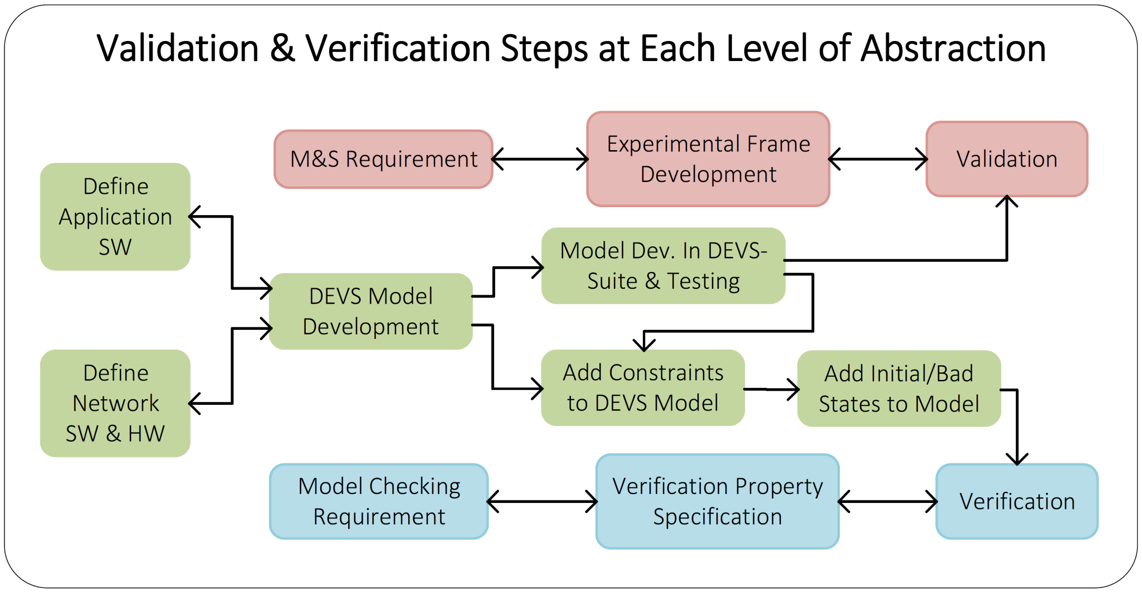

Validation & Verification Steps at Each Level of Abstraction
M&S Requirement
Experimental Frame Development
Validation
Define Application SW
DEVS Model Development
Model Dev. In DEVS-Suite & Testing
Define Network SW & HW
Add Constraints to DEVS Model
Add Initial/Bad States to Model
Model Checking Requirement
Verification Property Specification
Verification

Figure 1.6: Model Validation and Verification Processes at Each Abstraction Level

Chapter 2

BACKGROUND

2.1 Network-on-Chips

A Network-on-Chip (NoC) is a communication subsystem, which connects processing elements of a System-on-Chip (Hemani *et al.*, 2000). Communication and message passing between processing elements are packetized and sent over the network as flits or phits (Dally and Towles, 2004). Flits/phits are routed in the network using NoC routing schemes (oblivious or adaptive) to reach their destination (Rantala *et al.*, 2006). These flits are then assembled in the destination node and form the original message sent from the source node.

NoC models are developed in four levels of abstraction (Dally and Towles, 2004) as shown in Figure 2.1. At the highest level, the interface level, a highly abstracted specification of NoC is incorporated which models packet delivery, packet latency approximations, and network interface. At his level, decisions such as topology, delivery/retransmission policy, and number of nodes on the chip can be decided upon. The next level is capacity level in which new details such as buffer size, component latency, and error rate. At this level, the models are still lightweight. The next level is Flit level. This level is named after the unit of data, which is transferred between components. Since links in NoC are much smaller in bandwidth compared with large-scale networks, real NoCs use flits, which are smaller pieces of data transferrable on those links. In addition to the change in the unit of data, the components within the switch component (such as buffers, crossbar, and allocators), more accurate timing and latency information based on clock cycles, and virtual channels are specified

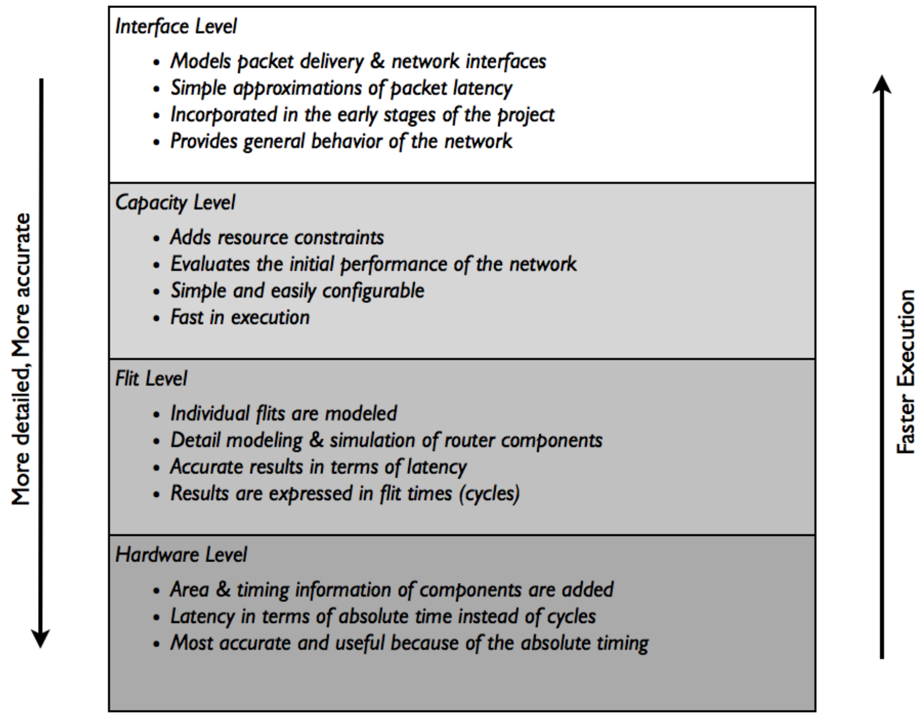

Interface Level
Models packet delivery & network interfaces
Simple approximations of packet latency
Incorporated in the early stages of the project
Provides general behavior of the network
Capacity Level
Adds resource constraints
Evaluates the initial performance of the network
Simple and easily configurable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

Figure 2.1: Abstraction Levels for NoC Modeling and Simulation

within Flit level NoC models. Finally, in Hardware level, the components are modeled closer to hardware by adding area information. The latency and timing aspects of components are also specified in terms of physical time, which is more realistic. A common trend in abstraction levels of any system is that the finer-grain the model, the lower the speed of evaluation. Whether the evaluation is of type simulation or model checking, the process is more time consuming due to the number of events and the size of the state space.

2.1.1 Network-on-Chip Model Validation and Verification

Evaluation of NoC may be done using validation or verification techniques. Validation of NoC models using simulation is very common in the community. NoC frameworks provide modeling/simulation capabilities for various designs of NoC. These

M&S environments enable designer/engineers to predict the performance and behavior before physical implementation. Verification techniques such as model checking may also be used for evaluating NoC models. Using model checking one can verify the correctness of certain important properties within the model such as deadlock freeness.

Evaluations via validation or verification greatly reduce the cost and time of developing new architectures. However, it is not always about developing new NoCs from scratch. In many cases, the designers change a few aspects of NoC as opposed to the entire system. These aspects may be routing schemes, flow control mechanisms, new buffers, links, flit sizes, bandwidths, etc. In these cases, NoC models at the right abstraction level can be used to evaluate the new aspect(s).

Existing NoC evaluation environments serve different purposes based on the abstraction level they support. NoC models specified at the Register Transfer Level (RTL) are closer to their physical realizations and therefore better for making RTL-level design decisions. Coarse-grain models provide a higher-level abstraction of models and are useful for a different set of design decisions such as routing algorithm and flow control mechanism. As shown in Figure 2.1, fine-grain models provide greater insight in the details of the system; therefore, evaluations (whether in the form of model checking or simulation) of finer-grain models are more time consuming for comparable scenarios.

2.1.2 Deadlock in Network-on-Chips

Deadlocks occur when a group of packets in an NoC are unable to progress toward destination while waiting indefinitely on resources. The deadlock condition is due to a circular dependency sequence among packets. Packets may be *waiting* for a resource or *holding* one. These are called *wait-for* and *holds* relations. Deadlock occurs when

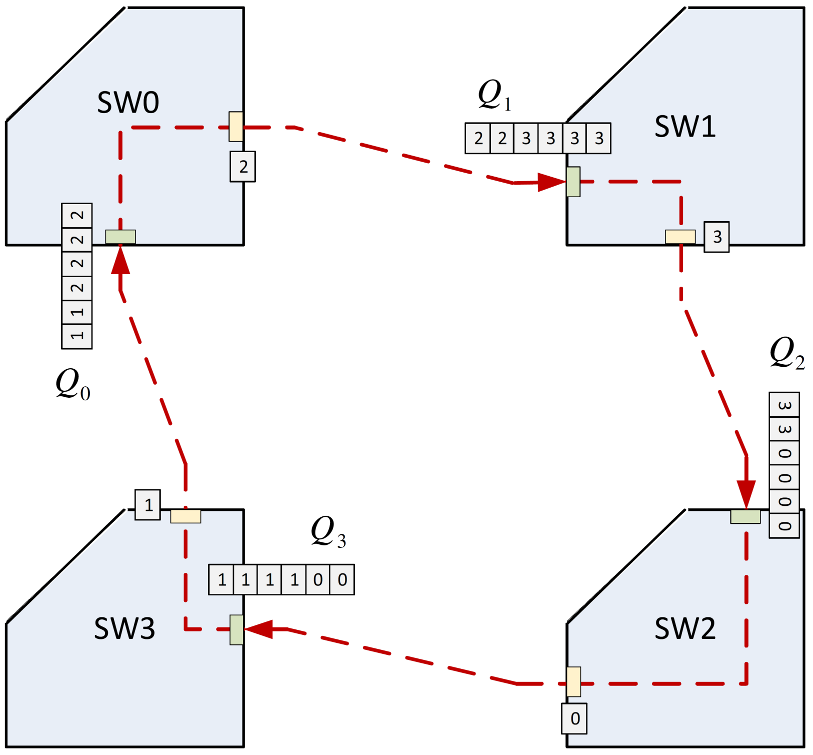


Figure 2.2: Deadlock Condition Implicating 4 Nodes

there is a loop in the resource *wait-for graph* (Dally and Towles, 2004). Without a proper deadlock recovery method, the packets in a deadlock scenario will never progress.

Figure 2.2 demonstrates a deadlock condition implicating four nodes. The input queues (FIFO) of each switch (Q_0, Q_1, Q_2, and Q_3 respectively) are shown with the flits currently stored in them (the numbers are the destinations of the flits). Similarly, the output buffers (the output port of these switches are single-buffered) and what is stored in them are depicted. This is a deadlock case since no packet can proceed due to circular waiting. We prove this deadlock using the wait-for graph for the state of the network is depicted in Figure 2.3. Circles are agents (packets), squares are resources (input queues), solid lines are *hold* relationships (allocated), and dashed lines are *wait-for* relationships. As mentioned earlier, a cycle in the wait-for graph is

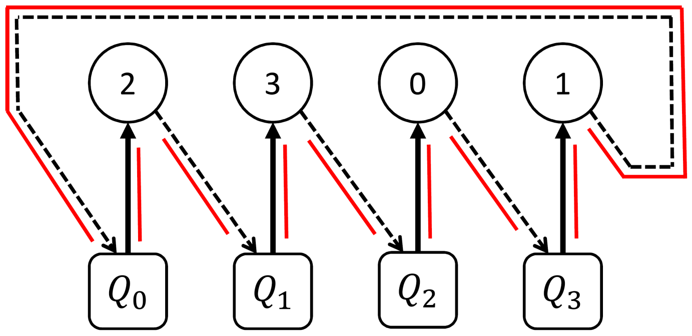


Figure 2.3: Wait-for Graph for the Network Shown in Figure 2.2

proof of a deadlock. The red dashed line in Figure 2.3 marks the cycle in this graph.

There are two methods for dealing with deadlock: 1) deadlock avoidance and 2) deadlock detection and recovery. In deadlock avoidance, one intends to design a network which is deadlock free. In this scenario the designer eliminates the possibility of deadlock occurrence by imposing restrictions on resource allocation. All avoidance methods introduce some sort of resource ordering to avoid forming cycles in the resource wait-for graph Dally and Towles (2004). In Deadlock detection and recovery, on the other hand, deadlocks may occur but the purpose is to detect and recover from them in an efficient manner. This may be more appealing to designers because of the excessive performance degradation of deadlock avoidance methods Dally and Towles (2004). Designers may choose not to impose many restrictions on the operation of the network but deal with deadlock cases if they occur during runtime to improve average-case performance.

Various methods for deadlock avoidance and detection/recovery are suggested. For deadlock avoidance, methods such as resource classes (such as distance or dateline) and restricted physical routes (such as dimension order routing and the turn model (Glass and Ni, 1992)) are used. As for deadlock recovery, *regressive* and *progressive* methods exist. Regressive methods break a deadlock cycle by dropping a number of flits. In order to reduce the cost of deadlock recovery, progressive methods resolve the deadlock situation without dropping flits. An example is the use of escape buffers in DISHA (Anjan and Pinkston, 1995).

2.2 Simulation and Validation

Validation aims at determining whether the model is an accurate representation of the real system/phenomenon. In other words, validation ensures that the right model is built while using verification techniques (Section 2.3) we verify whether the model is built right. There are various techniques by which models can be validated such as historical data validation, Turing tests, and predictive validation (Sargent, 2005). At the core of many of these approaches is a simulation model. Comparing the input/output relationships in the simulation model and the real system is the validation method using simulation (Naylor and Finger, 1967). A simulation model is an executable representation of a real system in a modeling framework. Some simulation models can be created in specialized tools such as BookSim (Dally and Towles, 2004), GEM5 (Binkert *et al.*, 2011), Modelica (Modelica Association et al, 2017). Some others may be based upon mathematical formalisms such as DEVS, Timed Automata, and Markov Chains. While specialized tools provide fast and easy simulations with animation, they cannot be easily generalized for other uses. Models developed using mathematical formalisms, on the other hand, are very customizable and platform independent. Simulation models may also be of different types with respect to the way they handle time, compositions, and control. Examples are hybrid models, agent based models, discrete event specifications, and continuous time specifications. The choice between the type of simulation, the modeling framework, and the simulation tool is all on the shoulders of the modeler. To make an informed decision on the validation environment, the modeler should evaluate the tools modeling capabilities,

Table 2.1: Evaluation Measures for NoC Simulation Environments

Simulation Performance	Area	Area requirements of the design
	Frequency	Minimum frequency to satisfy the application needs
	Latency	Average latency of delivering flits
	Power/energy	Average power usage w.r.t frequency
	Throughput	Transfer capability in the network
	Runtime	Application run-time
	Utilization	Utilization of the network capability
Modeling	Elements	Hardware components, network software, workload support
	Abstractions	Levels of abstraction supported in modeling components
	Co-design	SW and HW are modeled together, co-simulated, and verified
	MRM	Supporting spatial, temporal, object, and process dimensions

the simulation types it supports, and validation mechanisms it possesses.

As an example, in this research the target system is NoC. This kind of system is commonly characterized to have discrete dynamics, even though it also has continuous dynamics. Therefore, discrete event simulation is a suitable choice for developing and validating NoC models. One may choose a NoC modeling and simulation tool by examining its support for modeling alternative, complementary kinds of atomic and composed NoC components. For example, support for modeling software (as a necessity toward HW/SW co-design) is an advantage. Detailed modeling and simulation tools exist that support actual brands of hardware, real software, and physical characteristic modeling capabilities (e.g. GEM5). Table 2.1 highlights some key metrics for system performance evaluation and complementary modeling support.

2.3 Model Checking

Formal model checking aims at exhaustively determining whether a model of a system meets certain requirements. Theorem-proving approaches for mathematically proving a certain property for a certain model is undecidable and generally restrictive (Halpern and Vardi, 1991). Therefore, exhaustive model checking is usually used for critical systems as a full-proof verification method. State explosion problem is a common issue when model checking is applied to a complex system (Burch *et al.*, 1990). However, the use of model checking method is not entirely abandoned. In particular, for safety critical systems, it is still necessary to explore the entire state space. Methods for tackling the state explosion problem are introduced by the community and will be discussed later in this proposal. In addition, model checking can benefit large-scale, complex systems by applying it selectively to some parts of the model. In this approach, the modeler can separately verify the operation of some components, which are vital to the correct operation of the system.

Model checking of hardware systems corresponds to the reachability graph as one can convert the state space to a directed graph with nodes corresponding to states and edges to transitions. Many formal modeling approaches such as Timed Automata (Alur and Dill, 1994) and Petri net (James, 1981) can be verified exhaustively as they correspond to a finite state machine. As for DEVS, because of the continuity of time (for external inputs) and boundless state variables, the state space is unlimited. DEVS-based models, as normally specified, are well suited for simulation purposes. However, model checking algorithms require bounded state space in order to iterate all possible states and transitions.

The general DEVS formalism does not put any constraints on the range of values any of its state variables can have which results in unbounded state space. Unbounded state values are one of the reasons why the DEVS formalism does not lend itself to model checking. Even if the state values are finite, the total state Q is infinite i.e., input events can arrive at any instance of time. Furthermore, there are no restrictions on the input and output values for both atomic and coupled models.

However, constrained versions of DEVS are introduced to address this issue. There has been previous efforts such as FD-DEVS (Hwang and Zeigler, 2009) for supporting model checking. Although this approach is promising, there are two major shortcomings associated with it that we are trying to address. These two shortcomings are discussed below.

2.3.1 Support for Non-determinism and Stochasticity

DEVS formalism has comprehensive support for both deterministic and non-deterministic systems. In a non-deterministic model, several transitions could be possible for a set of internal/external events. Non-determinism adds a large number of possibilities to the state space, which could lead to the state explosion problem (Valmari, 1998). FD-DEVS is developed to address this problem. Predictability of deterministic systems is significant towards verification via model checking. Similar to non-determinism, stochasticity is another obstacle toward model checking. In Parallel DEVS (P-DEVS), internal and external transition functions are deterministic. However, non-determinism exists for external inputs. External inputs may be injected at any instance of time. This is the most obvious form of non-determinism in DEVS.

The randomness for choosing one transition among many, as defined in Finite Probabilistic DEVS (FP-DEVS), can also increase the state space of a model. One can reduce the stochasticity of the system by removing randomness. For example, consider a variation of Petri net (James, 1981) in which the firing of a transition is based on a random variable (Ciardo *et al.*, 1989; Marsan and Chiola, 1986). Compare this, with another variation in which a firing must take place if a transition is enabled such as deterministic timed Petri net (Marsan and Chiola, 1986). For the stochastic Petri net, any event can happen at any instance of time. It is decidable to solve reachability problems for both forms of Petri nets and they are both EXPSPACE hard (Mayr, 1984). However, the deterministic one possesses a much smaller state space. The smaller the state space, the less space it takes to verify the model and the faster it can be carried out.

2.3.2 Limited Property Checking Capability

Models are verified against properties that are defined by some reference model. In most model checking environments, a formal language encodes these properties such as Timed Computation Tree Logic (Alur *et al.*, 1993) in UPPAAL (Larsen *et al.*, 1997) or DEVS Natural Language (DNL) in MS4 Me (Seo *et al.*, 2013). Inventing a language or using a mathematical format are common ways for encoding properties. However, these methods have their own limitations. It can be challenging to encode a complex property in these formal languages. As an example, encoding the reachability problem, may not be so difficult as it deals with the collective state of the system which is already formally modeled. However, what if one needs to verify whether all flits in a Network-on-a-Chip can be delivered within some time window (satisfying a predefined QoS)? Or what if one needs to ensure whether all traffic is distributed fairly within the network and the load on one link (or a set of links) never exceeds a pre-determined threshold (to avoid overheating)? In these cases, formal languages such as DNL and LTL are helpless.

As mentioned earlier, Marculescu *et al.* categorize research issues in four major groups. In the 4th category, the authors briefly explore NoC verification and point out that this field has received less attention compared with other 3 major categories

(Marculescu *et al.*, 2009). There have been many works in this area, which mostly verify properties such as packet delivery and deadlock/livelock freedom. Although verification capability for packet delivery and deadlock/livelock is very useful, it is also limited considering the expected functionality of NoC. It would be helpful if one could perform model checking to verify the operation of the model for other properties such as worst-case flit latency and packet/flit loss ratio.

2.4 Multiresolution Modeling

Multi-resolution modeling (Davis and Bigelow, 1998) is used in various field of research such as graphics (Garland, 1999) and defense systems (Davis and Bigelow, 1998). The challenges associated with MRM have been recognized for many years and application domains (Davis and Tolk, 2007; Yilmaz *et al.*, 2007). MRM is concerned with gradual progression from low-resolution to high-resolution models where the elements of the system withstand appropriate details as measured by the modeling tasks. We can observe that lower-level resolution models are suited for verification using model checking while higher-level resolution models are suited for validation using simulation. Therefore, verification and simulation offer unique, complementary capabilities for designing systems.

Currently, there is no universally accepted definition for multi-resolution as different categorizations of system resolutions exist (Baohong, 2007). Each of these definitions/categorizations may be consistent with a set of systems. However, in this work, we are focusing on a class of systems (i.e., Network-on-Chip) which allows us to be specific and use domain knowledge to define a fitting multi-resolution model for NoC.

Davis and Bigelow recognize 4 dimensions to resolution: object, temporal, process, and spatial. The object dimension relates to the fragmenting components (atomic models to coupled models). By increasing object resolution, the number of components in the model increase and the functionality of each component becomes more detailed. The temporal resolution deals with the granularity of time. The higher the temporal resolution is the more fine grain the time is. The functionality of a component may change without changing object or temporal aspects. This is handled by process resolution, which directly deals with the functionality of each model. Finally, the physical aspects of the model (considering it is modeling a physical entity) are covered by spatial resolution. One important point to note is that no MRM frameworks have been proposed for NoC and System-on-Chips (Berekovic *et al.*, 2002). Without such a framework, it is difficult to classify what could be highest or lowest resolution models (Catania *et al.*, 2016). While one aspect of resolution (such as object, process, time, or space) of a model may be increasing, another aspect may be declining (Davis and Bigelow, 1998). The power of MRM lies in defining levels of model abstractions and more significantly how models that have different levels of resolution can be related based on their differences.

For modeling NoC in this research, we focus on the first three dimensions of resolution: object, temporal, and process. In the context of NoC, object dimension relates to the number of atomic models defining a certain component of the system. For example, the switch element can be modeled as an atomic model or as a coupled model with internal components (buffers, crossbar, routers, etc.). This is shown in Figure 2.4. The switch on the right hand side of this figure is clearly in higher resolution with respect to object dimension. While object resolution is only concerned with the components and couplings, the method/algorithm by which a component completes its task relates to process resolution. An example for this could be the routing algorithm in the switch. A simple algorithm can be X-Y routing in which the flit always takes a deterministic path toward destination. A more complex routing

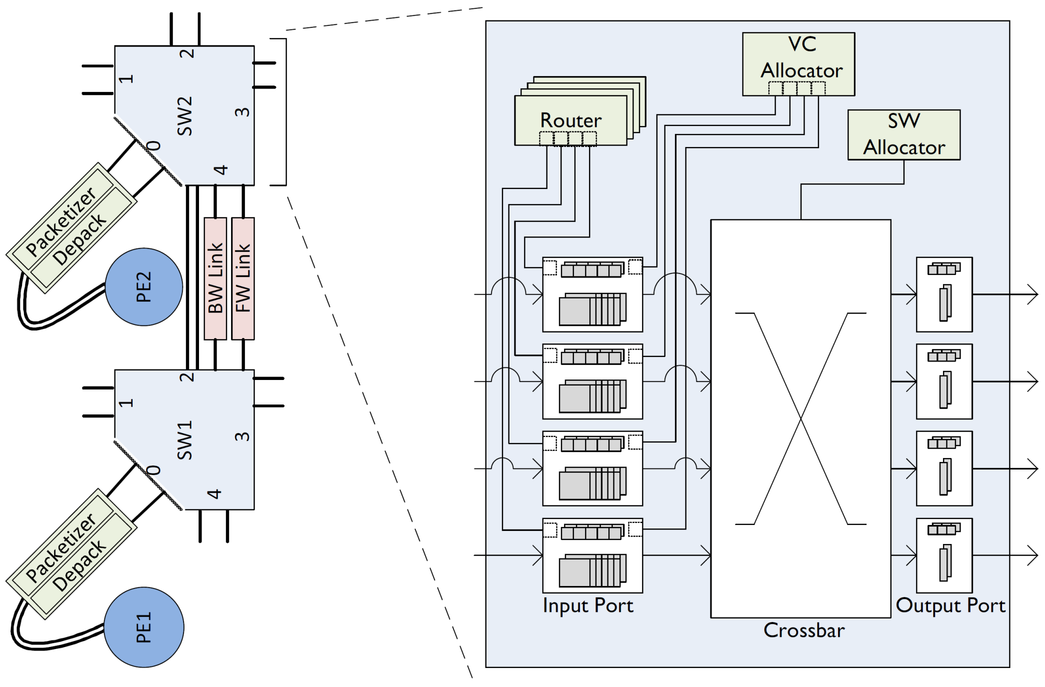


Figure 2.4: Increasing Object Resolution for the Switch Component

could be an adaptive routing algorithm, which considers the traffic while forwarding flits through the network. The second routing algorithm clearly has higher resolution compared to the first one with respect to process dimension. Finally, the temporal dimension relates to the granularity of the clock signal for the model of NoC.

2.5 Homomorphic Mapping

A system can be modeled in number of different ways with respect to modeling method, abstraction level, and methodology. One obvious way of having a number of models for one system is modeling at different levels of abstraction. This can be accomplished with MRM. As explained earlier, a family of models are developed each of which specifying the system at a certain resolution. In these cases, one or more of the object, temporal, process, or spatial aspects of resolution changes which makes one model finer-grained as the other. However, two models may still appear different even if they are modeled at the same level of resolution and have identical functionality. These models may appear different with respect to interface and state space but are still identical. A formal basis for capturing these forms of similarities is very useful. There are three forms of similarity relationships: homomorphism, isomorphism, and copies. These concepts are defined in **Definition 4** to **Definition 6**.

Definition 4 Homomorphism: a relationship between two atomic models when one model is a simplification or elaboration of the other one

Definition 5 Isomorphism: a relationship between two atomic models when two models are essentially the same (have the same functionality) but may have a different interface or notation

Definition 6 Copies: a relationship between two atomic models when two models are identical with respect to interface and functionality

Similarity modeling in the form of homomorphism, isomorphism, and copies are formulated for discrete (-time) systems. A discrete time atomic model is defined as follows (Wymore, 1993):

$$Z = \langle SZ, IZ, OZ, NZ, RZ \rangle$$

In this definition, *SZ* is the state set, *IZ* the input set, and *OZ* is the output set. The set *NZ* specifies the next state function. *NZ* is a function from state set and input set to the state set ($NZ : (SZ \times IZ) \to SZ$). *RZ* Is the output (readout) function, which is a function from state to output set ($RZ : SZ \to OZ$). Time is implicit in the definition of discrete systems. After each tick of the clock, the next state and output functions operate based on the current state and the input set.

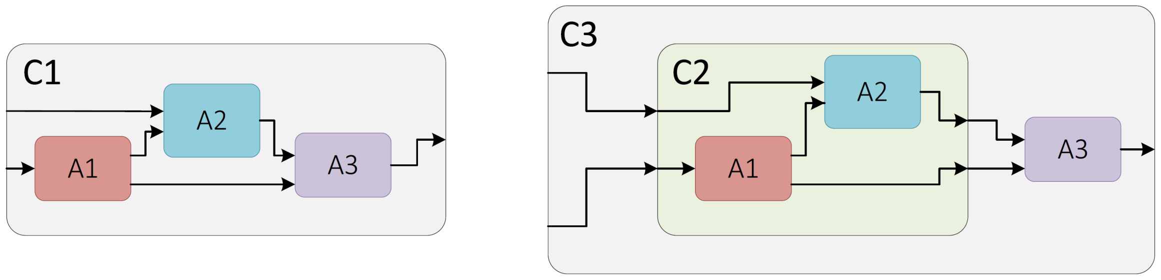


Figure 2.5: Two Isomorphic Models with Different Structures

Homomorphic mapping establishes similarity relationships between atomic discrete-time models by finding a mapping between all these 5 sets. Time is not directly taken into consideration since it is implicitly defined.

The interesting point here is that two models at different resolutions may be homomorphic, isomorphic, or none of the above. For example, models C1 and C3 in Figure 2.5 are isomorphic although they look very different structurally. C3 has an extra level of hierarchy, which changes the structure of the model though the functionality stays the same. At the heart of both models, are identical atomic models A1, A2, and A3, which result in identical functionalities. Therefore, although these models are different, they are isomorphic. So, if two models are only different with respect to object resolution (and nothing else), we can assume there is an isomorphic relationship between the two. The same thing applies to changes in temporal resolution. In the case of process resolution changes, the other model may end up being homomorphic, isomorphic, or not similar to the original model. Complex cases occur when two or more of resolution dimensions change. For those cases, we require a formal approach toward proving homomorphic or isomorphic relationships.

However, one might ask what the benefits of identifying these similarity relationships are. This could impact the validation and verification activities. If we have two models of a component at two levels of resolution and we prove that the two models are isomorphic, one can apply the same validation experiments to both models with no change. Similarly, if the two models are homomorphic, all validation experiments of the coarser-grain model can be applied to the finer-grain model. These similarity relationships enable us to reuse validation activities across different resolutions.

As mentioned in the definitions above, homomorphism relationship is defined between atomic models (Zeigler *et al.*, 2000). However, there has not been any previous work on modeling homomorphism between coupled/atomic models. One of our challenges is formulating a method for homomorphic mapping among coupled models.

Chapter 3

RELATED WORK

Network-on-chip is the central topic of this research. However, NoC design and implementation contains many research fields on structural aspects (such as topology), behavioral aspects (routing or flow control mechanisms), and physical aspects (hardware brand, area information). So many research questions are formulated based on these aspects of NoC (Marculescu *et al.*, 2009). In this research, we focus on multiresolution NoC modeling and V&V of NoC models. Our modeling approach is only limited to discrete event system specification (DEVS). In this section, we first review previous works on NoC verification and validation in Section 3.1. In Section 3.2 we focus on model checking and discuss various modeling methods proposed with support for model checking. Considering the fact that model checking capability should be developed on top of DEVS, we discuss DEVS-based model checking approaches in this section as well. Multiresolution modeling, as the next central subject of this research, is reviewed in Section 3.3.

3.1 Network-on-Chip Validation and Verification

One of the most prevalent ways of validating NoC models is by the use of simulation. There are numerous simulation engines in the community each in a certain level of abstraction. The coverage of simulation engines are not complete but considering the size and complexity of NoC, simulation engines are necessary for evaluating fine-grain models.

Booksim (Dally and Towles, 2004) framework is a flit-level simulation engine with support for throughput, latency, and utilization measurement. However, standard

Booksim lacks capabilities such as area measurement or power consumption. From modeling point of view, Booksim provides a succinct textual format for specifying the NoC, experiments, and the traffic pattern. The elements it supports are at the flit-level (internal switch components). However, it does not directly take into account multi-resolution modeling, co-design, and application software.

SystemC is a C++ library for creating cycle accurate models software and hardware architectures and simulating them (SystemC, 2017). Cycle accurate timing, reactive behavior, and concurrency are among the constructs added to C++ to better support system modeling. SystemC is not specific to any system but it has been used to develop several simulators for NoC. Noxim (Catania *et al.*, 2016) is developed on top of SystemC discrete event simulator. NoC models created within Noxim are configurable in terms of network size, injection rate, traffic pattern, and routing algorithm using a command line interface. Noxim shares similar capabilities and limitation of Booksim. Another example of a NoC simulator developed using SystemC is NIRGAM (Jain *et al.*, 2007).

Wormsim (CMU, 2017) is a cycle accurate simulator developed in C++. This simulator supports a wide range of topologies, routing algorithms, and switching policies while measuring basic performance characteristics of the network. The traffic generations can be also trace-based in addition to synthetic. Trace-based traffic gives the modeler the option of resembling the real application better than a synthetic workload. This simulator can be coupled with Orion (Kahng *et al.*, 2009) for NoC power modeling. This simulator also lacks multi-resolution modeling, co-design, and application software modeling.

TOPAZ simulator (Abad *et al.*, 2012) supports configuration parameters for various components of the network such as router, topology, and traffic. One can integrate this simulator with full-system simulation tools such as GEM5 (Binkert *et al.*, 2011) for holistic performance evaluation and Orion (Kahng *et al.*, 2009) for power analysis. This simulator supports router implementations in several resolutions. It supports a few well-known implementations and other network components are single resolution. TOPAZ does not support application software modeling although by integrating with full-system simulators it can run actual software.

DART (Wang *et al.*, 2011) simulator is unique among the ones introduced in this section for its support for hardware (FPGA-based) execution. This capability substantially increases the performance of the simulation itself compared to other simulators (such as Booksim, NoC-DEVS, and GEM5). The DART simulator specifies the NoC in flit-level as well and provides accurate performance measures. However, it has similar deficiencies as the Booksim simulator.

Arguing that model checking methods are not practical for proving performance constraints, Holcomb *et al.* (2011) formally model new traffic models that are suitable for performance analysis and simulate the RTL model of NoC. For this purpose, the authors introduced a formalism for modeling traffic (called TITAN) and they infer the model from a simulation trace.

There have been previous efforts on formal verification of network-on-chips. Some of these approaches work based on external model checker engines. Salaun *et al.* suggest a formal verification method for asynchronous architecture based on automatic translation from CHP to LOTOS (process algebra in CADP toolbox). Their method of model checking checks for deadlock freedom and protocol correctness (Salaun *et al.*, 2007). GeNoC (Schmaltz and Borrione, 2008) creates a meta-model of the NoC containing size, topology, routing algorithm, and switching mechanism. Then it uses ACL2 theorem prover (Kaufmann *et al.*, 2013) to prove whether data is correctly routed and reaches the intended destination. This approach does not check for performance properties or liveness/deadlock. In another work, the authors extended

GeNoC to support liveness and deadlock detection (Verbeek and Schmaltz, 2010). One limitation of this approach is that it only operates on deterministic routing schemes. Roychoudhury *et al.* use SVM symbolic model checker to verify and debug Advanced Micro-controller Bus Architecture (AMBA). Their approach toward model checking is only limited to starvation (Roychoudhury *et al.*, 2003).

Other approaches have incorporated standard modeling method with wide range support. Taktak *et al.* make use of graph theory and the concept of Strongly Connected Components (SCC) in order to develop an automatic deadlock detection mechanism for NoC. Performance evaluation using previously known link loads and deterministic tasks has been suggested in (Goossens *et al.*, 2005). They used VHDL and XML in order to configure and generate the NoC model. Performance verification is of utmost importance when designing a system with hard performance requirements. Although the approach suggested in (Goossens *et al.*, 2005) is simplistic due to the assumed determinism, it is important that we consider this approach and expand on it in order to bring forth more comprehensive performance verification methods. In another approach, Petri nets were used to verify routing and switching policies in NoC (Bazzaz *et al.*, 2009).

Arguing the excessive time required for verification of RTL models, in (Chatterjee *et al.*, 2012), system-level models are abstracted for deadlock and liveness checking. They identified primitive microarchitectural blocks that are necessary for modeling communication fabrics (not restricted to NoC). The glue logic for composing these models are taken into account in each of the blocks. Therefore, creating communication fabric is as simple as connecting these components using wires. This approach enables rapid model construction and limited model checking for system-level designs. The authors developed a C++ API which finds modeling bugs for automatically generated assertions.

3.2 Model Checking

Numerous modeling methods are introduced with support for model checking. Among the two popular ones are variants of Timed Automata and Petri nets. An implementation of Timed Automata in UPPAAL enables exploring a systems state space through model checking. UPPAAL provides great features such as Java programming (within each state), global time, and property checking using process algebra. UPPAAL uses a simplified version of Timed Computation Tree Logic (TCTL) (Alur *et al.*, 1993) as its property language. Properties in TCTL are in four major categories: state formulae, reachability, safety, and liveness (Behrmann *et al.*, 2014). However, one drawback of using a property language is its limited capability to express complex properties. For example, using TCTL, QoS properties for a network model are more complex to formulate than liveness or deadlock properties.

Petri nets (James, 1981) is a modeling language capable of describing distributed systems. A Petri net is made up of places, arcs, and transitions. This mathematical modeling language is good for verifying deadlock and liveness properties. There are many extensions to the original Petri net. Some of them are more expressive than the original version (Bouyer *et al.*, 2008). For the models that require timing, timed Petri nets were suggested (Holliday and Vernon, 1987). In this variation of Petri net, transitions may have timing. Although this extension of Petri net makes it very similar to Time Automata, there is no equivalency relationship between the two. Therefore, another variant of Petri net were suggested to establish equivalency relationship with Timed Automata. Read Arc Timed Petri net are capable of checking for the presence of tokens without consuming them. The bounded version of Read Arc Time Petri net is computationally equivalent to timed automata (Bouyer *et al.*, 2008). A Petri net is k-bounded if there exists a number k, which is the maximum number of tokens a place can have.

Both Petri nets and time automata are widely used to model and verify concurrent real-time systems. One downside, which both timed automata and Petri nets share, is their inability to handle complex data types. Timed Automata supports the exchange of simple signals and Petri nets can only operate using tokens. For example, a model of a network may require packets (as objects) to be exchanged between model components. This is a serious drawback in developing and verifying such systems. In addition, both Petri nets and Timed Automata only support modeling the behavior of the system. DEVS on the other hand supports structural modeling in addition to modeling the behavior of concurrent systems. However, as mentioned, DEVS in its original form is not suitable for model checking. Therefore, the community has introduced several extended DEVS formalisms with support for checking. What follows is a taxonomy of model checking efforts in the DEVS community.

3.2.1 DEVS-based Model Checking

A variant of DEVS called Finite-Deterministic DEVS (FD-DEVS) supports model checking. It allows finite state/event sets, rational or infinity state lifespans, and the internal schedule change rule. It has tool support through MS4 ME for various phases of model development design. Seo *et al.* identify three model development stages: 1) creating basic atomic models (with event handling logics), 2) adding behavior using state designer, and 3) adding experiment frame to include simulation control and data collection (Seo *et al.*, 2013). In spite of these capabilities offered by FD-DEVS and supported with MS4 ME tool, the description of internal transition, external transition, and advance functions in FD-DEVS do not have support for non-determinism. Many engineered and natural systems such as Networks-on-Chips are inherently non-deterministic. Determinism in FD-DEVS, a subclass of DEVS, is a key limiting factor for modeling stochastic systems. However, both safety and liveness for FD-DEVS are decidable (Hwang and Zeigler, 2009). FD-DEVS does not provide a separate method for arbitrary property checking. Therefore, in the case of Quality of Service properties such as, deadlines for flits or traffic distribution thresholds, FD-DEVS does not support property encoding.

An approach looks at FD-DEVS as target for model checking. Pasqua *et al.* introduce a model transformation approach by which UML sequence diagram models are transformed to FD-DEVS models. They create meta models for both FD-DEVS and sequence diagrams to automate the transformation phase. In addition they incorporate linear temporal logic (LTL) to specify undesired traces. This method can bring simulation and model checking to UML sequence diagrams (after transformation). However, it is still subject to the limitations noted earlier for FD-DEVS (Pasqua *et al.*, 2012).

Saadawi and Wainer introduce Rational Time-Advance DEVS (RTA-DEVS) in which only rational values are allowed for time advance function ta(s). RTA-DEVS introduces a graph-based representation. The authors provide a manual method for converting RTA-DEVS models to Timed Automata. Then, one can verify Timed Automata models in UPPAAL and effectively model check the original RTA-DEVS model. Using this method, the modeler can enjoy simulatability of RTA-DEVS model and verifiability of Time Automata for the same system (Saadawi and Wainer, 2013). However, one limitation of this model is that Timed Automata models are limited with respect to the data they can communicate. This is important for modeling systems such as Network-on-Chips. In DEVS (and all its variations), one can introduce complex data types to be transferred between models. However, by converting the model to Timed Automata this capability is lost. What we are trying to offer is a framework within which simulation and verification are both possible without com-promising the capabilities of the other. Finally, our description of DEVS models are not deviating from normal DEVS model specification. However, RTA-DEVS introduces a graph-based representation, which is somewhat different from the original specification.

Finite Probabilistic DEVS (FP-DEVS) (Seo *et al.*, 2015) is an extension of FD-DEVS in which the choice of next state is made probabilistically. In FD-DEVS, the choice of next state is deterministic through a lookup table. Whereas, in FP-DEVS, an atomic model determines the next state by creating a cumulative distribution function from each transitions probability value and then choosing one based on the value of a random number generator. Potentially, DEVS has support for probabilistic transitions. However, similar to FD-DEVS, FP-DEVS does not support non-determinism.

In addition, researchers have suggested several analytical approaches that can help verifying models as well. For example, in (Nutaro and Zeigler, 2015), a probabilistic confidence method is introduced by which one can evaluate the confidence level of a model. In this approach, one can decide whether a successful test should increase our confidence that the same test will run successfully on the real system. In addition, the probabilistic approach can be used to deduct conclusions based on running tests on a replacement model. Using this approach, the modelers can reach the required confidence level in the model before developing it into a physical/software system. However, this approach is still based on testing and does not address model checking. It can be most useful for evaluating the confidence level of a model at the end of the design process. Therefore, simulations and model checking approaches help designers to reach the final design while this approach can ensure that the final model hits the required confidence level before implementing it.

Simulation frameworks incorporate the concept of experimental frame (EF) for simulation validations. The EF provides an environment in which the model can evolve through experimentation and measurement. Therefore, the EF helps to validate the simulation model. However, if the EF has bugs the entire evaluation is compromised. Therefore, Foures *et al.* considered modeling the EF in formal languages and verifying it (Foures *et al.*, 2013). In other words, we can verify the method by which we validate our simulation models. Although this method does not touch on the matter of simulation model verification/model checking, it can act on top of our environment as a complementary verification phase. As we will elaborate later, EF plays a crucial role in our method of model checking. However, we do not propose a method for verifying it. What is proposed in (Foures *et al.*, 2013) on top of our model checking environment, can provide a higher level of trust with respect to model verification. Similarly, in (Foures *et al.*, 2016) a method for behavioral compatibility between a simulation model and its intended purpose (the real system) is introduced which again is based on the concept of EF.

As mentioned, for large hybrid systems prevalent verification techniques are not very useful. Another class of approaches for verifying such system is simulation guided-methodologies. They incorporate simulations as a means to verify large systems for which the use of formal methods is impractical (Kapinski *et al.*, 2015). Some incorporate falsification methods, which intends to search within a possibly infinite set of parameters and inputs and find unsafe states (Plaku *et al.*, 2009). Since the state space may be infinite, various optimization methods such as Simulated Annealing and Colony Optimization are suggested to find local minimums in which the system is more likely to be unstable (Dokhanchi *et al.*, 2015). Dokhanchi *et al.* suggest coverage metrics to guide the falsification method toward less-explored regions (while searching the state space trajectory). The result is development of a tool called S-TALIRO for coverage guided falsification for Simulink/Stateflow models (TaLiRo, 2017).

3.3 Multiresolution Modeling

MRM has been popular in many domains such as policy making, control, and training. The appropriate resolution of a model for simulation or model checking is decided based on the level of detail required and the properties we want to validate or verify. It is common for modelers to fast forward uninteresting events or components (for optimization purposes for example (Li *et al.*, 2016)) but also to closely examine some state variables, operations and processes. These models can be characterized as LREs (Low Resolution Entities) and HREs (High Resolution Entities (Natrajan *et al.*, 1997)).

Davis and Bigelow present a comprehensive introductory to MRM in (Davis and Bigelow, 1998). They provide basic definitions for resolution, detail, the importance of MRM, and the challenges. They identified four dimensions to resolution: object-related, process, temporal, and spatial. This categorization is important to provide regulate how models at various resolutions can be different from one another. The authors identify MRM to be very useful in economy, explanatory power, complex adaptive systems, and emergent behavior.

Multi-accuracy modeling is an attempt to enable the designer to switch model accuracy during simulation (Beltrame *et al.*, 2007). This approach is useful when some parameters or components of a system may change. Multi-accuracy modeling and simulation shares similarities with our MRM approach of M&S. However, the difference is our emphasis on cross resolution V&V and relationships between various resolutions. Multi-accuracy, on the other hand, intends to provide a convenient way for designers to explore model space. Our approach can be complementary to multi-accuracy by forming similarity relationships between models at various levels of resolution so that the system model is not just a collection of models but also a family of models with meaningful relationships between them. Similarly in (Eggenberger and Radetzki, 2013), an adaptive simulation approach is introduced which enables users to switch the precision of the simulation between RTL and gate-level.

Closer to the line of research presented in this proposal, Davis and Hillestad discuss variable-resolution modeling as a method for building models at various resolutions that are linked together. The user can change the resolution of the entire model by changing the sub-models. They also talk of seamless design, which permits changing resolution with smooth consistency of representation and consistency of prediction (Davis and Hillestad, 1993).

In some works, the concept of abstraction and resolution are used interchangeably. In (Caughlin and Sisti, 1997) a taxonomy of abstraction techniques is provided which has some similarities to the concepts presented in (Davis and Bigelow, 1998) for resolution. In this taxonomy, abstraction techniques are divided into two major categories of structural and behavioral. In behavioral abstractions, one can take the direct or inverse approach. In the direct approach, model abstraction begins with understanding the system and specifying the extracted behavior. However, in inverse modeling one looks at the system as a black box and abstracts the model from input/output data. Similarly, structural abstractions can relate to data as the interactions between the model and the environment and model that can be used to understand the behavior.

Chapter 4

MULTIRESOLUTION MODELING

In this research, we engage the problem of system-level modeling as an early but integral phase in system design. As the designers go through various modeling phases, models become more sophisticated. We propose using MRM to manage the increasing complexity of models and to establish relationships between them in various phases of modeling. MRM enables us to create a family of models within a single modeling framework, all of which represent the same system/phenomenon. Each model in this family captures some aspects of the system with respect to resolution dimensions (object, temporal, process, and spatial) discussed earlier. To ensure correctness of the design and the implementation of the model, every model has to be validated and verified. One important question that this research is attempting to answer is the impact MRM and V&V have on one another. Verification and validation are necessary methods to ensure correctness when moving toward fine-grain models.

To apply this approach to a real system, we chose the Network-on-Chip platform. Coarse-grain NoC models are developed at early stages of the design and then gradually transitioned toward finer-grain models. The need for validation and verification is well known for NoC design process; we introduce MRM to suggest a method to overcome some of the challenges remaining. In addition, we are promoting the use of formal modeling for NoC as opposed to programming based models.

There are three major efforts in this dissertation one of which is customized for Network-on-Chip and the other two are more general for DEVS-based models. The first is on multi-resolution modeling of NoC that enables the designer to model the system at different levels of detail. The theory of MRM and homomorphism specify similarity relationships between models at various levels of detail (this chapter). Similarity measures will be useful for reusing the validation efforts across levels of resolution. The second contribution is concerned with verification of DEVS models. For verification of DEVS models, we propose an approach toward constrained modeling and a DEVS-based model checking protocol (Chapter 5). Finally, our third effort is on developing the model checking engine for the DEVS-Suite simulator and creating verification scenarios for NoC-DEVS models within that tool (Chapter 6).

4.1 Multiresolution modeling

As mentioned earlier, MRM is concerned with gradual progression from low-resolution models to high-resolution ones and vice versa. In the area of NoC, depending on the choice of NoC abstraction (interface, capacity, flit, and hardware) and (structural and behavioral) model hierarchy, we can have different models in different levels of detail. A model hierarchy is defined to have 7 layers: 1) I/O Frame, 2) I/O Relation, 3) I/O Function, 3) Iterative I/O Specification, 5) I/O System Specification, 6) Coupled System, and 7) Coupled Network of Systems (Zeigler *et al.*, 2000).

As mentioned earlier, NoC abstractions are defined in four levels: 1) Interface, 2) Capacity, 3) Flit, and 4) Hardware (Dally and Towles, 2004). This is illustrated in Figure 2.1. A prototypical NoC component belongs to one of three categories: Switch (SW), Link, and Network Interface (NI). Often a fourth component known as processing element (PE) is used. The processing element paves the way for adding *application software* where specific task execution, scheduling, and communication are specified. Thinking of NoC as a 4-component system (PE, SW, NI, and Link), can be considered a high-level (coarse grain) NoC abstraction. Fine-grain resolution modeling of NoC reveals a number of sub-components in each of these components, new processes (network software), spatial information, and relationships. For example, the switch model at the flit level possesses additional subcomponents (e.g. Crossbar) and processes compared to the capacity level switch. MRM is supposed to help us with categorizing model changes between abstractions.

However, MRM role is not only limited to categorizing the changes between models. MRM is involved in a number of steps essential for NoC design. The three of them that we elaborate on here are: 1) MRM for model categorization, 2) MRM for model validation, and 3) MRM for model verification. In model categorization, the purpose is to identify dimensions by which resolution is recognized and defining relationships between models at different resolutions (using similarity modeling). MRM for validation introduces an approach by which one can reuse the validation approach applied to one model on another one in another resolution. Similarly, MRM for verification pursues methods for reusing the verification approach in a multiresolution setting. We discuss these methods below.

In the ad-hoc approach, the similarity relationship between two models at different levels of abstraction are not modeled. Multiresolution modeling aims at forming relationships between models at different abstraction levels. This is achieved by homomorphic mapping and model categorization. Here we discuss model *categorization*, which requires identifying dimensions of resolution.

In (Davis and Bigelow, 1998), resolution is defined to have four dimensions: Process, Object, Time, and Space. As we mentioned earlier, Data is considered as additional dimension of resolution for a system like NoC in which data is an integral part of the system. While abstraction levels of NoC add various details (such as physical, behavioral, structural, and temporal) at each level, it does not differentiate those details from one another. **Definition 7** through **Definition 11** describe physical, structural, behavioral, temporal, and data aspects of a system. For example, moving from flit level to hardware level, the model is equipped with additional components that are excluded from flit level (such as circular buffers at switch input port), physical information (such as area requirements and energy consumption), and detailed behavioral information (such as link reconfiguration (Dally and Towles, 2004)). The abstraction levels (in Figure 2.1) do not differentiate between physical, behavioral, structural, data, and temporal aspects of NoC. Using the dimensions of resolution, details added at each abstraction level, can be categorized as one of process, object, time, space, and data. We will describe and examine the uses of these concepts further in this proposal.

Definition 7 Physical aspect: is any feature of the system that relates to the spatial dimension of the resolution (if the system has any spatial dimension). For a model of NoC, the physical aspects would relate to properties such as the size, exact shape, and mass.

Definition 8 Structural aspect: is any feature of the system, which relates to the interface of models/components and the couplings between those components. These structural aspects can be shown with component diagrams. For an NoC model, structural aspects specify hierarchical components and the coupling between atomic/coupled models.

Definition 9 Behavioral aspect: : is concerned with the functionality of each component. This includes the collection of input set, output set, and the inner workings of the component. The behavioral aspect of a router component in NoC switch is the method it uses to forward traffic.

Definition 10 Temporal aspect: relates to the timing aspects of each component. Each functionality/operation within a component takes certain amount of time to complete. The granularity of time and the accuracy of time estimation for each operation defines the level of detail for the temporal aspects. Higher granularity of time results in higher granularity of functionality. For example, delivering a flit to downstream node in NoC can be modeled as one operation. In higher granularity, operations such as error checking and transmission can be modeled, separately.

Definition 11 Data aspect: focuses on the data as means of communication in network communication. The granularity of data impacts other aspects of a network system. Data defines the bandwidth of links as a physical aspect, routing method as behavior, and granularity of time required for handing data chunks as temporal aspect.

As for the use of MRM in validation or verification, evaluations (simulation or model checking) performed on a model may impact how models in other resolutions are evaluated. For example, the data applied to validate the coarse-grain model of the link may be used to test the finer-grain model of the link as well. As for verification, if we can identify what aspects of a model are changed and what aspects remain similar, there may be no need to repeat the verification process for all desired properties in finer-grain models. If a certain component or functionality does not change when moving from one resolution to another, the verification process can be reused without major changes. This would not be the case when abstracting models in an ad-hoc manner. Sometimes models are in two different modeling languages and therefore all properties need to be checked again. Even when models are in the same language,

Table 4.1: Application Software with Accuracy/Precision Resolutions

Software Application	**Description**	**Temporal**	**Process**	**Object**
Random Tasks	Random communication among tasks Single task assignments to processing elements			✓
Distributed Tasks	Communication volume among tasks Multiple tasks assignment to processing elements		✓	✓
Parallelized Probabilistic Modules	Probabilistic method calls Execution times Dependencies among tasks	✓	✓	✓

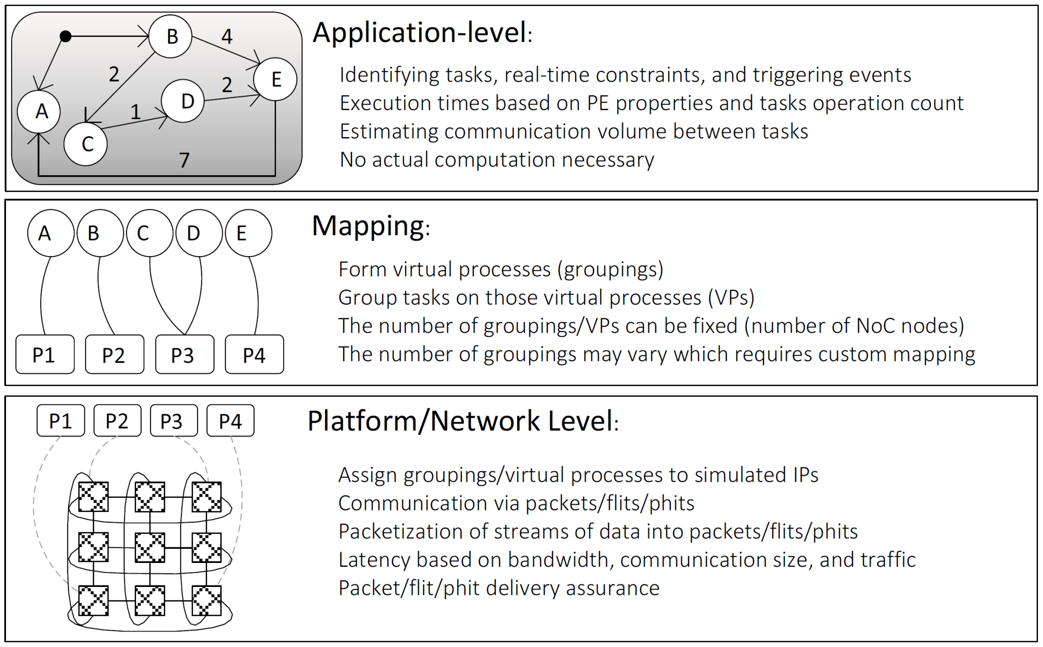
without homomorphic mapping, the relationships between models cannot be clearly defined and one cannot show whether a property will hold in the new model without verifying it again.

Our vision for using MRM in NoC design is to start with coarse-grain models at interface-level of abstraction and gradually increase object, process, and temporal resolution to reach hardware-level models. Increasing the object resolution is equivalent to replacing atomic components with hierarchical structures containing fine-grain components. Changing process resolution is equivalent to changing the functionality of components, which for NoC is the way incoming signals/flits are handled. Changing the communication data (from packets to flits and then to phits) also results in changing the process. Finally, changing the temporal resolution is key for modeling fine-grain actions in the circuit. We do not consider the spatial dimension of models within the bounds of this research.

Remember from Figure 1.5 that we categorized NoC to hardware, network software, and application software. For multiresolution modeling of NoC hardware and network software, resolution can be expressed in terms of abstraction levels (interface, capacity, flit, and hardware levels). We reviewed these abstraction levels in Figure 2.1. Mutli-resolution modeling can also lend itself to developing application software. Application software can be modeled as a set of distributed tasks, which transmit random messages, set of tasks with pre-defined communication volume, or a set of tasks with additional specifications on the execution times, threads, dependencies, and function calls (Butler, 1994). These three resolutions are illustrated in Table 4.1. The application software models vary in terms of Object, Process, and temporal dimensions of resolution. From object point of view, single tasks (atomic) in low resolution are converted to a collection of function calls and parallel threads (Butler, 1994). From process point of view, the software tasks are equipped with dependency, execution times, and probabilistic method calls as we model the software in higher-resolution. Finally, in the highest resolution, temporal aspects of software (such as conditional execution times) are modeled as well. The highest resolution software contains tasks, dependencies, communication volume, threads, and methods.

A complete NoC system model, contains a model for hardware and network software and another for application software. Using MRM, the designer usually starts from interface-level hardware model and random task application software and gradually move toward finer-grain models. Figure 4.1 shows mid-resolution application software (distributed tasks) executing on capacity-level hardware modules (inspired from (Salminen *et al.*, 2009)). At application level, software tasks are mapped to low-resolution hardware modules (PEs).

In case the software model is known and available before hardware design is started, one may incorporate fine-grain software models with coarse-grain hardware



B
4
2
2
E
A
1
D
C
7
Application-level:
Identifying tasks, real-time constraints, and triggering events
Execution times based on PE properties and tasks operation count
Estimating communication volume between tasks
No actual computation necessary
A
B
C
D
E
P1
P2
P3
P4
Mapping:
Form virtual processes (groupings)
Group tasks on those virtual processes (VPs)
The number of groupings/VPs can be fixed (number of NoC nodes)
The number of groupings may vary which requires custom mapping
P1
P2
P3
P4
Platform/Network Level:
Assign groupings/virtual processes to simulated IPs
Communication via packets/flits/phits
Packetization of streams of data into packets/flits/phits
Latency based on bandwidth, communication size, and traffic
Packet/flit/phit delivery assurance

Figure 4.1: Mid-resolution Application Software Model

models. The resolution of all components within a model may not be the same. A system model may contain a mix of fine-grain models and coarse-grain ones. While the hardware (and the network software) is modeled at the lowest resolution (interface level), the application software can be modeled at its highest resolution (parallelized probabilistic). The highest resolution application software contains tasks, dependencies, communication volume, threads, and methods. Similarly, the highest resolution of hardware (at hardware abstraction level) may be co-simulated with the least detailed application software (lowest resolution in Table 4.1) as the compatibility of hardware design with application software has been checked and confirmed with coarse-grain models.

In case fine-grain model of software is known and is used for co-simulation with coarse-grain hardware, one can extract a benchmark form the same software to co-simulate with finer-grain models of hardware. For this purpose, a transducer model records the packet communication between processing elements. This record is then used as application software model in a finer-grain model. The record can be used as only feed-forward (the software is reduced to tasks sending and receiving flits using the record) or a feedback-enabled software (tasks are still data sensitive, dependencies exist, and the record captures all of them). As a result, the high-resolution software application dynamics is highly restricted.

4.2 Multiresolution NoC Models

In order to develop multi-resolution model components of NoC, we ask two questions: 1) what can be the resolution of a component? and 2) what relationships model components at different resolutions can have?

For NoC, moving from Interface, to Capacity, to Flit, and to Hardware abstractions leads to more closely modeling physical aspects. Resolution for the hardware and network software (refer to Figure 1.5) is defined based on the abstraction levels introduced in (Dally and Towles, 2004). Earlier, we categorized resolution into 5 groups: object (structural aspects), time (temporal aspects), space (physical aspects), process (behavioral aspects), and data (data granularity). Knowing that dimensions of resolution must be customized to the system under modeling, Figure 4.2 depicts an example of what changing each aspect of resolution means for a system such as NoC.

Figure 4.2-a demonstrates the impact of increasing object resolution for an atomic component. Object resolution deals with the compartmentalization/decompartmentalization of models. Converting an atomic model to a coupled one with more granular components leads to a higher object resolution. Figure 4.2-b depicts temporal resolution. As defined earlier, the granularity of time and the accuracy of time estimation for each operation defines the level of detail for the temporal aspects. Increasing time resolution results in the model operating closer to reality. A simple view of process

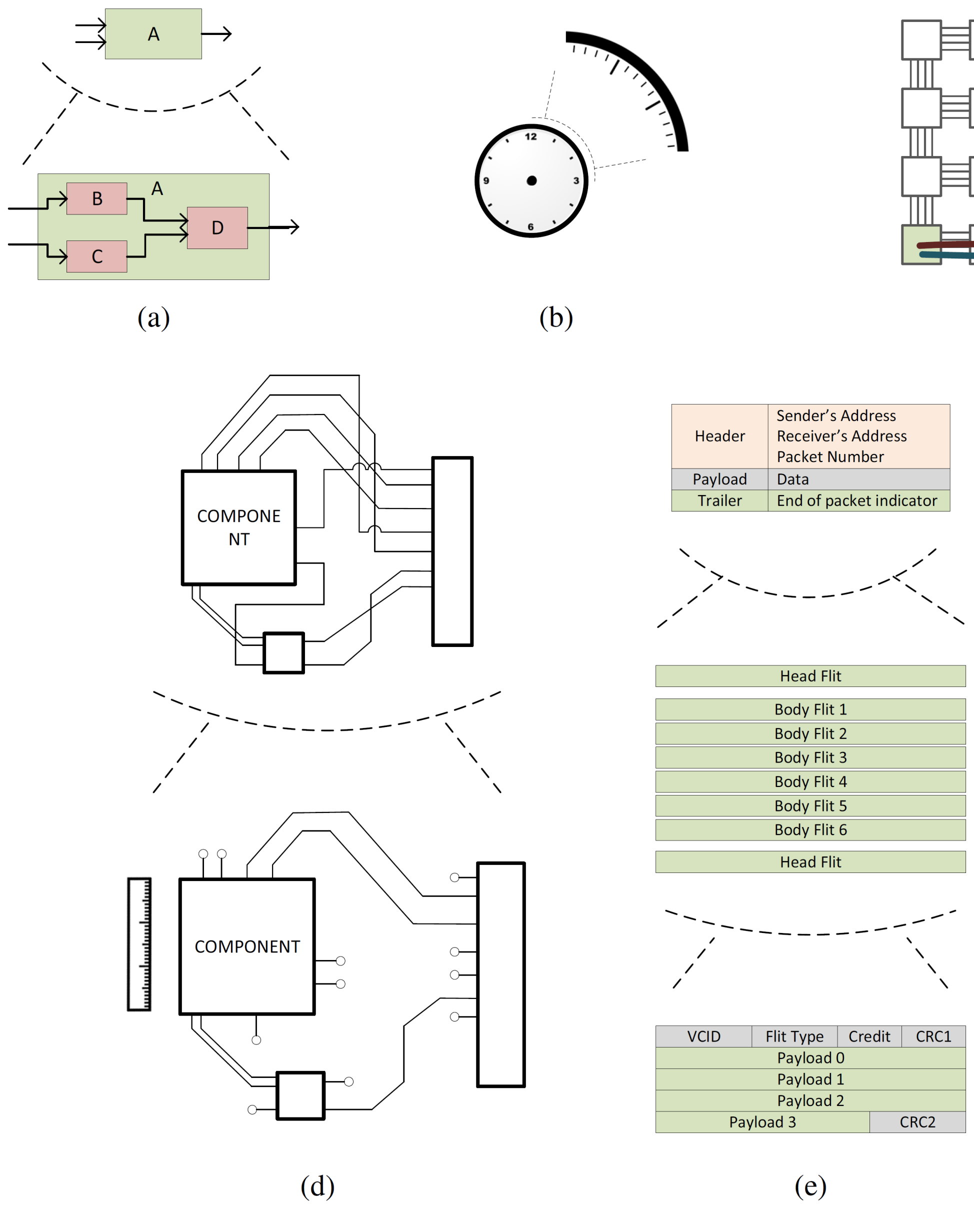


Figure 4.2: Changing Five Aspects of Resolution Visualized for Noc. (A) Increasing Object Resolution via Compartmentalization, (B) Increasing the Temporal Resolution by Increasing the Granularity of Time, (C) Increasing Process Resolution (in a Routing Algorithm) by Suggesting a More Intricate Approach, (D) Increasing Spatial Resolution by Adding Physical Information (Vertical Interconnection and Size), (E) Increasing the Resolution of Data by Incorporating More Granular Data Chunks

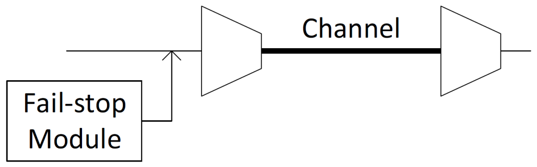
resolution for routing algorithms is shown in Figure 4.2-c. Compared to oblivious routing methods, adaptive routing considers more aspects of the network in routing packets around. One aspect of process resolution is the level of complexity of the operation carried out by the system. Spatial resolution deals with how closely physical aspects of network are modeled. In Figure 4.2-d (top part), two components are coupled with one another without considering layering (which is a physical aspect of the chip). The lower part of Figure 4.2-d, on the other hand, also takes into account VIAs (Vertical Interconnection Access) and chip size in modeling. Finally, 4.2-e demonstrates the granularity of data as the medium of communication between NoC components.

As defined, the Interface level model has the lowest level of resolution; its components are specified as objects without having Temporal, Process, and Spatial abstractions. Only a set of objects exchanging data in an ordered discipline. We do not consider this order as temporal specification because of its vast difference with temporal specifications in other hardware abstractions of NoC. The capacity level introduces timing for delivering messages between two nodes and higher resolution objects. The flit level extends the capacity level by introducing processes (for handling flits) and more detailed model of time, objects, and data. Finally, the hardware level, in addition to extending all models with higher resolution concepts, adds chip spatial information to the specification.

What follows is component view of some models for NoC components.

4.2.1 Link

The link model which is demonstrated here contains components from upstream and downstream nodes as well. Link component can be modeled as a uni-directional connector to a channel with wires, error checking module, error counter, reconfigu-



Header	Sender's Address Receiver's Address Packet Number
Payload	Data
Trailer	End of packet indicator

Figure 4.3: Low-resolution Link Model with the Type of Data It Communicates

ration logic, etc. Figure 4.3 depicts a low-resolution model of the link with the data (packets) that it can handle. A piece of network software exists for the *Fail-stop* module which is in charge of disabling the channel if need be. The network software inside the Fail-stop module is simple since the model is at the capacity level. The Fail-stop module communicates with the hardware at the channel entry point. That is where the software signal interacts with the hardware component and disables the transmission operation.

Listing 4.1: Model of the Link in DEVS

$$S = \overbrace{\{Active, Idle\}}^{\text{phase}} \times \overbrace{\sigma}^{\text{sigma}} \times \overbrace{\{0,1\}^{16}}^{\text{Phit}}$$

$$X = \big(in, \{0,1\}^{16}\big)$$

$$Y = \big(out, \{0,1\}^{16}\big)$$

$$\delta_{ext}\big(Idle, \sigma, Phit, e, (in, x)\big) = \big(Active, \delta t, x\big)$$

$$\delta_{int}\big(Active, \sigma, Phit\big) = \big(Idle, \infty, \varnothing\big)$$

$$\lambda(Active, \sigma, Phit) = (out, Phit)$$

$$P_p = 1 - (1 - Pe)^N$$

$$\text{Fail-stop Signal} = \begin{cases} 0\ (enable), P_p < 10^{-12} \\ 1\ (disable), P_p \geq 10^{-12} \end{cases}$$

The Fail-stop consists of a hardware logic which is governed by a network software. However, there are components in NoC that are purely hardware, such as buffers. For the low-resolution model of the link, channel is pure hardware while the Fail-safe module is both hardware and network software. The network software will be modeled as the behavior of the hardware component. The hardware component is defined by specifying the state space and input/output ports. The rest of the specification (how inputs are handled, how outputs are generated, how the state changes, etc.) is the network software which specifies how the hardware behaves. At each of the resolution levels (described below) the hardware and software components of the link are extended.

As a showcase of a simplified capacity-level model of the link, we modeled the channel using DEVS in Listing 4.1. The model is defined to have a state set (S), input/output ports, external transition (δ_{ext}), internal transition (δ_{int}), and output functions (λ). As for the network software operating on the Fail-stop module, we used the concept of BER (bit error rate). The software disables the channel if the actual frequency of channel malfunction (bit error) is greater than the packet error ratio (PER) which is characterized by P_p. The Fail-stop module can be developed within a DEVS model and thus simplify its composition with the channel model.

The link is modeled at higher resolution in Figure 4.4 (left) by increasing its details in the object and process resolution dimensions. The hardware is extended by flit buffer and additional logic for retransmission and error checking. The network software is also extended with error checking algorithm and retransmission decision making module. The retransmission logic, upon receiving an error signal from *Error Checking Module*, reconfigures the MUX to pass the buffer data through and enables the buffer to transmit the previously stored data. This way, the data sent in the previous cycle and rejected by the error checking module is retransmitted. In addition to the extensions made to hardware and software, the data which is communicated is changed to flit which is the breakdown of a packet into 8 chunks of smaller data. The higher resolution for the packet and flit structures are shown in the same figure.

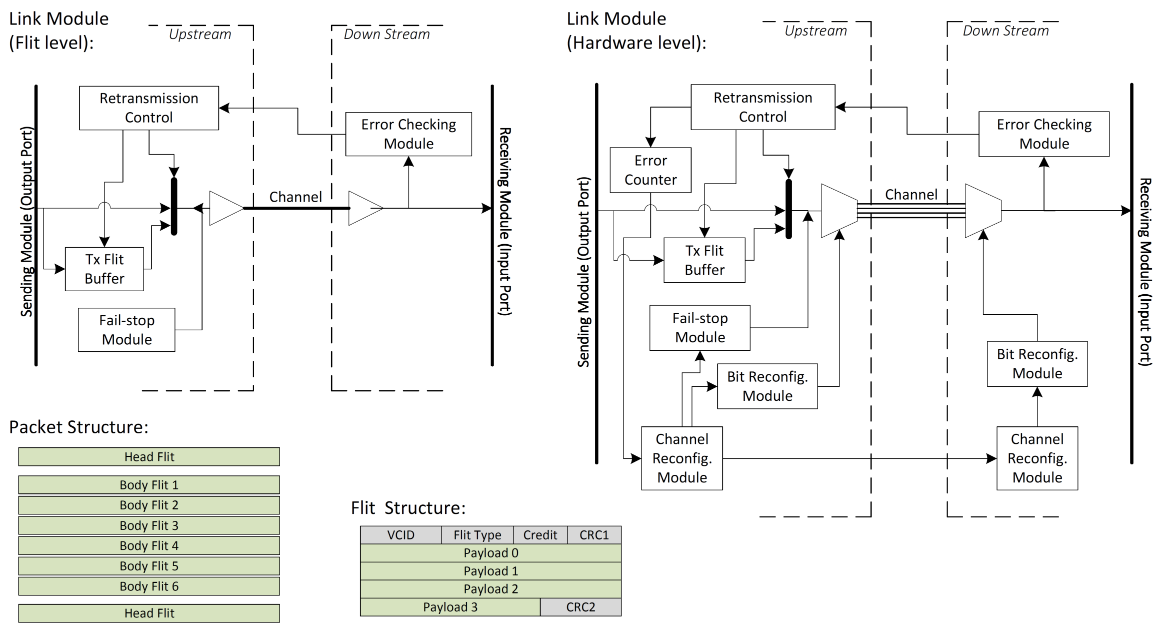


Figure 4.4: Link Models at the Flit and Hardware Abstraction Levels

Finally, in Figure 4.4 (right), we have modeled the link at higher resolution. The hardware is extended with additional necessary modules for wires, error counter module, and channel reconfiguration module. Consequently, the network software is also extended for channel reconfiguration management. The Fail-stop module works based on an error counter module. If the number of errors for the channel becomes greater than acceptable, the channel reconfiguration module orders the fail-stop module to block the channel. In bit reconfiguration scenario, the channel is reconfigured to change the data wires due to bit errors in one of them. Thus, the channel is reconfigured to use less or a different set of wires for data communication. In high-resolution, the communication unit of data is still the flit.

4.2.2 Switch

A switch component orchestrates the operation of NoC by routing flits in the network. The performance, latency, energy consumption, and other important char-

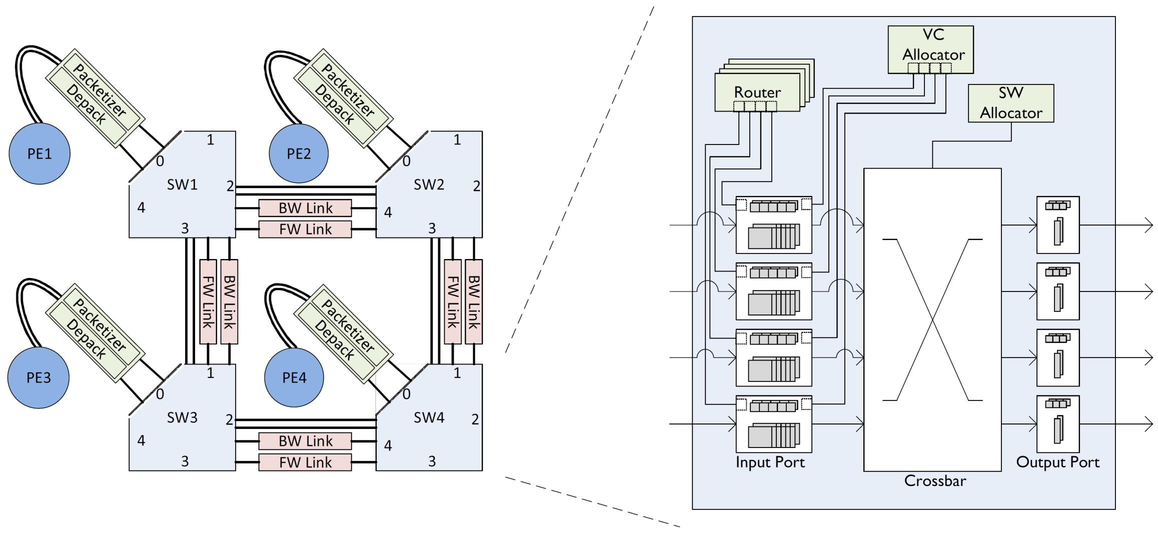


Figure 4.5: Capacity and Flit Level NoC Hardware Model Abstractions

acteristics of NoC are largely affected by switch components. The routing of a flit in a switch happens in 3 phases: 1) receipt of the flit and routing, 2) transferring via crossbar and allocation, and 3) storage in output port and transmission. A higher-level (coarse-grain) abstraction of the switch contains simple queues in input port, a router component, and single buffered output ports. There are no switch allocations, no virtual channels, and no pipelining.

An example IO System, flit-level model of the switch is depicted in Figure 4.5. At this resolution, pipelining, competition (for allocating resources), crossbar, and virtual channels are introduced. Flits from different virtual channels may compete to allocate an output port VC or the crossbar for transfer. Since the hardware-level switch model is very large, we only focus on the input and output port models. Figure 4.6 depicts a single input port and a single output port modeled in high resolution. The input/output ports contain 2 virtual channels each possessing its own status array and flow control logic. In this particular model, we used a circular buffer for the input port and a single buffer for the output port and on/off flow control policy.

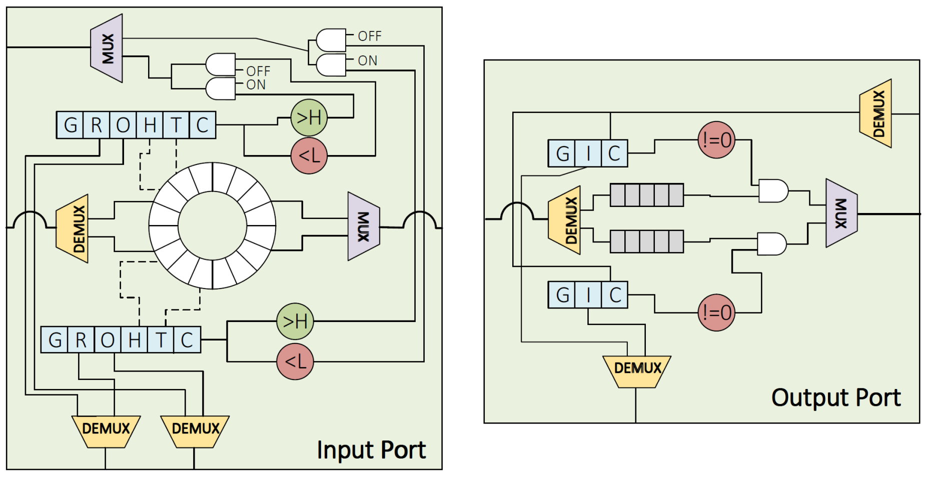


Figure 4.6: High-resolution Models of Input/Output Ports in the Switch

The input port receives 3 inputs (marked by the 3 DEMUXs) from three sources: router, VC allocator, and upstream switch. The two outputs (marked by the MUXs) are for outgoing flits and the flow control signal. As for the output port, there are 3 inputs for incoming flits, flow control signal, and receiving the input VC. There is only one output, which transmits the routed flits to the downstream switch (via a designated virtual channel).

4.2.3 Network Interface

The Network Interface (NI) component has two main functions: packetizing and depacketizing. The packetizing function, converts streams of data coming from the processing element into packets (flits) to be transmitted on the network. The depacketizing function converts flits of data into streams to be sent to the processing elements. Here, we model the Network Interface (with Packetizer and Depacketizer as inner components).

In the Capacity level, the NI is not independently modeled. A component in charge of converting streams of data to flits and vice versa is contained in PE. In this

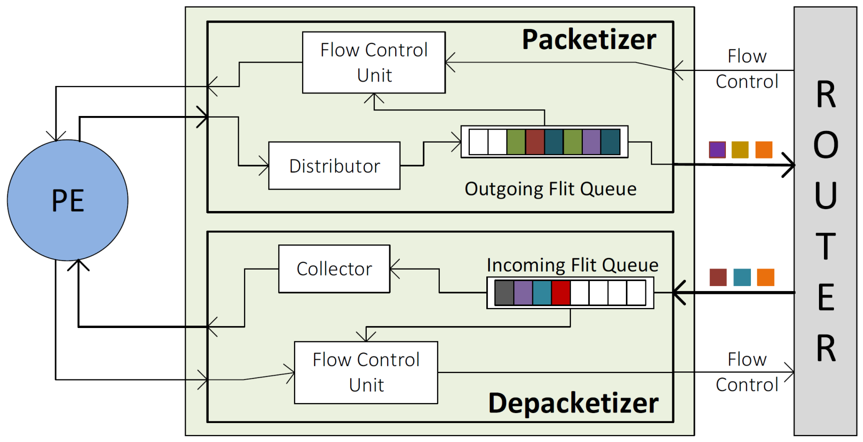


Figure 4.7: IO System, Flit-level Model of NI

level, streams of data are communicated with the central processing unit and packets with the router module. New capabilities are added to the IO System Flit-level NI. First, the data it communicates with the network is changed to flits. Second, the NI is modeled as composite, containing Packetizer and Depacketizer components. NI is placed between PE and switch. Flow control mechanism is added to the NI to ensure that neither the switch nor the PE are overwhelmed with flits/data. One way of designing such flow control mechanism is using ON/OFF method. For this, new links are connected from Router and PE to Packetizer/Depacketizer modules specifically intended for flow control to prevent data loss (see Figure 4.8).

Finally, the Packetizer/Depacketizer can be modeled in higher resolution by adding components such as Serializers, Deserializers, Distributers, and Collectors, and OR gates [14]. Also, the NI is equipped with multi-channel communication. As shown in Figure 4.8, there could be several wires to and from the PE or the Switch. However, the functionalities (such as the flow control) remain similar as those in Figure 4.7.

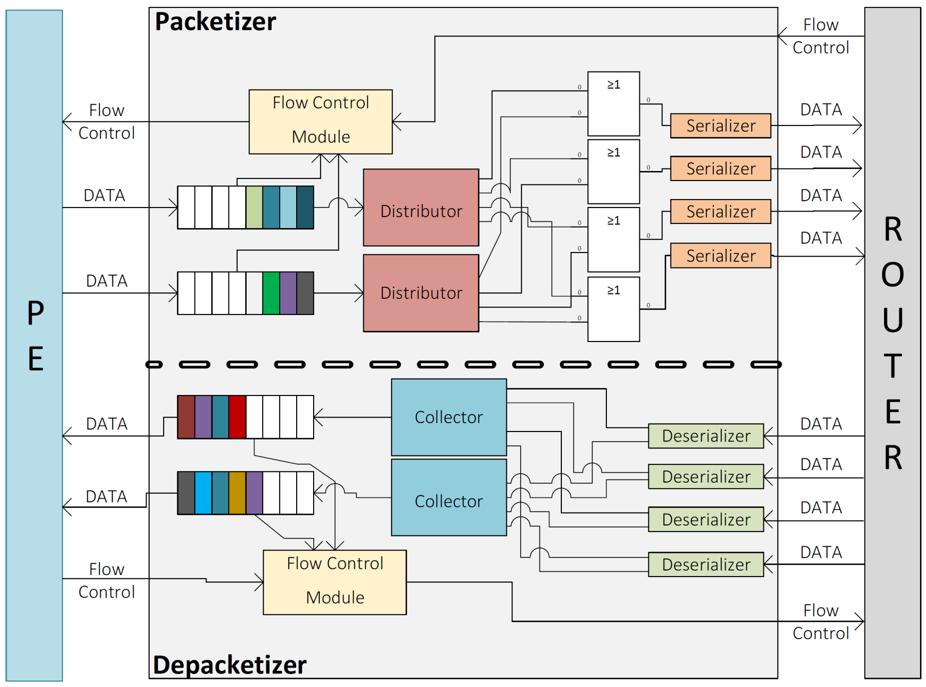


Figure 4.8: IO System, Hardware-level Model of NI

Chapter 5

CONSTRAINED MODELING AND VERIFICATION

To ensure correctness of the design and the implementation of the model, every model has to be validated and verified. One important question that this research is attempting to answer is the impact MRM and V&V have on one another. Verification and validation are necessary methods to ensure correctness when moving toward fine-grain models.

In contrast with multiresolution modeling, discussed in Chapter 4, our contribution in this section is not general and can be applied to any system modeled with discrete event methodology. For verification of DEVS models, we propose an approach toward constrained modeling and a DEVS-based model checking protocol.

In this chapter, we start with clarifying why Parallel DEVS modeling methodology and simulation protocol (as they are) are not well-suited for model checking in Section 5.1. Then, in Section 5.2 we discuss constrained modeling and how DEVS formalism is enhanced to support model checking. Later, in Section 5.3 we discuss the model checking protocol and how the model checking engine uses the simulation engine to verify properties of a Constrained-DEVS model. Additional features of Constrained-DEVS and the model checking protocol are articulated in Section 5.4. Finally, in Section 5.5, we discuss how MRM and V&V coexist for modeling, validation, and verification of DEVS models.

5.1 DEVS and Model Checking

In this section, we discuss how systems are specified using DEVS, how they are executed for simulation, and what is required for supporting model checking.

An illustrative example of a circular buffer is provided to show the modeling concepts discussed in the remainder of this chapter. Circular buffers are a common method of buffer implementation in hardware chips. We present this model from the library of Network-on-Chip (NoC) DEVS models (Gholami and Sarjoughian, 2016).

5.1.1 DEVS Modeling

Discrete EVent System Specification (DEVS) is a hierarchical, continuous time formalism devised for modeling and simulation of reactive systems. Systems can be modeled as a set of communicating automata in a hierarchical fashion using atomic and coupled DEVS models.

$$Atomic = \langle X^b, S, Y^b, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda, ta \rangle.$$

In this description, the input ports/events, output ports/events, and sequential state set are represented by X^b, Y^b, and S. External transition function is defined as $\delta_{ext} : Q \times X^b \rightarrow S$ where $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$. This function is a mapping between the occurrence of a bag of external events on one or more input ports and the sequential state set at any instance of time. The internal transition function, $\delta_{int} : S \rightarrow S$, defines how the model reacts to internal events. The confluent function, $\delta_{conf} : Q \times X \rightarrow S$, handles the occurrence of simultaneous (internal and external) events. The output function ($\lambda : S \rightarrow Y^b$) specifies output generation by mapping the state set to bag of output events on one or more output ports at any instance of time. Finally, the time advance function, $ta : S \rightarrow \mathbb{R}^+_{0,\infty}$, specifies the timing behavior of the system. In the remainder of this dissertation, X and Y replace X^b and Y^b for brevity.

Hierarchical structures in DEVS are made possible through coupling input and output ports of atomic/coupled models subject to no direct feedback coupling. Cou-

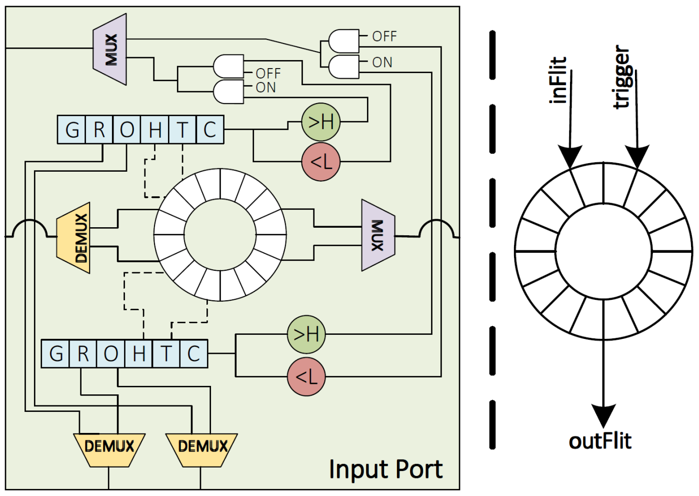


Figure 5.1: NoC switch input module utilizing a circular buffer (left) with its simplified diagram (right)

pled models do not contain state information; they only specify how components are placed and connected to one another in a strict hierarchical tree.

$$Coupled = \langle X, Y, D, \{M_d\}, EOC, EIC, IC \rangle.$$

In the description of DEVS coupled models, X and Y remain as input and output ports/events. D is the index set (component names) for internal atomic/coupled models. $\{M_d\}$ is the set of internal atomic/coupled models for which the index set D. Finally, the three sets EOC, EIC, and IC contain a set of external output port couplings, a set of external input port couplings, and a set of internal couplings (for internal couplings between atomic/coupled models within $\{M_d\}$), respectively.

To see DEVS in action, we provide a model of an NoC circular buffer. Circular buffer is one way of implementing the storage capability in an input module of an NoC router (Ni *et al.*, 1998). It is utilized to simplify the operation of NoC with multiple virtual channels. These buffers hold flits (packets of data) waiting to be routed. Figure 5.1 shows the circular buffer in an NoC input module.

We model a simplified version of the circular buffer with two input ports (i.e., one for storing input data (flit) and another for triggering sending out stored flits) and one output port for outputting flits. The buffer operates in FIFO mode (see Listing 5.1). Note that the modeling presented in Listing 5.1 is a DEVS model and only appropriate for simulation. Later, we present a model of circular buffer in Constrained-DEVS and its realization in DEVS-Suite suitable for both model checking and simulation.

Listing 5.1: Model of a Single Channel Circular Buffer in DEVS

$$S = \overbrace{\{Active, Idle\}}^{\text{phase}} \times \overbrace{\sigma}^{\text{sigma}} \times \overbrace{\{0,1\}^*}^{\text{flitBuffer}} \times \overbrace{\mathbb{N}}^{\text{head}} \times \overbrace{\mathbb{N}}^{\text{tail}} \times \overbrace{\beta}^{\text{outFlit}}$$

$$X = \big\{\big(inFlit, \{0,1\}^*\big), \big(trigger, 1\big)\big\}$$

$$Y = \big\{(outFlit, \{0,1\}^*)\big\}$$

$$\delta_{ext}\big((Idle, \sigma, flitBuffer, head, tail, \varnothing), e, (inFlit, x)\big) = \begin{cases} (Idle, \infty, flitBuffer.x, head, tail+1, \varnothing) & \textbf{if } head \neq tail \\ (Idle, \infty, flitBuffer, head, tail, \varnothing) & \textbf{if } FULL \end{cases}$$

$$\delta_{ext}\big((Idle, \sigma, flitBuffer, head, tail, \varnothing), e, (trigger, x)\big) = \begin{cases} (Active, .5, flitBuffer, head+1, tail, flitBuffer.head) & \textbf{if } head \neq tail \\ (Idle, \infty, flitBuffer, head, tail, \varnothing) & \textbf{if } EMPTY \end{cases}$$

$$\delta_{conf}\big((phase, \sigma, flitBuffer, head, tail, \beta), e, (PORT, x)\big) = \delta_{ext}\big(\delta_{int}, 0, (PORT, x)\big)$$

$$\delta_{int}\big((phase, \sigma, flitBuffer, head, tail, \beta) = (idle, \infty, flitBuffer, head, tail, \varnothing)$$

$$\lambda(Active, \sigma, flitBuffer, head, tail, \beta) = (outFlit, \beta)$$

$$ta(Active, \sigma, flitBuffer, head, tail, \beta) = 0.5$$

In the state set (S), variables for head index, tail index, and a data structure for holding the data is required. In this model, the receipt of a new flit invokes the external transition function (δ_{ext}). This transitions results in storing the flit and increasing the tail index. A trigger event invokes the external transition function which ultimately causes outputting the head of the queue in the description of the output function (λ). Obviously, inserting to a full buffer and obtaining from an empty one is not possible. We set the value of time advance function (ta) to 0.5 in state *Active*, which means, upon receiving the *trigger* signal, it takes 0.5 cycle for the buffer to output the head flit.

This model does not contain hierarchy and is modeled as an atomic model. We chose to exclude hierarchy from the example since this would be sufficient to show the model checking protocol with respect to the simulation protocol.

5.1.2 DEVS Simulation

The simulation protocol is responsible for executing DEVS atomic models. The reason we included this section is that we incorporate DEVS-Suite simulation engine within the model checking engine. Our state exploration protocol wraps around the simulation protocol and uses it for cycle-by-cycle execution. The mechanism will be elaborated later in this dissertation.

DEVS-Suite mechanism for DEVS execution relies on simulator and coordinator protocols. The execution of every atomic model is handled by its dedicated simulator module; similarly, every coupled model is assigned its coordinator module.

Simulator and coordinator modules manage timing and choice of functions to be executed as well as sending and receiving input/output events for atomic and coupled models they supervise. The behavior of any atomic model, specified as functions, is always invisible to the simulator protocol. The control of the simulator module on the timing aspect of an atomic model is depicted in Figure 5.2. Time of last event (tL), time of next event (tN), and external input events are used by the simulator module to control the execution of an atomic model. Among the internal/external events belonging to all atomic models, the ones with the earliest scheduled time (i.e.,

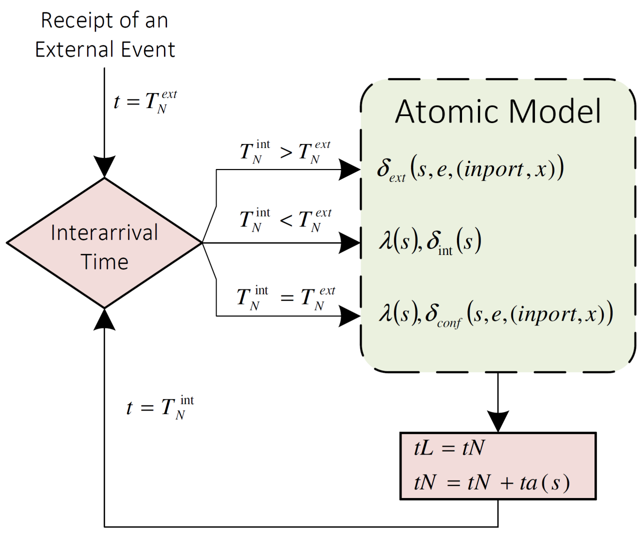

Receipt of an External Event
$t = T_N^{ext}$
Interarrival Time
$T_N^{\text{int}} > T_N^{ext}$
$T_N^{\text{int}} < T_N^{ext}$
$T_N^{\text{int}} = T_N^{ext}$
Atomic Model
$\delta_{ext}(s, e, (inport, x))$
$\lambda(s), \delta_{\text{int}}(s)$
$\lambda(s), \delta_{conf}(s, e, (inport, x))$
$tL = tN$
$tN = tN + ta(s)$
$t = T_N^{\text{int}}$

Figure 5.2: Time management in the coordinator module (with solid borders and in red) and invocation of the atomic model functions (with dashed borders and in green) in the simulator module

tN) will be executed. Depending on whether an external event, an internal event, or both occur at time instance tN, the appropriate functions within the atomic model are invoked.

This is the simulation protocol used in DEVS-Suite for DEVS model execution. However, this protocol is not appropriate for model checking. In Section 5.3, we describe how we incorporate this simulation protocol into the model checking protocol.

A sample simulation scenario for the model of circular buffer (presented earlier) is shown in Table 5.1. In this scenario a number of random external events are injected into the model. Some of the state variables and the output of the model are presented in the table to illustrate how the execution protocol simulates an atomic model.

Table 5.1: Sample Simulation Scenario for Circular Buffer

Time	inFlit	Trigger	Buffer Phase	Buffer Output	Head Index	Tail Index
0	-	-	*Idle*	-	0	0
2	A	-	*Idle*	-	0	1
3	B	-	*Idle*	-	0	2
3	C	-	*Idle*	-	0	3
4	D	1	*Active*	-	0	4
4.5	-	-	*Idle*	A	1	4
6	E	1	*Active*	-	1	5
6.5	-	1	*Active*	B	2	5
7	F	1	*Active*	C	3	6
7.5	-	-	*Idle*	D	4	6

5.1.3 Requirements for DEVS Model Checking

So far, we analyzed the modeling and simulation processes for DEVS specification. The question we would like to answer in this section is how can one use the DEVS specification and the simulation protocol (as presented earlier) for model checking?

But before answering the question, we first present the requirements that are needed for model checking. We also highlight expressiveness of the DEVS formalism and empowering DEVS-Suite to support Constrained-DEVS modeling and verification.

Considering any DEVS model's state space as a graph with transitions as arcs and states as nodes, the entire graph must be explored. Therefore, we need to ensure that the model possesses a finite state space. This requires the number of internal/external transitions (as arcs) and states (as nodes) to be finite.

For the finite internal transitions requirement, the time advance function for every atomic model must be defined relative to a discrete time base. Given $|S| < \infty$ for $ta : S \rightarrow R'$, where R' is a finite set (i.e., $R' \in \mathbb{R}$), there can exist a finite number of internal transitions. For the external transitions, DEVS allows input events to be received at any arbitrary instance of time. Therefore, external events are constrained to arrive only at discrete time instances. Also, the values for any input port must be bounded. As for the last condition, we use bounded state variables for representing state. These three constraints are discussed in more detail in Section 5.2.

DEVS-Suite as a simulation engine for DEVS models cannot enforce the above three constraints. Therefore, the changes we proposed (and discussed in detail in Section 6) are required for DEVS-Suite to support model checking.

We wish our tool (for modeling, simulation, and model checking) to support non-determinism, stochasticity, complex data transfer (information flow) and property checking capabilities beyond those supported in LTL, CTL, or TCTL. The restrictions of other approaches (DEVS-based and non-DEVS-based formalisms) for simulation and model checking is discussed in Section 3.

We will come back to the example of circular buffer presented in Listing 5.1. We will model it in Constrained-DEVS, realize it in DEVS-Suite, and perform experimentations in Chapter 6.

5.2 Constrained modeling

Model verification in DEVS entails four additional features (relative to the features already existing in DEVS) within the DEVS M&S framework: 1) bounded state configuration, 2) bounded input port configuration, 3) finite number of internal/external events, and 4) verification protocol. The first three are required to constrain state-space. Here we explain in detail our method for adding these four features to DEVS. In addition to these four features, we discuss two additional features which we view as important for a model checking engine. One for data exclusion, which contributes to reducing the size of the state space even further and the other for property statement language which is the way functional requirements are encoded for the verifier to check.

5.2.1 State Configuration

In theory, state variables and input values can be unbounded and can take any value. The verifier needs to know about the value set of each state variable. We bring value constrains to state variables. The verifier later leverages this Metadata for model verification.

State variables can be *Atomic* or *Compound*. An atomic state variable can only be of certain types including Character, Enumeration, Integer, and Boolean. Compound states are any combination of atomic and/or compound states. We use *regular expressions* to define a more compound state variables. As an example, consider a queue of size 8 that can hold strings (each of size 24). The specification is as follows.

$$\text{Atomic state:}\mathit{Char} \tag{5.1}$$

$$\text{Compound state 1:}\mathit{String} = (\mathit{Char})^{24} \tag{5.2}$$

$$\text{Compound state 2:}\mathit{Queue} = (\mathit{String})^{8} = ((\mathit{Char})^{24})^{8} \tag{5.3}$$

The first one is an atomic state for one character of a string. The type of the atomic state is *Char*. Each cell of the queue can hold a string of size 24. The second equation formulates the state variable of a string of size 24 in the form of a regular expression. Finally, the third equation is the compound state of the queue holding 8 strings of size 24.

A more advanced compound example would be a complex data structure which contains an array of string (of size 8 which holds strings of size 24) and a Map of integers and strings (integers of under 10 and strings of size 4).

$$\text{Array of strings:}((\mathit{Char})^{24})^{8} \tag{5.4}$$

$$\text{Map:}[(1|2|3|4|5|6|7|8|9) \in \mathit{Integer}] \times (\mathit{Char})^{4} \tag{5.5}$$

$$\text{Map:}((\mathit{Char})^{24})^{8} \times [(1|2|3|4|5|6|7|8|9) \in \mathit{Integer}] \times (\mathit{Char})^{4} \tag{5.6}$$

The verifier can easily calculate the number of states for the aforementioned state space and iterate through all of them. However, for atomic DEVS model, state variables are not the only elements that decide the state space. The ports and the time also affect the size of the state space.

5.2.2 Port Configuration

Similar to state variables, ports require accurate specification for their type and value sets. However, not all ports require bounded specification. Only the external input ports (coming from out of the model) require such specification and all internal couplings (input or output ports) are left unspecified. The reason for this is that the internal couplings are between atomic/coupled models. Model checking engine does not have control over these ports. These ports are driven by their source models. The model checking engine, can only manipulate the external inputs in order to iterate the entire state space.

The method introduced in for bounded state configuration can be applied here for bounded port configuration as well. Ports can transfer atomic or compound data types from user (or an external model) to the models under inspection. Value sets for these ports can also be specified via regular expressions. In the specification of input ports, an additional *NULL* ($\varnothing$) value is always needed for all cycles that the port carries no data. Therefore, for an input port of type string (of size 5), the regular expression defining it will look like the following.

$$\text{Port State Variable:}(Char)^5 \cup \varnothing \tag{5.7}$$

Considering 127 possible values (not null values) for *Char* the number of possible combinations of input (possible number of events) is:

$$\text{Port State Count:}127^5 + 1 = 33,038,369,408 \tag{5.8}$$

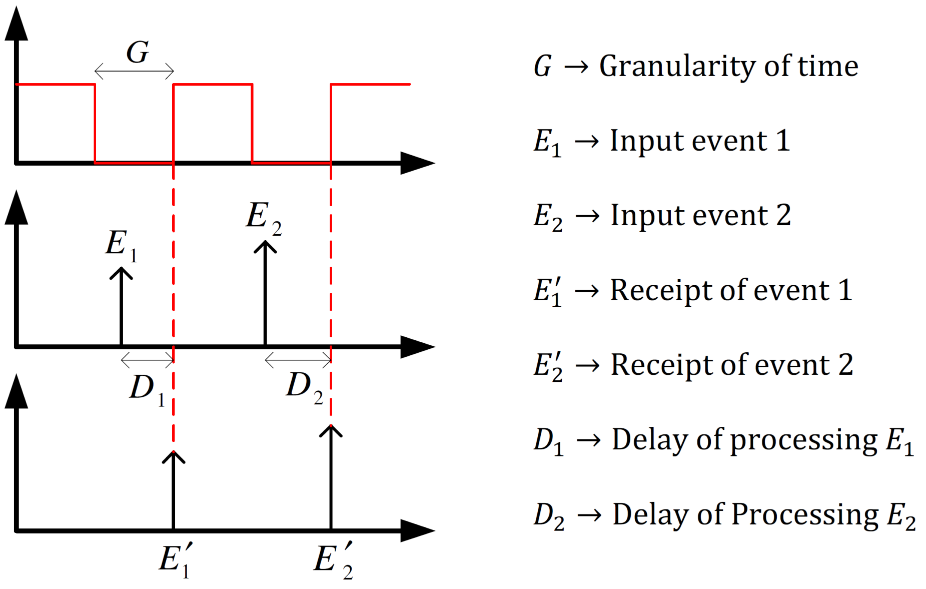
Any of these events can be applied to any given state of the model. The verification engine should apply all possible combinations of events to all reachable combinations of state variables for a reachability analysis.

5.2.3 Finite Number of Internal/External Events

As explained in Section 5.1.3, the number of internal transitions can be made finite by banning the use of continuous time-base for Time Advance (ta) function.

As for the external events, the receipt of external input events should become discrete to limit the number of transitions from each state. Similar to internal transitions, the use of a continuous variable for elapsed time (e) can result in having external output events at arbitrary time instances (i.e., infinite number of output events). The continuity of time for atomic/coupled DEVS models stays the same. As long as there are no converging sequences of time-advance function values, the state space of the model stays finite.

Discretizing the time of external inputs requires delaying event occurrences before they are received and processed e.g. if E_1 occurs at relative time instance $G - D_1$ and $G - D_1 \neq 0$ then E_1 is changed to $E\prime_1$ such that its time instance is the next integer multiplier of the models temporal resolution G (see Figure 5.3).



G
E_1
E_2
D_1
D_2
E'_1
E'_2
$G \rightarrow$ Granularity of time
$E_1 \rightarrow$ Input event 1
$E_2 \rightarrow$ Input event 2
$E'_1 \rightarrow$ Receipt of event 1
$E'_2 \rightarrow$ Receipt of event 2
$D_1 \rightarrow$ Delay of processing E_1
$D_2 \rightarrow$ Delay of Processing E_2

Figure 5.3: Discrete-time Processing of Events

5.3 Model Checking Protocol

A model checking algorithm is required to explore the state space of Constrained-DEVS models. What we elaborate in this section is the model checking algorithm for an atomic model. However, this approach can be applied to coupled models as well. The state of a coupled model is the collective state of the atomic/coupled models inside it. Having the state and the input ports configured using the bounded approach explained earlier, the same model checking algorithm can be applied to the coupled model as well.

A verifier model is in charge of the model checking activity. In order to verify the model entirely, the verifier should visit all states and all its transitions. Transitions between states could be the result of events coming in from outside the model or an internal transition within one or several components. As mentioned earlier, the verification algorithm should apply all possible combinations of inputs to all possible combinations of state variables to explore models state space fully. Therefore, the verifier holds two data structures for visited states and partially visited states. Partially visited states are those that the verifier should still investigate since some of their transitions are not explored yet. The visited states are those whose transitions are explored entirely. For safety analysis, another data structure for bad/invalid/unsafe states is instantiated and initialized by the user.

The state exploration algorithm repeats a set of tasks to cover the entire state space of the model. There are three state sets: unvisited (Q), visited (V), unsafe (U). The state exploration starts with a set of initial states stored in Q and repeats certain steps until Q is empty. During this process if any undesirable state is visited (stored in U), the process is terminated and the user is notified.

Listing 5.2, presents the state exploration protocol for Constrained-DEVS models (Gholami and Sarjoughian, 2017). MOD is the target model and GEN is the generator of external events. In steps 1 and 2, initial states and unsafe states are added to unvisited state set (Q) and unsafe state set (U), respectively. A while loop is devised to go through all reachable states (stored in partially visited state set). Steps 4 up to 17 are repeated as long as the the unvisited state set is nonempty. At each cycle of execution, a state is taken from Q (step 4) and all possible inputs are applied to it. Another while loop is responsible for applying all possible input values to the current state (steps 5-16). Remember from before that input ports are modeled using Constrained-DEVS as well. Therefore, the algorithm can determine and apply each of those input sets to the model. After setting the state of the model and giving the inputs to the GEN in steps 6 and 7, the simulator is called which simulates the model for one cycle. The resulting state is checked against the unsafe state set (steps 9-12). If the resulting state is an unsafe one, the algorithm terminates and alerts the user of this transition to an invalid state. Otherwise, in steps 13-15, if the resulting state is not seen before, it is added to the partially visited state set. After the termination of the inner loop (steps 5-16), all possible external events are applied to the chosen state (at step 6); therefore, the state is moved to the set of visited state set (step 17) and a new cycle is started (if Q is nonempty).

Listing 5.2: The State Exploration Protocol

```
Input: MOD: $Verifiable$, GEN: $VerifierGen$
Output: $invalidState$ : $StateVar$
Initialization: $instantiate$ Q, V, and U ; $invalidState \leftarrow null$

1. add MOD.$initialStates$ to Q
2. add MOD.$unsafeStates$ to U
3. while $Q \neq \emptyset$ do
4.     state-event $\leftarrow Q.head()$
5.     while state-event.$inputSet \neq \emptyset$ do
6.         MOD.$state \leftarrow$ state-event.$state$
7.         GEN.$output \leftarrow$ state-event.$inputSet.head(\,)$
8.         call $simulate(\,)$
9.         if MOD.$state \in U$ then
10.            $invalidState \leftarrow$ MOD.$state$
11.            return $invalidState$
12.        end if
13.        if MOD.$state \notin Q \wedge$ MOD.$state \notin V$ then
14.            add MOD.$state$ to Q
15.        end if
16.    end while
17.    add state-event to V
18. end while
19. return $invalidState$
```

In Figure 5.4 the model checking algorithm for reachability analysis is depicted. This visualization is identical to the protocol explained in Listing 5.2. The flowchart at the left hand side of this figure visualizes the model checking protocol provided earlier in Listing 5.2. The right hand side, provides the reader with a guide on the elements used in the flowchart. There are two datasets (Unvisited and Visited), MOD (the model under verification which could be atomic or coupled) and Gen (a generator which is automatically instantiated to inject inputs to MOD's input ports). There are 5 phases for the model checking protocol:

1. **Initialization:** during steps 1-1 and 1-2, initial states and unsafe states are identified and the initial ones are added to Q. Other steps during initialization are initialization of the EF (*Generator* and *Transducer*) and the *Verification Engine*.

2. **Main Loop (new state):** during steps 2-1 and 2-2, a state ($S_{current}$) is removed from Q and is set by the *Verification Engine* as the current state of MOD.

3. **Inner Loop (input injection):** during steps 3-1 and 3-2, a combination of possible input set is applied to MOD and the simulation engine is called to execute the model for a single cycle.

4. **Housekeeping:** at steps 4-1 up to 4-4, the resulting state is stored in the *Transducer*, the unvisited ones in Q, and if all input injections are applied to state $S_{current}$, that state is put into set V.

5. **Termination:** at steps 5-1 up to 5-3, the process examines termination conditions. The process continues at the *Main Loop* phase if Q is not empty. Otherwise, it proceeds to the final state and the transducer provides the user with trace and property checking results.

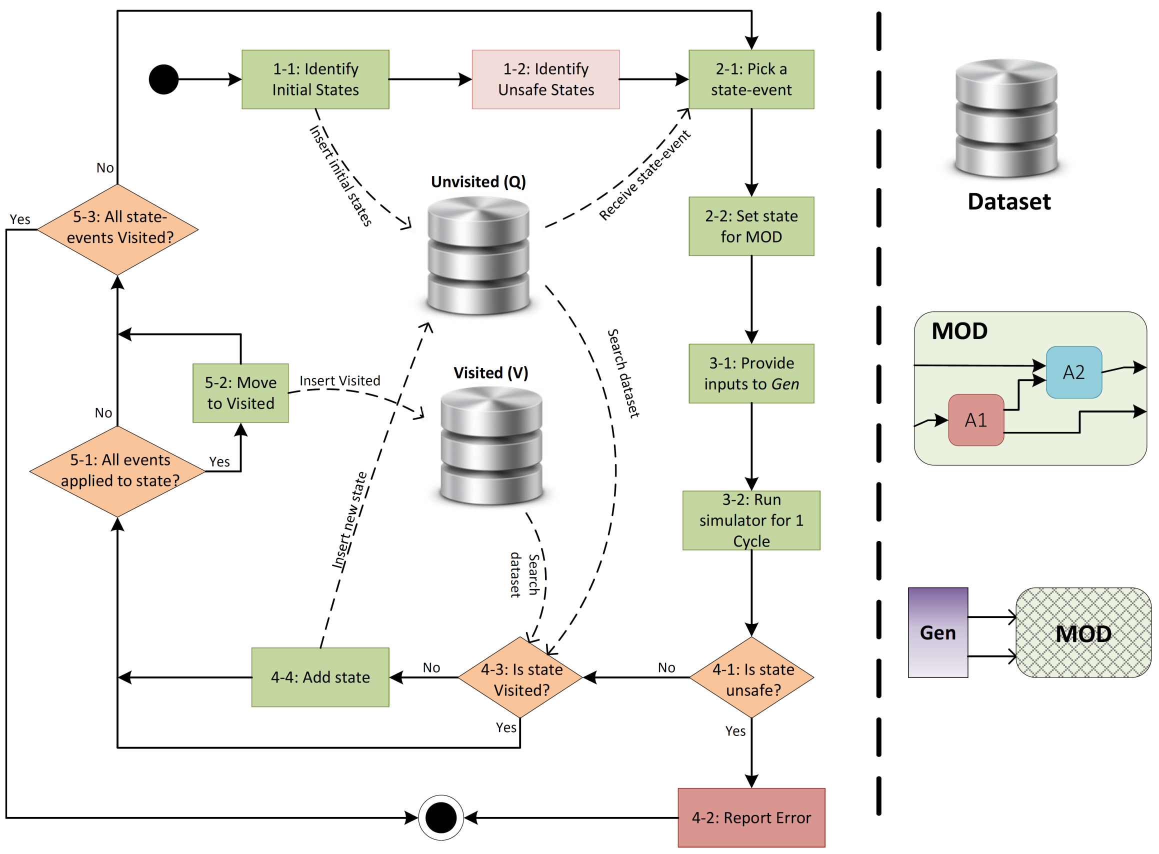


Figure 5.4: A Flowchart for the Constrained-DEVS Model Checking Protocol in DEVS-Suite

5.4 Additional Features of Constrained-DEVS Modeling and Verification

5.4.1 Selective State Exploration

One important aspect of modeling with Constrained-DEVS in DEVS-Suite is the flexibility it offers in the specification of state and input. Model state variables are marked as *explorable* or *unexplorable* (at initilization). An *explorable* state variable is considered as a part of the state of the model, therefore, it is explored and adds to the collective state space size. An *unexplorable* state variable operates very similar to the *explorable* one, however, it is invisible to the outside world.

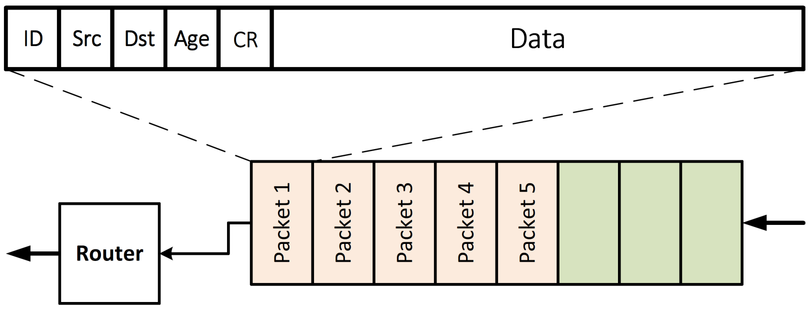


Figure 5.5: Waiting Queue of a Switch and the Structure of Packets It Stores

Within the model, both of *explorable* and *unexplorable* state variables are treated similarly. Their values are tracked, changed, and are used to make decisions. However, outside the model (particularly from the view point of the verification engine), a change in the value of an *explorable* state variable is considered a new state while the change in the value of the *unexplorable* one is ignored.

Depending on whether or not a state variable needs to be considered in state space, once should mark it as *explorable* or *unexplorable.* As an example, the queue of a switch component in NoC holds a number of packets waiting to be routed. This is depicted in Figure 5.5. As illustrated, each packets contains and ID (unique), source node, destination node, age, computation requirement, and data. Now, the state of a queue with capacity 8 is the possible combinations of data it can hold. This will make the number of states for this queue an immensely large number. For a network size of 6, 100 packets IDs, 5 values for RC, 3 values for age, and 2 bytes of data, queue's state space will be $100 \times 6^8 \times 6^8 \times 5 \times 3 \times 2^{16}$.

However, a switch does not care about the ID, source node, age, computation requirement, and the data which the packet carries. The only notable information for the switch is the destination node using which it routes the packet out. Therefore, the state space of the queue can be reduced to the combination of destination values it can hold. For example, for a network of 6 nodes, the state space of the queue can be rewritten as $6^8 \approx 1M$.

DEVS-Suite provides an easy way to change the role of each state variable at initialization. Therefore, the state space of the model can be easily managed for larger and more complex systems. Changing the role of a state variable does not impact the behavior of the model and its validation via simulation. This is a great capability compared with other approaches such as Timed Automata and Petri nets. In those modeling methods, one cannot change the size of state space of a model other than by creating a new model with fewer states.

Data Exclusion

In many systems verifying the operation of the system may not at all be related to the data it carries. In the case of Network-on-Chip, the data is only communicated and not at all modified in the network. Therefore, keeping the data in the flits as a part of state variables (in queues, links, etc.) is not necessary. Removing unnecessary data from the verification process could greatly improve the performance of verification as it can reduce the time of model checking for fully evaluating the state space.

For data exclusion, we use *Selective State Exploration*. In the case of flits, we consider source and destination nodes as part of the state space but remove data so that it would not contribute to enlarging the state space. Keep in mind that data will still be there and the simulation engine can use it for simulation purposes but the verification engine ignores the permutations of data from state space construction.

5.4.2 Trace Analysis

Using the protocol presented in Listing 5.2, the verification engine explores the entire reachable portion of the state space (starting from a set of initial states). During the exploration process, the transducer constructs the reachability graph. In

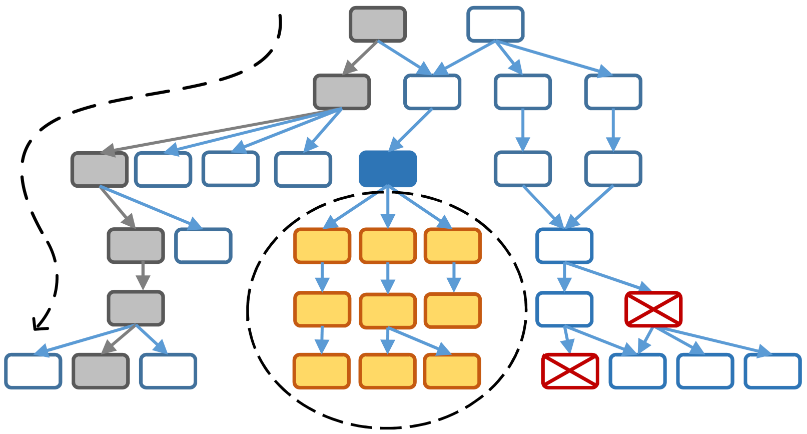


Figure 5.6: Trace Analysis on a Pruned Reachability Graph

this graph, nodes (set V) contain state information and edges (set E) are transitions. After the exploration is finished, the transducer may analyze the graph (using graph algorithms), draw conclusions, and verify properties.

By converting the graph to a set of traces (by pruning cycles), the transducer may verify properties related to individual states, paths, or subtrees. Figure 5.6 depicts a set of traces resulted from two initial states. Transducer verifies state-based properties by looking at individual states (nodes marked X), path-based properties by looking at one or more paths (dashed arrow), and tree-based ones by looking at a subtree (dashed circle).

In Chapter 6, we conduct state-based (unsafe/invalid states) and path-based (overall component utilization) property checking for models of Network-on-Chip.

5.4.3 Property Expressions

Model checking in particular and verification in general are purposed to check the validity of certain properties for the model. These properties are derived from functional requirements of the system and they may be related to various characteristics of the system, for example, they could be satisfying some performance measures or avoiding certain invalid states.

One problem, however, is how these properties are expressed for the model checking algorithm to process. There are different approaches for this. As mentioned, UPPAAL (Larsen *et al.*, 1997) uses a simplified version of Timed Computation Tree Logic (TCTL) (Alur *et al.*, 1993) to encode properties in four major categories: state formulae, reachability, safety, and liveness (Behrmann *et al.*, 2014). This although very useful and powerful, requires the modeler to learn another formalism/language for the model checking phase.

The other way is to use data collectors for this purpose. What we offer is the Experimental Frame (EF) (Rozenblit, 1991). The EF brings the concept of experimentation and measurement into the model by adding several components to the model of the system. Therefore, experimentation and measurement become parts of the model and not separate concepts. We use DEVS to specify the EF and Java to develop it within DEVS-Suite that gives the modeler (designer) flexibility and extensive support for creating experiments and conducting measurements. Normally, the EF contains one or more generators for creating experiments and transducers for collection and analysis of data.

The advantage of using the EF is that it can be used for both simulation and model checking without requiring any change in the process. In both simulation and model checking, the transducers are charged with the task of gathering data from the entire model (consequently receiving the collective state) and checking the data for satisfying some properties of interest. While in simulation only a subset of state space is explored, the entire reachable state space is visited and therefore verified in model checking. The modeler can implement complex properties with strong notations of DEVS and Java programming language. These include performance properties (maximum waiting time, average latency, etc.) that can be checked for network systems using this approach. For example, for measuring the collective delay of flits or the distribution of traffic, the transducer can have separate methods to gather the relevant data and validate whether they violate the expectation of the designer. The transducer validates one state at a time i.e., the models complete state space is explored and verified with the algorithm elaborated above.

5.5 Integrating MRM and V&V

In Figure 1.4, we depicted our approach toward multiresolution ESL design using V&V methods at each abstraction level. We also explained that we focus on the first three abstraction levels of NoC (interface, capacity, and flit). Here in Figure 5.7, we show in detail how V&V steps are carried out at each level of abstraction. At each level of abstraction, appropriate requirements are considered in modeling the application software, the network software, and the hardware. A DEVS model is specified based on those requirements, which is used for the validation phase. By adding type/value set information to the port and state variables, the DEVS model is transformed into Constrained-DEVS and is used for verification. Keep in mind that certain verification and validation efforts can be reused if homomorphic relationships are found between two models. This was discussed in detail in Section 7.1.1.

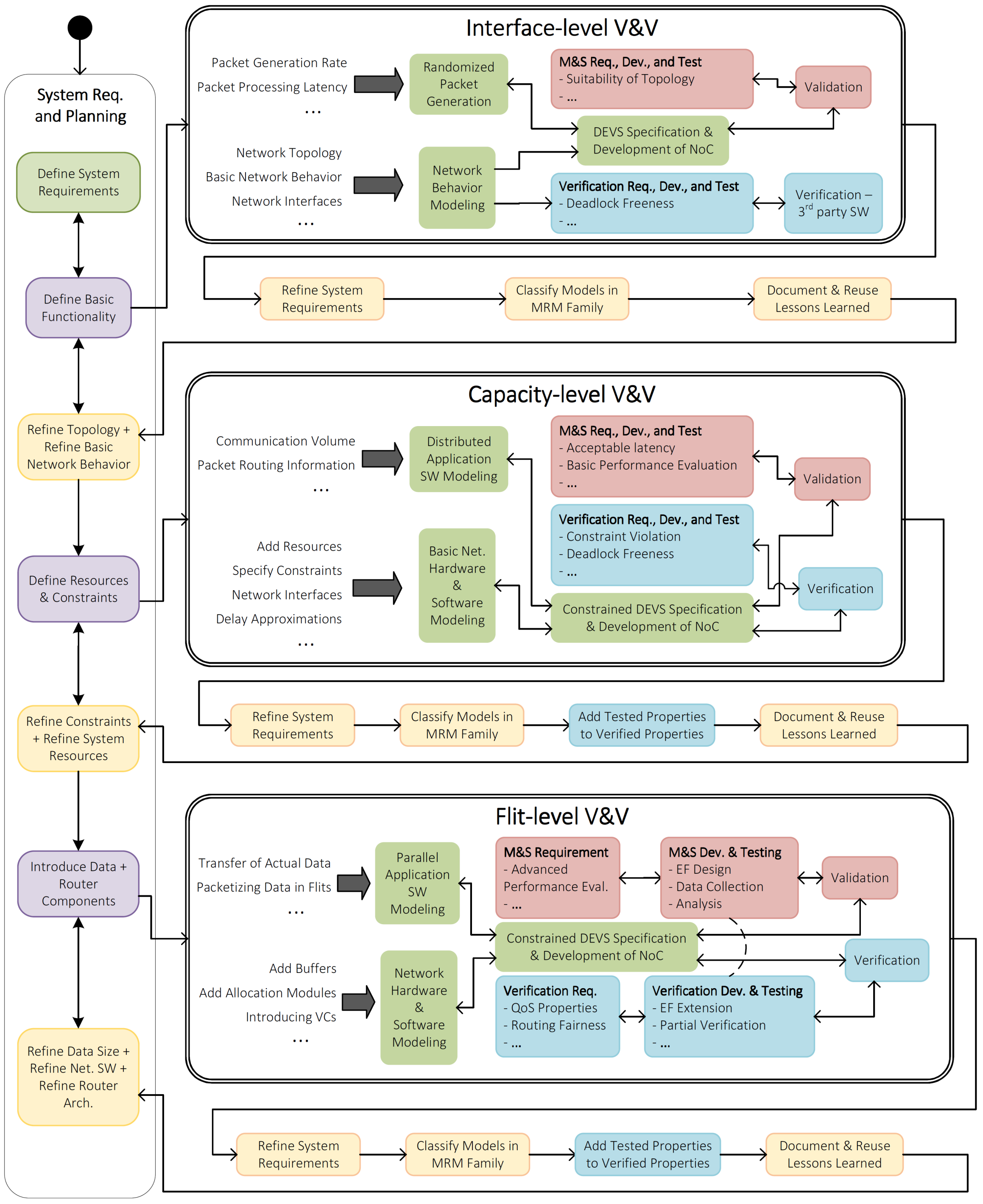

System Req. and Planning
Define System Requirements
Define Basic Functionality
Refine Topology + Refine Basic Network Behavior
Define Resources & Constraints
Refine Constraints + Refine System Resources
Introduce Data + Router Components
Refine Data Size + Refine Net. SW + Refine Router Arch.
Interface-level V&V
Packet Generation Rate
Packet Processing Latency
...
Randomized Packet Generation
M&S Req., Dev., and Test
- Suitability of Topology
- ...
Validation
DEVS Specification & Development of NoC
Network Topology
Basic Network Behavior
Network Interfaces
...
Network Behavior Modeling
Verification Req., Dev., and Test
- Deadlock Freeness
- ...
Verification – 3rd party SW
Refine System Requirements
Classify Models in MRM Family
Document & Reuse Lessons Learned
Capacity-level V&V
Communication Volume
Packet Routing Information
...
Distributed Application SW Modeling
M&S Req., Dev., and Test
- Acceptable latency
- Basic Performance Evaluation
- ...
Validation
Verification Req., Dev., and Test
- Constraint Violation
- Deadlock Freeness
- ...
Verification
Add Resources
Specify Constraints
Network Interfaces
Delay Approximations
...
Basic Net. Hardware & Software Modeling
Constrained DEVS Specification & Development of NoC
Refine System Requirements
Classify Models in MRM Family
Add Tested Properties to Verified Properties
Document & Reuse Lessons Learned
Flit-level V&V
Transfer of Actual Data
Packetizing Data in Flits
...
Parallel Application SW Modeling
M&S Requirement
- Advanced Performance Eval.
- ...
M&S Dev. & Testing
- EF Design
- Data Collection
- Analysis
Validation
Constrained DEVS Specification & Development of NoC
Verification
Add Buffers
Add Allocation Modules
Introducing VCs
...
Network Hardware & Software Modeling
Verification Req.
- QoS Properties
- Routing Fairness
- ...
Verification Dev. & Testing
- EF Extension
- Partial Verification
- ...
Refine System Requirements
Classify Models in MRM Family
Add Tested Properties to Verified Properties
Document & Reuse Lessons Learned

Figure 5.7: MRM-based Chip Design at ESL Phase with V&V Steps at Each Abstraction Level

Chapter 6

IMPLEMENTATION

In Chapters 4 and 5, main theoretical contributions of this dissertation were introduced. Here we discuss the extensions made to DEVS-Suite for multiresolution Network-on-Chip modeling, supporting Constrained-DEVS, validation, and verification. As explained earlier, the first extension made to DEVS-Suite is specific to NoC. Base models are created to support NoC modeling at various resolutions. The other extensions, however, are general and applicable to any system model.

In this chapter, we first discuss the extensions made for Constrained-DEVS modeling, validation, and verification in Section 6.1. DEVS-Suite's library for multiresolution models of NoC is later discussed in Section 6.2. We modeled NoC in another popular modeling environment (UPPAAL v. 4.0.14) and compared it with an NoC in DEVS-Suite in Section 6.3. Later in Section 6.4, few examples are provided to the reader (including the circular buffer example in Chapter 5) to demonstrate DEVS-Suites ability in multiresolution Constrained-DEVS modeling, validation, and verification.

6.1 Constrained-DEVS and Model Checking in DEVS-Suite

Constrained-DEVS modeling and the model checking algorithm discussed in Chapter 5 are at the level of theory. They should be realized in a tool to be usable for system design. We have extended DEVS-Suite to support Constrained-DEVS modeling, simulation, and model checking of those models.

There exists a clear distinction between modeling and simulation in the architecture of DEVS-Suite. The modeling architecture in DEVS-Suite should be extended to

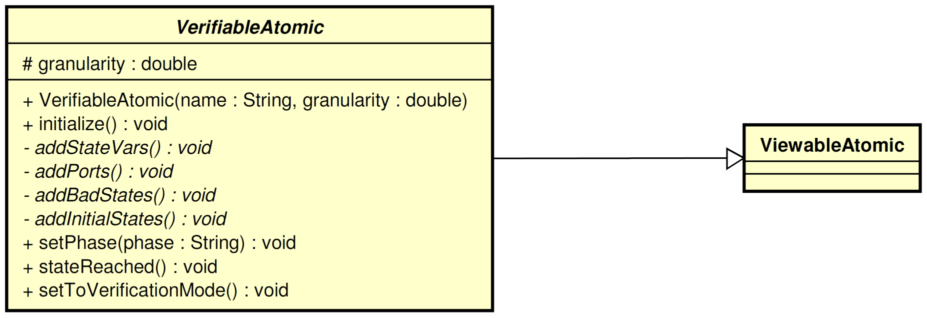


Figure 6.1: `VerifiableAtomic` Class for Validation and Verification

support Constrained-DEVS models. The simulation section of the tool is extended to provide execution (and model checking) of those Constrained-DEVS models. DEVS-Suite is extended to accommodate model checking by 1) developing Constrained-DEVS models and 2) their executions using the state exploration protocol provided in Listing 5.2. These two extensions are discussed in Sections 6.1.1 and 6.1.2. Later, in Section 6.1.3, we go back to the circular buffer model which was developed with DEVS in Chapter 5 and demonstrate how it is modeled in Constrained-DEVS and realized in DEVS-Suite for verification.

6.1.1 Constrained-DEVS Modeling in DEVS-Suite

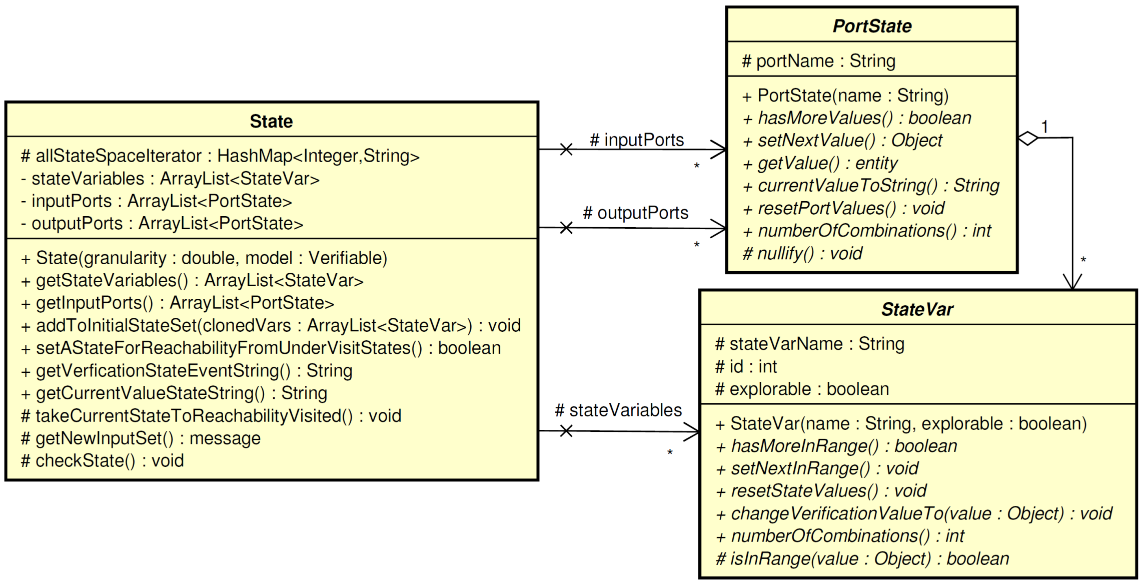
An atomic model for simulation is a class inherited from the `ViewableAtomic` class. Any attribute defined for any atomic model can be considered as its state variable. A variable type can be primitive (e.g., `int`, `float`, and `double`) or compound (e.g., `String`, `Map`, `List`, and user-defined). Although any of these data types are commonly used for simulation, they need to be constrained and identifiable for the model checking algorithm. For the model checking engine to work, it must be able to extract all state variables, all ports, and their values to be used for verification.

At the modeling level, base classes are added to support Constrained-DEVS.

At the atomic level, DEVS atomic models are represented in DEVS-Suite with a Java class called `ViewableAtomic`. It has predefined methods for external, internal, and confluent transition functions and time advance. A new base class called `VerifiableAtomic` is implemented which extends `ViewableAtomic` with some required verification means such as adding initial states, invalid states, and input/output ports. The class is shown in Figure 6.1. This summarized view of `Verifiable Atomic` also shows additional functionality for state transition checking (`stateReached`), initialization of state variables (`initialize`), and switching between simulation and model checking modes (`setToVerificationMode`).

This base class is capable of holding bounded state variables (defined by regular expressions) and using them in model execution. However, DEVS-Suite does not have support for bounded state variables. Therefore, some extensions are made. Figure 6.2 contains 3 classes necessary for state exploration. The `StateVar` abstract class is the basis for any state variable; in other words, any state variable that is required to be explored during model checking has to extend this class. Such bounded state variables is necessary for state exploration. Similarly, external ports should extend the `PortState` class. Similar to the `StateVar`, `PortState` implements bounded incoming ports. Instances of `message` class are communicated between models using ports. Each instance of message my contain a number of `content` objects which are the actual data communicated between two models. `PortState` ensures values taken by the `content` object are bounded. It are used by the verification engine to create all possible external events and injecting them into the model. `PortState` may contain one or more of `StateVar` instances for modeling bounded port values.

Another base class in relation to state variables is the *State* class. While a model may have a number of `StateVar` and `PortState` instances (for multiple state variables and input/output ports respectively), it can only have one one instance of *State*. The



PortState
# portName : String
+ PortState(name : String)
+ hasMoreValues() : boolean
+ setNextValue() : Object
+ getValue() : entity
+ currentValueToString() : String
+ resetPortValues() : void
+ numberOfCombinations() : int
# nullify() : void
State
# allStateSpaceIterator : HashMap<Integer,String>
- stateVariables : ArrayList<StateVar>
- inputPorts : ArrayList<PortState>
- outputPorts : ArrayList<PortState>
+ State(granularity : double, model : Verifiable)
+ getStateVariables() : ArrayList<StateVar>
+ getInputPorts() : ArrayList<PortState>
+ addToInitialStateSet(clonedVars : ArrayList<StateVar>) : void
+ setAStateForReachabilityFromUnderVisitStates() : boolean
+ getVerficationStateEventString() : String
+ getCurrentValueStateString() : String
# takeCurrentStateToReachabilityVisited() : void
# getNewInputSet() : message
# checkState() : void
# inputPorts
# outputPorts
# stateVariables
1
*
StateVar
# stateVarName : String
# id : int
# explorable : boolean
+ StateVar(name : String, explorable : boolean)
+ hasMoreInRange() : boolean
+ setNextInRange() : void
+ resetStateValues() : void
+ changeVerificationValueTo(value : Object) : void
+ numberOfCombinations() : int
# isInRange(value : Object) : boolean

Figure 6.2: *State*, *StateVar*, and *PortState* Classes for State Space Exploration

`State` class holds all state variables (instances of `StateVar` class), all ports and their possible values (instances of `PortState`), and data structures for visited and unvisited states (during state exploration). Figure 6.2 depicts the relationship among `State`, `StateVar`, and `PortState` classes.

The classes in Figures 6.1 and 6.2 show only some of their attributes and operations; including all their attributes, methods, and initializations is impractical. The functionalities left in the UML class diagram are related to the exploration functionality. More complete UML class diagrams, but without details of the classifiers, are provided in Section 6.1.2.

6.1.2 State Space Exploration Protocol

With the Constrained-DEVS modeling added to DEVS-Suite, next the model checking algorithm presented in Listing 5.2 must be added. The model checking algorithm uses the additional information provided in the models to conduct reachability analysis on the state space. The algorithm, with its main design concept and implementation, is highlighted here.

Before delving into extending the DEVS-Suite with execution of Constrained-DEVS models, major parts in the state exploration task are described. There following are needed for realizing the model checking protocol illustrated earlier in Figure 5.4:

$*$ **Verification Engine** : this is the module in charge of state exploration. It runs the protocol presented in Algorithm 1. Starting/ending the process is also the responsibility of this module. As explained earlier, the *Verification Engine* uses the simulation protocol to carry out state space exploration.

$*$ **Target Model** : MOD is the target model which the reachability analysis is applied to.

$*$ **Experimental Frame** : the *Generator* and *Transducer* pair which are in charge of generating possible combinations of input and collecting the state transition trace respectively.

$*$ **State Sets** : Q and V are required for keeping track of visited and unvisited states.

This protocol required adding some extensions to DEVS-Suite. These along with their relationship with classes (e.g., `StateVar` and `State`) for Constrained-DEVS are shown as a UML class diagram (see Figure 6.3). This partial class diagram highlights some of the classifiers and their relationships to support both simulation and model checking.

As explained earlier, the state variables should all be of type `StateVar`. This class mandates implementing bounds on the state variable along with an iteration capabil-

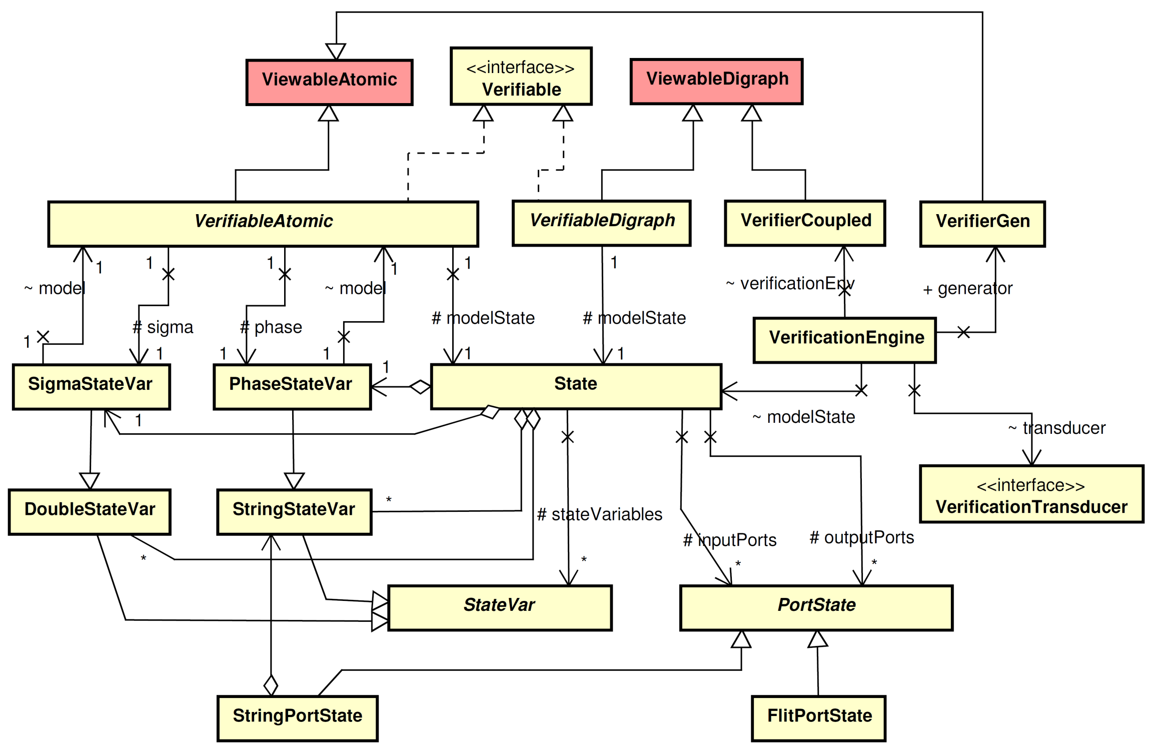


Figure 6.3: Partial UML Diagram of DEVS-Suite Modeling Class Structure (New Classes in Yellow)

ity (so that all values can be iterated). Two examples of state variables are provided in Figure 6.3: `DoubleStateVar` for double state variables and `StringStateVar` for string state variables. Other state variables can be added as long as they are inherited from the `StateVar`. Phase and Sigma (required for model checking and simulation) extend `StringStateVar` and `DoubleStateVar`, respectively.

The `State` class holds all state variables and belongs to all instances of `Verifiable Atomic` and `VerifiableDigraph`. The model under test can be an atomic or a coupled model which both implement the `Verifiable` interface.

Finally, the `VerificationEngine` class is instantiated for model checking scenarios and manages the entire process of verification. It instantiates the `VerifierGen` class which produces all possible combinations of input values, In addition, it creates

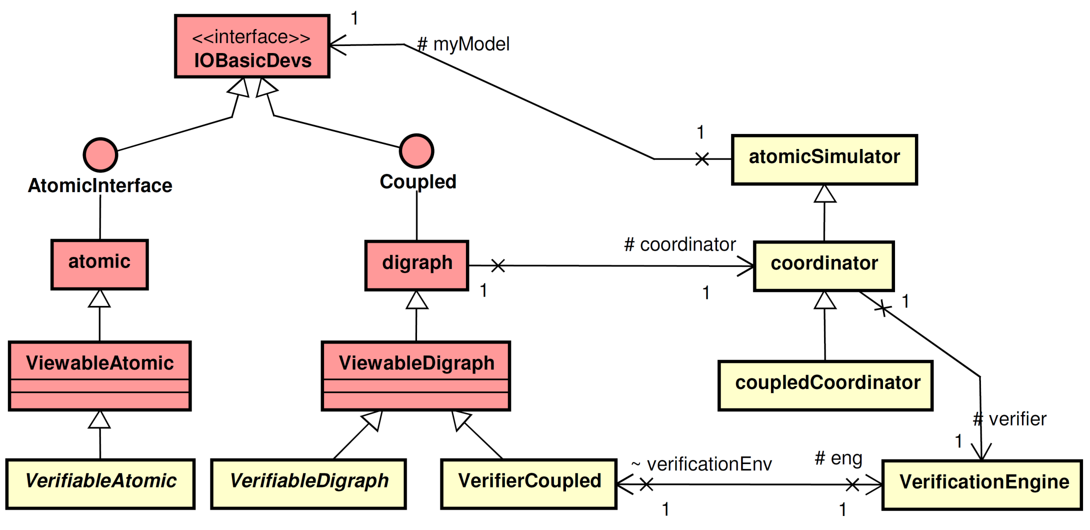


Figure 6.4: Partial UML Diagram of DEVS-Suite Model, Simulation, and Verifier Packages (New and Modified Classes in Yellow)

one or more instances of the Transducer class which are in charge of data collection, state-based analysis, and output validation. Finally, the responsibility of keeping track of visited and unvisited states and setting them (manually to the State class) for the model under test lies with `VerificationEngine`.

To keep the simulation functionality intact, we reuse DEVS-Suite atomic and coupled simulation capability. The `VerificationEngine` operates on top of these classes to facilitate model checking. Figure 6.4 provides an overview of the simulation package and how it relates to the modeling package. The bottom row of classes in the UML class diagram (plus minor changes without side effect in the `Coordinator` and `atomicSimulator` classes) pave the way for the added model checking functionality.

In order to reuse DEVS-Suite's animation and tracking capability, as illustrated in Figure 6.4, the `VerifiableAtomic` and `VerifiableDigraph` classes extend `Viewable Atomic` and `ViewableDigraph` classes, respectively that support animation as well as linear and superdense time trajectory generation (Sarjoughian and Sundaramoorthi,

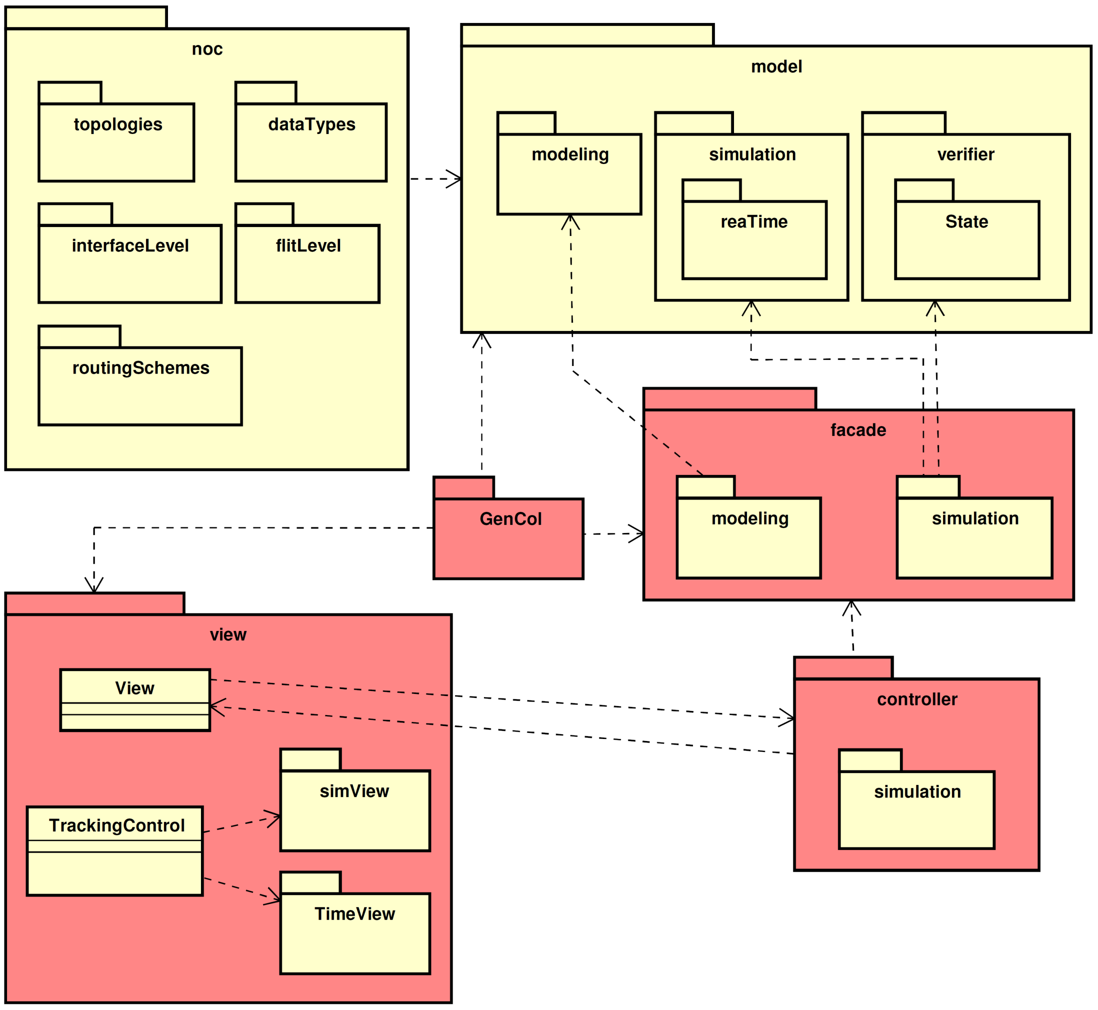


Figure 6.5: Partial Package Diagram of DEVS-Suite (New and Modified Packages in Yellow)

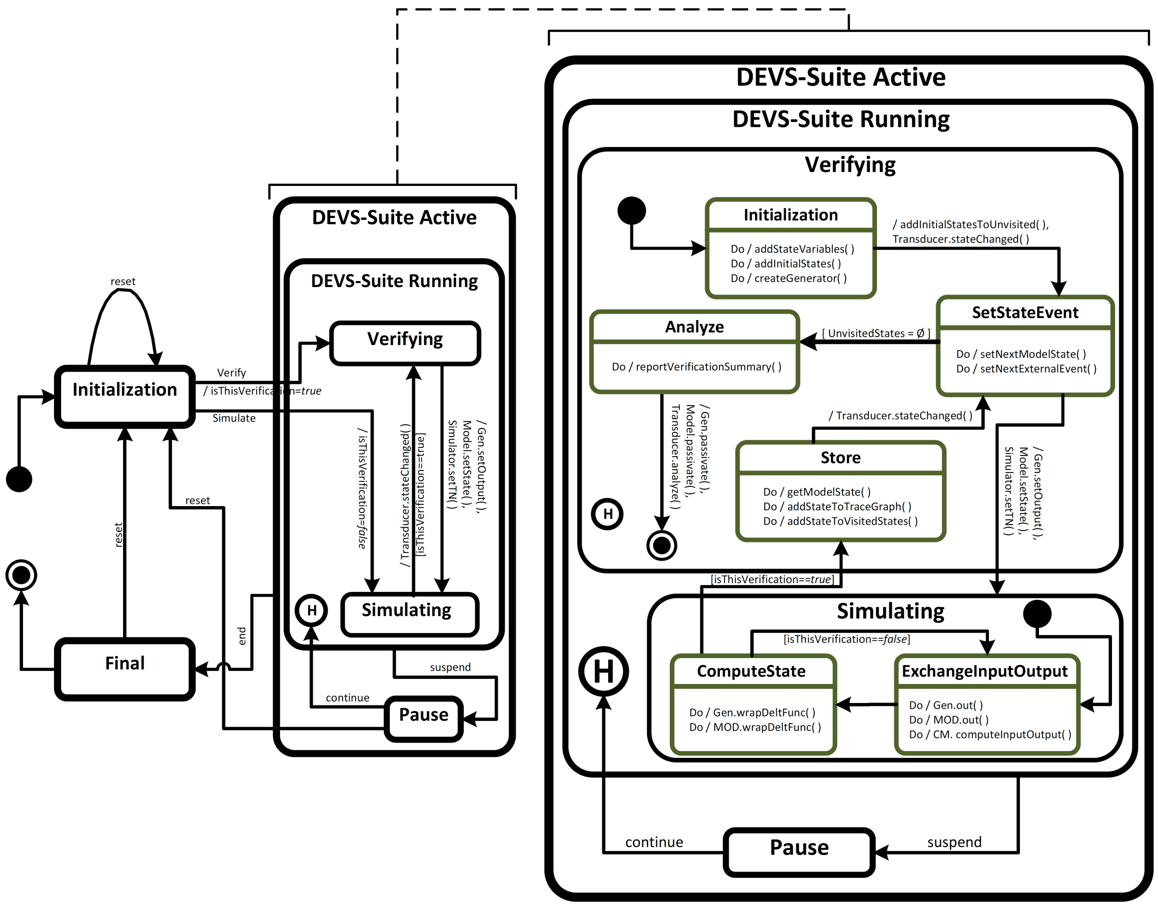
2015) and viewing at run-time. Also, this relationship facilitates reusing the simulation capability without requiring any changes to the Constrained-DEVS models. The package diagram depicted in Figure 6.5 illustrates the relationship between major packages of the extended DEVS-Suite. The `noc` package contains models of NoC created and experimented with later in this chapter.

The operation of the state exploration protocol as presented in Listing 5.2, is developed in DEVS-Suite. The execution of the verification protocol contains hundreds of calls and interactions. Here we present the reader with a statechart and a partial sequence diagram.

In the Statecharts diagram presented in Figure 6.6-a we show three major states in which DEVS-Suite may be in: *Initialization*, *Active*, and *Final.* To break it down further, an active DEVS-Suite may be in *Running* or in *Pause* mode and while in Running mode, DEVS-Suite is either in *Verifying* or *Simulating* mode. Transitions between these modes (in Figure 6.6-a) are all labeled with the signals that cause them and the actions that are executed due to them. Whether a model is being verified or simulated in the *Running* mode is decided at *Initialization.* The *simulate* or *verify* signals decide in which execution mode the model can be in. Other examples of signals are the *reset* and *suspend.* The *reset* signal (cause by a user action) always takes the model back to the Initialization state. Similarly, while in *Running* state (verifying or simulating a model), a *suspend* signal takes DEVS-Suite to *Pause* state. The *continue* signal brings the model back to the *Running* mode.

in Figure 6.6-b, we provide a detailed view of the *Active* state in which *Verifying* and *Simulating* compound states are shown in detail. The *Verifying* compound state, shows the phases that the verification engine goes through and the calls it makes to other processes. The process shown here is consistent with the exploration protocol in Listing 5.2 and its visualization in Figure 5.4. As explained earlier, the verification engine uses the simulation engine to run the model. This is evident here with the transition between the *SetStateEvent* (in *Verifying*) to the *Simulating* state and back. The verification engine picks a state event in *SetStateEvent*, the simulation engine runs the model for one cycle (in *Simulating*), and then the state is store in the suitable dataset (Visited or Unvisited) in the *Store* state.

The *suspend* signal brings DEVS-Suite to the *Pause* state from *Running.* In order to go back to the right state (depending on whether we were verifying or simulating



DEVS-Suite Active
DEVS-Suite Running
Verifying
Simulating
Initialization
Final
Pause
H
reset
Verify
/ isThisVerification=true
Simulate
/ isThisVerification=false
/ Transducer.stateChanged( ) [isThisVerification==true]
/ Gen.setOutput( ), Model.setState( ), Simulator.setTN( )
end
suspend
continue
Initialization
Do / addStateVariables( )
Do / addInitialStates( )
Do / createGenerator( )
/ addInitialStatesToUnvisited( ), Transducer.stateChanged( )
SetStateEvent
Do / setNextModelState( )
Do / setNextExternalEvent( )
Analyze
Do / reportVerificationSummary( )
[UnvisitedStates = ∅]
/ Gen.passivate( ), Model.passivate( ), Transducer.analyze( )
Store
Do / getModelState( )
Do / addStateToTraceGraph( )
Do / addStateToVisitedStates( )
/ Transducer.stateChanged( )
[isThisVerification==true]
[isThisVerification==false]
ComputeState
Do / Gen.wrapDeltFunc( )
Do / MOD.wrapDeltFunc( )
ExchangeInputOutput
Do / Gen.out( )
Do / MOD.out( )
Do / CM. computeInputOutput( )
(a)
(b)

Figure 6.6: DEVS-Suite V&V Statechart: (a) Partial Statechart Diagram for DEVS-Suite From Initialization to Termination, (b) Detailed View of the "DEVS-Suite Active" State (Containing Verification and Simulation Sub-states)

the model) when the *continue* signal is received, a history state is needed within the *Running* compound mode.

The sequence diagram in Figure 6.7 focuses on six major classes of the exploration protocol: `VerifiableAtomic` (MOD), `State` (containing the state variables and ports of MOD), `VerifierGen` (the generator of EF), `VerificationEngine` (orchestrator of the exploration process), and the two simulation classes uses by the verification engine: `atomicSimulator` and `coordinator`. Many objects and transitions within and among them that participate in a full cycle of state exploration must obviously excluded. The sequence diagram presented in Figure 6.7, contains the initialization and the exploration phases.

After the initialization phase is completed (state variables are identified, initialized, and put into the right datasets), the `coordinator` invokes the `Verification Engine` at every cycle to perform state exploration. The `VerificationEngine` sets a new state for MOD and selects a set of external events to be injected by the `VerifierGen`. The `coordinator` then kick starts a single-cycle simulation by calling on the `atomicSimualtor`. The `atomicSimulator` by itself calls the MOD to react to the external events and passage of time. Finally, the new state of the MOD is analyzed and added to the datasets for further analysis. This process is repeated until there are more states to visit.

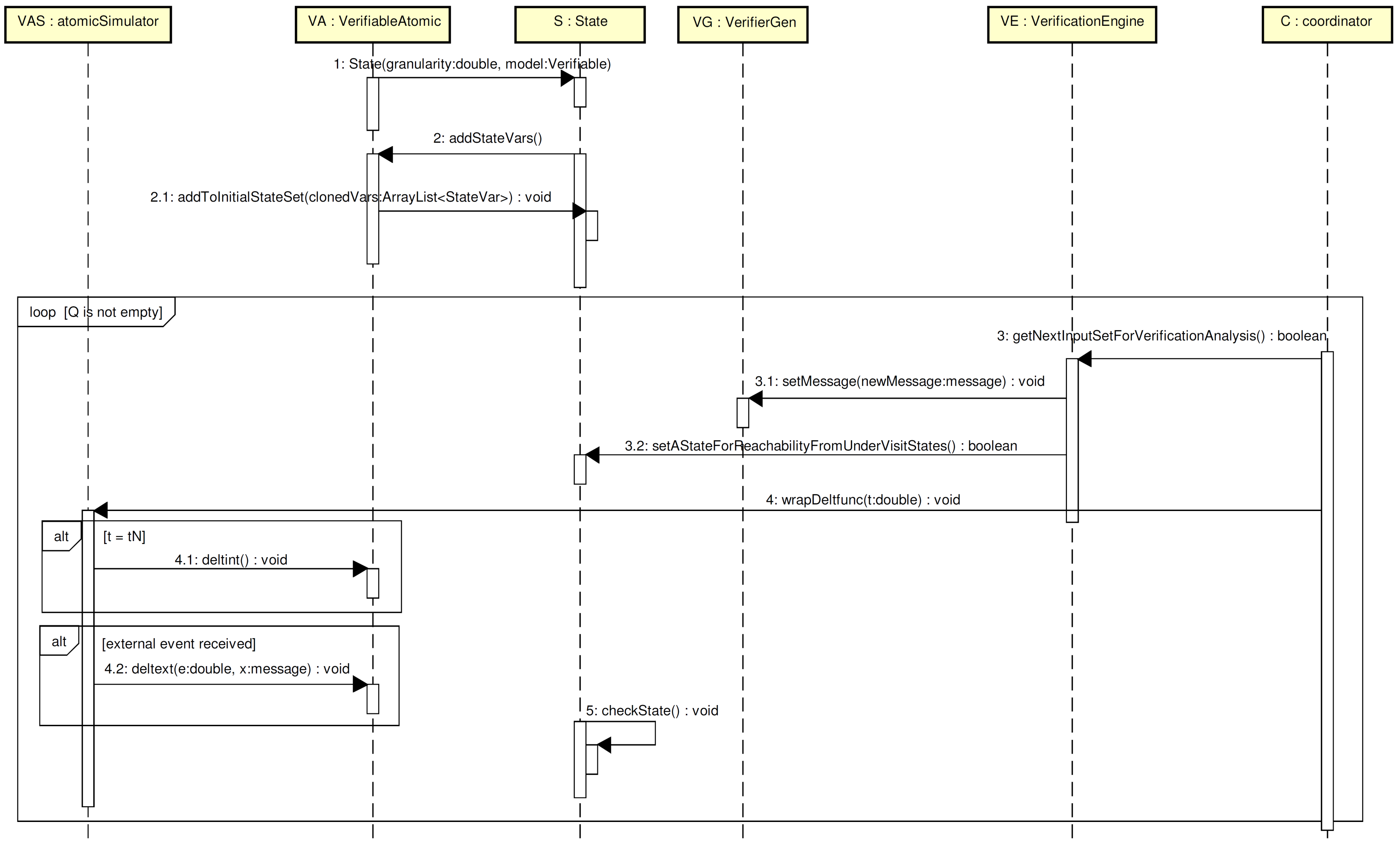

VAS : atomicSimulator
VA : VerifiableAtomic
S : State
VG : VerifierGen
VE : VerificationEngine
C : coordinator
1: State(granularity:double, model:Verifiable)
2: addStateVars()
2.1: addToInitialStateSet(clonedVars:ArrayList<StateVar>) : void
loop [Q is not empty]
3: getNextInputSetForVerificationAnalysis() : boolean
3.1: setMessage(newMessage:message) : void
3.2: setAStateForReachabilityFromUnderVisitStates() : boolean
4: wrapDeltfunc(t:double) : void
alt
[t = tN]
4.1: deltint() : void
alt
[external event received]
4.2: deltext(e:double, x:message) : void
5: checkState() : void

Figure 6.7: Partial UML Sequence Diagram Exemplifying the Execution of State Space Exploration Protocol

6.1.3 Circular Buffer Model

Here, we come back to the model of circular buffer in Chapter 5. This time it is modeled with Constrained-DEVS and implemented it in DEVS-Suite. This model is simple and is intended to highlight how the modeling, development, and verification phases work. The specification does not change much. Only restrictions are placed on state and input port values. In the case of circular buffer, we assume that the size of the buffer is 8. Therefore, *head* $\in \{0, 1, 2, ..., 7\}$, *tail* $\in \{0, 1, 2, ..., 7\}$, and *flitBuffer* $\in (\{0, 1\}^{24})^{8}$ (flits are assumed to be 24 bits). We also added a state variable called *bufferStatus* $\in$ *full*, *empty*, *normal* . As for ports, the port *trigger* can only send one type of signal and *inFlit* can send flits to a number of different destinations.

For implementing the circular buffer in DEVS-Suite, three state variables *head*, *tail*, and *bufferStatus* and two ports *trigger* and *inFlit* are modeled. Based on the number of possible inputs and the size of state variables (assuming 100 node network with 99 destinations), the total number of state-events (combinations of states and input events which correspond to all states and the transitions among them) is $100 \times 2 \times 8 \times 8 = 12600$. The state space exploration algorithm presented in Listing 5.2 iterates through all these states and gathers trace information.

The generator and the verification engine are automatically generated by DEVS-Suite. The coupled component view of the verification model is shown in Figure 6.8. Transducer records incoming data, state of the circular buffer, and outputs generated by it for further analysis.

After the model of the circular buffer is implemented, we have the option of developing a custom transducer as well. The transducer monitors the output of the circular buffer to ensure outputs are generated at the right time (takes 0.5 cycle from

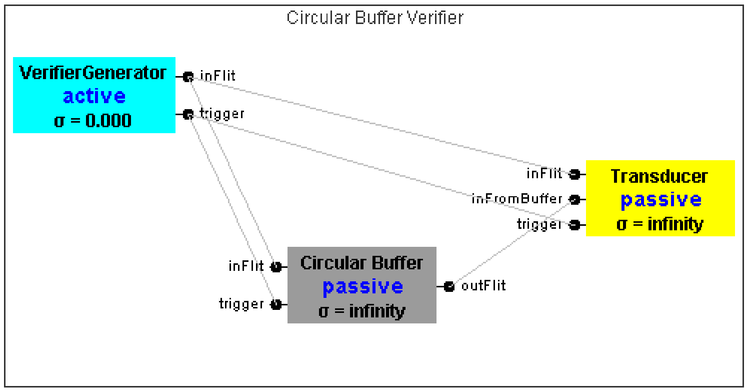


Figure 6.8: Circular Buffer Model Verification

the instance it is triggered), simultaneous events are handled properly (two events on two ports), correct setting of full/empty statuses, and correct updating of head/tail indexes. If developed correctly, the transducer is capable of finding model errors. As an example, for a basic circular buffer model, we had a small error in simultaneous event handling which could have easily gone unnoticed had it not been verified. The possibility of both simulating and model checking DEVS models in an integrated environment makes model development and evaluation simpler.

We ran a few instances of this verification with different parameters. Various scenarios can be created by changing configurations of the model under test (e.g., size of the buffer, types of incoming packets, and injection rate, ejection frequency). This is in part due to maintaining the modularity of DEVS-Suite while extending it to support model checking. In each of these cases, all states are explored and the correctness of functionality is ensured. A few of these instances are shown in Table 6.1.

The execution time for the state exploration linearly increases with state space size. Clearly, as the model gets larger, one must deal with the problem of state explosion. That is another part of our ongoing research on multiresolution modeling

Table 6.1: Sample Runs of the Circular Buffer Verification

Buffer Size	Number of Flit Types	State Space Size	Number of Cycles	Execution Time (seconds)
8	25	3200	3328.6	4.55
16	25	12600	13312.6	14.90
32	100	201600	206848.6	273.78

and verification. The idea is for DEVS-Suite to choose the highest abstraction possible for verifying a certain property to reduce computation time.

We devised another example in (Gholami and Sarjoughian, 2017) for minimal adaptive router. The model is specified in Constrained-DEVS and verified in DEVS-Suite. Also, it is worth noting that the state exploration algorithm and extend DEVS-Suite both support verifying coupled models as well. We have created and verified coupled models in this environment as well.

6.2 MRM NoC Library in DEVS-Suite

In order to enable NoC modeling at various levels of abstraction and mixing/-matching models of different resolutions, we created a modular packaging structure within DEVS-Suite modeling library. Each of these packages model an aspect of the network and relate to one or more of resolution dimensions (Object, Process, Time, Space, and Data) introduced in Chapter 2.

The behavioral, structural, and data aspects of NoC models are decoupled so that they can be used and reus4ed with one another. Most behaviors (such as routing, flow control, etc.) are added to components of NoC (containing structural information) at runtime using *Strategy* design pattern (Wolfgang, 1994).

6.2.1 NoC Components

NoC components were modeled at different levels of resolutions. There are 4 packages proposed for these components: Interface, Capacity, Flit, and Hardware (future work). At each package components of NoC were modeled. Multiple versions of each component may be possible. For example, at flit level, switch input ports can be modeled with various implementations of buffers (one being the circular buffer method we discussed earlier). Object and spatial dimensions of resolution can be altered when components of NoC are structurally modeled here.

An important point to note here is that these models are loosely coupled with one another as they stick to DEVS's strong modular model development regime. In DEVS models only communicate through ports and therefore are not directly dependent to one another. The other reasons for this is that the structure and behavior of these components are decided at runtime based on the configuration file provided by the user. features of the network such as number of virtual channels, queue capacity, and PE processing power are defined in Config files.

6.2.2 Topologies

Topology of a network relates to its spatial and object dimensions of resolution. In this package various topologies are placed. Similar to tools such as BookSim, well known topologies are placed here but the user-defined topologies using a configuration file is also possible. For this, a `CustomTopology` class is defined which reads the topology from file. In this topology, the connections between nodes (and the ports which connects them) are specified. The `CustomTopology` class builds a topology based on user description of the network.

One major capability of DEVS-Suite's library for topology is being dynamically configurable at runtime. Network topology may change temporarily at runtime due to component failure. These failures impact routing strategy and task distribution across the network. These cases can be simulated in DEVS-Suite and validated/verified.

6.2.3 Routing Methods and VC Allocation

Routing is a behavioral aspect of a network (behavior of the router) and relates to process and temporal resolutions. Various routing methods are realized here. This library is open to be used or extended by the user of the tool. Various source/logical routing methods based on turn model, adaptive routing, and oblivious routing may realized in this package. Switches can take be equipped with these routing methods at run time based on the configuration file.

The incredible flexibility of DEVS-Suite environment gives us additional capabilities not easily accessible in other tools. Equipping different switches with disparate routing methods, Changing the routing methods at run time (based on the state of the network or the changing topology), and even disabling parts of the network are among the additional capabilities provided to users in DEVS-Suite.

6.2.4 Flow Control Mechanisms

Similar to routing methods, flow control is a behavioral aspect of the network and therefore relates to process and time dimensions. Flow control mechanisms for wormhole switching are realized in another package and can be set for switches across the network. Not all networks use a flow control mechanism. Higher abstraction models (at interface/capacity level) do not consider flow controlling. Also, some networks (such as those which incorporate circuit switching) may not at all require a flow control policy. That is why switches are not directly dependent on the flow control mechanism. They can be instantiated (based on a configuration file) without

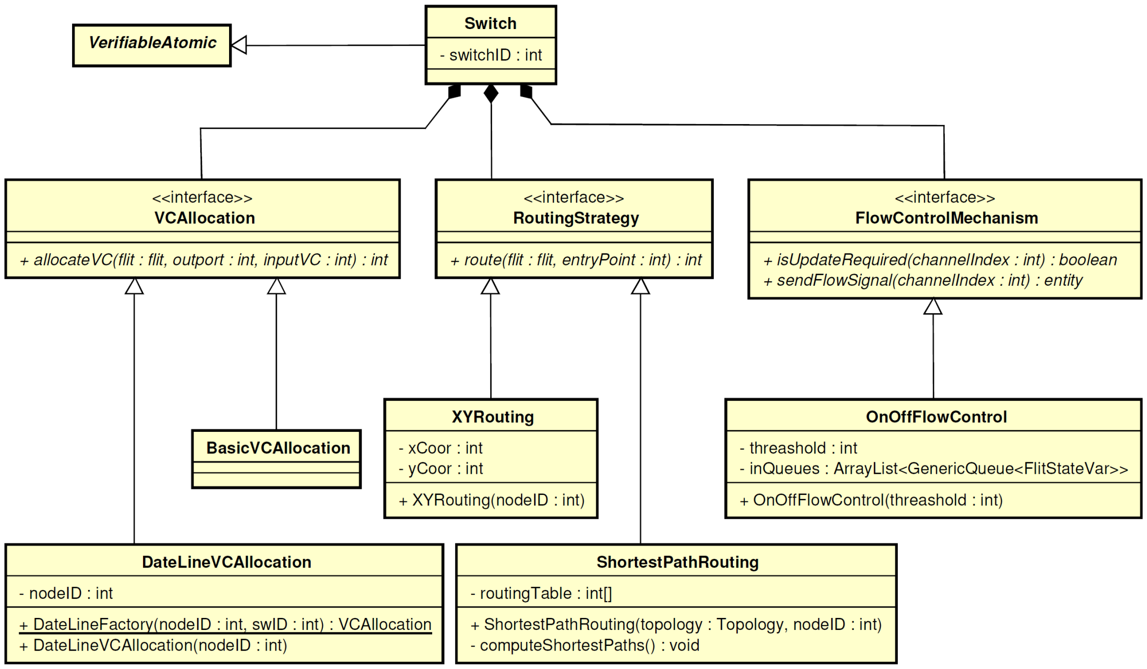

VerifiableAtomic
Switch
- switchID : int
<<interface>>
VCAllocation
+ allocateVC(flit : flit, outport : int, inputVC : int) : int
<<interface>>
RoutingStrategy
+ route(flit : flit, entryPoint : int) : int
<<interface>>
FlowControlMechanism
+ isUpdateRequired(channelIndex : int) : boolean
+ sendFlowSignal(channelIndex : int) : entity
BasicVCAllocation
XYRouting
- xCoor : int
- yCoor : int
+ XYRouting(nodeID : int)
OnOffFlowControl
- threashold : int
- inQueues : ArrayList<GenericQueue<FlitStateVar>>
+ OnOffFlowControl(threashold : int)
DateLineVCAllocation
- nodeID : int
+ DateLineFactory(nodeID : int, swID : int) : VCAllocation
+ DateLineVCAllocation(nodeID : int)
ShortestPathRouting
- routingTable : int[]
+ ShortestPathRouting(topology : Topology, nodeID : int)
- computeShortestPaths() : void

Figure 6.9: Strategy Design Pattern for Detaching Behavior from Structure and Flexibility

a flow control mechanism.

Figure 6.9 depicts how various behavioral aspects are attached to the switch component. Various strategies can be implemented for routing, VC allocation, and flow control (a few examples are provided). Any of these strategies can be attached to the switch. Furthermore, the strategy can be changed at runtime.

6.2.5 Data Types

So far we talked about various structural and behavioral aspects of the network which impact Object, Process, Temporal, and Spatial (Davis and Bigelow, 1998) aspects of the network. However, as previously stated, one aspect of NoC which cannot be ignored is data. Various data types, at different resolutions, may be communicated around the network or used by different components. In a separate package these data types are realized which can used by various physical (such as a link) or behavioral (such as a flow control mechanism) components. A few of these data types are: Tasks, Data Streams, Packets, and Flits.

Tasks are used by the PE models. Tasks may contain computation requirement, dependency, and age. A task may stay in a network and be processed by various processing elements up to a certain age (which can then be considered done). It also may be dependent on other tasks and have minimum processing time requirements. The processing power of the PE and the computational requirement of a task determines the computation time of that task in that processor. A task is built upon receiving a *Data Stream* which by itself is a collection of packets/flits.

Higher abstraction models, communicate using *Packets*. All the routing information are stored in the packet and they are communicated independently. Packets are converted to Data Streams when reaching their destination. *Flits* are similar to packets with a difference that they are more granular. A packet may contain a number of flits: one *Head* flit, a few *Body* flits, and a *Tail* flit. Only the head flit contains source/destination information but the data is divided among all flits. The head flit is ahead of the pack and allocates resources. The rest of flits follow the head flits without going through the allocation process again. The tail flit is in charge of deallocation of resources on its way forward.

All these various data types are equipped with state variable information for the verification process. As discussed earlier in Figure 5.5, state information in the data is flexible to be chosen by the user of the tool. One may control the number of states in the state space by controlling the contribution of data to the state size.

6.2.6 Transducers

Various aspects of the network can be measured during simulation or model checking. As state earlier, for this, we incorporate the concept of *Experimental Frame* with transducers and generators. Various transducers (each measuring a certain aspect of the network) are realized here. For example, transducers for deadlock avoidance or deadlock detection are developed here. One may incorporate one or more of these transducers based on the aspect of the network they want to measure, validate, or verify.

Like other aspects of this package, transducers can be modified or custom-made by the user for specific purposes.

6.2.7 Other Aspects

Other specific needs for simulation or model checking may be required. For mixing and matching models one may require data converters (one example was developed) or for more complete system simulations new components may be necessary. These custom needs may be added by the user. However, the important thing to note is that none of these new models in the library interfere with the simulation or model checking protocols. Modeling and execution at DEVS-Suite are two decoupled concepts. Therefore, custom needs of users can be easily added to the NoC library and they would work with the execution platform.

6.3 Modeling Comparison: UPPAAL vs. DEVS-Suite

In this section, we provide the reader with a model of Network-on-Chip (NoC) which embodies more complex characteristics compared with the circular buffer. Here we intend to 1) showcase the capabilities of extended DEVS-Suite in simulating and model checking NoC models and 2) compare these capabilities with that of another popular modeling and verification environment (UPPAAL). While the circular buffer model (presented in Chapter 5) provides basic understanding of how Constrained-DEVS and DEVS-Suite operate, it is not intricate enough to showcase the capabilities of DEVS-Suite.

As mentioned earlier, we compare DEVS-Suite with UPPAAL and not with any of the specialized NoC simulation or verification tools mentioned in the previous paragraph. There are two reasons for this: 1) none of the tools mentioned provide a comprehensive modeling, validation, and verification tool for NoC and 2) none of the tools are generic: their support for validation or verification is dedicated to NoC. Both DEVS-Suite and UPPAAL, on the other hand, are generic tools, supporting V&V of any system as long as they are modeled with their supported modeling languages (namely DEVS and Timed Automata).

Here, we first present the models of NoC in both of these environments and then compare them using a few simple experiments.

6.3.1 NoC modeling: Timed Automata & UPPAAL

For the model of NoC in UPPAAL, we captured the creation of data in processing element (PE), packetization in network interface (NI), and routing in switch. We developed this model in Timed Automata (TA) and created several sample properties using temporal logic. The model is then implemented in UPPAAL (v. 4.0.14) and the properties verified using UPPAAL verifier.

Data is first generated by the PE and sent to the packetizer. The packetizer stores the data and then converts it to packets in the order it receives them. Packets are then stored in a queue to be transmitted to the switch. Each packet, upon entering the switch from port 0, goes through several steps to be transmitted out. First, it

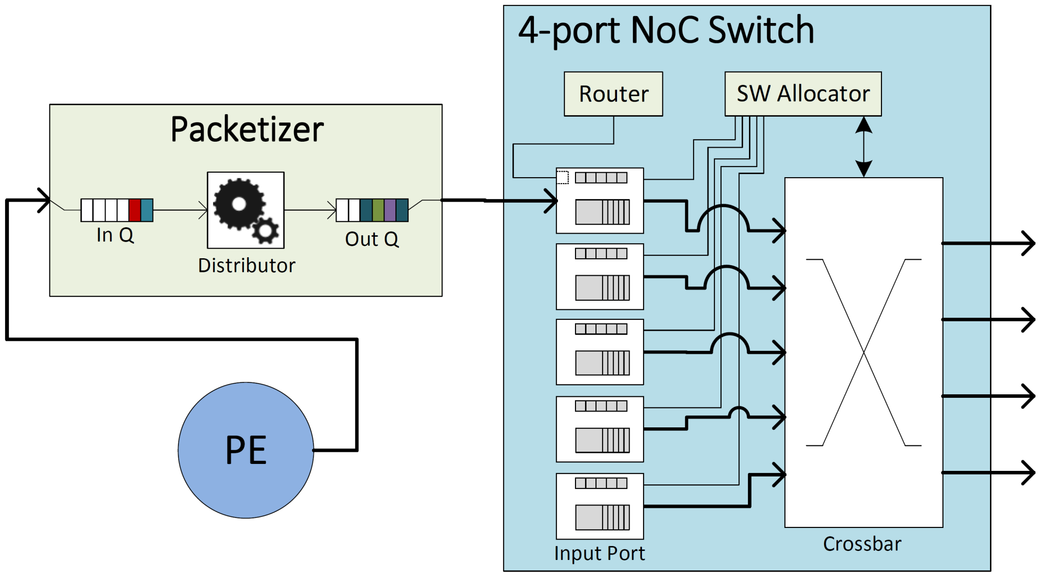


Figure 6.10: Detailed View of NoC as Modeled in Timed Automata. Each Component is a Separate Process

is stored in input port along with other packets. Then, the router determines the suitable outgoing port for the packet based on the destination address stored in the header. Next, a path in the crossbar switch is allocated in order to transfer the packet from input port to output port. After that, the packet is transferred through the crossbar to the output port. Finally, the output port sends the packet out of the router on the link. The model of NoC developed is depicted in Figure 6.10.

Several simplifications are made for modeling NoC. First, the data which is transferred from PE to the switch is only a simple integer value. The value can be 0, 1, 2, or 3 corresponding to the output port that it must be sent to. Therefore, the Router component just reads the port it should forward the packet to. Also, the Distributer component in the Packetizer only forwards it to the output queue. However, both of these components are modeled and time is allocated to the tasks they run. Finally, other input ports in the switch are not connected to outside ports but generate ran-

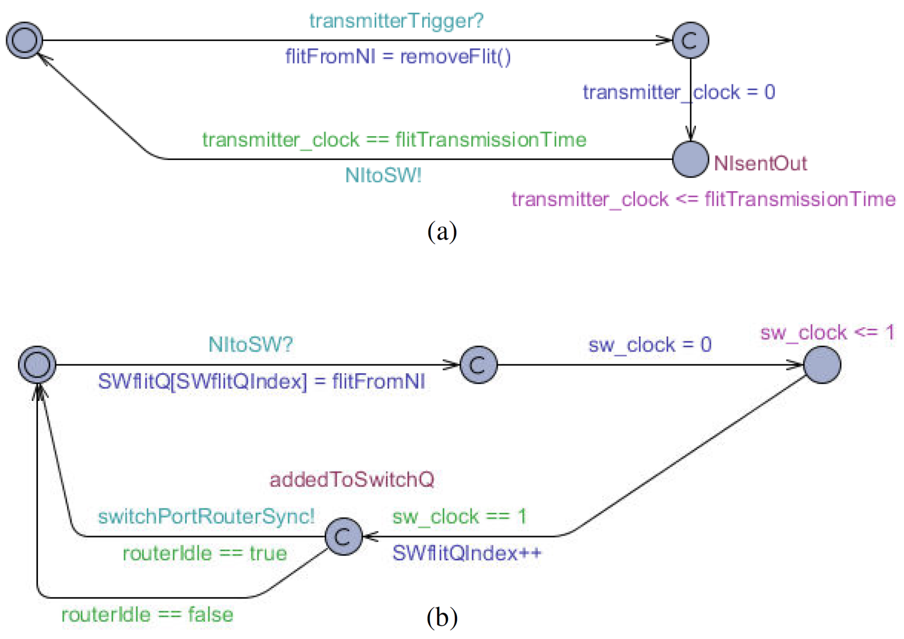


Figure 6.11: Two Sample Components of NoC and How They Are Connected via TA Channels: (a) NI Transmitter, (b) Swith Input Port

dom packets and allocate crossbar connections. This way contention is modeled and packets may be delayed. One important aspect of a switch that needs to be verified is the correctness of the switch allocation functionality.

Since these components are modeled as independent processes, a number of channels are introduced in timed automata for synchronization. For example, when PE generates a new packet, it stores it in a global variable and then notifies the NI receiver component that a new packet is available. Since information flow (i.e., there are no messages to be transmitted) is not really possible in UPPAAL's timed automata, global variables are defined and used to share data between two processes. As an illustration, in Figure 6.11, we provide two processes (NI Transmitter and Switch Input Port) and their communication through TA channels.

Various properties can be defined for this NoC. We devised several desirable properties in three categories. For safety, let us assume that a requirement of this NoC is not to drop packets. So, it would be a violation of safety if a full queue receives a new packet. For example, the index of switch input queue must always be smaller than the length of the queue. This can be formulated using temporal logic in the following way: $\Box(SWFlitQIndex < switchQSize)$

As for liveness, we formulate a property than ensures a packet entered a component eventually leaves it. For the network interface component, this property is written in temporal logic in the following way: $\Box(NI_Rec.addedToNIQ \rightarrow \diamond NI_Transmit.NIsentOut)$

6.3.2 NoC modeling: Constrained DEVS & DEVS-Suite

We developed a simple model of NoC using Constrained-DEVS and realized it in DEVS-Suite to showcase the additional capabilities provided to modelers in this environment. Contrary to the TA model, this one is a complete NoC of multiple nodes. One other difference here is that task generation is not random at the Processing Element (PE) level. Tasks come from outside and are distributed among PEs by a *Task Distributor* component. As it is shown later in this section, a task distributor may have various policies in regards to task distribution. In this model of NoC, we connect the task distributor to the first row of switches (the southern edge). Figure 6.12 depicts a 4-node network with the task distributor.

Every task entering the network has a life cycle of 3; meaning it is processed by 3 different processing elements (one after another) before being considered complete. Every PE after receiving a task, processes it, increments its age, and then forwards it to the next PE using the NoC. PEs have various processing power while tasks have computation requirement. The processing time of a task in a PE is $t = \dfrac{Computation\ Requirement}{Processing\ Power}$.

The model of NoC developed in DEVS-Suite is suitable for both simulation and

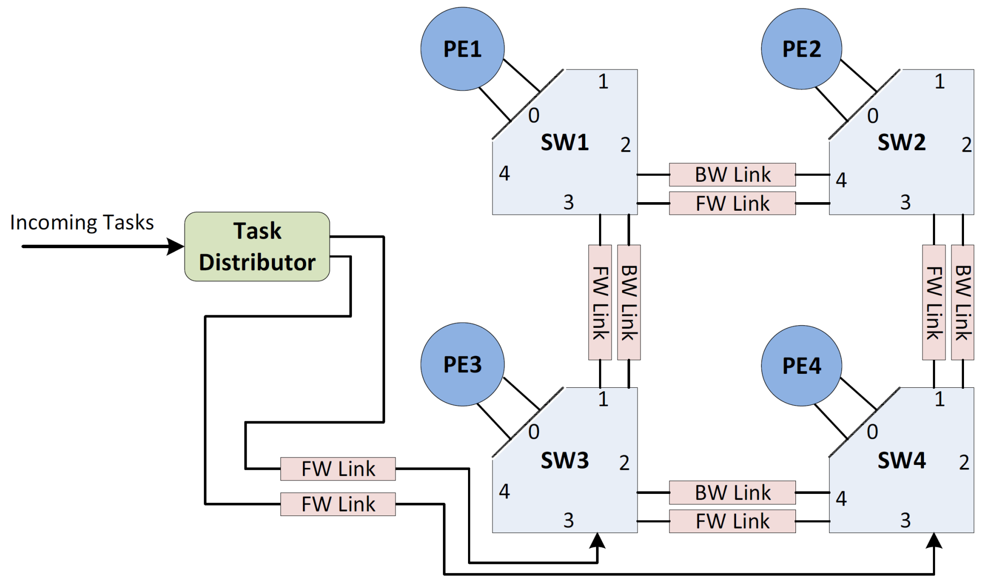


Figure 6.12: A 4-node Model of NoC With the Task Distributor Component

model checking. In the case of model checking, similar to the circular buffer, a *Generator* is automatically generated to inject various combinations of inputs. A view of NoC with only 4 nodes (visualized in DEVS-Suite) is shown in Figure 6.13. We conduct experiments on larger networks for scalability evaluations (see Figure 6.5).

The verification engine explores the entire state space and constructs the reachability graph. The transducer is notified after each state change and it stores data within nodes of the reachability graph. The data stored by the transducer relevant to the properties it is verifying for that model. For the network created here, one can observe properties such as queueing times, component utilization, and performance. Upon having the entire reachability graph, the transducer can analyze the graph and verify various properties.

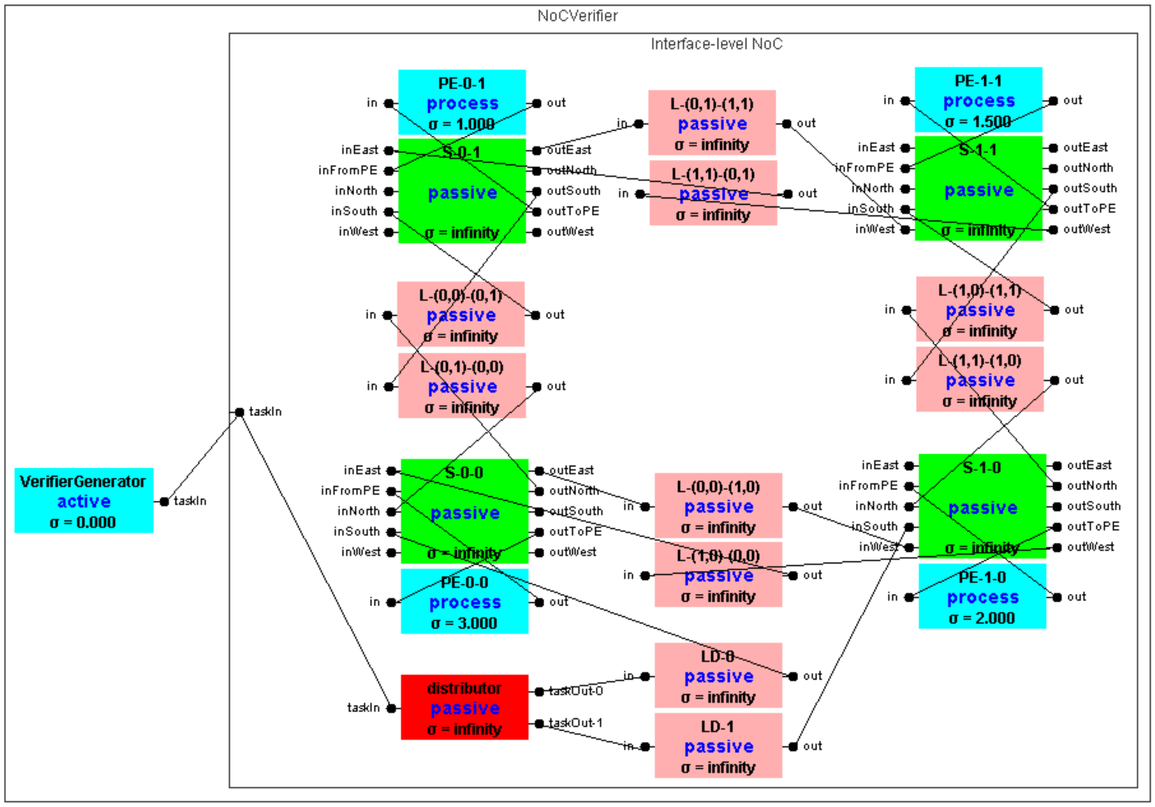


Figure 6.13: The Verification Environment for a 4-node Model of NoC With Major Components: PE, Switch, Link, and Task Distributor in DEVS-Suite

6.4 Experimentation

For both DEVS-Suite and UPPAAL there are some basic checks such as making sure no packets are dropped. We have developed additional experiments for DEVS-Suite. They are intended to demonstrate DEVS-Suite's capability in verifying complex properties. For this we consider maximum and minimum PE utilization and the impact of task distribution on this. Then we discuss deadlock avoidance/detection. Finally, an observation on the performance of verification in DEVS-Suite relative to the size of state space is provided in Section 6.4.4.

6.4.1 NoC Verification: PE Utilization and Task Distribution

As explained earlier, upon receiving new tasks, the task distributor component decides which processing elements should process the task first. This decision can be guided by different policies. We developed two task distributors with different policies. One uses the round robin approach and the other is adaptive. In the adaptive distribution policy, the distributor estimates the load on each processor by considering the history of distribution. Upon receiving a new task it assigns it to the PE with the estimated lightest load.

We conduct verification of a 4-node NoC for each of these task distributions. At the end of the verification process, the transducer calculates the highest and lowest possible utilization for each PE by applying DFS (Depth First Search) to the reachability graph. We would like to evaluate the impact of this adaptive policy on utilization and whether it contributes to a more computationally balanced network.

For this experiment, we only change the task distributor component and the rest of the network remains unchanged. It is worth noting that the adaptive task distributor requires new *explorable* state variables to hold its load estimates for PEs. We will analyze the impact of these new state variables on the size of the state space and the performance of the verification engine in Section 6.4.4. The execution platform for this experiment is a dual core-i5 3.5GHz machine running Java 7 and Windows 7.

Results

The use of round robin and adaptive distributors demonstrate very different behaviors in the network. Table 6.2 contains maximum and minimum utilization of each processing element using the two distribution methods. These results are extracted from the reachability graph by applying DFS on every reachable path. As shown in

Table 6.2: Maximum and Minimum Processor Utilization Using Round Robin (RR) and Adaptive Task Distributors with Total Execution Time

	RR Distribution		Adaptive Distribution	
	Min Util	Max Util	Min Util	Max Util
PE1	0.2	0.9	0.21	0.52
PE2	0.1	0.46	0.09	0.56
PE3	0.18	0.73	0.15	0.49
PE4	0.15	0.49	0.15	0.54
Overall	0.1	0.9	0.09	0.56
Duration	15.78s		245.20s	

the table, the adaptive approach results in a much more balanced load on processing elements (looking at maximum or minimum columns for all processors).

As explained earlier (and shown in Figure 6.12), the task distributor is only connected to the first row of switches (the southern edge). The round robin approach not only takes into account the load on each processor, it also does not consider the time it takes for the task to be transported by the network to PEs on upper rows. The adaptive distributor considers both of these and creates a more balanced distribution.

As discussed, the adaptive distributor requires new *explorable* state variables which increase the size of the state space. The last row in Table 6.2 shows the duration of model checking of these 4-node networks with each of these distributors. The use of adaptive distribution increases the size of the state space by a factor of 15 which consequently increases the duration of the verification process as reported in Table 6.2 (duration row).

6.4.2 Deadlock Avoidance Verification

It is possible to verify deadlock avoidance for special cases of routing and vc allocation without instantiating the model of the network and executing it. As introduced in Chapter 2, deadlocks may occur due to circular resource dependency. There are various methods for deadlock avoidance. Here two transducers are introduced which verify whether a pair of routing and VC allocation methods are deadlock prone or not.

The Turn Model

As introduced in Chapter 2, the turn model eliminates cycle-making turns from the routing algorithm in a k-ary n-mesh network. Turn model can be defined in our NoC models using predefined `enumeration` concept called *Turn*. The user can define a turn model by specifying allowable clockwise and counterclockwise turns using this enumeration.

When this model is defined, without knowing the adaptive part of the router, a deadlock avoidance transducer can be used to check whether this turn model is deadlock free in an infinite mesh. The search space however is bounded due to the the number of possible turns. For a turn model with 6 possible turns, any possible deadlock must happen with 6 turns. Therefore, the transducer searches the space of all 6-turn routes. In case a cycle is found the turn model is prone to deadlock.

As an example, the turn model shown in Figure 6.14 was described as a routing method. Deadlocks may occur since in this turn model since it allows circular waiting. Deadlock scenarios can be created for a 2D Mesh network. For this specific turn model, the transducer found a 6-turn (multiple of these are possible) depicted in Figure 6.15. The transducer stops after finding the first cycle.

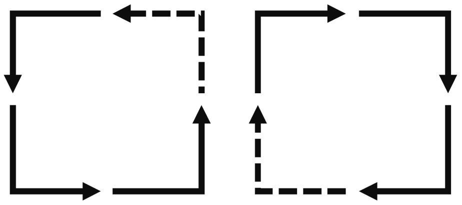


Figure 6.14: A Turn-model Rule Where Dashed Lines Show Forbidden Turns

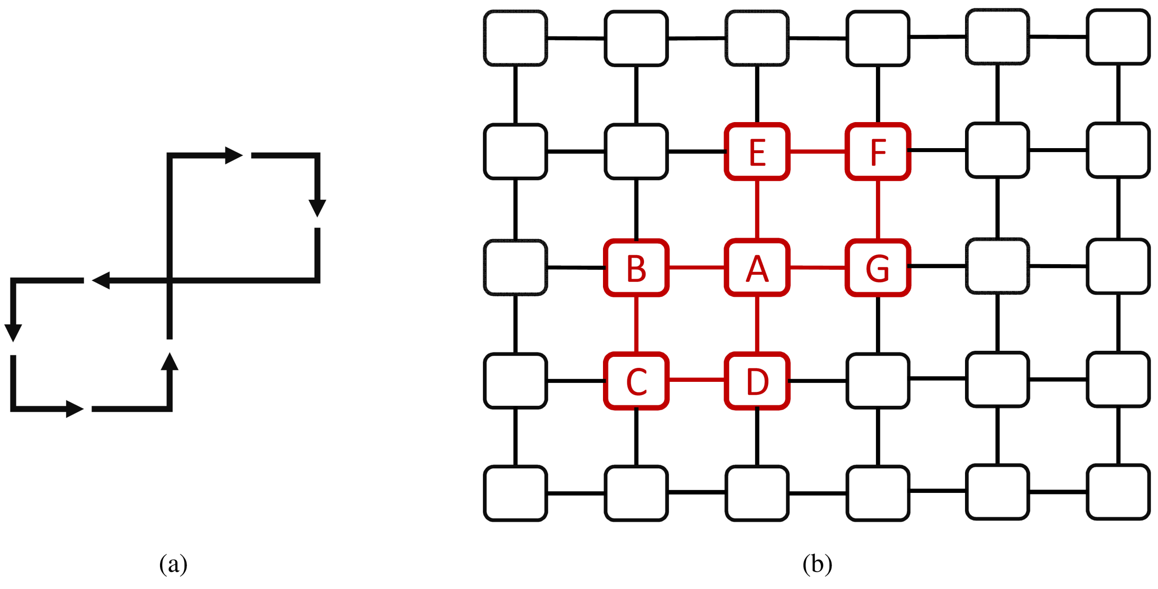


Figure 6.15: Deadlock Scenario for the Turn-model in Figure 6.14. (a) The Cycle Caused by the Turn-model, (b) The Deadlock in Effect in a Mesh Network with the Following Simultaneous Transmissions: $A \rightarrow C$, $B \rightarrow D$, $C \rightarrow E$, $D \rightarrow F$, $A \rightarrow F$, $E \rightarrow G$, $F \rightarrow A$, $G \rightarrow B$

Although there is an adaptive routing method to each turn model, the transducer does not require knowing that algorithm since it only checks whether the algorithm is prone to deadlock using allowable turns.

Black Box Deterministic Router/VC Allocator

The turn model deadlock avoidance is fairly limited. Many may not use turn models in their routing method design. Therefore, we created another transducer to detect possible deadlocks for black box routing and vc allocation. In this case, a topology, the routers and the vc allocators for each switch are instantiated and provided to the transducer. The transducer then determines whether this combination of topology, routers, and vc allocators is prone to deadlock.

Of course, these routers and VC allocators must be deterministic; meaning they do not consider the state of the network at runtime to make decisions. All routing decisions are made in advance and stored in routing tables. Examples of such routing algorithms are X-Y routing or predefined routing tables. As for the VC allocators simple strategy (forward on the same VC) and classified (such as dateline) are examples of deterministic VC allocators. The transducer in these cases does not understand the operation of these VC allocators and routers. They may be implemented using a programming language or configured using an external file. The transducer can only call the router and the VC allocator for a specific packet (going form a specific source to a destination) on an incoming channel to determine how it is routed and what VC is allocated to it.

The transducer must create channel dependency graph (explained in Chapter 2) in order to determine whether deadlocks are possible. For this packets are generated from every source to every destination and are given to the switches to deliver. The channel dependency graph is gradually built using the decisions made by the router and VC allocator at each node. After all combinations of source/destination nodes are explored, the CDG (Channel Dependency Graph) is complete and ready for analysis. As clarified in Chapter 2, any loop within the graph is a possible deadlock.

Algorithm 1 describes how the channel dependency graph is created and then checked for cycles using the `DFS` algorithm. The nested `for` loops after initialization create all packets for all combinations of source-destinations. These packets are routed to destination using the method `routeToDestination`. While routing the packets, the `routeToDestination` method constructs the `CDG` in Algorithm 1. The second part of the algorithm performs DFS on unvisited channels. If the `DFS` method returns `TRUE`, a cycle is found which is a possible deadlock.

Algorithm 1 Identifying Possible Deadlocks for Any Topology, Deterministic Routing, and Deterministic VC Allocation Using Channel Dependency Graph

```
Input: topology
Initialization: $n$ = topology.$size$
Instantiate: $switch_i$ $(0 \leq i \leq n)$, $CDG$, $Visited$
for $i = 0; i < n; i = i + 1$ do
    for $j = 0; j < n \wedge i \neq j; j = j + 1$ do
        instantiate $packet_{i,j}$;
        routeToDestination($switch_i, packet_{i,j}$);
    end for
end for
for all $channel_i \in CDG$ do
    if $channel_i \notin Visited$ then
        $loop = DFS(channel_i)$;
        if $loop = TRUE$ then
            return $channel_i$;
        end if
    end if
end for
return $\varnothing$;
```

The `routeToDestination` method is explained in Algorithm 2. This method routes the input packet to its destination by calling the routing and VC allocation methods of each switch on its way. With these information it constructs the channel dependency graph using channel IDs.

Algorithm 2 Expanding the `routeToDestination` Method in Algorithm 1. This Algorithm Constructs the Channel Dependency Graph in `CDG` Hash Structure

```
**Input Parameters:**$switch_i, packet_{i,j}$
$outport = switch_i.route(packet_{i,j});$
$outVC = switch_i.allocVC(packet_{i,j}, outport, 0);$
$nextChannelID = findChannelID(switch_i, outport, outVC);$
$currentSW = topology.nextSW(switch_i, outport, outVC);$
$inVC = outVC;$
$currentChannelID = nextChannelID;$
while $currentSW \neq switch_j$ do
    $outport = currentSW.route(packet_{i,j});$
    $outVC = currentSW.allocVC(packet_{i,j}, outport, inVC);$
    $nextChannelID = findChannelID(currentSW, outport, outVC);$
    $CDG.add(currentChannelID, nextChannelID)$
    $inVC = outVC;$
    $currentChannelID = nextChannelID;$
    $currentSW = topology.nextSW(currentSW, outport, outVC);$
end while
```

An example network topology is provided in Figure 6.16. This is a custom topology read from input file. The `CustomTopology` class reads this topology and creates the network based on it. We set the routing algorithm to be shortest path and simple VC allocation (forwards the packet on the same VC as the input). With this configuration, the transducer (containing Algorithms 1 and 2) constructs the channel dependency graph depicted in Figure 6.17.

The transducer will find one of the three loops (depending on which node it starts

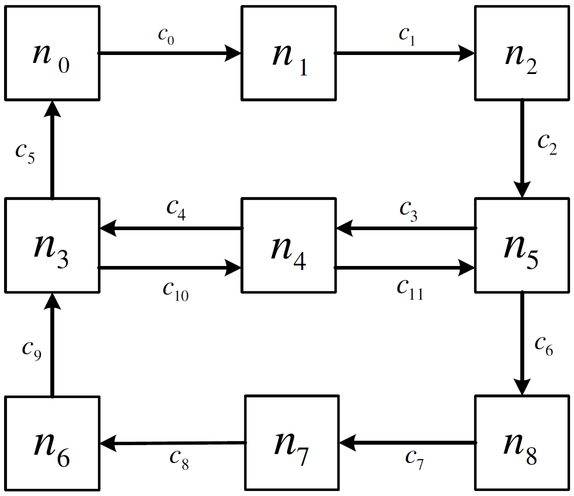


Figure 6.16: 2D NoC Custom Topology

with) and will alert the user of the possibility of deadlock.

For the deadlock scenario presented in Figure 2.2, the deadlock is detected by the transducer. However, assuming we have 2 virtual channel, when we change the VC allocation strategy to *Dateline*, the network becomes deadlock free and all flits reach their destination. The channel dependency graph for this network with Dateline at the channel between SW_3 and SW_0 is depicted in Figure 6.18.

Performance Evaluation for Deadlock Avoidance

Looking at the deadlock avoidance method (described above), one can guess that the complexity of the routing algorithm and the number of nodes directly impact the running time of the algorithm. Here we devised sample experiments to illustrate the complexity of deadlock avoidance method on various configurations. All experiments are conducted on DEVS-Suite 3.0.0 on a platform with Core i5 3.10 GHz and 8GB of physical memory.

In one experiment, the routing algorithm is set to be X-Y routing, and the VC

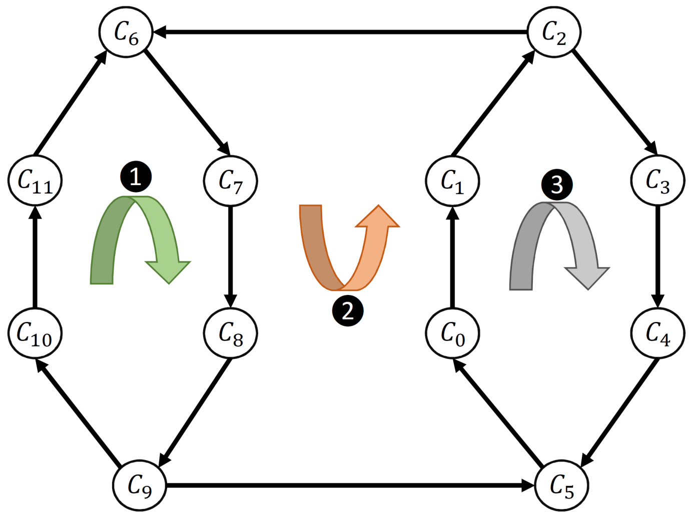
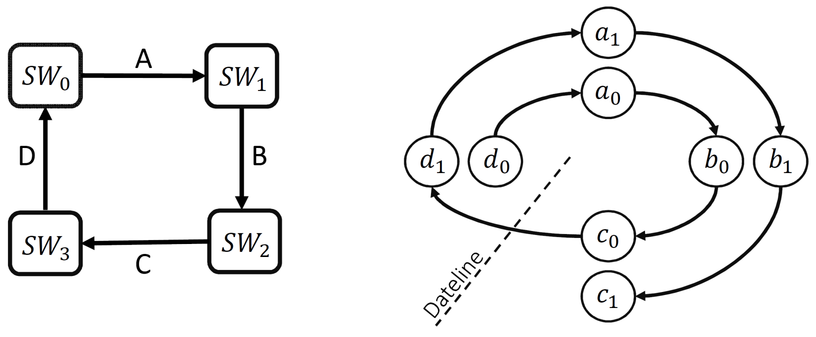


Figure 6.17: CDG For the Network Topology in Figure 6.16, Shortest Path Routing, and Simple VC Allocation



(a) (b)

Figure 6.18: Simple Ring with Two Virtual Channels and Dateline at Link *D*. (a) The Ring Network with 4 Nodes (Refer to Figure 2.2) (b) CDG For the Network with Virtual Channels Divided into Dateline Classes

Table 6.3: Deadlock Avoidance for Mesh with X-Y Routing

Number of Nodes	Execution Time (seconds)
25	0.003
100	0.004
400	0.01
900	0.018
1600	0.029
2500	0.055
3600	0.092
4900	0.13
6400	0.26
8100	0.36
10000	0.68

allocation to be simplistic: the first VC that is available. No pre-computation is required for this routing method or the VC allocation. Packets must be created from every source to destination and routed through the network. The routing path is then used for constructing the CDG. We applied this to a Mesh topology and the results are shown in Table 6.3. Execution times are depicted in a plot in Figure 6.19.

In the second experiment, we created networks with balanced tree topology with different sizes and applied shortest path routing to them. The number of nodes in a balanced tree is decided by the number of branches at each node and the depth of the tree. Shortest path algorithm is applied to all packets. We incorporated the simple all node shortest path (with time complexity of $O(n^3)$). The results are presented in

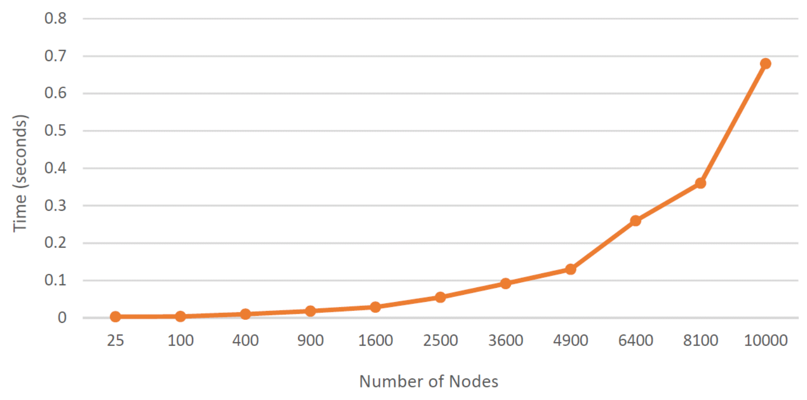


Figure 6.19: Plot for the Execution Times Presented in Table 6.3

Table 6.4: Deadlock Avoidance for The Balanced Tree with Shortest Path Routing

Depth	Branching	Number of Nodes	Execution Time (seconds)
4	3	121	0.24
3	6	259	2.2
5	3	364	9.43
3	8	585	38.5
4	5	781	140.47
6	3	1093	717.34

Table 6.4 and Figure 6.20.

As apparent from the results the pre-computation complexity for the routing algorithm has a great impact on the execution time of deadlock avoidance algorithm. This impact is much greater than the impact of network size as can be seen from the first experiment (see Table 6.3).

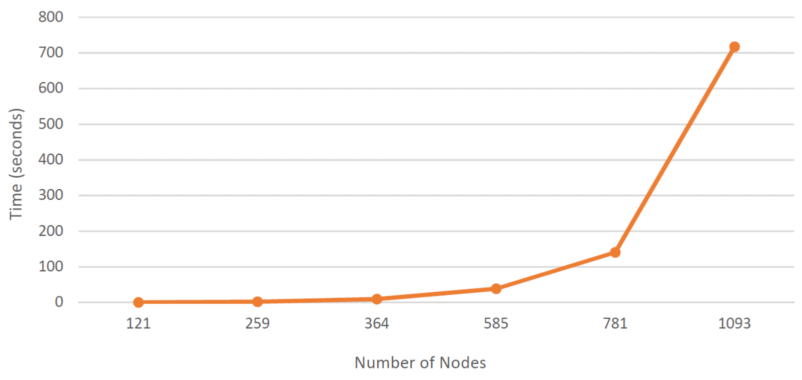


Figure 6.20: Plot for the Execution Times Presented in Table 6.4

6.4.3 Deadlock Detection

While the deadlock avoidance transducers are useful, they only operate on deterministic routing and VC allocation methods. For this reason, they do not require the fully operating network (only switches are instantiated). However, many adaptive routing and VC allocation methods operate based on the state of the network in runtime. In those cases, one cannot verify whether deadlocks are possible without a fully operating network.

A deadlock detection transducer is included in the model solely for detecting deadlocks. At any point in time, this transducer can analyze the state of the network and decide whether it is in a deadlock situation. In addition to routing and VC allocation, the flow control mechanism may also contribute to the occurrence of deadlock. The transducer regards the routing algorithm, flow control mechanism, and VC allocation as black boxes. Two conditions are necessary for a deadlock to occur: 1) there is a circular allocation of input ports and 2) the input queue of all input port in that circular allocation are full (cannot receive more from upstream).

We implemented this capability for flit-level NoC models. At an instance of time, the transducer first creates a graph of all allocations in the network. Then it searches

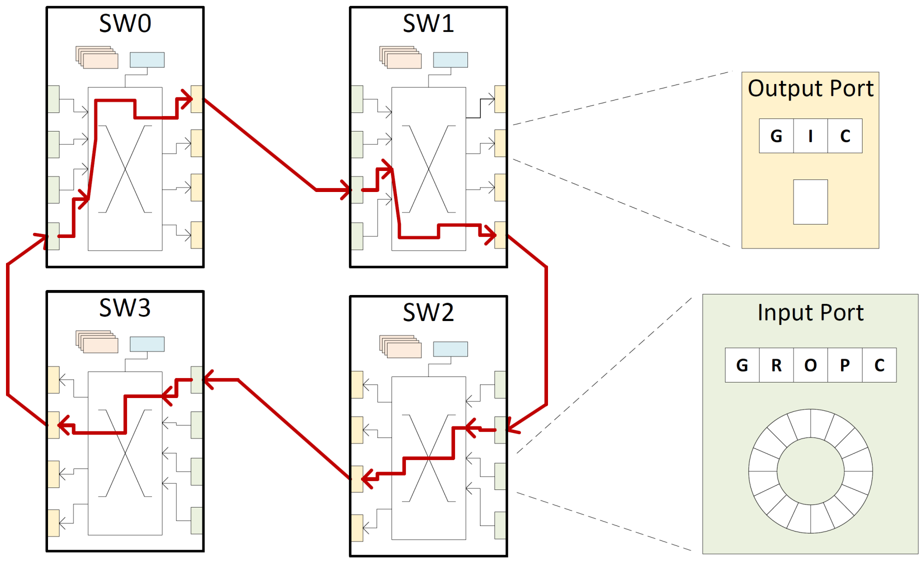


Figure 6.21: Deadlock Condition in a 4-node Flit Level Network. Input/Output Status Arrays and Buffers are Shown in Detail

the graph for cycles (first condition). In case a cycle exists, it checks whether the input queues for all input ports in that cycle are full (second condition). If such a case is observed, it is immediately reported to the user.

Status arrays in input and output ports contain the allocation information. Figure 6.21 depicts a deadlock condition among 4 nodes. The path of deadlock is shown in thick red line (shown as dark gray in B&W print). The allocation information is stored in R (for route) and O (for output VC). Variable C contains credit information (in case of credit-based flow control). A portion of a network is in deadlock if there is a circular allocation and all output ports are waiting for credit (which means buffers in input ports are full).

We successfully applied this transducer to various topologies with black box routing, flow control, and virtual channel allocation policies. It must be noted that this transducer can be used in both model checking and simulation scenarios. However, the state space of a fully operating flit-level network (even for 4 nodes)becomes excessively so large that deadlock analysis becomes computationally impractical using common computing platforms. Therefore, this capability is more appropriate for simulation scenarios.

6.4.4 Performance analysis

For this experiment, we only observe the performance of DEVS-Suite verification engine for different network sizes. Here we are not verifying any particular property; however, we intend to explore the entire state space and show the practicality of applying model checking to a system such as NoC. For this experiment, we increase the size of the network and collect performance metrics from the verification engine.

The model of NoC which is explored here considers every state variable within the network as *explorable* except for the processing queue of PE. We assume that any packet delivered to any PE will eventually be processed. So, phase, sigma, and buffers for switches, links, and PEs are all taken into account for model checking. As the reader can predict, the size of the state space will rapidly grow and cause the familiar state explosion problem.

The reader must notice that state exploration in DEVS-Suite is much more demanding as we are dealing with complex data types and more generally compound objects. Simpler modeling methods such as Timed Automata and Petri net may use primitive data types to represent the state of the system, however, in DEVS-Suite, in order to benefit from the capability of simulation and experimentation, the models are much more complex; thus, the exploration protocol, hashing method for visited and unvisited states, state modification, and model execution are all more demanding. This results in seeing longer execution times for exploring the state space of a model.

In this experiment, we will see the state explosion phenomenon for NoC models and present what DEVS-Suite and Constrained-DEVS can offer to make the verification of these complex models practical. Since this experiment is resource intensive, we ran these experiments on a Windows 7 machine with quad core Xeon 3.1GHz and 16GB of memory.

Results

The impact of increasing the size of the network on the size of the state space was as predicted. As expected, the size of the state space explodes as we add nodes to the network. The results are shown in Table 6.5. It is clear that exploring the entire state space of the network as a whole can become impractical. Of course there is room for improving the performance of the engine by refactoring, creating lower-level code (such as in C), and using a faster platform, however, one can easily see that scalability will always be a concern in exploring state spaces of some NoC systems.

The problem of state explosion is well-known and the results are expected. So now the question is, what is the use of such verification engine and is it at all practical to use this for a system such as NoC? We claim that using this environment is beneficial even in dealing with large and complex systems. Here are four reasons why Constrained-DEVS and DEVS-Suite are useful for modeling complex systems:

∗ **Partial Verification** : It may be unnecessary to explore the entire state space of a system such as NoC. We usually require model checking for small systems or portions of larger systems. For example, although model checking of networks larger that 16 nodes is impractical, model checking can be used to verify properties of individual components or a portion of the entire network.

∗ **Multiresolution Modeling** : Models created in Constrained-DEVS and DEVS-Suite can be created at various levels of abstraction. For the NoC example,

Table 6.5: Duration of Model Checking and Size of the Reachable State Space for Various Network Sizes

Network Size	Number of States	Duration
2×2	18339	14.62s
3×2	108432	111.59s $\approx$ 2mins
3×3	1748859	2029.11 $\approx$ 34mins
4×3	25255495	39211.6s $\approx$ 11hrs
4×4	–	–

we create models at different abstraction levels, each aims at possessing certain details. Increasing the resolution (lower abstractions) results in models with larger state spaces. Some properties (such as routing-related properties, PE utilization, and latency) can be verified at higher abstraction levels.

$*$ **Validation & Verification** : DEVS-Suite supports both validation (using simulation) and verification (using model checking) of Constrained-DEVS models. Models that lend themselves to model checking can be simulated without making any changes to them. This is beneficial for highly detailed models (high resolution) in which model checking a large portion is futile. Therefore, we can rely on a combination of small portion model checking and large scale simulation of high resolution models.

$*$ **Selective State Exploration** : As explained earlier, state exploration is selective in DEVS-Suite as modelers can mark state variables as explorable/unexplorable based on the properties they intend to verify.

Chapter 7

CONCLUSION AND FUTURE WORK

The overarching purpose of this dissertation is suggesting a new approach in order to reduce the separation between abstract models and the actual system. For this we incorporate multiresolution modeling in order to create a family of models capturing the target system in different resolutions. We are applying this approach to basic NoC models at interface, capacity, and flit abstraction levels. Instead of switching between different tools for creating models at each abstraction, we embed several phases of NoC component design within our tool. A library of basic NoC models is created in DEVS which upon implementation in DEVS-Suite can be verified or validated. This library contains models of NoC at different levels of abstraction (from coarse-grain, interface-level network models to finer-grain, flit-level models of NoC HW/SW) which can be mixed and matched for simulation or model checking. In this era of complex systems, multiresolution modeling is a requirement for the design process. One cannot comprehend all detailed aspects of a complex system at early stages. Multiresolution modeling is a gradual approach by which the designer reaches final design in a stepwise manner.

A Multi-Resolution Modeling with model checking approach, supported by the modular, hierarchical DEVS theory and framework, is proposed. A framework is grounded in this approach and implemented on top of the DEVS-Suite simulator. DEVS-Suite enables combined model validation and model checking for a class of DEVS models. Our approach is intended to simplify system-level design by customizing MRM and V&V. This approach, however, is not limited to a specific system. It proves useful during the design phase of any system given its compatibility with the

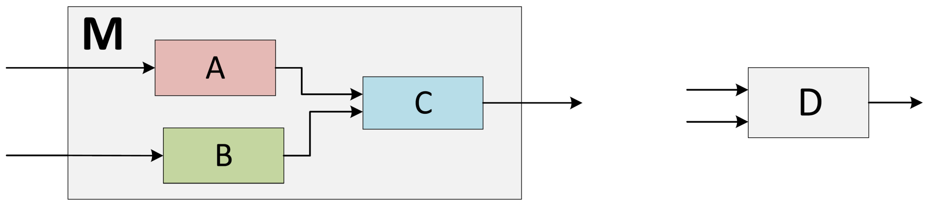


Figure 7.1: A Coupled Model and an Atomic One as Homomorphic Models

DEVS framework.

7.1 Future Work

This research can be continued at various fronts. Some of these future directions are provided below. These future directions are sorted based on the level of abstraction they apply.

7.1.1 Homomorphic Mapping

Homomorphic mapping of models at different levels of system hierarchy are discussed in (Wymore, 1993) extensively. Using the approach presented in (Wymore, 1993), one may show two atomic models to be homomorphic to each other. This concept has to be first extended to support coupled models as well as atomic ones. Being able to prove homomorphic relationships between models at different resolutions enables the modeler to reuse some of validation or verification methods for those models.

Consider the coupled model M and atomic model D in Figure 7.1. Models M and D have the same functionality: for the two inputs they receive (call them num_1 and num_2 the output is: $num_1^2 + num_2^2$. Model D, is an atomic model with the function mentioned above. Model M is a coupled model, which compartmentalizes the aforementioned function. It has two types of inner components: Adder and Square. Models A and B are of Square type. The Square type outputs the square value of the input it receives. Model C is of type Adder. It outputs the sum of the two inputs it receives. Clearly, these two models are homomorphic. However, we need the theory to support proving this relationship between atomic and coupled models as well. In this research, we extend the DEVS homomorphic mapping method to support coupled models as well as atomic ones.

When homomorphic relationship between two models are proven, one can reuse the validation/verification methods used on the simpler model for the more elaborate one. In the example provided in Figure 7.1, some validation experiment on model D can be applied on model M as well. However, model M requires some more scenarios to be validated since its functionality and behavior is more elaborate.

Figure 7.2 depicts homomorphic approach for validating a model of the channel at two levels of resolution. There are two models of the channel: one is an atomic model at flit level (left hand side) and the other is a coupled model at hardware level (right hand side). Assuming these two models are developed so that the hardware-level channel is an elaboration of the flit-level one and their homomorphic relationship is proven in DEVS (using our proposed approach), one can reuse the validation methods/experiments applied on the flit-level channel for the hardware-level channel. First the input/output pairs for the flit-level channel are gathered using probes. These probes can be put in place using programming techniques. Then, an experimental frame validates the hardware-level channel by injecting the inputs using a generator and validating the outputs via a transducer. Obviously, the hardware-level channel is more elaborate and requires further validation for its additional functionalities but homomorphic mapping helps in reusing some of the validation effort and avoiding redundancy.

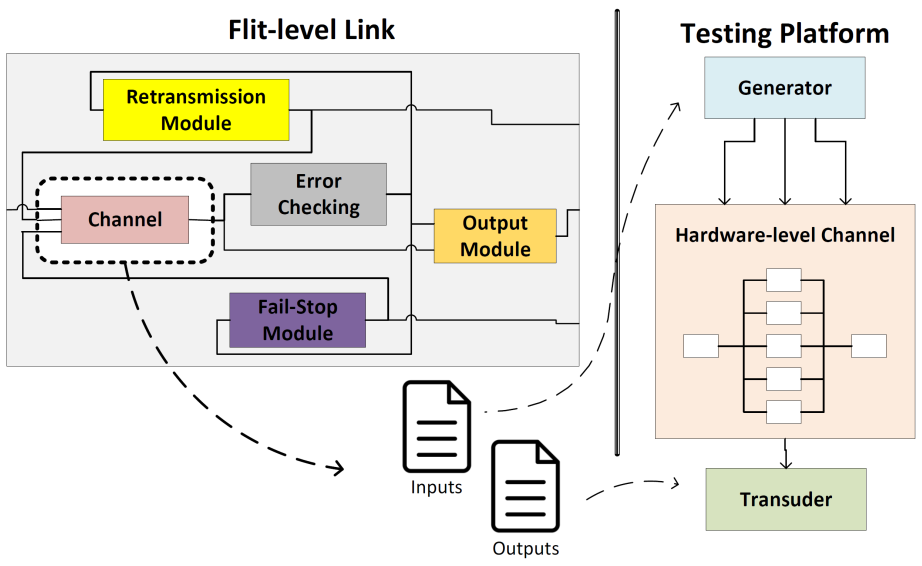


Figure 7.2: Validation of Simulation Models Across Different Resolution Using Homomorphic Mapping

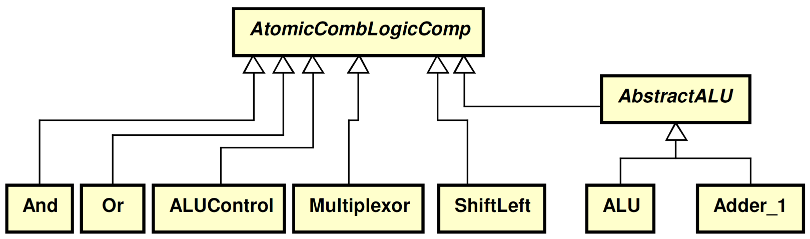


Figure 7.3: Select Combinational Logic Models from MIPS-DEVS Library

7.1.2 Gate-level Models

Since we were focusing on higher-level aspects NoC modeling, validation, and verification, we did not create hardware-level models of NoC at DEVS level. Hardware level components can be modeled in DEVS as long as discrete-event property holds for them. Adding these models to the existing ones creates a complete NoC model library containing models at all levels of abstraction.

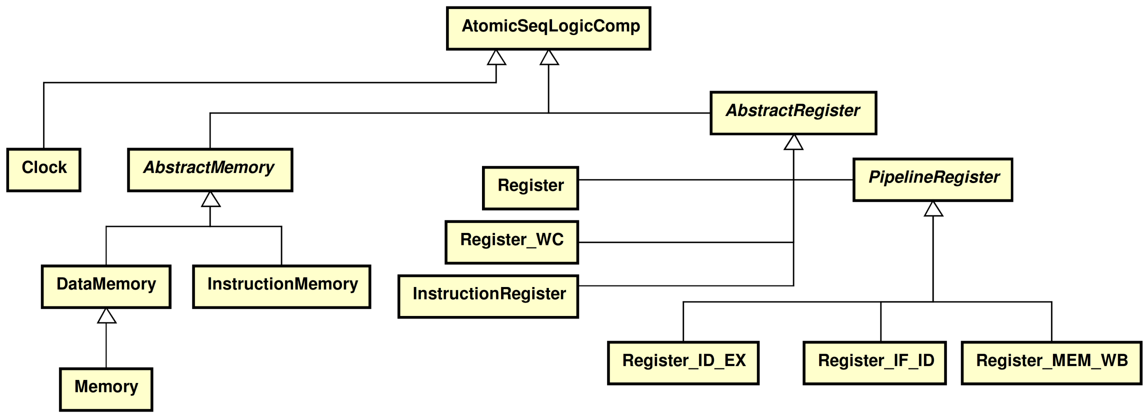


Figure 7.4: Select Sequential Logic Models from MIPS-DEVS Library

Chen and Sarjoughian created a library of gate-level components which they used for modeling MIPS32 in DEVS. Figures 7.3 and 7.4 show partial UML diagrams for the combinational and sequential logic components developed in DEVS (Chen and Sarjoughian, 2010). These elements are coupled together using DEVS coupling mechanism to form various pipeline MIPS32 architecture. Notice that a clock element is also included in the model although DEVS operates based on discrete events. This is necessary for the operation of sequential models (some shown in Figure 7.4).

The same library can be used/extended to model components of NoC at hardware level. This new package of hardware-level models automatically is executable in DEVS-Suite for both simulation or model checking.

7.1.3 DEVS to Hardware Description Language

Another interesting direction would be converting hardware-level models to a hardware description language. This is attractive since hardware-level model execution is very slow in software. If models can be converted to VHDL or Verilog, they can be synthesized and executed on an FPGA. Also, the capabilities of both DEVS and HDL complement each other and enable designers to do much more than is available in one of them. Model transformation between DEVS and HDL will be limited as the two are fundamentally different. Because of these inherent differences, transformations between the two will not be possible for all models. In order to have an effective transformation, models need to represent well-defined components (such as those presented in Figures 7.3 and 7.4) for which there are clear models/descriptions in both DEVS and HDL.

Moreover, co-simulation of models in FPGA and DEVS-Suite is another direction that can be taken. Control aspects or software modules can be simulated in DEVS-Suite while the HDL model acts as the underlying hardware. This requires a cable-based communication between the module running DEVS-Suite and the FPGA in which the hardware model is synthesized. This is toward high-level hardware/software co-design. Real-time simulation of models in DEVS-Suite is required. Toward this, we have introduced Action-Level Real-time DEVS (ALRT-DEVS) (Sarjoughian and Gholami, 2015; Gholami and Sarjoughian, 2012). Models developed using ALRT-DEVS have been used for hardware-software real-time co-simulation (Sarjoughian *et al.*, 2013).

REFERENCES

Abad, P., P. Prieto, L. G. Menezo, A. Colaso, V. Puente and J. A. Gregorio, "TOPAZ: An open-source interconnection network simulator for chip multiprocessors and supercomputers", pp. 99–106 (IEEE Computer Society, Los Alamitos, CA, USA, 2012).

ACIMS, "DEVS-Suite simulator, version 3.0.0", URL `https://acims.asu.edu/software/devs-suite` (2017).

Alur, R., C. Courcoubetis and D. Dill, "Model-checking in dense real-time", Information and computation **104**, 1, 2–34 (1993).

Alur, R. and D. L. Dill, "A theory of timed automata", Theoretical computer science **126**, 2, 183–235 (1994).

Anjan, K. and T. M. Pinkston, "An efficient, fully adaptive deadlock recovery scheme: DISHA", in "ACM SIGARCH Computer Architecture News", vol. 23, pp. 201–210 (ACM, 1995).

Antle, J. M., S. M. Capalbo, E. T. Elliott, H. W. Hunt, S. Mooney and K. H. Paustian, "Research needs for understanding and predicting the behavior of managed ecosystems: lessons from the study of agroecosystems", Ecosystems **4**, 8, 723–735 (2001).

Baohong, L., "A formal description specification for multi-resolution modeling based on DEVS formalism and its applications", The Journal of Defense Modeling and Simulation **4**, 3, 229–251 (2007).

Basdogan, C., C.-H. Ho and M. A. Srinivasan, "Virtual environments for medical training: Graphical and haptic simulation of laparoscopic common bile duct exploration", IEEE/Asme Transactions On Mechatronics **6**, 3, 269–285 (2001).

Bazzaz, H., M. Sirjani, R. Khosravi and S. Taheri, "Modeling networking issues of network-on-chip: a coloured Petri Nets approach", in "Proceedings of the 2nd International Conference on Simulation Tools and Techniques", pp. 22–32 (ICST, 2009).

Behrmann, G., A. David and K. G. Larsen, "A tutorial on UPPAAL 4.0 (2006)", URL http://www. it. uu. se/research/group/darts/papers/texts/new-tutorial. pdf (2014).

Beltrame, G., D. Sciuto and C. Silvano, "Multi-accuracy power and performance transaction-level modeling", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **26**, 10, 1830–1842 (2007).

Berekovic, M., H.-J. Stolberg and P. Pirsch, "Multicore system-on-chip architecture for MPEG-4 streaming video", IEEE Transactions on Circuits and Systems for Video Technology **12**, 8, 688–699 (2002).

Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill and D. A. Wood, "The GEM5 simulator", SIGARCH Comput. Archit. News **39**, 1–7, URL `http://doi.acm.org/10.1145/2024716.2024718` (2011).

Bouyer, P., S. Haddad and P.-A. Reynier, "Timed petri nets and timed automata: On the discriminating power of zeno sequences", Information and Computation **206**, 1, 73–107 (2008).

Bricaud, P., *Reuse methodology manual: for system-on-a-chip designs* (Springer Science & Business Media, 2012).

Burch, J. R., E. M. Clarke, K. L. McMillan, D. L. Dill and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond", in "Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science (LICS'90)", pp. 428–439 (IEEE, 1990).

Butler, J. M., "Quantum modeling of distributed object computing", ACM SIGSIM Simulation Digest **24**, 2, 20–39 (1994).

Catania, V., A. Mineo, S. Monteleone, M. Palesi and D. Patti, "Cycle-accurate network on chip simulation with Noxim", ACM Trans. Model. Comput. Simul. **27**, 1, 4:1–4:25, URL `http://doi.acm.org/10.1145/2953878` (2016).

Caughlin, D. and A. F. Sisti, "Summary of model abstraction techniques", in "AeroSense'97", pp. 2–13 (International Society for Optics and Photonics, 1997).

Chatterjee, S., M. Kishinevsky and U. Y. Ogras, "xMAS: Quick formal modeling of communication fabrics to enable verification", IEEE Design & Test of Computers **29**, 3, 80–88 (2012).

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

Ciardo, G., J. Muppala and K. Trivedi, "SPNP: stochastic Petri net package", in "Petri Nets and Performance Models, 1989. PNPM89., Proceedings of the Third International Workshop on", pp. 142–151 (IEEE, 1989).

CMU, "Wormsim 4.2", URL `http://www.ece.cmu.edu/~sld/software/worm_sim.php` (2017).

Dally, W. and B. Towles, *Principles and practices of interconnection networks* (Morgan Kaufmann, 2004).

Davis, P. K. and J. H. Bigelow, "Experiments in multiresolution modeling (MRM)", Tech. rep., RAND Corp., Santa Monica, CA (1998).

Davis, P. K. and R. Hillestad, "Families of models that cross levels of resolution: Issues for design, calibration and management", in "Proceedings of the 25th conference on Winter simulation", pp. 1003–1012 (ACM, 1993).

Davis, P. K. and A. Tolk, “Observations on new developments in composability and multi-resolution modeling”, in “Simulation Conference, 2007 Winter”, pp. 859–870 (IEEE, 2007).

Dokhanchi, A., A. Zutshi, R. T. Sriniva, S. Sankaranarayanan and G. Fainekos, “Requirements driven falsification with coverage metrics”, in “Proceedings of the 12th International Conference on Embedded Software”, pp. 31–40 (IEEE Press, 2015).

Eggenberger, M. and M. Radetzki, “Fine grained adaptive simulation with application to nocs”, in “Specification & Design Languages (FDL), 2013 Forum on”, pp. 1–8 (IEEE, 2013).

Foures, D., V. Albert and A. Nketsa, “Simulation validation using the compatibility between simulation model and experimental frame”, in “Proceedings of the 2013 Summer Computer Simulation Conference”, p. 55 (Society for Modeling & Simulation International, 2013).

Foures, D., V. Albert and A. Nketsa, “A new specification-based qualitative metric for simulation model validity”, Simulation Modelling Practice and Theory **66**, 1–15 (2016).

Garland, M., “Multiresolution modeling: Survey & future opportunities”, State of the art report pp. 111–131 (1999).

Gholami, S. and H. Sarjoughian, “Real-time network-on-chip simulation modeling”, in “SIMUTools, Desenzano, Italy”, pp. 103–112 (ICST, 2012).

Gholami, S. and H. S. Sarjoughian, “Hybrid multi-resolution modeling: a network-on-chip model”, in “Proceedings of the 2016 Winter Simulation Conference”, pp. 1499–1510 (IEEE Press, 2016).

Gholami, S. and H. S. Sarjoughian, “Modeling and verification of network-on-chip using constrained-DEVS”, in “Proceedings of the Symposium on Theory of Modeling & Simulation”, pp. 1–9 (Society for Computer Simulation International, 2017).

Glass, C. J. and L. M. Ni, “The turn model for adaptive routing”, ACM SIGARCH Computer Architecture News **20**, 2, 278–287 (1992).

Goossens, K., J. Dielissen, O. P. Gangwal, S. G. Pestana, A. Radulescu and E. Rijpkema, “A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification”, in “Proceedings of the conference on Design, Automation and Test in Europe-Volume 2”, pp. 1182–1187 (IEEE Computer Society, 2005).

Halpern, J. Y. and M. Y. Vardi, “Model checking vs. theorem proving: a manifesto”, Artificial intelligence and mathematical theory of computation **212**, 151–176 (1991).

Hemani, A., A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg and D. Lindqvist, “Network on chip: An architecture for billion transistor era”, in “Proceeding of the IEEE NorChip Conference”, vol. 31 (2000).

Holcomb, D., B. Brady and S. Seshia, "Abstraction-based performance verification of NoCs", in "Proceedings of the 48th Design Automation Conference", pp. 492–497 (ACM, 2011).

Holliday, M. A. and M. K. Vernon, "A generalized timed Petri net model for performance analysis", IEEE Transactions on Software Engineering , 12, 1297–1310 (1987).

Hwang, M. H. and B. P. Zeigler, "Reachability graph of finite and deterministic DEVS networks", IEEE Transactions on Automation Science and Engineering **6**, 3, 468–478 (2009).

Jain, L., B. Al-Hashimi, M. Gaur, V. Laxmi and A. Narayanan, "Nirgam: a simulator for noc interconnect routing and application modeling", in "Design, Automation and Test in Europe Conference", pp. 16–20 (2007).

James, L. P., *Petri net theory and the modeling of systems* (Prentice-Hall, Inc. Englewood Cliffs, NJ, 1981).

Kahng, A. B., B. Li, L.-S. Peh and K. Samadi, "Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration", in "Proceedings of the conference on Design, Automation and Test in Europe", pp. 423–428 (European Design and Automation Association, 2009).

Kapinski, J., J. Deshmukh, X. Jin, H. Ito and K. Butts, "Simulation-guided approaches for verification of automotive powertrain control systems", in "American Control Conference (ACC), 2015", pp. 4086–4095 (IEEE, 2015).

Kaufmann, M., P. Manolios and J. S. Moore, *Computer-aided reasoning: ACL2 case studies*, vol. 4 (Springer Science & Business Media, 2013).

Kim, S., H. Sarjoughian and V. Elamvazhuthi, "DEVS-Suite: a simulator supporting visual experimentation design and behavior monitoring", in "Proceedings of the 2009 Spring Simulation Conference", pp. 29–36 (Society for Computer Simulation International, 2009).

Larsen, K., P. Pettersson and W. Yi, "UPPAAL in a nutshell", International Journal on Software Tools for Technology Transfer (STTT) **1**, 1, 134–152 (1997).

Li, H., G. Pedrielli, M. Chen, L. H. Lee, E. P. Chew and C.-H. Chen, "V-shaped sampling based on kendall-distance to enhance optimization with ranks", in "Winter Simulation Conference (WSC), 2016", pp. 671–681 (IEEE, 2016).

Marculescu, R., U. Y. Ogras, L.-S. Peh, N. E. Jerger and Y. Hoskote, "Outstanding research problems in NoC design: system, microarchitecture, and circuit perspectives", Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **28**, 1, 3–21 (2009).

Marsan, M. A. and G. Chiola, "On Petri nets with deterministic and exponentially distributed firing times", in "European Workshop on Applications and Theory in Petri Nets", pp. 132–145 (Springer, 1986).

Martin, G., B. Bailey and A. Piziali, *ESL design and verification: a prescription for electronic system level methodology* (Morgan Kaufmann, 2010).

Mayer, G. R., H. S. Sarjoughian, E. K. Allen, S. Falconer and M. Barton, “Simulation modeling for human community and agricultural landuse”, SIMULATION Series **38**, 2, 65 (2006).

Mayr, E. W., “An algorithm for the general Petri net reachability problem”, SIAM Journal on computing **13**, 3, 441–460 (1984).

Department of Defense, “DoD modeling and simulation (M&S) glossary”, DoD Publication 5000.59-M (1998).

Modelica Association et al, “Modelica”, URL `https://www.modelica.org/` (2017).

Myers, G. J., C. Sandler and T. Badgett, *The art of software testing* (John Wiley & Sons, 2011).

Natrajan, A., P. F. Reynolds and S. Srinivasan, “MRE: a flexible approach to multi-resolution modeling”, in “Parallel and Distributed Simulation, 1997., Proceedings., 11th Workshop on”, pp. 156–163 (IEEE, 1997).

Naylor, T. H. and J. M. Finger, “Verification of computer simulation models”, Management Science **14**, 2, B–92 (1967).

Ni, N., M. Pirvu and L. Bhuyan, “Circular buffered switch design with wormhole routing and virtual channels”, in “Computer Design: VLSI in Computers and Processors, 1998. ICCD’98. Proceedings. International Conference on”, pp. 466–473 (IEEE, 1998).

Norlin, K. A., *Flight Simulation Software at NASA Dryden Flight Research Center* (National Aeronautics and Space Administration, Dryden Flight Research Center, 1995).

Nutaro, J. J. and B. P. Zeigler, “Towards a probabilistic interpretation of validity for simulation models”, in “Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium”, pp. 197–204 (Society for Computer Simulation International, 2015).

Pasqua, R., D. Foures, V. Albert and A. Nketsa, “From sequence diagrams uml 2. x to FD-DEVS by model transformation”, in “European Simulation and Modelling Conference”, pp. pp–37 (2012).

Plaku, E., L. E. Kavraki and M. Y. Vardi, “Hybrid systems: from verification to falsification by combining motion planning and discrete search”, Formal Methods in System Design **34**, 2, 157–182 (2009).

Rantala, V., T. Lehtonen and J. Plosila, *Network on chip routing algorithms* (Citeseer, 2006).

Roychoudhury, A., T. Mitra and S. R. Karri, “Using formal techniques to debug the amba system-on-chip bus protocol”, in “Proceedings of the conference on Design, Automation and Test in Europe-Volume 1”, p. 10828 (IEEE Computer Society, 2003).

Rozenblit, J. W., “Experimental frame specification methodology for hierarchical simulation modeling”, International Journal Of General System **19**, 3, 317–336 (1991).

Saadawi, H. and G. Wainer, “Principles of discrete event system specification model verification”, Simulation **89**, 1, 41–67 (2013).

Salaun, G., W. Serwe, Y. Thonnart and P. Vivet, “Formal verification of CHP specifications with CADP illustration on an asynchronous network-on-chip”, in “Asynchronous Circuits and Systems, 2007. ASYNC 2007. 13th IEEE International Symposium on”, pp. 73–82 (IEEE, 2007).

Salminen, E., C. Grecu, T. D. Hämäläinen and A. Ivanov, “Application modelling and hardware description for network-on-chip benchmarking”, IET computers & digital techniques **3**, 5, 539–550 (2009).

Sargent, R. G., “Verification and validation of simulation models”, in “Proceedings of the 37th conference on Winter simulation”, pp. 130–143 (winter simulation conference, 2005).

Sarjoughian, H. S. and S. Gholami, “Action-level real-time devs modeling and simulation”, Simulation p. 0037549715604720 (2015).

Sarjoughian, H. S., S. Gholami and T. Jackson, “Interacting real-time simulation models and reactive computational-physical systems”, in “Proceedings of the 2013 Winter Simulation Conference: Simulation: Making Decisions in a Complex World”, pp. 1120–1131 (IEEE Press, 2013).

Sarjoughian, H. S. and S. Sundaramoorthi, “Superdense time trajectories for DEVS simulation models”, in “Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium”, DEVS ’15, pp. 249–256 (Society for Computer Simulation International, San Diego, CA, USA, 2015), URL `http://dl.acm.org/citation.cfm?id=2872965.2872999`.

Schmaltz, J. and D. Borrione, “A functional formalization of on chip communications”, Formal Aspects of Computing **20**, 3, 241–258 (2008).

Schruben, L. W., “Establishing the credibility of simulations”, Simulation **34**, 3, 101–105 (1980).

Seo, C., B. P. Zeigler, R. Coop and D. Kim, “DEVS modeling and simulation methodology with MS4 Me software tool”, in “Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium”, p. 33 (Society for Computer Simulation International, 2013).

Seo, C., B. P. Zeigler, D. Kim and K. Duncan, “Integrating web-based simulation on IT systems with finite probabilistic DEVS”, in “Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium”, pp. 173–180 (Society for Computer Simulation International, 2015).

Stoy, J. E., *Denotational semantics: the Scott-Strachey approach to programming language theory* (MIT press, 1977).

SystemC, “Accellera System Initiative”, URL `http://accellera.org/downloads/standards/systemc` (2017).

Taktak, S., J.-L. Desbarbieux and E. Encrenaz, “A tool for automatic detection of deadlock in wormhole networks on chip”, ACM Transactions on Design Automation of Electronic Systems (TODAES) **13**, 1, 6 (2008).

TaLiRo, “TaLiRo (TemporAl LogIc RObustness) Tools”, https://sites.google.com/a/asu.edu/s-taliro/, URL `https://sites.google.com/a/asu.edu/s-taliro/` (2017).

Valmari, A., “The state explosion problem”, Lectures on Petri nets I: Basic models pp. 429–528 (1998).

Verbeek, F. and J. Schmaltz, “Formal specification of networks-on-chips: deadlock and evacuation”, in “Proceedings of the Conference on Design, Automation and Test in Europe”, pp. 1701–1706 (European Design and Automation Association, 2010).

Wang, D., N. E. Jerger and J. G. Steffan, “DART: a programmable architecture for NoC simulation on FPGAs”, in “Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip”, NOCS ’11, pp. 145–152 (ACM, New York, NY, USA, 2011), URL `http://doi.acm.org/10.1145/1999946.1999970`.

Whitner, R. B. and O. Balci, “Guidelines for selecting and using simulation model verification techniques”, in “Proceedings of the 21st conference on Winter simulation”, pp. 559–568 (ACM, 1989).

Wolfgang, P., “Design patterns for object-oriented software development”, Reading Mass (1994).

Wymore, A. W., *Model-based systems engineering*, vol. 3 (CRC press, 1993).

Yilmaz, L., A. Lim, S. Bowen and T. Oren, “Requirements and design principles for multisimulation with multiresolution, multistage multimodels”, in “Simulation Conference, 2007 Winter”, pp. 823–832 (IEEE, 2007).

Zeigler, B. P., H. Praehofer and T. G. Kim, *Theory of Modeling and Simulation* (Academic Press, Inc., Orlando, FL, USA, 2000), 2nd edn.