VERIFICATION OF CONSTRAINED-DEVS NETWORK-ON-CHIP MODELS

Soroosh Gholami
ACIMS
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, AZ, USA
sgholami@asu.edu

Hessam S. Sarjoughian
ACIMS
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, AZ, USA
sarjoughian@asu.edu

ABSTRACT

Verification of models is necessary for some classes of systems to guarantee safety properties in addition to satisfying functional requirements. Exhaustive model checking is a full proof method for verifying lack of undesirable behavior for dynamical systems. In this work, we present a model checking verification method for Network-on-Chip (NoC) models. For this purpose, a constrained version of the atomic DEVS modeling formalism is formulated and applied to verification of an NoC router. This is achieved by adding constraints on the state size and the number of transitions in the atomic model. A model-checker is introduced into the DEVS-Suite simulator for verifying constrained DEVS models using the DEVS simulator protocol. The model-checker is used to verify a model of NoC router component at different scale. The state space size, number of state transitions, and execution time metrics, without and with fault, demonstrate the verification of constrained DEVS models.

Keywords: Constrained-DEVS, DEVS-Suite Simulator, Model Checking, Verification, Network-on-chip

1 INTRODUCTION

Both validation and verification techniques are used to achieve some degree of assurance that models are accurate representations of systems of interest. Based on the nature of models for a system such as Network-on-Chip (NoC) one can use various V&V techniques (Sargent 2005, Whitner and Balci 1989). Model validation determines the degree to which some model along with its data is an accurate representation of a mental or physical system from the perspective of its intended use. For model verification the goal is to show a system's desired structure and behavior are correctly specified.

Network-on-Chip (NoC) (Hemani, et al. 2000) acts as a communication subsystem between Intellectual Properties (IPs) communicating on a System-on-a-Chip. Communication is treated as series of packets sent and received using an underlying network. Four open research and future challenge categories have been identified 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). In the 4th category, the authors briefly explore NoC verification and point out that this field has received less attention compared with other research categories. Research in model verification includes showing correct delivery of packets while ensuring absence of deadlock/livelock. Although verification capability for packet delivery and deadlock/livelock is necessary, it is insufficient considering the expected functionality of NoC. It would be helpful if one could also verify correctness of other properties such as worst-case flit latency and packet/flit loss ratio.

In this research, we use model checking as the verification method for Network-on-Chip models. The verification capability incorporates exhaustive model checking to verify model properties (based on system requirements). We propose using the DEVS modeling method by extending it to support verification

through specifying model states and I/O properties and introducing a model-checker to the DEVS-Suite simulator.

This paper is organized as follows. We discuss the background on model checking, DEVS, and verifiable versions of DEVS in Section 2. In Section 3, we closely examine related previous works on model checking and NoC verification. In Section 4, we introduced our approach toward constrained modeling and verification protocol. In Section 5, we present a simple model of a router verified in terms of state reachability under constrained state and I/O configurations. Finally, in Section 6, we summarize this research and discuss some future work.

2 BACKGROUND

Discrete EVent System Specification (DEVS) is a formalism for specifying modular, hierarchical coupled models composed of atomic models. Parallel atomic DEVS is a defined as $\langle X, S, Y, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta \rangle$ in which *X* and *Y* are input and output sets, S is the state set, $\delta_{int}: S \to S$ is the internal transition function, δ_{ext} is the external transition function, δ_{conf} is the confluent transition function, $\lambda: S \to Y$ is the output function, and $ta: S \to R_0^{+\infty}$ is the time advance function.

Formal model checking aims at determining whether a model of a system meets certain requirements. Theorem-proving approaches for mathematically proving/disproving 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 method of verification. State explosion problem is a common issue when model checking is applied to a complex model. However, the use of model checking method is not entirely abandoned. In particular, for a safety critical system, model checking is still necessary for exploring some parts of the entire state space of the system's model.

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 state transitions. Formal modeling approaches such as Timed Automata (Alur and Dill 1994) and Petri net (Peterson 1981) can be verified exhaustively as they correspond to a finite state machine. As for DEVS, because of the continuity of time for external input events and state transitions, the state space is unbounded. DEVS-based models, as normally specified, are well suited for simulation purposes. However, model-checking algorithms require bounded state space in order to iterate through all possible states and state transitions.

For example, consider the model of a stack that can store up to eight natural numbers at any given time (see Listing 1). The input port "*in*" is used to inserting numbers into the stack while the input port "*out*" is used for removing the numbers. The output port "*pop*" is used for requesting an output. This port is used to send output an already stored number (if any). If the stack is empty, no output will be sent. Similarly, if a number is received in input port "*in*" when the stack is full, the number is ignored. We assume whenever a signal is received in the input port "*pop*", the model outputs a stored number after a finite time period, δt.

This stack model has input X and output Y. The state of the model S consists of phase, sigma, buffer, index, and the popped number. The phase is only "*active*" when the pop signal is received. In phase active, the output is the popped number and thereafter the phase is set to "*idle*". The δ_{ext} defines receiving new entries from input port "*in*". When an external event is received with some elapsed time e (i. e., $0 \leq e \leq ta(s)$) and the stack is not full then the input value is put into the array, otherwise it is ignored and sigma is updated by e. Similarly, another δ_{ext} is defined for the case a pop signal is received. In this scenario, if the stack is not empty, the value of the *popped* state variable is set to the value from top of stack for transmission. After δt, the popped number is sent out and then the state is updated as defined in δ_{int}.

This Stack model exemplifies the unsuitability of the DEVS formalism for model checking if the *index* state variable is unbounded (i.e., the index can be any number between zero and infinity inclusive). However, we are well aware of the fact that this stack can store at most eight values, which constrains the value of index to a number between zero and seven. The general DEVS formalism does not place any constraint on

the range of values any of its state variables can have. In particular, the total state Q is infinite because input events can arrive at any instance of time even if every other state variable is restricted to have a finite range of values (i.e., the state space cardinalities for atomic, and therefore coupled, models are infinite).

Listing 1: A Simple Stack Model

$$S = \overbrace{\{Active, Idle\}}^{Phase} \times \overbrace{\sigma}^{sigma} \times \overbrace{\mathbb{N}^8}^{\text{Buff}} \times \overbrace{\mathbb{N}}^{\text{index}} \times \overbrace{\beta}^{\text{popped}} \ , \ X = \{(input, \mathbb{N}), (pop, 1)\} \ , \ Y = \{(output, \mathbb{N})\}$$

$$\delta_{ext}\big((Idle, \sigma, \text{Buff}[0..7], index, \emptyset), e, (input, x)\big) = \begin{cases} (\dots \text{Buff.add}(x), index + 1, \emptyset) \ \textbf{if} \ index < 7 \\ (\dots, index, \emptyset) \ \ \textbf{if} \ index = 7 \end{cases}$$

$$\delta_{ext}\big((Idle, \sigma, \text{Buff}[0..7], index, \emptyset), e, (pop, x)\big) = \begin{cases} (Active, \dots, index - 1, \text{Buff}[index]) \ \textbf{if} \ index > 0 \\ (Idle, \dots, index, \emptyset) \ \ \textbf{if} \ index = 0 \end{cases}$$

$$\delta_{int}(Active, \sigma, \text{Buff}[0..7], index, popped) = (Idle, \infty, \text{Buff}[0..7], index, \emptyset)$$

$$\lambda(Active, \sigma, \text{Buff}[0..7], index, popped) = (output, popped)$$

However, constrained versions of DEVS are introduced to make the cardinality of state space finite. There have been previous efforts such as FD-DEVS (Hwang and Zeigler 2009) for supporting model checking. Although this approach is useful, it has two shortcomings as described below.

2.1.1 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. Similar to non-determinism, stochasticity is another obstacle toward model checking. The randomness for choosing one transition among many, as defined in FP-DEVS, can also increase the state space of a model. One can reduce the stochasticity of the system by removing randomness.

In 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. We believe non-determinism support in a model checking engine is important. This leads to support for a wider range of real systems.

2.1.2 Limited Property Checking

Models are verified against properties defined by the modeler. In most model checking environments, a formal language encodes the desired properties of the systems. Examples are Timed Computation Tree Logic (Alur, Courcoubetis and Dill 1993) in UPPAAL (Larsen, Pettersson and Yi 1997) or DEVS Natural Language (DNL) in MS4 Me (Seo, Zeigler and Coop, et al. 2013). Inventing a language or using a time-based logic 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)? In these cases, formal languages such as LTL are helpless.

3 RELATED WORK

3.1 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. A variant of Timed Automata with UPPALL is a popular toolset for 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, Courcoubetis and Dill 1993) as its language for property expressions.

Petri nets (Peterson 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 supports verifying deadlock and liveness properties. There are many extensions to the original Petri nets modeling language, some of which are more expressive (Bouyer, Haddad and Reynier 2008). Since there were no equivalency relationship between any variant of Petri net and Timed Automata, , the bounded Read Arc Timed Petri net capable of checking the presence of tokens without consuming them was developed (Bouyer, Haddad and Reynier 2008). This variant of Petri net is computationally equal to Timed Automata. 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. In addition, both Petri nets and timed automata only support modeling the behavior of the system. DEVS on the other hand supports both structural and behavioral modeling of concurrent systems.

3.1.1 DEVS-based Model Checking

A variant of DEVS called Finite-Deterministic DEVS (FD-DEVS) is introduced to support model checking (Hwang and Zeigler 2009). It allows for finite state/event sets, rational or infinity state lifespans, and next-state scheduling. In spite of the FD-DEVS safety and liveness being decidable, non-determinism for internal transition, external transition, and advance functions are not accounted for. Many engineered and natural systems such as Networks-on-Chips are inherently non-deterministic. In addition, as mentioned earlier, arbitrary property checking is not possible through FD-DEVS.

An approach looks at FD-DEVS as the target for model checking. In (Pasqua, et al. 2012) the authors introduce an approach where 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. 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.

In (Saadawi and Wainer 2011) the authors 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 method for manually converting RTA-DEVS models to Timed Automata. Models that are converted to Timed Automata are limited with respect to the data they can communicate. Finite Probabilistic DEVS (FP-DEVS) (Seo, Zeigler and Kim, et al. 2015) is an extension of FD-DEVS in which the choice of next state is made probabilistically.

3.2 NoC Verification

There have been previous efforts on formal verification of network-on-chips. Some of these approaches work based on external model checker engines. In (Salaun, et al. 2007), the authors 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. GeNoC (Schmaltz and Borrione 2008) creates a meta-model of the NoC and then uses ACL2 theorem prover to prove whether data is correctly routed and reaches the intended destination. In (Roychoudhury, Mitra and Karri 2003) the authors use SVM symbolic model checker to verify and debug Advanced Micro-controller Bus Architecture (AMBA). Their approach is limited to starvation.

Other approaches have incorporated standard modeling method with wide range support. In (Taktak, Desbarbieux and Encrenaz 2008), the authors 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).

4 APPROACH

Model verification in DEVS entails adding four additional features to the DEVS M&S framework: 1) state configuration, 2) input port configuration, 3) discretized time for external inputs, and 4) verification protocol. The first three are required to constrain state space. In addition to these features, we discuss two additional features which we view to be important for developing the verification protocol for the constrained DEVS-Suite model checking engine. One for data exclusion, which further reduces the state space size and another for functional requirements of the system. These are defined in models and checked for by the verification engine.

4.1 State Configuration

As we elaborated in the simple stack example in Section 2, the unconstrained DEVS specification does not lend itself to model checking. A verification engine needs to know about the value set of each state variable. We introduce value constrains to state variables. The verification protocol can leverage these constrained state variables for model verification.

State variables can be ***Primitive*** or ***Compound***. An primitive state variable can only be of certain types including Character, Integer, and Boolean. Compound states are any combination of primitive states. We use ***regular expressions*** to define 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.

Primitive state 1: $Char$
Compound state 1: $String$: $(Char)^{24}$
Compound state 2: $Queue$: $(String)^{8} \rightarrow ((Char)^{24})^{8}$

The expression for the primitive state 1 is for one character of a string. In the Java programming language, the type of the primitive state variable 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 using a regular expression. Finally, the third equation is the compound state of the queue holding 8 strings of size 24.

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

4.2 Port Configuration

Similar to state variables, ports require accurate specifications for their types and value sets. However, not all ports require bounded specification. Only the external input ports require such specification. Internal couplings and external outputs do not define port types. They define source/destination model components and their corresponding ports. The reason for this is that the internal couplings are between atomic/coupled models. 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 for bounded state configuration can be applied here for bounded port configuration as well. Value sets for these ports can also be specified via regular expressions. In the specification of input

ports, an additional *NULL* (∅) 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 be $(Char)^5 \cup \emptyset$. Couplings can transfer primitive or compound data.

Considering 127 possible values (not null values) for *Char*, the number of possible combinations of input (possible number of events) is 3.3×10^{10}. For reachability analysis, the verification engine should apply all possible combinations of events to all reachable combinations of state variables.

4.3 Finite Number of Internal/External Events

For discrete event system specification, the time advance function is continuous. An external input event can happen at any instance of time. This can result in having an infinite number of external events and therefore an unbounded state space. In order to have a bounded state space, we define time as a discrete function for external inputs. Internal events, they are finite if the time advance is restricted to have a finite number of values as well as satisfying finiteness on all other variables that form the state space.

The receipt of external input events should become discrete to limit the number of transitions from each state. Otherwise, there will be infinite number of transitions from each state of the model as external inputs can be received at any time. Similarly, the number of internal transitions should be finite; otherwise, the state space will be again infinite. So, modeling the Time Advance function as a random function from state to rational numbers set is considered not verifiable.

4.4 Model Checking Algorithm

Beside constraining the state space of models, a verification engine is also required. The verification algorithm introduced here can be applied to both atomic and coupled models. In constrained modeling of any coupled model, its state is the collective state of the atomic/coupled models inside it and the bounded definition of ports remains the same.

The verification engine is in charge of the model checking activity. In order to verify a model entirely, the verifier should visit all states and all its state transitions. Transitions could be the result internal/external transitions within one or several components. The verifier holds two data structures for visited states and unvisited states. Unvisited states are those which the verification engine should iterate on since some of the transitions related to those states are not explored yet. The visited states are those whose transitions are explored entirely. For safety analysis, another data structure for unsafe states is instantiated and initialized by the user.

The model-checking algorithm for safety analysis is given in Algorithm 1. It receives the model (MOD) and the generator (GEN) as inputs. The output is an invalid state if one is found during the exploration of state space. If the model is safe, the algorithm returns null. There are three sets (all three of type Hash) for keeping the unvisited (Q), visited (V), and unsafe states (U) in this algorithm. In steps 1 and 2, initial states and unsafe states are added to unvisited state and unsafe state sets, respectively. A while loop is devised to go through all reachable states (stored in unvisited state set). Steps 4 through 16 are repeated as long as the under visit state set is nonempty. At each cycle of the loop, one state is chosen (step 4) and all possible inputs are applied to it (steps 5-16). Another while loop is responsible for applying all possible input values to the current state. Since the value set for input port are constrained, the engine can determine and apply each of input value 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 9-11). If the resulting state is an unsafe state, the algorithm terminates and alerts the user of the unsafe state, the source state, and the input that caused it. Otherwise, in steps 12-14, if the resulting state is not seen before, it is added to the unvisited state set. In line 16, after all the inputs are applied to the current state, it is added to the visited state set and the algorithm moves on to the next state.

Algorithm 1: Model Checking Algorithm for Safety Analysis of Constrained DEVS Models

Input: MOD: *Verifiable*, GEN: *VerifierGen*

Output: invalidState: *StateVar*

Initialization: instantiate Q, V, and U

```
1:  add model.initialStates to Q
2:  add model.unsafeStates to U
3:  while Q ≠ ∅ do
4:      state-event ← Q.head ( )
5:      while state-event.inputSet ≠ ∅ do
6:          MOD.state ← state-event.state
7:          GEN.output ← state-event.inputSet.head ( )
8:          call simulate ( )
9:          if MOD.state ∈ U then
10:             return MOD.state
11:         end if
12:         if MOD.state ∉ Q ∧ MOD.state ∉ V then
13:             add MOD.state to Q
14:         end if
15:     end while
16:     add state-event to V
17: end while
18: return null
```

4.5 Data Exclusion

In most systems that process, communicate, or store data, the state space is especially bigger because of all the state variables added by data. Going back to the stack example in Section 2, the data stored in the stack increases the state space by a large factor. Assuming the numbers that can be stored in the stack could only be between 0 and 9, the state space is multiplied by a factor of 10^8. This could prove problematic for verification.

In some systems verifying the operation of the system, the specifics of data may not be needed. 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. This suggests storing data as state variables for model verification; the data can still be used for model validation.

The process of removing data from verification is simple. In our constrained DEVS modeling approach, each state variable marks those variables that belong to the state space. In the case of flits, we consider source and destination nodes as part of the state space by excluding the specifics of the data that is communicated within the NoC. The data can be used by simulation engine but the verification engine ignores the data permutations for constructing state space.

4.6 Property Expressions

The aim of modeling checking is to check the validity of certain dynamic properties of models. These properties they are related to various characteristics of the system, such as satisfying performance requirements or prevent certain state transitions. One problem, however, is how these properties are expressed for the model checking engine to process. There are specific languages for property expressions such as TCTL.

The other way is to use data collectors. 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 (generator and transducer). Therefore, experimentation and measurement become parts of the model and not separate concepts. We use Java to implement EF in the

DEVS-Suite simulator that gives the modeler (designer) flexibility and support for creating experiments and calculating measurements based on the states of models.

In the proposed model-checking framework, the generator inject various combinations of inputs to the model and the transducers are charged with the task of gathering data from the model. These transducers can collect state data as well as data derived from states and check them against some desired properties. The modeler can specify and implement complex state-based properties using programming languages such as Java. Properties can include performance measures (maximum waiting time, average latency, etc.) that can be checked for network systems for specified time instances and intervals. For example, for measuring the total delay of flits or the distribution of flit traffic, one or more transducers can have separate calculations to gather the needed data and determine whether the measured data satisfy their expected values.

4.7 DEVS-Suite Extension

The DEVS-Suite simulator does not provide built-in constructs to constrain state variables for atomic models (ACIMS 2016). An atomic model 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 model checking. For the model checking engine to work, it must be able to extract all the state variables and their values to be used for verifying allowed state transitions.

The DEVS-Suite simulator framework is extended to support specifying constrained atomic DEVS models as well as executing them according to the model checking algorithm 1 (see sub-section 4.4). The challenge is to add the model checking capability to the framework while keeping the simulation capabilities intact. With both of these features at our disposal, the user can use a constrained DEVS model for both model checking and simulation. A class diagram is depicted in Figure 1. We created the *VerifiableAtomic* class as a child class of *ViewableAtomic*. This class has a single instance of a *State* class for implementing its state. The *State* class holds all state variables (descendants of abstract *StateVar* class). For each state variable, the user must create an instance of *StateVar* type. Three possible candidates for state variables are supported: *IntStateVar*, *DoubleStateVar*, and *StringStateVar*. These are basic examples but one can create other complex state variable classes (for any compound data type) as long as it inherits from the *StateVar* class which adds constraints to state variables. The same idea is used for input ports. We define the concept of state for the input ports as well which is the most recent value they injected into the atomic model. The *PortState* class is the counterpart of the *State* class for input ports; it contains all port state variables which similar to state variables belong which can have primitive and compound types.

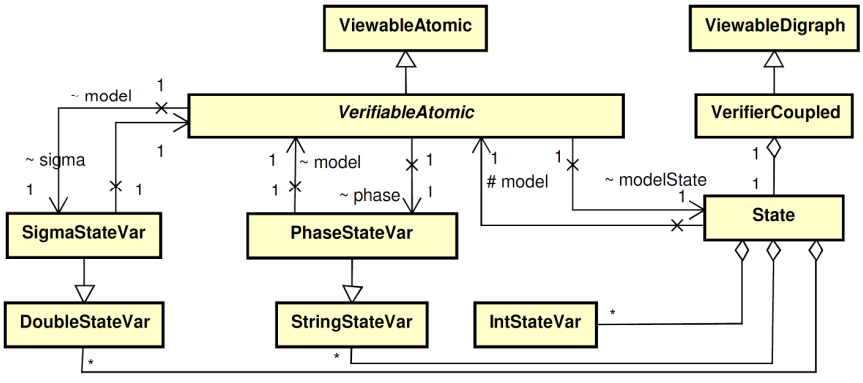

Figure 1: Partial UML diagram of DEVS-Suite modeling class structure

Each descendant of *StateVar* class must implement a method for adding valid range of values to the state variable. For example, for the *IntStateVar* class, the *addRange* method receives minimum and maximum values as arguments with other values within these two values added to define a valid set of integer values.

The method can be called several times for adding valid value sets for each state variable. The *VerifiableAtomic* class has functionality for adding state variables, port values, initial state(s), and unsafe state(s). The verification engine requires to know which state(s) to start with and which states are considered as unacceptable. During model-checking, whenever the model enters into an unsafe state, the engine alerts the user and terminates; otherwise the verification stops if no unsafe state is found after all possible state transition paths are traversed to completion.

5 AN ATOMIC NOC ROUTER MODEL EXAMPLE

In order to demonstrate how the modeling and implementation of constrained DEVS models are carried out for NoC components, we modeled and verified an atomic router model in the DEVS-Suite simulator. A switch in NoC contains input/output buffers, virtual channel allocators, routers, crossbar, and switch allocator components. Next we consider a router which analyzes the header of each incoming flit and determines the output port it should be sent to (via crossbar).

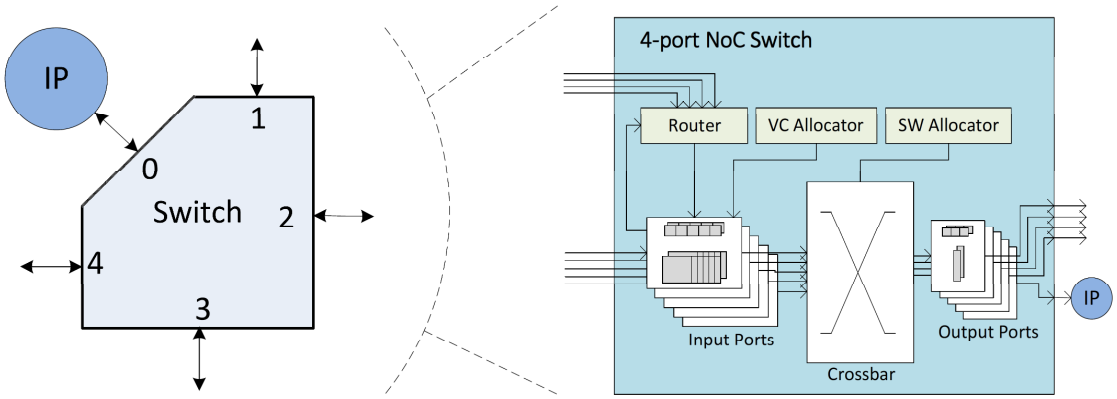

Figure 2: A Mesh switch with its internal components

We consider a 5×5 mesh topology. In order to have all possible permutations considered, we verify the router in the switch (see Figure 2). In a mesh topology, the switch has four output ports named North, South, West, East. These ports are coupled to links that are in turn coupled with 4 other switches. The switch also has an output port for the Intellectual Property (IP). The routing scheme used is called Minimal Adaptive Routing (Dally and Towles 2004). It locally optimizes the routing of flits based on the traffic on the outgoing links. The router component receives the traffic information via four input ports; each can receive flits from a designated link as specified in the NoC mesh. Another input port is dedicated to the flit header. The router send a flit on a locally optimized path toward its destination. The state of the router component is modeled as shown in the following equation.

$$S = \overbrace{\{Active, Idle\}}^{\text{Phase}} \times \overbrace{\sigma}^{\text{Sigma}} \times \overbrace{\{1,2,3\}}^{\text{Load East}} \times \overbrace{\{1,2,3\}}^{\text{Load North}} \times \overbrace{\{1,2,3\}}^{\text{Load West}} \times \overbrace{\{1,2,3\}}^{\text{Load South}} \times \overbrace{\{0,1,2,3,4\}}^{\text{Target Port}} \times \overbrace{\{0 \le x < 5\}}^{\text{xPos}} \times \overbrace{\{0 \le y < 5\}}^{\text{yPos}}$$

Traffic on a link is represented by a number between 1 and 3. A higher number represents a more congested link. The target output port can take 0 (for the link to the IP), 1 (for the west bound link), 2 (for north bound), 3 (for the east bound link), and 4 (for the south bound link). Upon receiving the traffic info, it is stored in its designated state variable and used for routing the current flit and possibly future flits. For simplicity, the flit header only contains the destination information.

The verification engine iterates on all states possible for the router. The input ports for the router can inject 4×4×4×4×25 combinations of input values. The verification engine automatically instantiates a generator (VerifierGenerator) and identifies all possible combinations of its output for the Router (see Figure 3). All of these inputs are injected one by one in some order into the router (see Algorithm 1). All outputs of the VerifierGenerator are applied to each initial state of the Router to ensure correct functionality in all possible combinations of state transitions. The verification engine has built in checks for unsafe state transitions.

However, for more functional properties a transducer is required (see Figure 3). The Transducer determines whether or not the router behavior is satisfying a-priori defined functional properties.

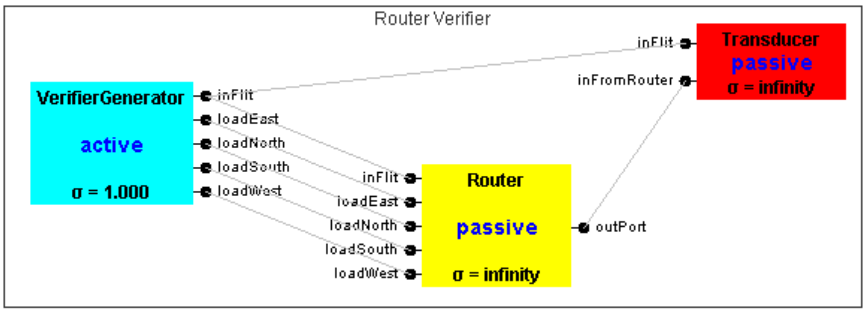

Figure 3: Component view of the router verifier model in the DEVS-Suite simulator

We created several different scenarios and verified the switch model in DEVS-Suite. For these experiments, we used DEVS-Suite 3.0.0 on Java 8. The operating system was Windows 7 Enterprise on an Intel Core i5-2400 (3.10 GHz) processor and 8 GB of physical memory.

Table 1: Sample runs of the RouterVerifier model

Size	Routing Errors	Output	State Space Size	No. of Iterations	Execution Period (seconds)
3×3	None	All reachable states traversed!	2,560	3,585	3.58
5×5	North bound output port	Wrong routing action for: 12→3 North bound is identified	--	--	0.44
5×5	None	All reachable states traversed!	6,400	9,729	5.73
10×10	None	All reachable states traversed!	25,600	38,529	25.47

Some errors (intentionally) introduced in the Router component can be identified and reported by the Transducer. As an example, an error is introduced to the Router model. The router forwards a flit to the north port instead of the south port. This error was captured and reported by the verification engine. Table 1 reports four different runs for the router. For each run, the state space size, the number of times the verification is iterated, and the execution period for all the verification iterations are measured. These experiments are error-free for three different network sizes. The other experiment reports the introduced error once it is found and the verification terminates (no information is provided for the state space size and the number of iterations quantities). These experiments show the usefulness of verification engine. Before components such as Router are simulated along with the rest of the system, it can be verified for correctness.

6 CONCLUSION

In this work, we presented extensions to both DEVS modeling framework and DEVS-Suite simulation engine for model checking with a Network-on-Chip exemplar. Using the constrained DEVS modeling method, NoC components are modeled in finite state space. These models are suitable for model checking. DEVS-Suite is equipped with a verification engine capable of realizing, simulating, and model checking DEVS models such as NoC models. Little is sacrificed for making DEVS models verifiable; to the extent that the constraints introduced to prototypical atomic DEVS models do not disrupt their execution as standalone simulations in the DEVS-Suite simulator. In addition, non-determinism and stochasticity can still be modeled in constrained DEVS and verified. We believe containing both simulation and verification of models in frameworks such as DEVS-Suite simulator can greatly reduce development time while increasing model reliability.

As for future work, a DEVS-based multiresolution modeling approach toward developing today's class of complex systems such as NoC may benefit from model verification. A multiresolution modeling approach may contain several phases of model development in a single tool as opposed to several. We will explore the implications of validation (via simulation) and verification (via model checking) of multiresolution models in our future works.

REFERENCES

ACIMS. 2016. "DEVS-Suite Simulator 3.0.0." *http://acims.asu.edu/software/devs-suite/.*

Alur, Rajeev, and David L. Dill. 1994. "A theory of timed automata." *Theoretical computer science* 126 (2): 183-235.

Alur, Rajeev, Costas Courcoubetis, and David L. Dill. 1993. "Model-checking in dense real-time." *Information and computation* 104 (1): 2--34.

Bazzaz, Hamid Hajabdolali, Marjan Sirjani, Ramtin Khosravi, and Shamim Taheri. 2009. "Modeling networking issues of network-on-chip: a coloured petri nets approach." *Proceedings of the 2nd International Conference on Simulation Tools and Techniques.*

Bouyer, Patricia, Serge Haddad, and Pierre-Alain Reynier. 2008. "Timed Petri nets and timed automata: On the discriminating power of zeno sequences." *Information and Computation* 206: 73-107.

Dally, William J., and Brian P. Towles. 2004. *Principles and practices of interconnection networks.* Morgan Kaufmann.

Goossens, Kees, John Dielissen, Om Prakash Gangwal, Santiago G. Pestana, Andrei Radulescu, and Edwin Rijpkema. 2005. "A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification." *Proceedings of DATE'05.* 1182-1187.

Halpern, Joseph Y., and Moshe Y. Vardi. 1991. "Model checking vs. theorem proving: a manifesto." *Artificial intelligence and mathematical theory of computation* 212 (1): 151-176.

Hemani, Ahmed, Axel Jantsch, Shashi Kumar, Adam Postula, Johnny Oberg, Mikael Millberg, and Dan Lindqvist. 2000. "Network on chip: An architecture for billion transistor era." *Proceeding of the IEEE NorChip Conference.*

Hwang, Moon H., and Bernard P. Zeigler. 2009. "Reachability Graph of Finite and Deterministic DEVS Networks." *IEEE Transactions on Automation Science and Engineering* (IEEE) 6 (3): 468-478.

Larsen, Kim G., Paul Pettersson, and Wang Yi. 1997. "UPPAAL in a nutshell." *International Journal on Software Tools for Technology Transfer (STTT)* 1 (1): 134-152.

Marculescu, Radu, Umit Y. Ogras, Li-Shiuan Peh, Natalie Enright Jerger, and Yatin Hoskote. 2009. "Outstanding research problems in NoC design: system, microarchitecture, and circuit perspectives." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (IEEE) 28 (1): 3-21.

Pasqua, Roberto, Damien Foures, Vincent Albert, and Alexandre Nketsa. 2012. "From sequence diagrams uml 2.x to FD-DEVS by model transformation." *European Simulation and Modelling Conference.*

Peterson, James L. 1981. *Petri net theory and the modeling of systems.* Prentice Hall PTR.

Roychoudhury, Abhik, Tulika Mitra, and Satyanarayana R. Karri. 2003. "Using formal techniques to debug the AMBA system-on-chip bus protocol." *Proceedings of DATE'03.* 828-833.

Rozenblit, Jerzy W. 1991. "Experimental frame specification methodology for hierarchical simulation modeling." *International Journal Of General System* 19 (3): 317--336.

Saadawi, Hesham, and Gabriel Wainer. 2011. "Principles of discrete event system specification model verification." *Simulation* (SAGE Publications) 41-67.

Salaun, Gwen, Wendelin Serwe, Yvain Thonnart, and Pascal Vivet. 2007. "Formal verification of CHP specifications with CADP illustration on an asynchronous network-on-chip." *13th IEEE International Symposium on Asynchronous Circuits and Systems.* 73-82.

Sargent, Robert G. 2005. "Verification and validation of simulation models." *Proceedings of the 37th Winter Simulation Conference.* 130-143.

Schmaltz, Julien, and Dominique Borrione. 2008. "A functional formalization of on chip communications." *Formal Aspects of Computing* 20 (3): 241-258.

Seo, Chungman, Bernard P. Zeigler, Doohwan Kim, and Kenneth Duncan. 2015. "Integrating web-based simulation on IT systems with finite probabilistic DEVS." *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium.* Society for Computer Simulation International. 173-180.

Seo, Chungman, Bernard P. Zeigler, Robert Coop, and Doohwan Kim. 2013. "DEVS modeling and simulation methodology with MS4 Me software tool." *Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium.* Society for Computer Simulation International.

Taktak, Sami, Jean-Lou Desbarbieux, and Emmanuelle Encrenaz. 2008. "A tool for automatic detection of deadlock in wormhole networks on chip." *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 13 (1): 6:1-6:21.

Whitner, Richard B., and Osman Balci. 1989. "Guidelines for selecting and using simulation model verification techniques." *Proceedings of the 21st Winter Simulation Conference.* 559-568.

AUTHOR BIOGRAPHIES

Soroosh Gholami is a Computer Science PhD student at Arizona State University. He can be contacted at <sgholami@asu.edu>.

Hessam Sarjoughian is Associate Professor of Computer Science & Engineering at Arizona State University and Co-Director of the Arizona Center for Integrative Modeling and Simulation. His research focuses on modeling & simulation theory, model composability, distributed co-design modeling, visual simulation modeling, and agent-based simulation. He can be contacted at <sarjoughian@asu.edu>.