Modeling and Verification of Network-on-Chip using Constrained-DEVS

Soroosh Gholami
Hessam S. Sarjoughian

School of Computing, Informatics, and Decision Systems Engineering
Arizona Center for Integrative Modeling and Simulation
Arizona State University

Spring Simulation Multi-Conference

April 23-26, 2017
Electronic Complex Systems

• Network/interaction is inherent to electronic complex systems
• Complexity arises from:
  • Complexity of individual components
    • Functionality of individual components
    • Software, hardware, or physical
  • Interactions between these components
    • Time-sensitive information
    • Overall functionality
• Development steps:
  • Identifying requirements
  • Multiple phases of modeling using variety of methods
  • Multiple phases of model validation and verification
  • Conversion of models to HW/SW pieces
  • Develop communication modules
  • System/subsystem validation and verification
  • Deployment
Complexity and Network-on-Chips

• NoC is a communication system, connecting components of a chip
• NoC design requires
  • design of individual components within the network
  • design of the communication subsystem and protocols
• SoC as a set of software and hardware components interacting through NoC
  • Switches, Processing Elements, and Network Interfaces communicate through links
• Integrated Chip design process has three major phases
  • Electronic System Level (ESL) Design
  • Register Transfer Level (RTL) Design
  • Physical Design
V&V for NoC Models

• Models evaluation based on requirements
  • Verification: building the model correctly
  • Validation: building the correct model
  • Model complexity should not be sacrificed for the sake of V&V
  • Unified framework support is desirable
Overview

• Problem Description/Goals
• Background
• Proposed Research
  • Approach
• Conclusion and Future Work
Limitations of V&V for NoC Design

• Verification is not trivial for DEVS
  • DEVS language is undecidable
  • It is continuous time
  • Simulation is the major means for model evaluation

• Model Evaluation is limited
  • Models are repeatedly abstracted for evaluation

• Complex property (compound) expression
  • Aspects required to check for them are not even modeled (exclusion of information flow)
  • No method to check for them, no language to express them
Scope & Goals

- We limit the scope of the problem to:
  - Modeling framework: Discrete Event System Specification (DEVS)
  - Target system: Network-on-Chip + Processing Element (PE)
  - Validation method: Discrete Event Simulation
  - Verification method: Model Checking
  - Tool: DEVS-Suite\textsuperscript{1,2}

- Goals:
  - Extending DEVS modeling with model checking capabilities
  - Extending DEVS-Suite with both modeling checking and simulation of constrained-DEVS

\textsuperscript{1} ACIMS, DEVS-Suite Simulator, https://sourceforge.net/projects/devs-suitesim/
Elements of Research

Constrained-DEVS Modeling & Simulation
Constrained-DEVS Model Checking (State exploration)
Timed event-handling

Support for DEVS Simulation
Support for DEVS model checking
Experimental frame-based evaluation
Background
Network-on-Chip (1)

- Works as a communication subsystem for SoC
  - Design factors
    - Topology, routing algorithm, flow control, buffer size, hardware brand, flit size, ...

- Major parts:
  - Chip Hardware
    - The electronic components of the circuit
  - Network Software
    - The software modules controlling the circuit
  - Application Software
    - The software running on this base
Network-on-Chip (2)

• Similar to combinational logic, parts (or the entire) NoC may operate independent of a clock signal
  • Globally Asynchronous Locally Synchronous (GALS) for large chips
  • Clock signal propagation issues

• NoC evaluation targets various aspects:
  • Performance
    • avg. latency, worst case latency, queueing time, network capacity
  • Functionality
    • deadlock freeness of routing, fairness of arbitration, error correction
  • Time
    • In time delivery of time sensitive information
  • Physical
    • Energy consumption, heat generation
Model Checking (1)

• Exhaustively determining whether a model meets certain properties
  • Properties are derived from requirements (QoS, safety, liveness, etc.)
  • Why? Deciding whether a system meets a certain property is undecidable
  • When? For critical systems as a full-proof method of verification

• Issues
  • State explosion problem
    • The state space rapidly grows in size
  • Various methods to manage the size
    • Symbolic model checking
    • Bounded model checking
    • Abstraction
Model Checking (2)

- Various formalisms/method are introduced for model checking systems:
  - Timed Petri nets
  - Timed Automata (and its variations)
  - DEVS-based approaches (FD-DEVS, FP-DEVS)

- Major efforts for model checking
  - Use abstraction to simplify the model
    - Abstracting out information flow in basic Petri net and TA
  - Remove stochasticity
    - FD-DEVS\(^1\) (finite deterministic DEVS)
  - Use model conversion
    - Conversion to timed automata for RTA-DEVS; model check using UPPAAL
    - Conversion to non-deterministic automata for FD-DEVS; model check using SPIN/PROMELA

---

DEVS M&S (1)

• Parallel DEVS models are made by atomic/coupled models

\[ \text{Atomic DEVS: } \langle X, S, Y, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda, ta \rangle \]

- Input Events
- State Set
- Output Events
- External Transition Function
- Internal Transition Function
- Time Advance Function
- Output Function
- Confluent Transition Function

\[ ta: S \rightarrow R_{0,\infty}^+ \]

\[ \delta_{ext}: Q \times X \rightarrow S \text{ where } Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\} \]
Coupled DEVS models define couplings between Atomic/Coupled models

• No behavior (external/internal transition functions or output function) for coupled models

\[
\text{Coupled DEVS} : \langle X, Y, D, \{M_d\}, EIC, EOC, IC \rangle
\]
DEVS M&S (3)

• DEVS Modeling
  • Features
    • Continuous time, discrete event
    • Parallel
    • Synchronized time between models
    • Reactive
  
• DEVS Simulation
  • Can be conducted in
    • Logical time: time is advanced to the most urgent event
    • Real-time: simulation time is synchronized with the physical clock
  
• Various implementations
  • eCD++, DEVS-Suite, MS4Me
DEVS M&S (4)

- DEVS-Suite
  - Model development through coding
  - Discrete Event Simulation
  - Model visualization, Simulation animation
  - Tracking
    - Time View (basic types)
    - Superdense time
  - Add-on libraries
    - Real-time simulation
    - Network-on-Chip
    - Real-time hardware interaction
    - RTL DEVS
    - EMF-DEVs (Eclipse Modeling Framework)

Constrained DEVS and Model Checking
Model Checking in DEVS – Example

• DEVS models are not well-suited for model checking

\[ S = \left\{ \text{Phase} \right\} \times \sigma \times \text{values} \times \text{index} \times \text{popped} \]

\[ X = \left\{ (\text{input, } \mathbb{N}), (\text{pop, 1}) \right\} \]

\[ Y = \left\{ (\text{output, } \mathbb{N}) \right\} \]

\[ \delta_{\text{ext}} \left( (\text{Idle, } \sigma, \text{values}[0..7], \emptyset), e, (\text{input, } x) \right) = \begin{cases} (\ldots, \text{index} + 1, \emptyset) & \text{where values[\text{index}] = x if index < 7} \\ (\ldots, \text{index}, \emptyset) & \text{if index = 7} \end{cases} \]

\[ \delta_{\text{ext}} \left( (\text{Idle, } \sigma, \text{values}[0..7], \emptyset), e, (\text{pop, } x) \right) = \begin{cases} (\text{Idle, } \ldots, \text{index} - 1, \text{values}[\text{index}]) & \text{if index > 0} \\ (\text{Idle, } \ldots, \text{index}, \emptyset) & \text{if index = 0} \end{cases} \]

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

\[ \lambda (\text{Active, } \sigma, \text{values}[0..7], \text{index}, \text{popped}) = (\text{output, popped}) \]
Model Checking in DEVS – Shortcomings

• Earlier approaches have certain shortcomings
  • Non-determinism and stochasticity
    • Stochasticity: randomness in models
    • Non-determinism: possibility of multiple states at one instance of time

• Property checking capabilities
  • Specific languages for model checking
  • Limited expressive power
    • Deadlock detection vs. minimum accepted quality of service for specific data

• Data exchange constraints
  • Some modeling languages do not support complex data flow
    • Such as Petri net and timed automata
  • NoC component models requires exchanging complex data types
    • How does one verify those models?
Model Checking in DEVS – Requirements

• What do we need to make DEVS verifiable (via model checking)?

• Answer:
  • Constrain state set and input set values
  • Discretize time for input events
  • Finite number of internal transitions

• Example:
  • Complex data type containing an array of strings (of size 8 holding strings of size 24) and integers under 10

    Array of strings: \((Char^{24})^8\)
    Numbers: \([1|2|3|4|5|6|7|8|9] \in Integer\]

    Entire state space: \((Char^{24})^8 \times (1|2|3|4|5|6|7|8|9) \rightarrow (Char^{24})^8 \times [Integer < 10]\)
Model Checking in DEVS – NoC

• How the stack model changes?
• Answer:
  • No more than 8 numbers in the stack
  • Only positive numbers less than 10
  • Time resolution for input events (new input or pop) has the granularity of 1 cycle

• How do we leverage this for modeling NoC?
  • Data is only limited to flits and flow control signals
  • Events can only happen at clock edges

• What is our property checking method?
  • We use the experimental frame (EF)
Tool Support
DEVS-Suite Extensions

• DEVS-Suite were extended to support

1. Constrained-DEVS modeling
   • Base classes for constrained state variables
   • Invalid state specification
   • Initial state set
   • Input/output value sets

2. Constrained-DEVS execution
   • State space exploration for model checking mode
   • Invalid state reporting for model checking mode
   • Parallel DEVS execution for simulation mode
   • Model checking engine uses the simulation for state exploration
DEVS-Suite State Space Exploration Protocol

- In model checking mode, DEVS-Suite carries out the following steps:
  - Initialization
    - Model is loaded, state variables are recognized, input ports identified
    - *Verification Engine* and *Generator* models are instantiated
    - Initial states are put into *Unvisited* data structure
  - Main Loop: take state from *Unvisited*, set the state of the model
    - Nested Loop: apply all combinations of input to the model
      - Store resulting states (if not seen before) into the *Unvisited*
    - Add the original state to the *Visited* data structure
    - Continue until *Unvisited* is empty

- Transducer model(s) stores the trace and verifies properties
1. Add initial states

2. Instantiate Verification Engine and Generator classes

3. Take (remove) a state

4. Set the state

5. For every input combination

6. Intermediate States to Unvisited

7. Move State to Visited

8. If empty
Atomic Model Verification

• DEVS-Suite experimentation is based on Experimental Frame (EF)
  • Data generation by *Generator*
  • Data collection and analysis by *Transducer*

• Model checking a minimal adaptive router
  • The *Generator* injects flits and traffic information
  • *Transducer* collects outgoing flits and verifies whether the routing decision is correct
Adaptive Router – DEVS Model

\[ S = \{\text{Active, Idle}\} \times \mathbb{D} \times \{1,2,3\} \times \{1,2,3\} \times \{1,2,3\} \times \{0,1,2,3,4\} \times \{x < 10\} \times \{y < 10\} \]
\[ X = \{(\text{inFlow}, \{0.1\}^4), (\text{loadEast}, \{1,2,3\}), (\text{loadNorth}, \{1,2,3\}), (\text{loadWest}, \{1,2,3\}), (\text{loadSouth}, \{1,2,3\})\} \]
\[ Y = \{(\text{outPort}, \{0,1,2,3,4\})\} \]

\[ \delta_{\text{ext}}((\text{Idle, } \sigma, LE, LN, LW, LS, \text{targetPort}), e, (\text{loadEast}, x)) = (\text{Idle, } \sigma, x, LN, LW, LS, \text{targetPort}) \]
\[ \delta_{\text{ext}}((\text{Idle, } \sigma, LE, LN, LW, LS, \text{targetPort}), e, (\text{loadNorth}, x)) = (\text{Idle, } \sigma, LE, x, LW, LS, \text{targetPort}) \]
\[ \delta_{\text{ext}}((\text{Idle, } \sigma, LE, LN, LW, LS, \text{targetPort}), e, (\text{loadWest}, x)) = (\text{Idle, } \sigma, LE, LN, x, LS, \text{targetPort}) \]
\[ \delta_{\text{ext}}((\text{Idle, } \sigma, LE, LN, LW, LS, \text{targetPort}), e, (\text{loadSouth}, x)) = (\text{Idle, } \sigma, LE, LN, LW, x, \text{targetPort}) \]
\[ \delta_{\text{ext}}((\text{Idle, } \sigma, LE, LN, LW, LS, \text{targetPort}), e, (\text{inFlow}, x)) \]

\[ \delta_{\text{int}}(\text{Active, } \sigma, LE, LN, LW, LS, \text{targetPort}) = (\text{Idle, } \omega, LE, LN, LW, LS, \text{targetPort}) \]
\[ \lambda(\text{Active, } \sigma, LE, LN, LW, LS, \text{targetPort}) = (\text{outPort, targetPort}) \]

\[ (\text{Active, } r\text{Delay, LE, LN, LW, LS}) \]
\[ \begin{cases} 0 & \text{if } x\text{Pos} = x \land y\text{Pos} = y \\ 1 & \text{if } x\text{Pos} > x \land y\text{Pos} = y \\ 2 & \text{if } x\text{Pos} = x \land y\text{Pos} < y \\ 3 & \text{if } x\text{Pos} < x \land y\text{Pos} = y \\ 4 & \text{if } x\text{Pos} = x \land y\text{Pos} > y \\ 1 & \text{if } x\text{Pos} > x \land y\text{Pos} < y \land LW \leq LN \\ 1 & \text{if } x\text{Pos} > x \land y\text{Pos} > y \land LW \leq LS \\ 2 & \text{if } x\text{Pos} > x \land y\text{Pos} < y \land LN \leq LW \\ 2 & \text{if } x\text{Pos} < x \land y\text{Pos} < y \land LN \leq LE \\ 3 & \text{if } x\text{Pos} < x \land y\text{Pos} < y \land LE \leq LN \\ 3 & \text{if } x\text{Pos} < x \land y\text{Pos} > y \land LE \leq LS \\ 4 & \text{if } x\text{Pos} < x \land y\text{Pos} > y \land LS \leq LE \\ 4 & \text{if } x\text{Pos} > x \land y\text{Pos} > y \land LS \leq LW \end{cases} \]
Coupled Model Verification

- Works similar to the atomic version
  - The *generator* injects data based on the input ports of the coupled model
  - The state of the coupled model is the aggregate state of inner models

- Model checking a coupled model with two inner components
  - *VerifierGenerator* injects all combinations of input values for model1
Analyzing Traces
Analyzing Traces
Analyzing Traces
Demo
Conclusion & Future Work
Conclusion

• Model checking capability
  • Constrained-DEVS formalism for model checking
  • State exploration algorithm for constrained-DEVS models

• An attempt toward unified design environments
  • With support for simulation & model checking
  • EF-based experimentation and model evaluation
Future Work

• Ongoing
  • Hardware-level model library for NoC using Constrained-DEVS
  • Integration with multiresolution modeling – the right abstraction is chosen automatically based on the property which is being verified

• A new version of DEVS-Suite (v 4.0) is scheduled for release by the end of summer 2017
  • Contains the verification engine for Constrained-DEVS models
Thank You