Symbolic Analysis of Round-Robin Arbitration in Networks

Yongjian Li, Naiju Zeng, William N. N. Hung, Xiaoyu Song, and Zhiping Shi

***Abstract*—A round-robin arbitration is a key component in network-based systems. This paper presents an automated formal verification method for hardware design of round-robin arbiter. The approach is based on symbolic trajectory evaluation (STE) which performs symbolic simulation for liveness checking of round-robin arbitration. The experiments demonstrate that the enhanced STE specification for realistic hardware design can be completed in a reasonable time. The work gives the first attempt to verify liveness properties for round-robin arbiters by STE.**

I. Introduction

A fast arbiter is one of the most dominant parts for high performance network switches such as ATM (Asynchronous Transfer Mode) network [1], [2]. Fast and efficient switch arbiters play a key role in switching packets for Network-on-Chip (NoC) [3]. In order to provide a high speed and cost-effective implementation of the arbitration scheme where multiple packets from different input ports compete for the same output port, designers usually implement these arbiters directly in hardware. However, these hardware arbiter designs are very tedious and error-prone. Design errors in network components may have disastrous effects, especially if networks are used in financial or safety-critical applications, where communication errors could cause loss of money or life.

In order to guarantee the correctness of the arbiter hardware design, simulation and testing approaches are traditionally adopted. However, it is practically impossible to run exhaustive tests or simulations for all the routing cases under arbitration. For instance, in a $N \times N$ round-robin arbiter studied in this paper, where N is the number of input/output ports, the number of the exhaustive simulation cases in Boolean domain is $2^N \times N$. Thus, the use of formal verification with proper abstraction or reduction techniques is gaining interest because the correctness of a formally verified design implicitly involves all cases regardless of the input cases.

Symbolic trajectory evaluation (STE) is an efficient formal hardware verification method that has grown from the combination of multi-valued simulation and symbolic simulation [4]. It has shown great promise in verifying medium to large scale industrial hardware designs with a high degree of automation. STE has been in active use in Intel, Motorola, and IBM. At Intel, for instance, STE was used to verify a floating point arithmetic unit against IEEE standard 754 and a complex IA instruction length decoder unit [5], [6]. In addition, the FORTE formal hardware verification tool, which combines STE and theorem proving in a higher-order logic, has been developed at Intel [7]. Generalized Symbolic Trajectory Evaluation (GSTE) is an extension of STE that can deal with properties ranging over unbounded time [8], [9], [10]. In GSTE, specifications on circuits are given by assertion graphs, which are $\forall$-automata[11].

In this work, we present our results on formally verifying hardware design of a round-robin arbiter which is the core component in many real-world network systems. Our approach is based on STE, which is a marriage of ternary-valued and symbolic simulation. The first arbiter under study is a core component of the Fairisle switch fabric. The Fairisle ATM network was designed and used at the Computer Laboratory of the University of Cambridge. The arbiter performs round-robin arbitration among requests for a single output port. Then we generalize our designs to $N \times N$ configuration size, and give a parametric proof script to verify such a generalized design.

The main contributions of our work are twofold. The first one is natural specifications of the expected behaviors of the arbiter. We formalize the specifications on one round arbitration by an STE assertion, and those on the sequential behavior and the response property of the arbiter by GSTE graphs. The second is the fully symbolic simulation of the arbiter against the proposed specification. We emphasize that fully symbolic simulation in this context means exhaustive simulation for all input patterns. STE plays a key role in our verification. It reduces the complexity of the verification from $2^N \times N$ to $N \times N$ for a $N \times N$ round-robin arbiter. Due to this exponential reduction, our verification is fully automated by running FORTE, which is an industrial STE tool originated from Intel. At last, our verification is scalable in the sense that we can verify a $N \times N$ round-robin arbiter by instantiating the parameterized proof script with the parameter N. Our experiments show that the parameter N can reach a modest size in industrial design.

A. Related work

Although the principles of round-robin arbitration are viewed as one of the most important technology in global communications, there exist few published results addressing a thorough formal verification of a hardware design for round-robin arbiter in an automated way. To our best knowledge,

Yongjian Li and Naiju Zeng are with State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of sciences , Beijing, China. E-mail: lyj238,zengnj@ios.ac.cn

William N. N. Hung is with Synopsys Inc., Mountain View, California, USA. E-mail: william_hung@alumni.utexas.net

Xiaoyu Song is with ECE Department, Portland State University, Portland, Oregon, USA. E-mail: song@ece.pdx.edu

Zhiping Shi, College of Information Engineering, Capital Normal University, Beijing, China. E-mail: shizhiping@gmail.com

there are only a few cases that sideswipe this topic in the literature.

In [2], [12], Curzon formally verified the 4×4 fabric of the Fairisle switch using the HOL theorem prover. As a core component of the switch, the arbiter is verified by describing the behavioral and the structural specifications of the arbiter and proving the HOL formula of the structure implying that of the behavior. The proof style is from bottom to up, the components of the arbiter are verified down to the gate level. The separated proofs were then combined to prove the correctness of the arbiter. As a component of the switch, the proofs of the arbiter itself are used to prove the correctness of the whole switch.

Chen et al. at Fujitsu exploited symbolic model checking to detect a design error in an ATM circuit [13]. Using SMV, they identified the design error by checking properties described in computation tree logic (CTL) [14]. In order to avoid state space explosion, they abstracted the data width of the addresses from 8 bits to 1 bit, and the number of addresses in the WAF (Write Address FIFO) from 168 to 5. However, due to the above reduction, the circuit model is too coarse, and some fine-grained properties cannot be checked. Therefore a more detailed (gate-level) model was needed for a thorough verification of ATM switch circuit.

Tahar et al. presented several techniques for modeling and formally verifying the Fairisle ATM switch fabric using multiway decision graphs (MDG) [15]. They modeled and verified the switch fabric at three levels of abstraction: behavior level, register transfer level (RTL), and gate level. In the first stage, they validated the high-level specification by checking specific safety properties that reflect the behavior of the fabric in its real life operating environment. Using the intermediate RTL model, they hierarchically completed the verification of the original gate-level implementation of the switch fabric against the behavioral specification. Since MDG avoided model explosion induced by data values, this work demonstrates the effectiveness of MDG-based verification as an extension of Reduced Ordered Binary Decision Diagram (ROBDD) based approaches [16].

Among the aforementioned work, the verification of round-robin arbitration is exhaustive in [2], [12] due to the strong modeling and reasoning ability of HOL theorem proving. However, the use of HOL is interactive and requires much expertise to guide the verification process manually. Besides, the proofs were not scalable. That is to say, the proofs are only for the arbiter with 4×4 configuration size. When tracking the design changes and proving the properties of the arbiter with 16×16 configuration size, the original proof of the 4×4 arbiter must be fundamentally modified [17].

In contrast, the model checking approaches adopted by Chen et al. and Tahar et al. were automatic. Since the complexity of all the arbitration cases of the arbiter is $2^N \times N$ in terms of the number of requests N, it would be difficult to verify the correctness of round-robin arbitration in a common Boolean value or first order value based model checker. Due to state space explosion, the model-checking work can not complete exhaustive exploration of arbitration cases in a reasonable time. For instance, in [15], the authors only verified a concrete routing case from input port 0 to output port 0 in Property 3. They did not verify all the routing cases.

To the best of our knowledge, little work has applied STE to the verification of the hardware design of components of network systems including the round-robin arbiter. Our work also demonstrates that STE/GSTE can be effectively applied in the verification of some control-dominated circuits (not data-dominated circuits such as memories or CAMs). No work has been done to apply GSTE to the verification of a response property before in the literature.

Presentation of the paper: In this work we use notations in functional programming style to illustrate both our theory and implementation. Lemmas about lists, sets, etc., are polymorphic, and a function is usually defined in a curried form instead of a tupled form. That is, we often use the notation $f\ x\ y$ to stand for $f(x, y)$. The advantage of a curried function is to allow a partial function application [18]. For a pair (a, b), we have $\mathsf{fst}\ (a, b) \equiv a$ and $\mathsf{snd}\ (a, b) \equiv b$. Some other notations are listed in Table I.

TABLE I
SOME IMPORTANT NOTATIONS

Notation	Value
$xs[0:m]$	$[\ xs[0], \ldots,\ xs[m]\ ]$
$x : xs[0:m]$	$[\ x,\ xs[0], \ldots,\ xs[m]\ ]$
$xs[0:m]\ @\ ys[0:n]$	$[\ xs[0], \ldots,\ xs[m],\ ys[0], \ldots,\ ys[n]\ ]$
$0\ \mathsf{upto}\ i$	$[0, ..., i]$
$xs_i\ (xs\ !\ i)$	$xs[i]$
$x\ \mathsf{mem}\ ls$	$x \in ls$
$\mathsf{len}\ xs[0:m]$	$m\ +\ 1$
$\mathsf{last}\ xs[0:m]$	$xs[m]$
$\mathsf{flat}\ L$ $(L = [xs,\ ys, \cdots])$	$xs\ @\ ys\ @ \ldots$
$\mathsf{zip}\ \ xs[0:m]\ \ ys[0:m]$	$[\ (xs[0],\ ys[0]), \ldots,\ (xs[m],\ ys[m])\ ]$
$\mathsf{map}\ f\ xs[0:m]$	$[\ (f\ xs[0]), \ldots,\ (f\ xs[m])\ ]$

Boolean expression type is written as bool, which may be constructed as follows: true, false, bvariable string and AND($\wedge$), OR($\vee$), NOT($\neg$). Function encode i $width$ encodes an integer i into a bit vector with length $width$. Another function decode $vect$ transforms a boolean vector $vect$ to an integer. For instance, encode 2 2 $=$ [ff, tt], and decode [ff, tt] $= 2$.[1] Here, ff and tt are standard binary values which stand for false and true.

The remainder of this paper is organized as follows: Section II introduces the preliminary theory of round-robin arbitration of a switch fabric. Section III formalizes the STE specification of the round-robin arbitration in one round. Section IV uses GSTE to model check the sequential behaviors of the round-robin arbitration. Section V formalizes the response property. Section VII shows the experimental results. Section VIII concludes the paper.

[1] In an encoding vector $vect$, the least significant bit is corresponding to the index 0, and the most significant bit len $vect - 1$. In the truth tables in section III, in order to follow the convention, we write the most significant bit firstly, then the second for the vector $grant$.

II. Background

A. $N \times N$ Switch Fabric and Arbitration

In this section, we first briefly introduce some background on a Switch Fabric and Arbitration problem.

A switch fabric, which is shown in Fig. 1, consists of three types of components: input port controllers, output port controllers and a switch fabric[2]. Each input (output) port controller is connected to one input (output) link of the switch, and to the switching fabric. A cell arrives at an input port controller on an incoming transmission line. It then passes through the fabric and arrives at an output controller. The output port controller transmits the cell on its outgoing transmission line.

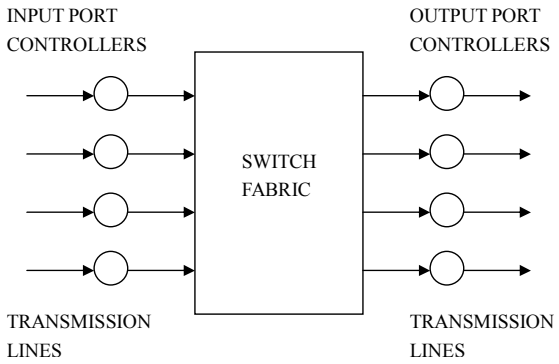

Fig. 1. An Overview of the Switch Fabric

The switch fabric switches cells from input controllers to output controllers. If different input controllers inject cells into the fabric at the same time which are destined for the same output port controller, then only one will initially succeed. The others will be rejected and must retry later. The fabric arbitrates between such cells.

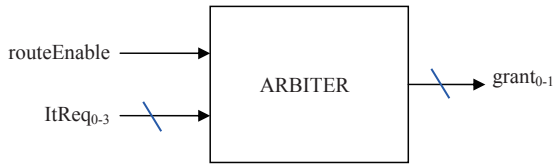

Fig. 2. An Overview of the Arbiter

In fact, there are N arbiters in a $N \times N$ fabric switch, each of which is in charge of the arbitration for requests from all input ports which are destined for an output port. Because the structure of an arbiter is the same as that of any other one, we only analyze one arbiter. An overview of the structure of an arbiter design is shown in Fig. 2.

The arbiter enforces an arbitration policy for a single output port. It takes inputs as a request vector, $ItReq$, indicating which inputs are making requests for the output.

Its output is a grant vector which indicates the encoding of the input whose request is currently valid. For instance, $ltReq[0:3] = [\mathsf{tt}, \mathsf{ff}, \mathsf{tt}, \mathsf{ff}]$, and after one cycle of arbitration, the output changes to the following status: $grant[0:1] = [\mathsf{ff}, \mathsf{ff}]$. This means that the first and third input ports make requests, and the first input port is granted to transfer the data.

Round-robin is an important arbitration policy which is extensively adopted in network systems. In general, the request, which has the highest priority in round-robin order, will be granted in each cycle. We will explain formally the round-robin policy in the following sections. This policy guarantees fairness (no starvation) among requests and allows a request granted whose round-robin turn is later and who is ready now. The ready request will be granted at last if this request is kept ready. The worst-case waiting time of this request can also be predicted. This reliable prediction is another advantage of the round-robin policy. Obviously, this is a response property. The round-robin arbiter works as follows. In each cycle, one of the requests (in round-robin order) has the highest priority to be granted. If the token-holding master does not need the resource in this cycle, the master with the next highest priority who sends a request can be granted the resource, and the highest priority master then passes the token to the next master in round-robin order.

B. STE and GSTE

STE is a formal verification technique that is based on ternary symbolic simulation. In STE, specifications are given as assertions of the form $ant \rightsquigarrow cons$ where both ant and $cons$ are trajectory formulae. ant is called the antecedent, which specifies with symbolic values that are used to drive the simulation. $cons$ is called the consequent, which specifies the expected results of the simulation. An STE tool such as FORTE can automatically check whether a given circuit C satisfies a given STE assertion. If so, we write $\mathsf{cktSat}\ C\ ant \rightsquigarrow cons$.

Four values ff, tt, X, and $\top$ are used in STE simulation [4]. ff and tt have the same values as before. The third value X stands for an unknown value, while the fourth value $\top$ a clash value. Formally, we define $\mathbb{V} =_{df} \{\mathsf{ff}, \mathsf{tt}, \mathsf{X}, \top\}$.

The first concept is the circuit model used for STE. A circuit is modeled by a netlist, which is a set of nodes (or wires) connected by logical entities such as gates and one-phase delays. Gates describe combinational logics deciding the relationship between values of nodes. Delays refer to all sequential elements which can keep a "state". A circuit state is an instantaneous snapshot of a circuit behavior given by an assignment of $\mathbb{V}$ to nodes of the circuit.

Here we use a simple example to illustrate the concepts used in STE verification. Fig. 3 shows a netlist of 2-cell single-bit memory, which is modified from a GSTE tutorial [19].

Let $write = \mathsf{AndList}[\mathsf{Is1}\ wr, din\ \mathsf{isB}\ bD, a\ \mathsf{isB}\ bA]$, $retain = \mathsf{AndList}[wr\ \mathsf{isB}\ bWr,\ When\ bWr\ (a\ \mathsf{isB}\ \neg bA)]$, $read = \mathsf{AndList}[\mathsf{Is0}\ wr, a\ \mathsf{isB}\ bA]$, $outResult = dout\ \mathsf{isB}\ bD$. Here $\mathsf{Is1}(\mathsf{Is0})\ wr$ simply specifies that the value of the node wr is $\mathsf{tt}(\mathsf{ff})$, and $\mathsf{AndList}\ fs$ is a conjunction of trajectory formulae list fs. $\mathsf{When}\ b\ f$ specifies that only when the Boolean expression b is true, the trajectory formula f holds, otherwise all the

[2]This figure is taken from [2], which illustrated the Fairisle network. Here we generalize it to a $N \times N$ network switch.

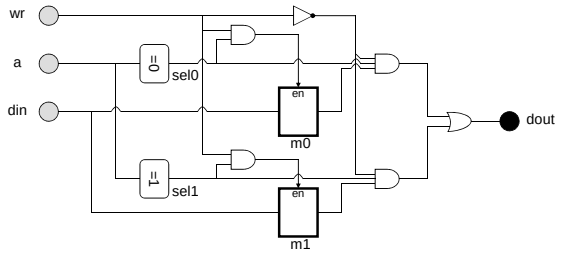

Fig. 3. 2-cell single-bit memory

nodes are set the value X. The notation din isB bD simply abbreviates AndList [When bD (Is1 din), When $\neg bD$ (Is0 din)], which assigns a symbolic value bD to the node din. A novel trajectory formula chaos is introduced here to represent a state where the values of all the nodes in the circuit are unknown. We also define Next^1 f = Next f, Next^{N+1} f = Next (Next^N f). For notation convenience, we also introduce a syntactical abbreviation for symbolic value assignments to vectors : ns bvAre $bvs \equiv$ AndList (map isB (zip ns bvs)).

Let ant=AndList[$write$, Next $retain$, Next^2 $read$], $cons$ = Next^2 $outResult$. The STE assertion $ant \rightsquigarrow cons$ specifies that if a value is written to a memory cell, and no other writes to the cell in the next cycle, then the read from the cell immediately later will return the value.

One of the main limitations of STE is that it can only deal with properties ranging over a finite number of time steps. Generalized Symbolic Trajectory Evaluation (GSTE) is an extension of STE that can deal with properties ranging over unbounded time [8], [9], [10]. In GSTE, specifications on circuits are given by assertion graphs, which are $\forall$-automata. Fig. 4 shows an assertion graph for the netlist in Fig. 3. This assertion graph specifies that if a value is written to a memory cell, and no other writes to the cell in the next cycle, then the read from the cell immediately will return the value.

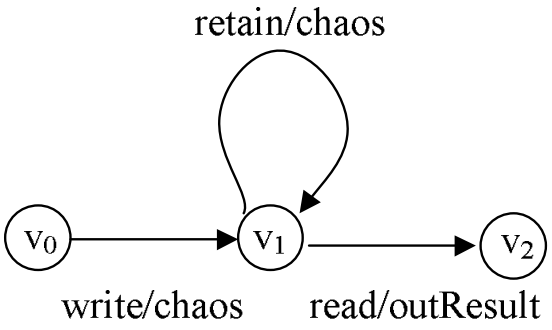

Fig. 4. GSTE assertion graph for memory cell

An STE assertion such as the aforementioned one $ant \rightsquigarrow cons$ can be seen as a linear assertion graph shown as (b) in Fig. 5. While a GSTE assertion graph with loops such as Figure 4 can be seen as a collection of linear assertion graphs.

C. HOL specification of One Round-Robin Arbitration

This subsection is mainly taken from HOL specification of round-robin arbitration in the counterpart section in [2]. Given

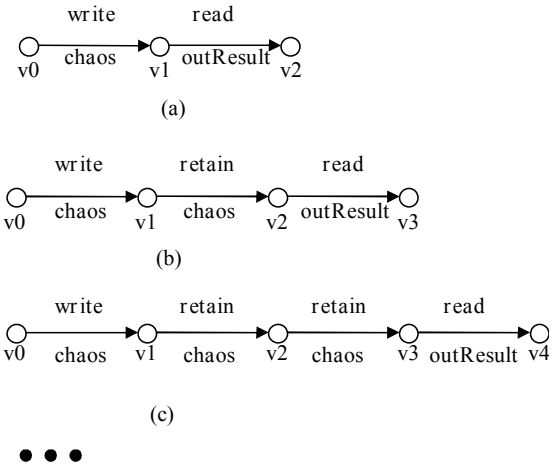

Fig. 5. STE assertions and GSTE assertion graphs

an indication of the last value selected, round robin arbitration returns the next highest value requested, with suitable wrap around from the highest possible request to the lowest.

```
let SUC_MODN N last =
       (last + 1 = N) =>  0|  (last + 1);

letrec RoundRobin 0 requestSet last N = 0
/\ RoundRobin n requestSet last N =
       let tryNext = SUC_MOD N last in
       ((tryNext mem requestSet) =>  tryNext
       |RoundRobin (n - 1) requestSet tryNext N);

let RoundRobinArbiter N requestSet last =
       (requestSet = []) => NORESULT|
       (RESULT (RoundRobin N requestSet last N));
```

The round-robin arbitration is specified by the function RoundRobinArbiter which has several arguments: the number of input ports, N; a list giving the input ports making requests, $requestSet$; and the last successful input port in the previous arbitration. It returns either an indication that it cannot make a selection if the request set is empty (NORESULT); otherwise it returns the result of the round robin arbitration. A result is of a datatype defined as follows:

ROUNDTTYPE = NORESULT | RESULT int

The function RoundRobin is called in RoundRobinArbiter. It tries successively higher values above the last successful request until it finds a proper request in the request set. It is defined in terms of a counter ensures that the function does terminate. Provided that the counter is initially set the value of the highest request possible and the request set is not empty, it will not terminate until a result is obtained.

For instance, for an arbitrator of the 4×4 switch fabric, the last successful input port number in the previous arbitration is 0, and the current request set is [0,2,3], we call RoundRobinArbiter 4 $[0, 2, 3]$ 0=2, which returns 2 as the result in the current round of arbitration.

The function RoundRobinArbiter gives a good explanation on round-robin arbitration in mathematical notation in the sense that a request and a grant result are represented by an integer and request set an integer set. However, this notation is far away from the structure of the arbiter shown in Fig.

2. For instance, request sets are represented by a vector of inputs $req_0 - -req_3$ and the number of the granted input port is encoded in a vector in Fig. 2. We need to give another specification which considers more details of hardware implementation of the arbiter.

```
let RequestsToArbitrate 0 reqVect = []
/\RequestsToArbitrate n reqVect =
       (reqVect ! (n − 1)) =>
       ([n − 1] union (RequestsToArbitrate (n−1) reqVect)
       |RequestsToArbitrate (n − 1) reqVect
```

The argument of $requestSet$ in the function RoundRobinArbiter is a list of integer numbers. However, the request set is represented by the Boolean values of a request vector in the hardware implementation of the arbiter. Let $reqVect$ be a Boolean vector with each bit indicating whether a corresponding input is making a request, RequestsToArbitrate returns the set of requests by scanning each bit in turn. If it holds the value tt, then its position number is added to the request set. For instance, let $reqVect[0 : 3] = [\mathsf{tt}, \mathsf{ff}, \mathsf{tt}, \mathsf{tt}]$, then RequestsToArbitrate 4 $reqVect = [0, 2, 3]$.

```
let SucessFulInput last reqVect =
  (let requestSet =
  RequestsToArbitrate (len req) reqVect in
  RoundRobinArbiter (len req) requestSet last

let GrantForOut reqVect grantVect =
  let sucIn = SucessFulInput (decode grantVect) reqVect in
  (sucIn = NORESULT) => grantVect
   |(val (RESULT result = sucIn) in
     encode (len grantVect) result))
```

Giving a $grantVect$ indicating the value of the last successful input port in the previous arbitration, and $reqVect$ be a Boolean vector with each bit indicating whether a corresponding input is making a request, GrantForOut $reqVect$ $grantVect$ first converts $grantVect$ to an integer, then calls $SucessFulInput$ to compute the successful input, and converts the integer result to another Boolean vector, which encodes the result of this arbitration. For instance, let $reqVect[0:3] = [\mathsf{tt}, \mathsf{ff}, \mathsf{tt}, \mathsf{tt}]$, $grantVect[0:1] = [\mathsf{ff}, \mathsf{ff}]$, then GrantForOut $reqVect$ $grantVect[0:1] = [\mathsf{ff}, \mathsf{tt}]$ which denotes that $reqVect[2]$ is granted.

III. STE Specification of One Round of the Round-Robin Arbitration

According to the above discussion, we can show the truth table of the round-robin arbitration function in Table II, which is corresponding to one round of round-robin arbitration GrantForOut req $grant$. Here req stands for a vector $req_0 - -req_3$, and $grant$ a vector of $grant_0, grant_1$. The returned result is another vector $grant'_0, grant'_1$. This is also the base of the implementation of the logics of the arbitration. Among the six arguments, $req_0 - -req_3$ stand for four request inputs, and $grant_0, grant_1$ for the encoding of the granted request in the previous arbitration round. $grant'_0, grant'_1$ are the outputs to indicate the encoding vector of the number of the granted ports. Here we omit the other inputs $frameStart$, $routeEnable$ and $reset$ of the arbiter, which is not the main factors affecting the state space. This table is a typical use of the special value X, which significantly reduces the size of the truth-table and the synthesis size of the logic of the arbiter.

TABLE II
Ternary-valued Truth Table of the Round-robin Arbiter

req0	req1	req2	req3	$grant_0$	$grant_1$	$grant'_0$	$grant'_1$
X	ff	ff	ff	ff	ff	ff	ff
X	tt	X	X	ff	ff	tt	ff
X	ff	tt	X	ff	ff	ff	tt
X	ff	ff	tt	ff	ff	tt	tt
ff	X	ff	ff	tt	ff	tt	ff
X	X	tt	X	tt	ff	ff	tt
X	X	ff	tt	tt	ff	tt	tt
tt	X	ff	ff	tt	ff	ff	ff
ff	ff	X	ff	ff	tt	ff	tt
X	X	X	tt	ff	tt	tt	tt
tt	X	X	ff	ff	tt	ff	ff
ff	tt	X	ff	ff	tt	tt	ff
ff	ff	ff	X	tt	tt	tt	tt
tt	X	X	X	tt	tt	ff	ff
ff	tt	X	X	tt	tt	tt	ff
ff	ff	tt	X	tt	tt	ff	tt

Note that this truth-table is exhaustive, which explores all the cases of the input patterns. Here we briefly analyze why the truth-table is exhaustive. Note that we have to analyze all the possible boolean assignments for the nodes $grant_0, grant_1$, namely, X is not used to assign the value of the two nodes. X value is mainly used for value assignments of the nodes $req_0 - -req_3$. For instance, when (a) $grant_0 = \mathsf{ff}$ and $grant_1 = \mathsf{ff}$, then we have either (0) $req_1 = \mathsf{ff} \wedge req_2 = \mathsf{ff} \wedge req_3 = \mathsf{ff}$, or (1) $req_1 = \mathsf{tt}$, or (2) $req_2 = \mathsf{tt}$, or (3)$req_3 = \mathsf{tt}$. The four cases are corresponding to lines from the first one to the fourth one in the table II, and also list all the possible cases when $grant_0 = \mathsf{ff}$, and $grant_1 = \mathsf{ff}$. In case (0), the input value of req_0 is not cared by us because the arbitration value of $grant'_0$ and $grant'_1$ will be the same as $grant_0$ and $grant_1$ respectively in both input cases $req_0 = \mathsf{ff}$ and $req_0 = \mathsf{tt}$. In case (1), req_1 is the highest priority request in the current round, and will be granted immediately once $req_1 = \mathsf{tt}$. Therefore, the values assignments for the nodes req_2, and req_3 are not cared by us. In case (2), req_2 is the highest priority request in the current round if $req_1 = \mathsf{ff}$, and will be granted immediately. Therefore, the values assignments for the nodes req_3 are not cared by us. In case (3), req_3 is the highest priority request in the current round if $req_1 = \mathsf{ff}$ and $req_2 = \mathsf{ff}$, and will be granted immediately.

Similarly, we can analyze the cases of the other boolean assignments for the nodes $grant_0, grant_1$. In fact each of these cases is symmetric to case (a) in some sense. For instance, the input assignments for nodes $req_0 - -req_3$ can also be divided into four subcases in the case where $grant_0 = \mathsf{tt}$ and $grant_1 = \mathsf{ff}$, each of which is corresponding to the counterpart one in the case (a) by shifting the input assignments for nodes $req_0 - -req_3$ by one bit in the right direction in Table II. At last, we analyze the complexity of the verification in the term of the number of requests N. The number of all the Boolean simulation cases is $2^N \times N$, however that of the ternary valued simulation cases is only $N \times N$ with the help of the X value

TABLE III
AN STE ASSERTION FOR ONE-ROUND ARBITRATION

```
constr_0_0 = ¬grantV0 ∧ ¬grantV1 ∧ ¬reqV1 ∧ ¬reqV2 ∧ ¬reqV3
constr_0_1 = ¬grantV0 ∧ ¬grantV1 ∧ reqV1
constr_0_2 = ¬grantV0 ∧ ¬grantV1 ∧ ¬reqV1 ∧ reqV2
constr_0_3 = ¬grantV0 ∧ ¬grantV1 ∧ ¬reqV1 ∧ ¬reqV2 ∧ reqV3
constr_1_1 = grantV0 ∧ ¬grantV1 ∧ ¬reqV2 ∧ ¬reqV3 ∧ ¬reqV0
constr_1_2 = grantV0 ∧ ¬grantV1 ∧ reqV2
constr_1_3 = grantV0 ∧ ¬grantV1 ∧ ¬reqV2 ∧ reqV3
constr_1_0 = grantV0 ∧ ¬grantV1 ∧ ¬reqV2 ∧ ¬reqV3 ∧ reqV0
constr_2_2 = ¬grantV0 ∧ grantV1 ∧ ¬reqV1 ∧ ¬reqV3 ∧ ¬reqV0
constr_2_3 = ¬grantV0 ∧ grantV1 ∧ reqV3
constr_2_0 = ¬grantV0 ∧ grantV1 ∧ ¬reqV3 ∧ reqV0
constr_2_1 = ¬grantV0 ∧ grantV1 ∧ reqV1 ∧ ¬reqV3 ∧ ¬reqV0
constr_3_3 = grantV0 ∧ grantV1 ∧ ¬reqV1 ∧ ¬reqV2 ∧ ¬reqV0
constr_3_0 = grantV0 ∧ grantV1 ∧ reqV0
constr_3_1 = grantV0 ∧ grantV1 ∧ reqV1 ∧ ¬reqV0
constr_3_2 = grantV0 ∧ grantV1 ∧ ¬reqV1 ∧ reqV2 ∧ ¬reqV0
cons0 = When constr_0_0  (AndList[Is0 grant0, Is0 grant1])
cons1 = When constr_0_1  (AndList[Is1 grant0, Is0 grant1])
cons2 = When constr_0_2  (AndList[Is0 grant0, Is1 grant1])
cons3 = When constr_0_3  (AndList[Is1 grant0, Is1 grant1])
cons4 = When constr_1_1  (AndList[Is1 grant0, Is0 grant1])
cons5 = When constr_1_2  (AndList[Is0 grant0, Is1 grant1])
cons6 = When constr_1_3  (AndList[Is1 grant0, Is1 grant1])
cons7 = When constr_1_0  (AndList[Is0 grant0, Is0 grant1])
cons8 = When constr_2_2  (AndList[Is0 grant0, Is1 grant1])
cons9 = When constr_2_3  (AndList[Is1 grant0, Is1 grant1])
cons10 = When constr_2_0  (AndList[Is0 grant0, Is0 grant1])
cons11 = When constr_2_1  (AndList[Is1 grant0, Is0 grant1])
cons12 = When constr_3_3  (AndList[Is1 grant0, Is1 grant1])
cons13 = When constr_3_0  (AndList[Is0 grant0, Is0 grant1])
cons14 = When constr_3_1  (AndList[Is1 grant0, Is0 grant1])
cons15 = When constr_3_2  (AndList[Is0 grant0, Is1 grant1])
ant = AndList[grant bvAre grantV, req bvAre reqV]
cons = AndList[ cons0,  cons1, ...,  cons15]
assert = ant ⇝  Next cons
```

assignments. Notice that this is a substantial decrease at the exponential scale.

According to table II, we can easily write an STE assertion, as shown in table III:

Here $grant$ is the vector of nodes $[grant_0, grant_1]$, $grantV$ the vector of symbolic values $[grantV_0, grantV_1]$, req the vector of nodes $[req_0, ..., req_3]$, $reqV$ the vector of symbolic values $[reqV_0, ..., reqV_3]$. In FORTE, a node is simply of type string. The antecedent ant simply assigns symbolic values to the corresponding nodes. For instance, node $grant_0$ is assigned the symbolic value $grantV_0$. The consequent is a conjunction of trajectory formulae, each of which is a guarded formula of the form When $constr_i$ $cons_i$ and corresponding to each line of the truth table listed in Table II. The assertion $assert$ is an STE assertion which should be satisfied by one run arbitration of the 4×4 round-robin arbiter. For such an STE assertion, we can directly run the tool FORTE to verify it.

IV. GSTE SPECIFICATION OF THE SEQUENTIAL BEHAVIORS OF THE ARBITER

In Table II, we only list the arbiter's behavior in one round. In fact, its behavior is a typically sequential, or reactive. It will accept the requests and make arbitration in each cycle. The sequential behavior can be precisely captured by the following GSTE assertion graph in Fig.6.

TABLE IV
ANTECEDENTS OF THE GSTE SPECIFICATION IN FIG. 6

```
let trans_0_1 = AndList[Is1 req1];
let trans_0_2 = AndList[Is0 req1, Is1 req2];
let trans_0_3 = AndList[Is0 req1, Is0 req2, Is1 req3];
let trans_0_0 = AndList[Is0 req1, Is0 req2, Is0 req3];
let trans_1_2 = AndList[Is1 req2];
let trans_1_3 = AndList[Is0 req2, Is1 req3];
let trans_1_0 = AndList[Is0 req2, Is0 req3, Is1 req0];
let trans_1_1 = AndList[Is0 req2, Is0 req3, Is0 req0];
let trans_2_3 = AndList[Is1 req3];
let trans_2_0 = AndList[Is0 req3, Is1 req0];
let trans_2_1 = AndList[Is0 req3, Is0 req0, Is1 req1];
let trans_2_2 = AndList[Is0 req3, Is0 req0, Is0 req1];
let trans_3_0 = AndList[Is1 req0];
let trans_3_1 = AndList[Is0 req0, Is1 req1];
let trans_3_2 = AndList[Is0 req0, Is0 req1, Is1 req2];
let trans_3_3 = AndList[Is0 req0, Is0 req1, Is0 req2];
```

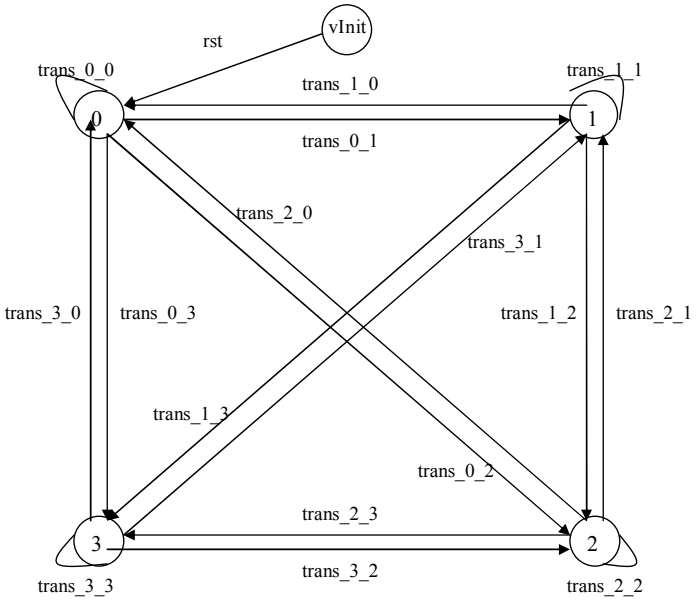

Fig. 6. GSTE specification of the Arbiter

The edge from $init$ to v_0 stands for the reset action which sets the initial value of the register $grant_0, grant_1$ ff respectively. The node v_0 is corresponding to the state where $grant_0 = \mathsf{ff}, grant_1 = \mathsf{ff}$, the edge (v_0, v_1) stands for the transition from state v_0 to v_1 where $grant_0 = \mathsf{tt}, grant_1 = \mathsf{ff}$. The antecedent of the transition is $trans\_0\_1$ which sets $req1$ tt, and doesn't care the values of any other inputs. At state v_0, once the inputs $req1 - -req3$ are set ff, the state $grant_0 = \mathsf{ff}, grant_1 = \mathsf{ff}$ will hold. This is captured by the self-loop (v_0, v_0) whose antecedent is $trans\_0\_0$. Here we omit the consequents below each edge in the graph because they only specify the values of nodes $grant_0, grant_1$ respectively. These values are implicitly indicated by the state nodes which the edges come from.

Similarly, we can analyze the other states and transitions.

TABLE V
ANTECEDENTS AND CONSEQUENTS OF GSTE SPECIFICATION IN FIG. 7

```
let rst = Is1 reset;
let antSet = AndList [Is0 req0, Is1 req1, Is0 req2, Is0 req3]
let ants_vSet_2 = AndList[Is1 req3, Is1 req2];
let ants_vSet_3 = AndList[Is1 req3, Is0 req2];
let ants_2_3 = AndList[Is1 req3];
let cons_3 = AndList[Is1 grant0, Is1 grant1];
```

TABLE VI
ANTECEDENTS AND CONSEQUENTS OF GSTE SPECIFICATION IN FIG. 8

```
let rst = Is1 reset;
let antSet = AndList [Is0 req0, Is0 req1, Is1 req2, Is0 req3]
let ants_vSet_3 = AndList[Is1 req2, Is1 req3];
let ants_vSet_0 = AndList[Is1 req2, Is0 req3, Is1 req0];
let ants_vSet_1 = AndList[Is1 req2, Is0 req3, Is0 req0,
                  Is1 req1];
let ants_vSet_2 = AndList[Is1 req2, Is0 req3, Is0 req0,
                   Is0 req1];
let ants_3_0 = AndList[Is1 req2, Is1 req0];
let ants_3_1 = AndList[Is1 req2, Is0 req0, Is1 req1];
let ants_3_2 = AndList[Is1 req2, Is0 req0, Is0 req1, ];
let ants_0_1 = AndList[Is1 req2, Is1 req1];
let ants_0_2 = AndList[Is1 req2, Is0 req1];
let ants_1_2 = AndList[Is1 req2];
let cons_2 = AndList[Is0 grant0, Is1 grant1];
```

V. GSTE SPECIFICATION OF THE RESPONSE PROPERTY OF THE ARBITER

For the 4 round-robin arbiter, a running state of arbiter is determined by the state variable vector $grant$. Therefore we sometimes call a state by the decoding number of value of the $grant$ vector at the state in the following part of this subsection. For instance, If we call a state 2, we mean that the decoding number of the value of the $grant$ vector of the arbiter is 2.

In this part, we will introduce how to model the response property of the arbiter. This property specifies that once a request req_i is set high from a state i and kept high, then the request will be granted after several cycles. The worst-case waiting time for the request to be granted can also be predicted. Before we formally define the response property by GSTE assertion graphs, we give two examples to illustrate the response property.

Example 1: Consider a state where the value of $grant$ is $[\mathtt{tt}, \mathtt{ff}]$, which means that the last request granted is req_1, if the request req_3 is set high and kept high, then the request will be granted (or the value of $grant$ is set $[\mathtt{tt}, \mathtt{tt}]$) after at most 2 cycles. Namely, the worst-case waiting time for this request to be granted is 2.

Example 2: Consider a state where the value of $grant$ is $[\mathtt{ff}, \mathtt{tt}]$, which means that the last request granted is req_2, if the request req_2 is set high again and kept high, then the request will be granted (or the value of $grant$ is set $[\mathtt{ff}, \mathtt{tt}]$ again) after at most 4 cycles. Namely, the worst-case waiting time for this request to be granted is 4.

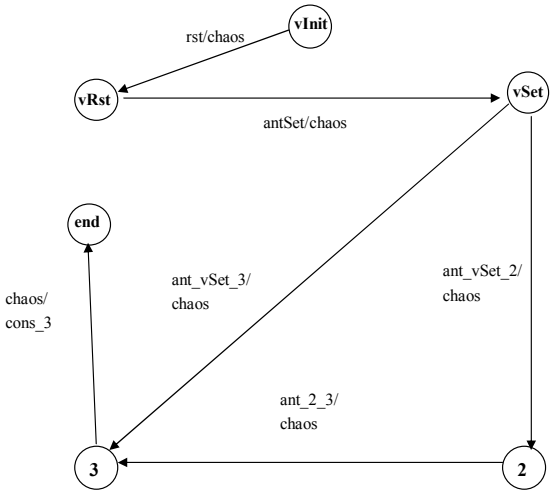

Fig. 7. STE specification of the response property in Example 1

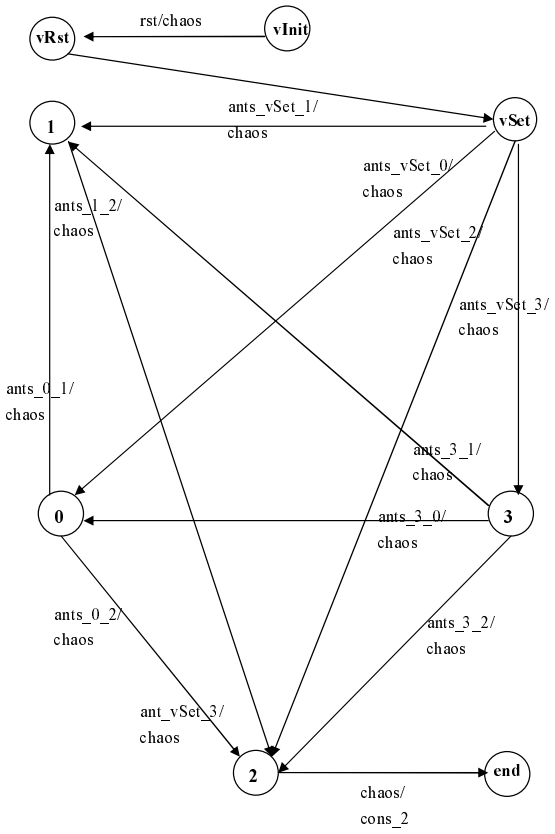

Fig. 8. GSTE specification of the response property in Example 2

We can formally model the two response properties in two GSTE graphs, which are shown in Fig. 7 and Fig. 8. Let us explain the reason why the two assertion graphs can naturally specify the response properties.

In each of the GSTE assertion graphs, there is a state where the decoding value of $grant$ vector is set as some integer s and the request req_j is set high and kept high until this request req_j is granted. There are four distinguishing features in the graphs:

(1) The first edge $(vInit, vRst)$ represents an action of reset, and the second edge $(vRst, vSet)$ sets the state of the arbiter to some initial value. For Fig. 7, in order to set the initial value of state variable vector $grant$ $[\mathsf{tt}, \mathsf{ff}]$, we simply set the simulation constraint by setting the antecedent of edge $(vRst, vSet)$ as AndList $[\mathsf{Is0}\ req_0, \mathsf{Is1}\ req_1, \mathsf{Is0}\ req_2, \mathsf{Is0}\ req_3]$, which means that only request req_1 is set high and the others are set low. Thus the request req_1 will be granted immediately, and the grant vector will be set $[\mathsf{tt}, \mathsf{ff}]$ accordingly.

(2) There is a node list seq which represents the state which may be reached from node $vSet$ by granting requests in a descending priority order. Let $seq' = ([vSet]@seq)$, for any two indices k, h such that $0 \leq k < h <$ len seq', there is an edge (seq'_k, seq'_h) whose antecedent must include a formula $\mathsf{Is1}\ req_j$, which means that the request req_j is kept high in the simulation steps. The edge (seq'_k, seq'_h) represents a simulation step where a request $req_{seq'_h}$ is granted from state node seq'_k. In Fig. 7 and Fig. 8, seq is $[2, 3]$ and $[3, 0, 1, 2]$ respectively. In Fig. 7, the edge $(vSet, 3)$ represents a simulation step whose constraint is AndList $[\mathsf{Is0}\ req_2, \mathsf{Is1}\ req_3]$, while $(2, 3)$ a simulation step whose constraint is AndList $[\mathsf{Is1}\ req_3]$.

(3) The last element of the list seq' represents a state after req_j is granted. It is connected with any other node v in seq' by an edge. From this state node, we can test the value of $grant$ by an edge which is from it to end whose consequent assigns the decoding number of j to $grant$. This means that the request req_j is granted at last. In Fig. 7 and Fig. 8, the last nodes of seq' are labeled by 3 and 2 respectively.

(4) The edges starting from each node in seq' except the last one should include all possible one-simulation step patterns under the constraint $req_j = \mathsf{tt}$, That is to say, the simulation should exhaustively enumerate all possible input patterns under the constraint $req_j = \mathsf{tt}$. For instance, for the node $vSet$ in Fig. 7, $(vSet, 2)$ and $(vSet, 3)$ represent exhaustive input patterns from a state in one-simulation step which starts from the state $vSet$, as shown in Table VII. Although there are only two rows, Table VII lists all input patterns under the constraints $req_3 = \mathsf{tt}$ and $grant = [\mathsf{tt}, \mathsf{ff}]$. Under the two constraints, we only need to care the value of the input req_2 because the values of req_0 and req_1 have the lower priority than req_3 and will not affect the simulation. A more complicated example is shown in Table VIII, which stands for all possible one-simulation step patterns which starts from the state $vSet$ in Fig. 8.

Requirements (2)(3)(4) guarantee that for any simulation sequence starting from $vSet$, $grant$ vector will be eventually changed into the decoding value of j once the request req_j is set high and kept high. The worst-case waiting time for the request req_j to be granted is the length of the longest path from $vSet$ to the last element of seq. Therefore, GSTE graphs such as Fig. 7 and Fig. 8 accurately capture the meaning of the response property of the round-robin arbiter.

TABLE VII
TERNARY-VALUED TRUTH TABLE OF ONE STEP SIMULATION CASES FROM STATE $vSet$ UNDER THE CONSTRAINT $req_3 = \mathsf{tt}$ IN FIG. 7

req0	req1	req2	req3	$grant_0$	$grant_1$	$grant'_0$	$grant'_1$
X	X	tt	tt	tt	ff	ff	tt
X	X	ff	tt	tt	ff	tt	tt

TABLE VIII
TERNARY-VALUED TRUTH TABLE OF ONE STEP SIMULATION CASES FROM STATE $vSet$ UNDER THE CONSTRAINT $req_2 = \mathsf{tt}$ IN FIG. 8

req0	req1	req2	req3	$grant_0$	$grant_1$	$grant'_0$	$grant'_1$
X	X	tt	tt	ff	tt	tt	tt
tt	X	tt	ff	ff	tt	ff	ff
ff	tt	tt	ff	ff	tt	tt	ff
ff	ff	tt	ff	ff	tt	ff	tt

VI. PARAMETERIZED VERIFICATION SCRIPT OF A $N \times N$ ROUND-ROBIN ARBITER

For a $N \times N$ round-robin arbiter, we can construct a corresponding STE assertion and GSTE assertion graphs for the aforementioned one-round arbitration and sequential behavior and response property correspondingly.

A. STE Assertion of a $N \times N$ Round-Robin Arbiter

An STE assertion of a $N \times N$ Round-Robin arbiter is shown in Table IX. Suppose that vs is a vector of ternary values, and $symbVs$ a vector of symbolic values. According to vs, constrOfReq vs $symbVs$ returns a conjunction of literals according to the ternary values in vs, which is used to define the boolean guard of $constr\_i\_j$ for some $i, j < N$, which is shown in Table III. One literal is ether $symbVs_i$ or $\neg symbVs_i$ for some $i <$ len $symbVs$. For instance, constrOfReq $[\mathsf{X}, \mathsf{tt}, \mathsf{ff}, \mathsf{ff}]$ $reqV = reqV_1 \wedge \neg reqV_2 \wedge \neg reqV_3$.

consConstrIJ $width$ N $reqVs$ $grant$ $grantV$ i j returns a pair to construct one conjunct item in the consequent. The first element is a trajectory formula assigning new values to the $grant$ vector after one arbitration. The second one is a guard $constr\_i\_((i+j)\%N)$ for some $i, j < N$, which is shown as Table III. For instance, consConstrIJ 2 4 $reqV$ $grant$ $grantV$ 1 1 = (AndList $[\mathsf{Is0}\ grant_0, \mathsf{Is1}\ grant_1], grantV_0 \wedge \neg\ grantV_1 \wedge reqV_2$).

B. GSTE Assertion Graph of Sequential Behaviors of a $N \times N$ Round-Robin Arbiter

A GSTE assertion of a $N \times N$ Round-Robin Arbiter is shown in Table X. Here translJ i j $width$ N describes a state transition from state i to $(i+j)\%N$, where $N = 2^{width}$ is the number of the input ports and $width$ is the width of the bit vector which is needed to encode the number.

TABLE IX
STE ASSERTION OF A $N \times N$ ROUND-ROBIN ARBITER

```
letrec constrOfReq [] [] = T
/\constrOfReq (v : vs) (symbV : symbVs) =
  ((v = X) => constrOfReq vs symbVs|
  (v = tt) => symbV ∧ (constrOfReq vs symbVs)|
  (¬symbV) ∧ (constrOfReq vs symbVs))

let consConstrIJ width N symbReqs grant grantV i 0 =
  let last = encode i width in
  let newLast = encode i width in
  let reqV' = map (\k.(k = i) => X|ff)(0 upto (N − 1)) in
  let reqConstr = constrOfReq reqV' symbReqs in
  (grant bvAre newLast, (grantV bvEq last) ∧ (reqConstr))
/\consConstrIJ width N symbReqs grant grantV i j =
  let last = (encode i width) in
  let j' = (i + j)%N in
  let newLast = encode j' width in
  let negReqs = 1 upto (j − 1) in
  let negReqs = map (\k.(k + i)%N) negReqs in
  let reqV' = map (\k.(mem k negReqs) => ff|
    (k = j') => tt|X)
    (0 upto (N − 1)) in
  let reqConstr = constrOfReq reqV' symbReqs in
  (grant bvAre newLast,
  (grantV bvEq last) ∧ (reqConstr))

let consConstrI width N symbReqs grant grantV i =
  map (\j.consConstrIJ width N symbReqs grant
    grantV i j)(0 upto (N − 1))
let vect2Val vect = map(\str.bvariable str) vect

let STEAssert grant req width N =
  let reqV = vect2Val req in
  let grantV = vect2Val grant in
  let Ant = AndList [grant bvAre grantV,req bvAre reqV]in
  let consConstrs = flat (map
    (\k.consConstrI width N reqV grant grantV k)
    (0 upto (N − 1))) in
  let consCons = map
    (\apair. (Guard (snd apair) (fst apair)))
    consConstrs in
  Ant ⇝ (Next (AndList consCons))
```

TABLE X
GSTE ASSERTION GRAPH OF A $N \times N$ ROUND-ROBIN ARBITER

```
let transIJ i 0 width N req =
  (i, i,
  AndList ((map Is0 (req subtract [req!i]))),
  (grant bvAre (encode i width)))
/\ transIJ i j width N req =
  let j' = (i + j)%N in
  let negReqs = 1 upto (j − 1) in
  let negReqs = map (\k.(req!k + i)%N) negReqs in
  let ant = AndList ((map Is0 negReqs)
    union [Is1 (req!j')]) in
  let newLast = (encode i width ) in
  let cons = (grant bvAre newLast) in
  (i, j', ant, cons);

let transFromI WIDTH req i =
  let NUM_PORTS = 2 * *WIDTH in
  map (\j.transIJ i j WIDTH NUM_PORTS req)
  (0 upto (NUM_PORTS − 1));

let transFrom WIDTH req =
  let NUM_PORTS = 2 * *WIDTH in
  flat (map (transFromI width req) (0 upto (NUM_PORTS − 1)));
let ag WIDTH req =
  let initEdge = (0, 1, Is1"reset") in
  initEdge@
  (map (\(from, to, ant, cons).
  (from + 1, to + 1, ant, cons)) transForm WIDTH req);
```

transFromI $width\ i$ generates all the edges starting from state i. The whole assertion graph is simply all the transition edges from each state i in addition to the $init$ edge which is the reset transition. A GSTE specification on the arbiter with $N \times N$ configuration size is precisely captured by assertion graph $ag\ width\ req$.

C. GSTE Assertion Graph of Response Properties of a $N \times N$ Round-Robin Arbiter

Consider a running state s of a $N \times N$ Round arbiter, and a request req_j. In order to model the response property that the request req_j will be granted once it is set high and kept high from the state s, we introduce FL code, shown as Table XI:

Because the response property requires that req_j is set high and kept high, only some requests have a chance to be granted while the others have no such a chance. The above function priorList $s\ j\ N$ defines a list of requests in a descending priority order, which have a chance to be granted until req_j is granted.

According to the Round-Robin arbitration policy, the request $req_{(s+1)\%N}$ has the highest priority, so $(s+1)\%N$ is the first element in priorList $s\ j\ N$. j is the last element of the list. The numbers of all the requests are included in priorList $s\ j\ N$ whose priority is not lower than req_j, and not higher than $req_{(s+1)\%N}$.

For instance, prePriorList $1\ 3\ 4 = [2, 3]$, prePriorList $2\ 2\ 4 = [3, 4, 5, 6]$, priorList $1\ 3\ 4 = [2, 3]$, and priorList $2\ 2\ 4 = [3, 0, 1, 2]$.

The above instances mean that for a 4×4 Round-Robin arbiter, for the state 1, if req_3 is set high and kept high, then only requests req_2 and req_3 can be granted until req_3 is granted; for state 2, if req_2 is set high and kept high, then all the requests have a chance to be granted until req_2 is granted.

For convenience, in a GSTE assertion graph which specifies the aforementioned response property, the first node, the second node, the third, and the last node are named $vInit$, $vRst$, $vSet$, and end which are not in priorList $s\ j\ N$. We directly use the numbers in priorList $s\ j\ N$ to name the other state nodes, which can be reached from the state node $vSet$. For instance, in Fig. 7, we use 2 and 3 to name the state node which can be reached from $vSet$ respectively. Notice that the state number k is just the number of the request which is granted from a previous node by granting the request req_k, and is also the decoding number of the $grant$ vector at that state.

priorList $s\ j\ N$ is just the seq list which is defined in section V. Let $seq' = [vSet]@seq$. For any index h, k such that $0 \leq h < k <$ len seq', trans $h\ k\ seq'\ req$ defines a transition edge from seq'_h to seq'_k. Its antecedent is composed of two parts: Is1 req_j and ants $h\ k\ seq'\ req$. The latter model key simulation constraints which enable the transition from seq'_h to seq'_k. Formally, ants $h\ k\ seq'\ req$=AndList[Is0 $req_{seq'_{h+1}}$, ..., .,Is0 $req_{seq'_{k-1}}$,

TABLE XI
GSTE ASSERTION GRAPH OF A $N \times N$ ROUND-ROBIN ARBITER'S RESPONSE PROPERTY

```
let prePriorList s j N =
  (s + 1)%N = j => [s + 1]|
  [s + 1]@(prePriorList (s + 1) j N);

let modMap N i = i %N;

let priorList s j N = map (modMap N) (prePriorList s j N)
let constructIs0Ant L req =
  let mapf k = Is0 (req!k) in
  map mapf L

let  ants h k seq req =
  let hkList = (h + 1) upto (k − 1) in
  let hkSeq = map (\i.seq!i) hkList in
  let is0Ants = constructIs0Ant hkSeq req in
   is0Ants@[Is1 (req!(seq!k))]
let consJ j N grant =
  let jCode = encode j N in
  let zipTwo = zip jCode grant in
  let  mapf v = (fst v = ff) => Is0 (snd v)| Is1 (snd v
  in map mapf zipTwo
let  trans h k seq req j =
  (h, k, [Is1 (req!j)]@(ants h k seq req j), chaos)

let set s req =
  let L = 0 upto len (req − 1)
  let f i = (i = s) => Is1 req!i|Is0 req!i
  in map f L

let  responseAg N req grant s j  =
  let rstEdge = (vInit, vRst, rst, chaos) in
  let setEdge = (vRst, vSet, set s req, chaos) in
  let seq′ = [vSet]@(priorList s j N) in
  let transh h = map
    (\k.(trans h k seq′ req j))
    ((h + 1) upto len seq′ − 1) in
  let otherTrans = map (\h.transh h) (0 upto len seq′ − 2)
  let lastEdge = (j, last, chaos, consJ j N grant)
  in [rstEdge, setEdge, lastEdge]@otherTrans

let  allResponse N req grant =
  let L = 0 upto (N − 1) in
  let f s = map (responseAg N req grant s) L in
  flat (map f L)
```

TABLE XII
EXPERIMENTS

N	GSTE time sec.	GSTE memory MB	STE time sec.	STE memory MB	RESPONSE GSTE time sec.	RESPONSE GSTE memory MB
8	0.1	12.3	0.02	8.2	1.7	17.8
16	1.2	19.2	0.05	10.3	35.2	22.8
32	34.2	50.5	0.34	20.8	1274.1	83.7
64	1217.9	361.3	4.43	73.0	99982.9	366.5

$\mathsf{ls1}\ req_{seq'_k}]$, which specifies that $req_{seq'_k}$ is set high, and $req_{seq'_l}$ is set low for any l such that $h < l < k$. For instance, in Fig. 7, $seq' = [vSet]@[2, 3]$, then ants 0 2 seq' req= AndList[ls0 req_2, ls1 req_3]. Recall that $seq'_0 = vSet$, $seq'_1 = 2$, and $seq'_2 = 3$ here.

For any index $0 \leq h < \mathsf{len}\ seq' - 1$,transh h represents all the transitions from a non-terminal node seq'_h, which exhaustively enumerate all possible input patterns in one-simulation step from the state seq'_h under the constraint req_j is set high. From the state, only requests $req_{seq'_{h+1}}$,..., $req_{seq'!(\mathsf{len}\ seq'-1)}$ can be granted, the antecedents ants h $(h + 1)$ seq' req,..., ants h $(\mathsf{len}\ seq' - 1)$ seq' req just enumerate simulation constraints to grant the corresponding request. Therefore the parameterized GSTE graph satisfies the requirement (4) for the exhaustive simulation from any node except the last one.

The last element of seq', $seq'!(\mathsf{len}\ seq' - 1)$, is just j. The state j is connected with any other element state node in seq' because there is a transition $(seq'_h, seq'_{\mathsf{len}\ seq'-1})$ for any $0 \leq h < \mathsf{len}\ seq' - 1$. Besides, there is an edge $lastEdge$ from it to the end node whose consequent tests whether the decoding number of $grant$ vector is j.

$rstEdge$ and $setEdge$ are simply edges for actions of reset and initializing values. $otherTrans$ are the other transitions, each of which models a transition from a node in seq' except the last element of seq'. The GSTE graph for the response property is simply the combination of $[rstEdge, setEdge, lastEdge]$ and $otherTrans$.

allResponse N req $grant$ checks whether the response property responseAg N req $grant$ s j hold for all states s and requests j. Notice that the evaluation of responseAg N req $grant$ s j and responseAg N req $grant$ s' j' are independent of each other if $s \neq s'$ or $j \neq j'$. Therefore we can divide the verification task of allResponse N req $grant$ into subtasks and distribute them into different machines to verify when N become greater. Another interesting point is the symmetry between the subproblems responseAg N req $grant$ s j and responseAg N req $grant$ s' j' for some s, s', j, j'. For instance, assertion graph responseAg N req $grant$ 0 6 and responseAg N req $grant$ 1 7 are symmetry to each other [3]. Therefore it is enough for us to verify the cases where $s = 0$ for all $0 \leq j < N$. We can apply the symmetry reduction technique used in [21], [22] to reduce the complexity of verification further.

VII. EXPERIMENTS

We have conducted experiments to verify the aforementioned STE assertions and the assertion graphs for arbiters with varying numbers of requests. The detail experimental codes and data, such as Verilog codes, BLIF codes of the arbiter circuits and verification scripts, can be found in [20]. Table XII shows the verification result for STE assertions for one round arbitration and GSTE assertion graphs for sequential behaviors of arbiters with different requests number N.

For the response property, we notice that the verification runtime grows faster than the memory used. But this is not a big problem because the big verification task allResponse N req $grant$ can be divided and distributed if we have a parallel verification platform.

VIII. CONCLUSION

Round-robin arbitration is a very important routing scheme which is extensively used in real-world network systems such

[3]Reader can refer to [20] for justifying the symmetry of the two assertion graphs.

as ATM and NOC. Despite its extensive application, little work has given a thorough formal verification for the hardware design of a round-robin arbiter. The difficulty lies in the exhaustive simulations. Take one round arbitration of a $N \times N$ arbiter for example, the number of simulation cases is $2^N \times N$. However, we reduce the complexity in this work. Our approach is enhanced STE, which explores fully symbolic simulation for not only one round of round-robin arbitration, but also the sequential behaviors of the arbiter. Our approach enhances the simulation procedure in the sense that it is both exhaustive and effective. The key points which come from the ternary-value based abstraction not only reduce the number of input patterns from 2^N to N (or a number which is less than N) in a state, but also guarantee the exhaustive enumeration of simulation patterns. This approach effectively reduces the complexity of verification from exponential scale to linear scale. Another advantage of our approach is STE/GSTE, which are naturally oriented to the industrial verification because STE assertions or GSTE graphs can be easily understood and used by hardware engineers [19]. For instance, the GSTE graphs directly enumerate all state nodes and the transitions between state nodes. A state transition represents a simulation step. The antecedents and consequents clearly specify the input stimulus and the corresponding outputs in each simulation step. Our experiments demonstrate that the enhanced STE specification for real-world hardware design can be finished automatically in a reasonable time and memory usage.

REFERENCES

[1] D. Ginsburg, *ATM: Solutions for Enterprise Internetworking*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] P. Curzon, "The formal verification of an atm network," in *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*. ACM Pres, August 1994, p. 392.

[3] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*, 2001, pp. 684 – 689.

[4] C.-J. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, 1995.

[5] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating-point hardware," *Intel Technology Journal*, vol. Q1, pp. 147–190, 1999.

[6] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Combining theorem proving and trajectory evaluation in an industrial environment," in *DAC '98: Proceedings of the 35th annual conference on Design automation*. New York, NY, USA: ACM, 1998, pp. 538–541.

[7] *Forte/fl user guide*, 2003rd ed., Technical Publications and Training, Intel Corporation.

[8] J. Yang and C.-J. H. Seger, "Introduction to generalized symbolic trajectory evaluation," *IEEE Trans. VLSI Syst.*, vol. 11, no. 3, pp. 345–353, 2003.

[9] ——, *Generalized Symbolic Trajectory Evaluation*, Intel SCL Technical Report.

[10] J. Yang and A. Goel, "Gste through a case study," in *ICCAD*, L. T. Pileggi and A. Kuehlmann, Eds. ACM, 2002, pp. 534–541.

[11] Z. Manna and A. Pnueli, "Specification and verification of concurrent programs by a ∀-automata," in *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '87. New York, NY, USA: ACM, 1987, pp. 1–2. [Online]. Available: http://doi.acm.org/10.1145/41625.41626

[12] P. Curzon, "The formal verification of the fairisle atm switching element," Ph.D. dissertation, University of Cambridge, 1994.

[13] B. Chen, M. Yamazaki, and M. Fujita, "Bug identification of a real chip design by symbolic model checking," in *European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings.*, feb-3 mar 1994, pp. 132 –136.

[14] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, Apr. 1986. [Online]. Available: http://doi.acm.org/10.1145/5397.5399

[15] S. Tahar, X. Song, E. Cerny, Z. Zhou, M. Langevin, and O. A. Mohamed, "Modeling and formal verification of the fairisle atm switch fabricusing mdgs," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 18, no. 7, pp. 956–972, 1999.

[16] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, Aug. 1986. [Online]. Available: http://dx.doi.org/10.1109/TC.1986.1676819

[17] P. Curzon, "The formal verification of the fairisle atm network," http://www.eecs.qmul.ac.uk/~pc/research/atmproof/atmproof.htm.

[18] L. C. Paulson, *ML for the working programmer*. University of Cambridge Press, 1996.

[19] J. Yang, "Verification challenges and opportunities in the new era of microprocessor design," in *ATVA*, ser. Lecture Notes in Computer Science, S. Graf and W. Zhang, Eds., vol. 4218. Springer, 2006, pp. 6–7.

[20] Y. Li and N. Zeng, "Enhanced symbolic simulation of a round-robin arbiter," 2011, http://lcs.ios.ac.cn/~lyj238/atmArbiter.html.

[21] Y. Li, W. Hung, X. Song, and N. Zeng, "Exploring structural symmetry automatically in symbolic trajectory evaluation," *Formal Methods in System Design*, pp. 1–27, 2011, 10.1007/s10703-011-0119-z. [Online]. Available: http://dx.doi.org/10.1007/s10703-011-0119-z

[22] Y. Li, N. Zeng, W. Hung, and X. Song, "Combining symmetry reduction with generalized symbolic trajectory evaluation," 2012, manuscript, Accpted in the Computer Journal, Oxford university press.