A High-Level Modeling and Simulation Approach Using Test-Driven Cellular Automata for Fast Performance Analysis of RTL NoC Designs

Moon Gi Seok, Hessam S. Sarjoughian
ACIMS lab., Arizona State University, Tempe, USA
mseok@asu.edu, Hessam.Sarjoughian@asu.edu

Daejin Park
Sch. of EE., Kyungpook National University, Rep. of Korea
boltanut@knu.ac.kr

***Abstract*—The simulation speedup of designed RTL NoC regarding the packet transmission is essential to analyze the performance or to optimize NoC parameters for various combinations of intellectual-property (IP) blocks, which requires repeated computations for parameter-space exploration. In this paper, we propose a high-level modeling and simulation (M&S) approach using a revised cellular automata (CA) concept to speed up simulation of dynamic flit movements and queue occupancy within target RTL NoC. The CA abstracts the detailed RTL operations with the view of deciding a cell's state of actions (related to moving packet flits and changing the connection between CA cells) using its own high-level states and those of neighbors, and executing relevant operations to the decided action states. During the performing the operations including connection requests and acceptances, architecture-independent and user-developed routing and arbitration functions are utilized. The decision regarding the action states follows a rule set, which is generated by the proposed test environment. The proposed method was applied to an open-source Verilog NoC, which achieves simulation speedup by approximately 8 to 31 times for a given parameter set.**

I. Introduction

The network-on-chip (NoC) paradigm has been an important design methodology using on-chip communications among large numbers of intellectual properties (IPs) to mitigate the design complexity and scalability issue of a system on a chip (SoC) due to its advantages including communication throughput, flexibility, and scalability. Designers have commonly implemented various NoCs using hardware description languages (HDLs) at register-transfer level (RTL) for automatic synthesis to circuits.

Generally, RTL NoC design requires iterative simulations to find acceptable communication performance using a set of parameters for the participation of potential IPs or to optimize the design parameters (e.g., input buffer size) for designated IPs. However, slow RTL simulation speed causes reducing the parameter set space or limits having a sufficient parameter exploration for optimization. To increase the simulation speed, the high-level modeling can be a solution. Typically, the high-level NoC modeling approaches can be divided into two main categories: queuing modeling and architectural- and algorithm-level hardware modeling using high-level modeling languages, such as SystemC.

The queueing modeling studies have tried to develop various parametric equations to observe the average service time and queue utilization based on the queueing theory [1], [2]. The previous studies show relative sound accuracies (≈ 90-92%) and negligible computation overhead due to their analytic characteristics comparing with the RTL simulation, which iteratively processes events for signal value transitions at every simulation step. However, the previous studies have limitations in evaluating dynamic changes in packet transmission, such as buffer status, so it is not able to monitor hotspot situations at a specific time. Moreover, the statistical approach of queueing models cannot guarantee the exact communication performance results compared to the RTL simulation.

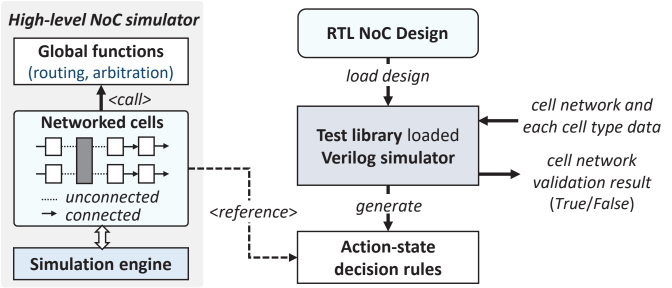


Fig. 1: Proposed CA modeling process using RTL NoC.

The architectural- and algorithm-level NoC models are usually developed using SystemC at the early stage of NoC designs. The high-level models have loosely-/approximated-timed or cycle-level accuracy, and utilized to explore design spaces and verify functionalities for assisting implementation of RTL models as intermediate models [3], [4]. Due to the algorithm-level abstraction of RTL operations, the models increase simulation speed with reduced modeling efforts. However, some operations of final RTL models and their high-level intermediate models might differ, so it cannot guarantee matched packet transmission results between two models. Moreover, the high-level modeling requires in-depth knowledge about detailed and specific operations of target RTL design.

In this paper, we propose a high-level and cycle-accurate modeling and simulation (M&S) method for the fast evaluation of packet transmissions of target RTL NoC design using an extended concept of cellular-automata (CA) and the proposed process, as shown in Fig. 1. The proposed methods abstracts detailed RTL operations into 1) cell's action-state decision based on high-level state (that are flit and capacity information) and 2) executes operations (that are flit fetching and connection change) corresponded to decided action state using a built-in flit-fetching function and user-developed global functions. Compared to the previous algorithm-level modeling, the CA model is reversely derived from final RTL design, not an intermediates model toward the RTL design. Without in-depth knowledge of all RTL operations, functions of action-state decision (related to the action-execution timing) reference the derived rules by the proposed test environment using the user-specified information of CA network and RTL probing signals of cells' high-level states, and the implementation of global functions requires the knowledge about the routing and arbitration algorithm of target RTL design.

The CA network structure is identified by the designer using the information of RTL arbitration units, FIFO queues, or other units that cause propagation delay due to the flit-storing flipflops (FFs). Similar to the conventional CA concept, each cell determines the action state based on neighbors' and own state at each time step according to a rule [5]. The central CA extensions consist of two parts: 1) neighbors are dynamically determined in runtime to reflect the NoC arbitration and 2) each cell has an influence function, which enables the simulation engine to find and schedule only state-changeable neighbors after a cell's action state is changed in order to prevent unnecessary executions of CA to achieve scalability.

A decision rule consist of multiple coupled states of the target cell and its neighbors and associated action states. For the decision-rule generation, the proposed test environment observes RTL signals that are related to cells' high-level state and action state and stores the pair of high-level and action states during the RTL NoC simulation of randomly injected packets. Moreover, the CA network is validated by checking next position of flits to be moved during the test. After the test, the environment generates compact decision rules through checking a state dependency.

The rest of the paper is organized as follows. Section II describes the proposed CA modeling and simulation method. Section III describes the proposed test environment and rule generation. Section IV applies the proposed approach to an open source RTL NoC model. Section V concludes the paper.

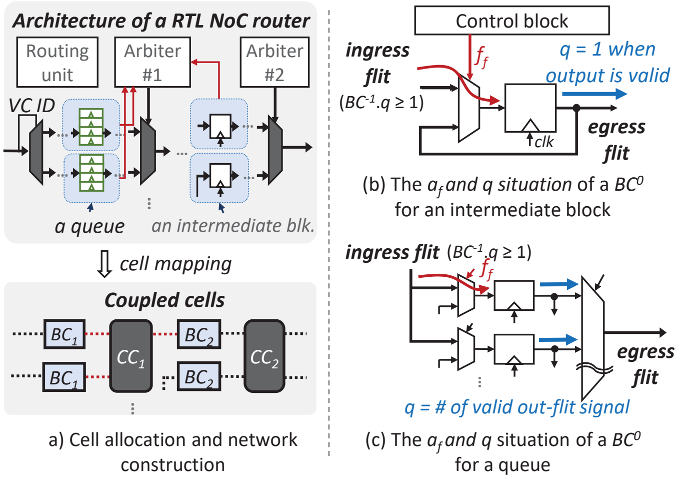
II. Cellular Automata Modeling and Simulation of NoC RTL design

A. CA Modeling Perspectives and Formal Specifications

We propose high-level CA modeling for RTL NoC designs, which have the following characteristics.

- NoC designs consist of synchronous routers.
- NoC designs have a flit-based transmission mechanism, which breaks a large packet into small flits for low latency.

All CA can be classified into two types: a buffer cell (BC) and a coupling cell (CC). Each BC has a FIFO buffer to store one or multiple flits. The BC represents queues or intermediate units where each FFs holds a flit and causes cycle-level delay, as shown in Fig. 2(a). The CC is responsible for establishing the connection or disconnection between two specific BCs among two or multiple adjacent BCs. The cell coupling between CC and BC can be established using input and output information of the representing arbiter unit of the target RTL NoC. The specific high-level states of BC and CC are defined as below.



a) Cell allocation and network construction

(b) The a_f and q situation of a BC^0 for an intermediate block

(c) The a_f and q situation of a BC^0 for a queue

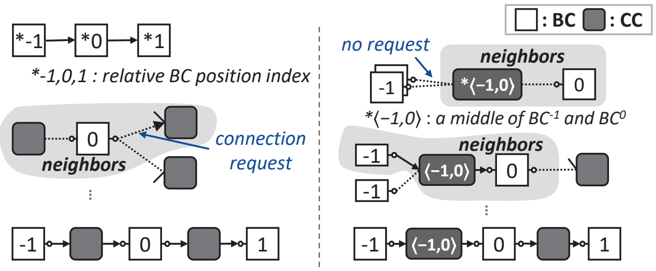
Fig. 2: Cell allocation, and f_f and q value of a BC (BC^0).

Definition 1: A high-level state (S_b) of BC is represented using a 4-tuple $< id, F, Q, FT >$ and a high-level state (S_c) of CC is represented as a 3-tuple $< id, C, P >$, where

- id : the cell identification index;
- F : the set for the flit-fetching action states (f_s) of BC, $F = \{f_i, f_f\}$, f_i: idle, f_f: state of flit fetching at the next clock trigger;
- Q : the set of possible number of flits (q) in the FIFO buffer within BC;
- FT : the set of the front flit types, $FT = \emptyset \cup \{$*header, payload, last, single*$\}$;
- C : the set of connectivity action states (c_s) of CC, $C = \{c_c, c_d, c_{c(0)}, c_{d(1)}\}$, c_c: connected state, c_d: disconnected state, $c_{c(0)}$: newly connected state at the current clock cycle, $c_{d(1)}$: to be disconnected state at the next clock trigger;
- P : the id pair of connected BCs, $\emptyset \cup \{BC.id, BC.id\}$.

The $f_f (\in F)$ means to bring a flit from previous BC (BC^{-1}) to current BC (BC^0) at the next clock trigger, and the $q (\in Q)$ is related to the number of valid output flits, as shown in Fig. 2(b)(c). The RTL signal of a cell's f_f is related to write-enable conditions of an input flit at the clock trigger. The action state of a cell, f_s and c_s, is decided by checking the current values of its own and neighbors' state based on the described mapping condition in its decision rule.

Depending on the existence the connection states of adjacent CCs, neighbors of a cell can be variously coupled, as Fig. 3(b). For a BC, if an adjacent CC is disconnected, the BC cannot reach another BC across the CC as a neighbor, and an adjacent CC can be a neighbor after requesting a connection. The connection request to next CC is executed by the simulation engine when a flit is fetched and a next CC is not designated. Depending on the number of next reachable



(a) Various BC^0 neighbor couplings (b) Various CC$^{\langle -1,0 \rangle}$ neighbor couplings

Fig. 3: Various neighbor couplings of BC and CC.

CCs, multiple connection requests can be performed based on the destination information of the front flit, which is obtained using the global routing function. The current BC (BC^0) and current CC ($CC^{\langle -1,0 \rangle}$, connecting BC^{-1} to BC^0 and BC^0) can reach BC^{-1} and BC^1 at most as neighbors.

The background philosophy referring three BCs at most is to find the relationship between status of sender (BC^{-1}) and availability of receiver (BC^0), and the receiver's availability sometimes depends on the action of BC^1. For example, a BC^0 represents an intermediate unit that has a FF to store a flit. A new flit arrival is possible when the current FF's output flit is ready to exit at the next clock trigger to prevent the flit loss, the output-flit exit is related to the f_s of BC^1. In general, two cells should be referred to decide the availability.

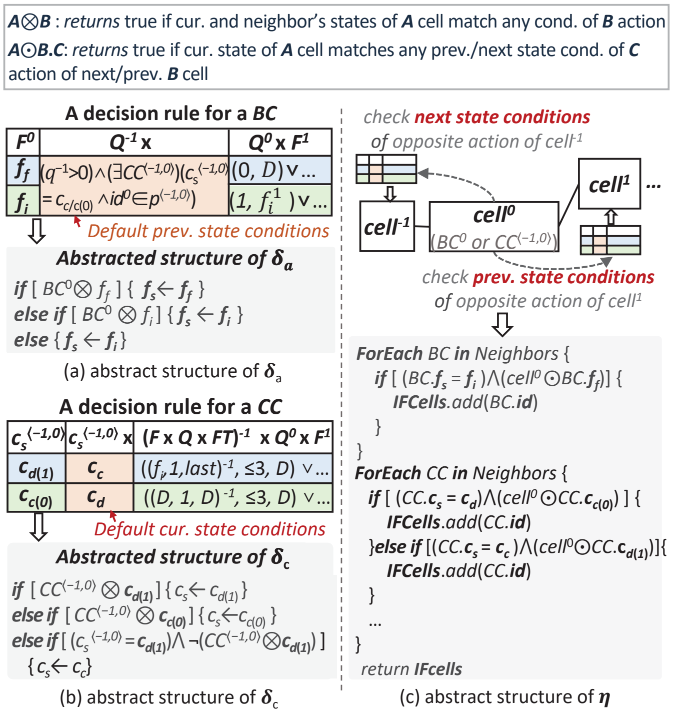
Based on the defined high-level states and neighbor couplings, we define the decision function of the action state and an influence function for BC and CC, as shown below.

Definition 2: The simulation engine invokes the BC functions δ_a and η as well as the CC functions δ_a and η, where

- $\delta_a : \overbrace{Q^{-1}}^{prev.} \times \overbrace{Q^0}^{cur.} \times \overbrace{F^1}^{next} \rightarrow F^0$, the f_s decision function;
- $\delta_c : \overbrace{((F \times Q \times FT)^{-1})^p}^{prev.\,states} \times \overbrace{C_s^{\langle -1,0 \rangle}}^{cur.} \times \overbrace{(Q^0 \times F^1)^n}^{next\,states}$, the c_s decision function, p/n: the number of previous/next neighbor BCs;
- $\eta : \overbrace{S^0_{b/c}}^{cur.} \times \overbrace{S^m_{b/c}}^{prev./next} \rightarrow \{id\}^r$, the influence function, m: the number of neighbors, r: the number of neighbors whose action state can be changeable due to the current state ($S^0_{b/c}$)

Each of the δ_a, δ_c, and η functions has an *if-then* structure based on corresponding rule-describing conditions of previous, current and next states, and the abstracted structures of those functions are shown in Fig. 4(a)(b)(c). In the decision rules of CC and BC, default conditions are defined for trivial action-changing situations.

After simulation time has advanced, the f_ss of all BCs are set to f_i before making decisions. Depending on whether previous, current and next states meet any condition of f_f, the f_s can be newly updated to f_s. The c_s of CCs can be also updated to $c_{d(1)/c(0)}$ or remain unchanged depending on the current and neighbor's states. The $c_{c(0)}$ affects the f_s state of a BC^1, but the $c_{d(1)}$ does not affect any neighbor cells.


check next state conditions of opposite action of cell-1
cell-1
cell0
(BC0 or CC⟨-1,0⟩)
cell1
...
check prev. state conditions of opposite action of cell1

A⊗B : *returns* true if cur. and neighbor's states of **A** cell match any cond. of **B** action

A⊙B.C: *returns* true if cur. state of **A** cell matches any prev./next state cond. of **C** action of next/prev. **B** cell

A decision rule for a *BC*

F^0	Q^{-1} x	Q^0 x F^1
f_f	$(q^{-1}>0) \wedge (\exists CC^{\langle -1,0 \rangle})(c_s^{\langle -1,0 \rangle}$	$(0, D) \vee \ldots$
f_i	$= c_{c/c(0)} \wedge id^0 \in p^{\langle -1,0 \rangle})$	$(1, f_i^1\ ) \vee \ldots$

Default prev. state conditions

Abstracted structure of δ_a

```
if [ $BC^0 \otimes f_f$ ] { $f_s \leftarrow f_f$ }
else if [ $BC^0 \otimes f_i$ ] { $f_s \leftarrow f_i$ }
else { $f_s \leftarrow f_i$ }
```

(a) abstract structure of δ_a

A decision rule for a *CC*

$c_s^{\langle -1,0 \rangle}$	$c_s^{\langle -1,0 \rangle}$ x	$(F \text{ x } Q \text{ x } FT)^{-1}$ x Q^0 x F^1
$c_{d(1)}$	c_c	$((f_i, 1, last)^{-1}, \leq 3, D) \vee \ldots$
$c_{c(0)}$	c_d	$((D, 1, D)^{-1}, \leq 3, D) \vee \ldots$

Default cur. state conditions

Abstracted structure of δ_c

```
if [ $CC^{\langle -1,0 \rangle} \otimes c_{d(1)}$ ] { $c_s \leftarrow c_{d(1)}$ }
else if [ $CC^{\langle -1,0 \rangle} \otimes c_{c(0)}$ ] { $c_s \leftarrow c_{c(0)}$ }
else if [ $(c_s^{\langle -1,0 \rangle} = c_{d(1)}) \wedge \neg (CC^{\langle -1,0 \rangle} \otimes c_{d(1)})$ ]
  { $c_s \leftarrow c_c$ }
```

(b) abstract structure of δ_c

```
ForEach BC in Neighbors {
  if [ $(BC.f_s = f_i) \wedge (cell^0 \odot BC.f_f)$ ] {
    IFCells.add(BC.id)
  }
}
ForEach CC in Neighbors {
  if [ $(CC.c_s = c_d) \wedge (cell^0 \odot CC.c_{c(0)})$ ] {
    IFCells.add(CC.id)
  }else if [ $(CC.c_s = c_c) \wedge (cell^0 \odot CC.c_{d(1)})$ ]{
    IFCells.add(CC.id)
  }
  ...
}
return IFcells
```

(c) abstract structure of η

Fig. 4: Abstract structures of $\delta_{a/c}$ and η.

Due to the setting that the f_s of all BCs are initialized into f_i before action decisions, the $c_{c(0)}$ decision is performed conservatively. In detail, the $c_{c(0)}$ decision is related to the receiver's availability of a BC^0, and the availability can depend on the f_s of BC^1. In this situation, if a BC^0 of a CC is actually available at the time, and the δ_c of the CC is invoked before δ_a of BC^1 during simulation, c_s would be unchanged as c_d. This conservative decision leads that the c_s as $c_{c(0)}$ would be retained at the clock cycle. On the other hand, $c_{d(1)}$ is determined aggressively, so c_s can be set as $c_{d(1)}$ to c_c interchangeably at the current time.

After the decision of f_s or c_s and the execution of state-related operations by the engine, the changed current state can affect some neighbors' state decisions, and η is defined to provide the information of influenced neighbor cells from the current change. If the updated current cell state meets any pre-/next conditions of the opposite action of a next/previous cell, the cell is considered as a changeable cell, as shown in Fig. 4(c), and the group of changeable cells is informed to the simulation engine.

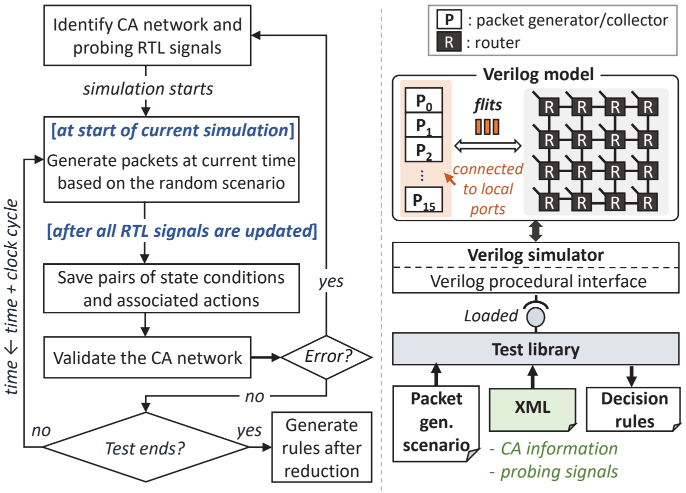
B. Scheduling-based Simulation Algorithm

The CA high-level states, except for c_s or f_s, are updated by the simulation engine after performing operations of the flit fetching and the connection change. The overall algorithm for the CA state transition at each iteration, is described in Algorithm 1. After the action state decisions of CA end at a current time, the time advances to the next clock-trigger time, and the action transitions of CA are repeated.

```
Algorithm 1: Simulation algorithm at each time step.
1  $L_p$ : a list of previously scheduled CA
2  $L_n$ : a list of newly scheduled CA for decisions
3  $L_i$ : a list of influenced CA
   /* Move each flit or reflect connection changes */
4  foreach cell in $L_p$ do
5      if cell type is $BC$ then
6          $BC^0.buffer.insert(BC^{-1}.buffer.pop())$
7          $BC^0.f_s \leftarrow f_i$, $BC^{-1}.f_s \leftarrow f_i$
8          if $BC^0.q > 0$ and next $CC$ is not designated then
9              request connection(s) to next $CC$(s) using the
               global routing function
10             schedule next $CC$(s) to to $L_n$
11         schedule $BC^0$ and $BC^{-1}$ to $L_n$
12     else
13         if $CC^{\langle -1,0\rangle}.c_s = c_{c(0)}$ then $CC^{\langle -1,0\rangle}.c_s \leftarrow c_c$
14         else $CC^{\langle -1,0\rangle}.c_s \leftarrow c_d$
   /* Decide new $f_s$ and connection state */
15 $L_p \leftarrow \emptyset$
16 foreach cell in $L_n$ do
17     if cell is not executed as an influenced cell then
18         if cell type is $BC$ then
19             execute $BC^0.\delta_a$
20             if $BC^0.f_s = f_f$ then sched. cell to $L_p$
21         else
22             execute $CC^{\langle -1,0\rangle}.\delta_c$
23             if $CC^{\langle -1,0\rangle}.c_s = c_{d(1)}$ then sched. cell to $L_p$
24             else if $CC^{\langle -1,0\rangle}.c_s = c_{c(0)}$ then
25                 update $CC^{\langle -1,0\rangle}.p$ using an arbitration
                   function, then schedule cell to $L_p$
26         schedule $cell.\eta$ to $L_i$
           /* Process the influenced cells */
27         while $L_i$ is not empty do
28             $cell_i \leftarrow L_i.pop()$
               /* Repeat upper sequential sentences
                  from line No. 17 to 25 */
29             exec. $cell_i.\delta_{a/c}$, ... and sched. $cell_i.\eta$ to $L_i$
```

The overall algorithm can be divided into two phases: the first phase to perform flit-fetching operations and reflect the connect change and the second phase to decide f_s and c_s state. Regarding the flit flow, the first phase is to move flits at the clock trigger and the second phase is to identify the flit-fetching BCs at the next clock trigger. After fetching a flit, f_ss of related BCs are initialized and connection requests to reachable CCs are performed. In the perspective of connection, the first phase removes the connection for new connectivity and stabilizes $c_{c(0)}$ by setting c_c to prevent the change of connected BC pair until c_s is set to $c_{d(1)}$.

Since the execution order of BC can be different at each iteration, the buffer of each BC has one redundant slot because a flit can arrive before an existing flit being fetched. The f_s or c_s of each cell in L_n, which is a list of state-changed cells while executing operations in the first phase, is decided if the cell is not already executed as an influenced cell. The execution priority of the influenced cells is higher because those cells are determined as changeable cells using η, so early detection of active cells whose action state are f_f or $c_{c(0)}$ can eliminate useless iterations. Therefore, the influenced cells are executed in the *while-loop*. Depending on the decided action state, cells are scheduled to L_p for the execution of action operations at the next time.


Identify CA network and probing RTL signals
simulation starts
[at start of current simulation]
Generate packets at current time based on the random scenario
[after all RTL signals are updated]
Save pairs of state conditions and associated actions
Validate the CA network
Error?
yes
no
Test ends?
no
yes
Generate rules after reduction
time ← time + clock cycle
P : packet generator/collector
R : router
Verilog model
P0
P1
P2
⋮
P15
flits
connected to local ports
R
Verilog simulator
Verilog procedural interface
Loaded
Test library
Packet gen. scenario
XML
Decision rules
- CA information
- probing signals
(a) Overall test process

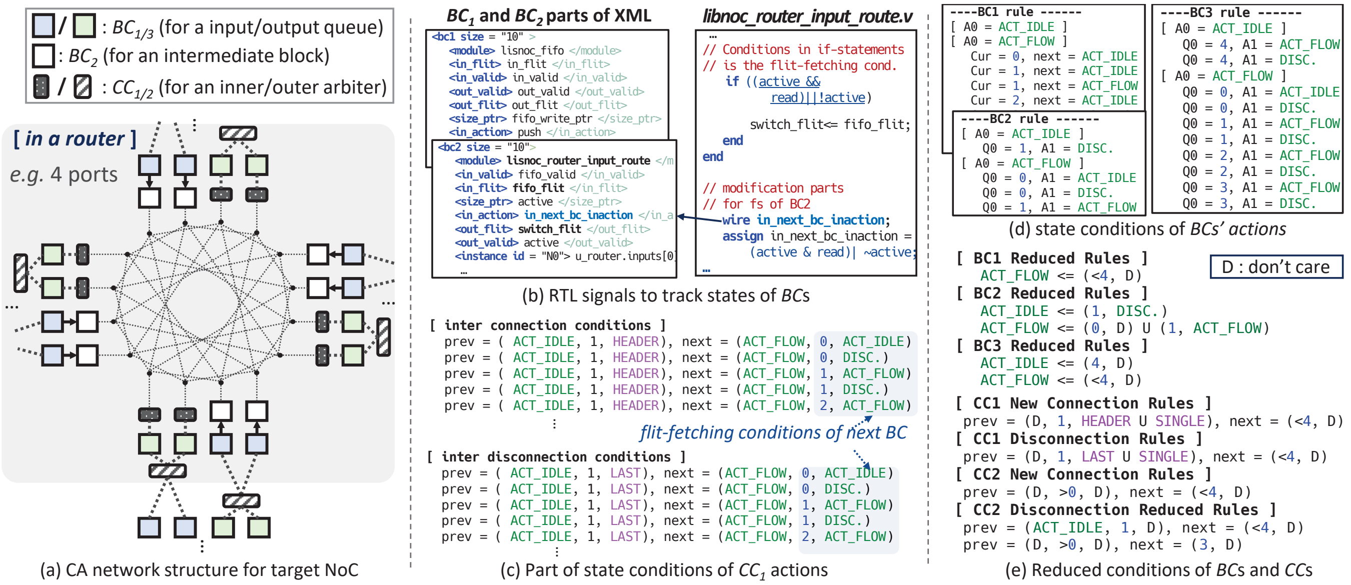
(b) Test environment structure

Fig. 5: Overall test process and environment structure.

III. Test-driven Rule Set Generation using RTL NoC Designs

The purpose of the test environment is to validate the CA network and to provide rules for the CA action-state decision. The overall test process is shown in Fig. 5(a). First of all, the designer identifies the RTL modules that hold the flits using FFs for BCs and arbiter modules for CCs by analyzing a simulation result of a few routers for a small sample of packet transmissions. Second, the designer constructs networks using the simulation result and the input and output information of arbiter modules. Using codes of the RTL modules for BCs, the designer identifies probing RTL signals that correspond to the CA states.

Based on the obtained CA network and probing RTL signal information, a network validation and saving pairs of conditions and their associated actions are performed. These are achieved using the proposed test environment with runtime NoC simulation for randomly injected packets. After the simulation, the environment generates decision rules for each cells after reducing the saved condition-action pairs. The overall structure of the test environment is illustrated in Fig. 5(b). To intervene during RTL simulation, the test library should be developed using a programming language interface (PLI) of the simulator, such as the Verilog procedure interface (VPI). The PLI enables an external library to access RTL signals and register functions of the library as callbacks to be invoked during the runtime simulation.


🟦 / 🟩 : $BC_{1/3}$ (for a input/output queue)
□ : BC_2 (for an intermediate block)
▮ / ▨ : $CC_{1/2}$ (for an inner/outer arbiter)
[in a router]
e.g. 4 ports

(a) CA network structure for target NoC

BC_1 and BC_2 parts of XML

```
<bc1 size = "10" >
   <module> lisnoc_fifo </module>
   <in_flit> in_flit </in_flit>
   <in_valid> in_valid </in_valid>
   <out_valid> out_valid </out_valid>
   <out_flit> out_flit </out_flit>
   <size_ptr> fifo_write_ptr </size_ptr>
   <in_action> push </in_action>
```

```
<bc2 size = "10">
   <module> lisnoc_router_input_route </m
   <in_valid> fifo_valid </in_valid>
   <in_flit> fifo_flit </in_flit>
   <size_ptr> active </size_ptr>
   <in_action> in_next_bc_inaction </in_a
   <out_flit> switch_flit </out_flit>
   <out_valid> active </out_valid>
   <instance id = "N0"> u_router.inputs[0]
    …
```

libnoc_router_input_route.v

```
 …
// Conditions in if-statements
// is the flit-fetching cond.
   if ((active &&
         read)||!active)

         switch_flit<= fifo_flit;
   end
end

// modification parts
// for fs of BC2
   wire in_next_bc_inaction;
   assign in_next_bc_inaction =
        (active & read)| ~active;
…
```

(b) RTL signals to track states of *BCs*

```
[ inter connection conditions ]
  prev = ( ACT_IDLE, 1, HEADER), next = (ACT_FLOW, 0, ACT_IDLE)
  prev = ( ACT_IDLE, 1, HEADER), next = (ACT_FLOW, 0, DISC.)
  prev = ( ACT_IDLE, 1, HEADER), next = (ACT_FLOW, 1, ACT_FLOW)
  prev = ( ACT_IDLE, 1, HEADER), next = (ACT_FLOW, 1, DISC.)
  prev = ( ACT_IDLE, 1, HEADER), next = (ACT_FLOW, 2, ACT_FLOW)
                  ⋮
```

flit-fetching conditions of next BC

```
[ inter disconnection conditions ]
  prev = ( ACT_IDLE, 1, LAST), next = (ACT_FLOW, 0, ACT_IDLE)
  prev = ( ACT_IDLE, 1, LAST), next = (ACT_FLOW, 0, DISC.)
  prev = ( ACT_IDLE, 1, LAST), next = (ACT_FLOW, 1, ACT_FLOW)
  prev = ( ACT_IDLE, 1, LAST), next = (ACT_FLOW, 1, DISC.)
  prev = ( ACT_IDLE, 1, LAST), next = (ACT_FLOW, 2, ACT_FLOW)
                  ⋮
```

(c) Part of state conditions of CC_1 actions

```
----BC1 rule ------
[ A0 = ACT_IDLE ]
[ A0 = ACT_FLOW ]
   Cur = 0, next = ACT_IDLE
   Cur = 1, next = ACT_IDLE
   Cur = 1, next = ACT_FLOW
   Cur = 2, next = ACT_IDLE
```

```
----BC2 rule ------
[ A0 = ACT_IDLE ]
   Q0 = 1, A1 = DISC.
[ A0 = ACT_FLOW ]
   Q0 = 0, A1 = ACT_IDLE
   Q0 = 0, A1 = DISC.
   Q0 = 1, A1 = ACT_FLOW
```

```
----BC3 rule ------
[ A0 = ACT_IDLE ]
   Q0 = 4, A1 = ACT_FLOW
   Q0 = 4, A1 = DISC.
[ A0 = ACT_FLOW ]
   Q0 = 0, A1 = ACT_IDLE
   Q0 = 0, A1 = DISC.
   Q0 = 1, A1 = ACT_FLOW
   Q0 = 1, A1 = DISC.
   Q0 = 2, A1 = ACT_FLOW
   Q0 = 2, A1 = DISC.
   Q0 = 3, A1 = ACT_FLOW
   Q0 = 3, A1 = DISC.
```

(d) state conditions of *BCs' actions*

D : don't care

```
[ BC1 Reduced Rules ]
   ACT_FLOW <= (<4, D)
[ BC2 Reduced Rules ]
   ACT_IDLE <= (1, DISC.)
   ACT_FLOW <= (0, D) U (1, ACT_FLOW)
[ BC3 Reduced Rules ]
   ACT_IDLE <= (4, D)
   ACT_FLOW <= (<4, D)

[ CC1 New Connection Rules ]
 prev = (D, 1, HEADER U SINGLE), next = (<4, D)
[ CC1 Disconnection Rules ]
 prev = (D, 1, LAST U SINGLE), next = (<4, D)
[ CC2 New Connection Rules ]
 prev = (D, >0, D), next = (<4, D)
[ CC2 Disconnection Reduced Rules ]
 prev = (ACT_IDLE, 1, D), next = (<4, D)
 prev = (D, >0, D), next = (3, D)
```

(e) Reduced conditions of *BCs* and *CCs*

Fig. 6: Overall CA network of target RTL NoC and test-driven state conditions for action decision.

To achieve the test purpose, the proposed library references three types of time-related callbacks: a callback called at a specific time before any RTL signal updates, a callback called after all RTL signals are updated at the time, and a callback called at the end of the simulation. For Verilog, the callback types are *cbAtStartOfSimTime*, *cbReadOnlySynch* and *cbEndOfSimulation*, respectively.

At every clock cycle before RTL simulation, a callback in the test library is invoked to check the existence of scheduled packets at the invoked time based on the input scenario. If any scheduled packets exist, the callback requests packet generations to relevant RTL generators by manipulating their input signals.

After the current time RTL simulation, a callback is invoked to save states of BCs and connected BC pairs of CCs to identify state conditions of actions. The state of BCs can be determined by checking the values of probing signals, and the connected BC pairs can be obtained by comparing the valid output and input flit values between candidate BC pairs. The connection pair enables us to validate the CA network. The test library manages the current and previous states and connected BC pairs.

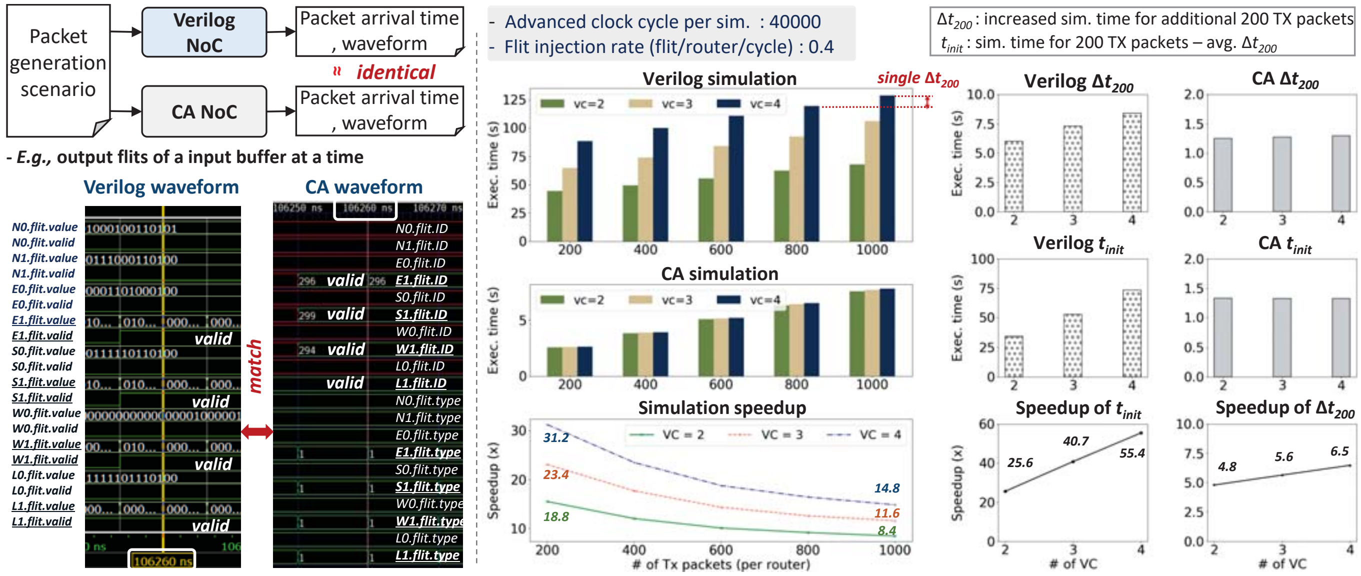
To detect conditions of $c_{c(0)}$ and $c_{d(1)}$, the test library compares current BC pairs with the pairs at the previous clock cycle. If the previous connection pair of the $CC^{\langle -1,0\rangle}$ does not exist or does not match the current connection, the connected state of the CC is newly changed and the neighbors' states at the clock cycle are one of the $c_{c(0)}$ conditions. If the previous pair does not match the current pair, the previous neighbor states are one of the $c_{d(1)}$ conditions if the previous f_s of the BC^0 was f_f, which means that a last transmission has happened at the current clock trigger. If the previous f_s of the BC^0 was f_i, we do not consider the current neighbor states of the CC as a recent disconnection. After updating the connection pairs of CC, if the default pre-condition of a BC is met, related current and next states are saved as one of the conditions of the BC action. The conditions are managed by multiple rule sets that are shared by multiple cells based on the symmetry of representing RTL units.

After testing until the number of conditions of each rule is converged, we can reduce conditions using the following principle: if one or multiple state values of conditions are identical and all possible values of rest states are observed, then the rest states do not need to be considered during the action-state decision. For example, in the situation that two conditions (0, f_i) and (0, f_f) are observed, and f_i and f_f are all possible cases when the first state is 0, the action decision only reference the first state.

IV. Experimentation

We applied the proposed M&S approach to an open source Verilog NoC, CONNECT [6]. In this experiment, we targeted an NoC with a 4-by-4 routers topology. We identified a CA network validated by the test environment, as shown in the Fig. 6(a). There are three kinds of BCs, which represents an input queue, an output queue, and an intermediate unit and two types of CCs, which serve an arbiter to resolve contention between the input ports and another arbiter to resolve contention between the virtual channels (VCs) at output ports. We developed three global functions: a routing function based on the XY routing scheme and two round-robin arbitration functions for the two arbiters.

To track the states of BCs, we identified RTL signals; the input XML for the signals is described in Fig. 6(b). The input and output queue instances of the routers share the same module. The test library was developed using the VPI provided by the *ModelSim* 10.5c in Quartus®Prime Pro 17.1. Using the test environment with input XML and various packet



(a) Simulation result comparison process

(b) Comparison of simulation execution time between Verilog and CA NoC

Fig. 7: Comparison of simulation results between Verilog and CA NoCs.

generation scenarios, we obtained various conditions for CA actions, and parts of the conditions are shown in Fig. 6(c)(d). After sufficient random testing and the convergence for the conditions, overall rules of each cell were obtained after the reduction, as shown in Fig. 6(e).

Based on the rule set, we developed the high-level CA and simulation engine following the proposed method using C++. To check and debug the equivalence of flit transmission between the Verilog and the CA NoC model, we enabled the NoC CA simulator to share packet generation scenarios with the Verilog test environment and to visualize the FT states of all BCs. For various packet generation scenarios, we confirmed identical arrival times of the packets between the Verilog and CA model, and an example of the matched output flit values of a queue is illustrated in Fig. 7(a).

For the simulation speed comparison, we measured the execution time of Verilog and CA NoC models by increasing the number of injecting packets per node, keeping the 0.4 average flit-injection rate per cycle to prevent packet loss caused by traffic congestion. We used an experimental machine whose CPU is Intel®Xeon®E3-1505M 3GHz and memory size is 64 GB. Using the proposed M&S approach, we obtained the speedup from 8.4 to 31.2×, as shown in Fig. 7(b).

We observed that the number of increased packets and increased execution time have a linear relationship. Based on the linearity, we defined two parameters, which are Δt_{200} to measure the runtime speedup and t_{init} to measure initialization speedup. Based on the obtained values of Δt_{200} and t_{init}, we can expect that if the number of injected packets goes zero, the CA speedup reaches about 24.6/40.7/55.4× (for the number of VCs is 2/3/4). If the number of injected packets grows arbitrarily large, the speedup will be converged to about 4.8/5.6/6.5×.

The proposed M&S method requires the designers to develop the routing and arbitration functions. Depending on the target NoC, various routing and arbitration functions, such as adaptive routing and quality-of-service supporting arbitration algorithms can be implemented. We will validate the flexibility by applying various RTL NoCs. Moreover, we will improve the test environment to support the automatic identification of the CA network.

V. Conclusion

We proposed a high-level M&S method using a newly extended CA to reduce the RTL simulation execution time focusing on packet transmissions. The proposed CA defines detailed RTL operations in two steps: First, the action states are determined (that are related to the flit-fetching and connection changes) using the defined high-level states of current and neighbor cells based on test-driven rule sets. Second, the detailed operations corresponding to the action states are simulated. For the scalable simulation, we proposed an influence function to schedule changeable cells to be mainly activated by the simulation engine. Our evaluation showed various CA speedup (up to 31 times) relative to the fixed NoC size and the number of injected packets with valid transmission results.

References

[1] A. E. Kiasari, Z. Lu, and A. Jantsch, “An Analytical Latency Model for Networks-on-Chip,” *IEEE Trans. on Very Large Scale (VLSI) Syst.*, vol. 21, pp. 113–123, 2013.

[2] E. Fischer and G. P. Fettweis, “An Accurate and Scalable Analytic Model for Round-Robin Arbitration in Network-on-Chip,” in *NoCS*, 2013.

[3] A. Portero, R. Pla and J. Carrabina, “SystemC Implementation of a NoC,” in *ICIT*, 2005.

[4] M. Briere et al., “System Level Assessment of an Optical NoC in an MPSoC Platform,” in *DATE*, 2007.

[5] T. Toffoli, “Cellular Automata Machines,” *Cambridge*, MA: The MIT Press, 1987.

[6] M. K. Papamichael and J. C. Hoe, “CONNECT: Re-examining conventional wisdom for designing NoCs in the context of FPGAs,” in *FPGA*, 2012.