Modeling and Verification of NoC Communication Interfaces

Vinitha Palaniveloo\textsuperscript{1} Arcot Sowmya\textsuperscript{1} Sridevan Parameswaran\textsuperscript{1}

\textsuperscript{1} University of New South Wales, Australia
{vinithaap, sowmya, sridevan}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-0903
October 2009

THE UNIVERSITY OF NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia
Abstract

The projected demands of a 21st-century road map in semiconductor industry for processing power, performance, power consumption reflects the need to integrate millions of transistors on a single chip. The major markets for semiconductors are networking and communication, medical, defense, automotive and consumer electronics. To satisfy the processing power required by those segments, the industry is proposing techniques such as parallel chip architecture, multiple data processing engines, more memory interface and chip-to-chip interface. Although recent technological advances allow integration of billions of transistors into a single chip the existing electronic design automation tools are not advanced enough to handle complex chip designs. Hence, the focus has shifted to providing a communication infrastructure for massive real time parallel processing, where communication and computation can be designed independently. The concept of Network on Chip (NoC) addresses the communication requirements on chip and decouples it from computation.

One of the challenges faced by designers of NoC integrated circuits is verifying the correctness of the communication scheme for an NoC Architecture. NoCs are on-chip communication networks that borrow the networking concept from computer networks to interconnect complex Intellectual Property (IP) on chip. Therefore, the applications on IP cores communicate with peer applications through communication architecture that consists of layered communication protocol, routers and switches. The absence of an integrated architectural model poses the challenge of performing end-to-end verification of communication scheme.

The formal models of NoC proposed so far in the literature focus on modeling parts of communication architecture such as the specific layers of communication protocol or routers or network topology but not as a integrated architectural model. This is attributed to the absence of expressive modeling language to model all the modules of the NoC communication architecture. The NoC communication architecture is heterogeneous as it consists of synchronous and asynchronous IP cores communicating through heterogeneous communication pipelines. We propose a heterogeneous modeling language for modeling and verification of NoC communication architecture. The proposed modeling language is based on formal methods as they provide precise semantics, mathematics based tools and techniques for specification and verification.
1 Introduction

The aim of this research is to design formalism for modeling Network-on-chip (NoC), a concept of interconnecting components on a chip using routers similar to computer networks [1, 2, 3, 4, 5, 6]. The basic concept idea NoC architecture is to separate the computation and communication modules of a system on chip. The basic differences between on-chip and computer networks such as influence of wiring delay, retransmission in communication protocol and guaranteed throughput inhibit us from using the design and validation techniques already used for in computer networks.

The NoC, on-chip communication architecture interfaces to IP cores through Network Interface (NI) as shown in Figure 1. The network interface can be part of IP cores or NoC or router. The IP core implements the application layer and physical layer to the network interface. The Network interface implements packetization, routing algorithms specific to the network layer and data layer and physical layer to the router. The router implements switching scheme and data layer, physical interface to router and Network interface. The Network Interface implements clock crossover algorithms to match clock between IP core and the clock of NoC communication architecture.

![Figure 1.1: NoC Communication Architecture](image.png)

2 Research Problem

NoCs incorporate numerous design variables such as topology, number of routers, router architecture, buffers at the router, switching schemes, routing schemes and clocking schemes. These design variables can be altered to obtain NoCs with different quality of service. NoCs with synchronous clock schemes are SPIN[7], Aethereal[8], xPIPES[9], NOSTRUM[10], HERMES[11]. NoCs with asynchronous clock schemes are MANGO[12], QNOC[13], ANOC[14] and HERMES-GLP [15].

Synchronous NoCs are in theory, implemented with isochronous clock (frequency and phase locked) throughout the chip, but practically the implementation of synchronous clocking scheme is limited by factors such as clock skew, delays, synchronization issues and synchronous latency insensitivity circuits. Asynchronous NoCs are based on the concept of confining the clocks to cores and allowing the
network to be clock-less and is called Globally Asynchronous and Local Synchronous (GALS) models [16]. Moreover, the clock scheme in futures NoC is still unpredictable [17]. However, GALS is a very powerful concept that can enable integration of more components as it is not restricted by wiring delays and synchronization issues and therefore is a prime area of research.

The modeling of synchronous NoC communication architecture can be done using synchronous formalism, since the on-chip communication network from the output of Network interface through router to another network interface is synchronous. This would enable verification of only the communication protocol at the physical interface and not end to end communication, since end-to-end communication requires modeling clock cross over. Moreover, the switching at the router can have arbitrary delays depending on the traffic, hence modeling it based on synchronous clock would not be appropriate. While performing end to end verification the synchronous NoC architecture also represents an heterogeneous architecture. Since both the synchronous and asynchronous noc represent an heterogeneous architecture we need an heterogeneous modeling language.

3 Related Work

There are many simulation tools for verification of NoC. But the simulation tools using System C and VHDL require RTL-level implementation and details for verification and these tools do not provide techniques to analyze the reason for failure. The communication scheme can be modeled with different levels of abstraction can be done using formal methods. The need for formal methods at various phases of NoC design and development is emphasized in the literature[18]. This research is aimed at identifying a suitable formal method to model and validate NoC. Simulation based design exploration frameworks such as OCCN [19] and ProtoNoC [20] are suitable for a specific communication scheme and NoC architecture. Secondly, they do not provide techniques to analyze the reason for failure since simulation-based techniques are semi-informal.

There are a number of existing works on formal modeling and verification of NoCs [21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]. However, the formalisms proposed so far, are those already developed for other applications that are control-flow or data-flow based and that are not network based. NoC is a networked data flow application. The formalisms proposed for modeling NoCs in the literature are graphs, finite state machines (FSM) or Petri-Nets (PN). The NoCs are modeled and verified using verification systems such as PVS Theorem prover [32], ACL2 Theorem prover[33], Finite State Process (FSP) [9], Communicating Hardware Process (CHP) [13], B-Method [12] and Specification Description Language (SDL)[10] that has a specification language, integrated support tools such as theorem prover for verification. Some verification systems such as FSP and CHP does not provide integrated verification tools therefore separate verification tools based on automated theorem proving and model checking are used.

The graph and FSM formalisms were proposed for modeling control applications. An extended graph based formalisms called data flow graphs was proposed to model data flow applications. Hence, graph and FSM models can be used to model simple data flow applications but they are not suitable for
modeling a network of data-flow application such as NoC. Due to this limitation in the selected formalism only an highly abstract model of NoC with a subset of communication or NoC architecture design variables has been modeled. The abstractions depend on the tools provided by formal methods for modeling and secondly on the criteria used to select design variables for modeling. Since NoC design variables are interrelated, these highly abstract models created with limited design variables do not guarantee the results of validation when the parameters change.

The formal models developed so far are for a specific NoC architecture to verify a specific communication scheme or a communication interface. The NoC communication scheme layered similar to the computer networks. The layered concept of networking was developed to accommodate changes in technology. Each layer of a specific network model may be responsible for a different function of the network. Each layer will pass information up and down to the next subsequent layer as data is processed. The application layer allows decoupling functional application from target hardware. The verification of NoC communication scheme is usually done as peer-to-peer communication at different levels of abstraction for each layer and is not interconnected to another layer as modeling each layer requires different formal methods and interaction between different formalisms. The verification done for specific layers do not hold for a integrated system. Since, each communication scheme requires different formalism, the formal models cannot be reused for the same NoC when different communication scheme is used. Therefore, the focus is on system level modeling and verification of NoC.

In order to validate a new architecture or communication scheme or for changing a topology, a completely new model becomes necessary. Although, an attempt to define generic models exists [24], it does not consider translation of a generic model to a specific NoC. The proposed generic model in ACL2 was used to verify message ordering in the network layer of the protocol stack. The generic model does not support any notion of time, multiple active networked nodes, irregular network topology or bounded buffer size at the nodes and the granularity of switching is limited to a packet.

NoC communication schemes are packet based, the indeterminism in routing and arbitration makes synchronous network as good as asynchronous networks except for the physical layer switching [28]. Therefore, we propose to use a formalism with a notion of time and allow representation of indeterminism at switching layer besides representation of synchronous and asynchronous communication interfaces at the physical layer of the network. The formal methods proposed so far are for either synchronous NoCs or asynchronous NoC. Graphs, finite state machines were used for modeling synchronous NoC[21, 22, 23, 24, 25, 26]. CHP and ASC an extension of System C, for modeling asynchronous circuits in asynchronous modes of NoC [30, 31]. Little research is targeted at modeling GALS based NoCs; Haskell based ForSyde [32], proposes an extension of synchronous formalism by refinements to incorporate multiple clock domains, channel delays, jitters and channel mapping.
4 Proposed work

Heterogeneous Protocol Automata (HP A) is a formalism recently designed by
the group for modeling communication interfaces of heterogeneous modules in
Network on chip. HPA is an extension of Synchronous Protocol Automata [30] a
synchronous formalism that was proposed to model hardware bus architectures
in system on chip. SPA formalism models communication on buses at a low
level of abstraction, describing behavior of signals for every clock tick. SPA
is a proven formalism that is used to synthesise protocol converters between
incompatible bus protocols in [31, 32].

Modeling network of heterogeneous modules at the lower level system lay-
ers requires modeling the communication bus, wrapper interface between the
heterogeneous modules, bus protocol and system. SPA formalism is proposed
for modeling synchronous bus interfaces only. In HPA we will use the formal
notation and semantics of SPA to model synchronous buses; in addition, we
will extend the formal semantics to model asynchronous communication buses.
Since, the formal semantics of both synchronous and asynchronous interfaces
use the same formal notation, problem of modeling wrapper interface which
is essential for interaction between heterogeneous modules is simplified. Thus,
HPA is extended to support modeling both synchronous and asynchronous com-
munication between heterogeneous processes using a single formalism.

In HPA the processes are represented as finite state machines (FSM) that can
transition based on clock tick or events. The clocked communication processes
are modeled as FSMs that transition at every clock tick based on the process’s
clock; at the transition there are guards and communication action actions that
are performed before transiting to the next state. The un-clocked communi-
cation processes are modeled as FSMs that makes a transition for events; at
the transition there are pre-guards and post-guards that must be satisfied be-
fore and after the communication action is performed. The status of the signals
broadcast to all the FSMs and the signals themselves can be modeled to be read
instantaneously when written, as in synchronized message passing and CSP style
rendezvous message passing or the signals can be read after they are written
using operations such as polling the status of the signal. The formalism has
semantics to check the status of the signal in the previous clock cycle, to check
if there was any edge transition in case of event triggered FSM.

5 Formal Definition

The GALS communication interfaces of NoC are modeled as a combination
of Synchronous Finite State Machines (SFSM) and Asynchronous Finite State
Machines (AFSM) . The transitions in SFSM are triggered by the clock tick.
The transitions in AFSM are triggered by events. The modeling language used
to model both the asynchronous and synchronous functions is known as Het-
erogeneous Protocol Automata (HPA).The interaction at the communication
interfaces are through an electrical link, which acts as a buffer to store status of
signal. The control signals are single length buffered communication channels.

The following example demonstrates handshake between microprocessor and
slave devices to request bus control. In the synchronous model, the actions are
performed at the clock tick. In asynchronous model, the action are performed

on edge transitions of the signal. The difference between the synchronous and asynchronous model is determinism in time.

5.1 Definition Adapted from SPA

The HPA retains some of the formal definitions of SPA formalism such as types of automata communication channel, synchronized read and write on control channels. The HPA has two types of channels: control and data channels. These channels are further classified into input and output channels based on the read or write actions performed on the channel. Therefore, the automata has input control channel, output control channel, input data channel and output data channel.

The actions on an input control channel are: reading presence of a signal a?, reading absence of a signal #a. The read action on the transition acts as a guard takes the transitions immediately when the guard becomes true.

The actions on the output control channel are: write presence of a signal a!. The write action on output channel permits delayed write if there is a silent transition (Tau) in the state or synchronous write where the write happens in the next clock tick and it is not permitted to stay in the state for more than one clock tick unless it has an explicit self-loop.

The write actions on output data lines is denoted by d!. The read actions on input data lines is denoted by d? reading for electrical lines at the same instant or some time later. so its like reading from a single size memory. The data lines are non-blocking. The write actions are non-blocking, the read channel act as guards on the transition and hence they are blocking.

5.2 Definition proposed in HPA

New definitions are introduced in HPA to enable the same modeling of both synchronous and asynchronous interfaces in the same formal language. The difference between synchronous and asynchronous model is specified in automata definition. Similar to SPA automata, the synchronous automata is defined to executed in locked step at every clock tick. The asynchronous automata is defined to execute independent of clock based on signal transitions. The semantics
differentiates synchronous and asynchronous model with the definitions of automata.

In the output control channel, an action is defined for writing absence of a signal #a!. This action is added as our definition assumes that signal presence is sustained until the absence is written explicitly in contrast to definition of SPA where assertion has to be written every cycle to sustain a signal or it automatically de-asserts in the next cycle. Similarly in Esterel, sustain(S) is defined to sustain presence of signal until deasserted.

In the input control channel additional actions are defined to enable delayed reading of signals. The new definitions are: delayed reading of signal presence a?? and delayed reading of signal absence #a??!. The delayed reading takes the transition if the signal is present or awaits till the signal is present. The read can happen either before the channel is written or after the channel is written but either way it takes the transition after it is true. The signal written once can be read multiple times, but every delayed read should have a write in it is path since the initial state. This is definition performs action similar await(S) and present(S) in Esterel.

All the control signals in the input control channel has a signal register. The register stores the status of control signal in the previous clock cycle in SFSM and previous transition in AFSM. The register updates every clock tick in SFSM and it is updated during state transitions in AFSM. The register associated with the control channel can be read using $channel_name. Esterel language uses pre(S) to store signal values in the previous clock cycle.

In an FSM the states indicate processing and transition indicate input or output actions required to go to next state. The processing at the state can take \( n \) clock cycles, for \( n \geq 1 \) clock cycle. If the processing takes more than 1 clock cycle it is indicated by a self-loop on that state with a \( Tau \) action in SPA, but in HPA we denote states having explicit self-loop with a suspend action denoted by suspend. In SPA, the states with \( Tau \) action comes out of the self-loop if any of the other outgoing transition is true; whereas in HPA the outgoing transition is taken only after the computations in the state are complete therefore they are called suspend action.

The complementary actions on data channels d! and d? need not happen at the same instant, they can be synchronized or asynchronous. Each data channel has a type and length associated with it. channel_type(d) can be serial or parallel. size(d) is integer which denotes width of d in parallel type, size(d) is not used for serial data channel. access_type(d) can be virtual channel or time multiplexed or simple circuit switched. The slot(d) and slotid(d) are properties of multiplexed access type only; slot(d) denotes maximum slot for multiplexed access, slotid(d) current access slot. The channel(d) and priorityid(d) are properties of virtual channel types only; channel(d) denotes maximum number of virtual channels permitted for the channel, channelid(d) denotes the virtual circuit id of the current item.

The data channels with Multiplexed, Pipelined data interfaces require FIFOs at source and destination. The example of multiplexed data channel is TDMA interface and pipelined data interface is AMBA processor bus interface. The data channel with virtual channel interfaces require FIFO only at the source. Simple point to point data channels that transmit and receive one data requires no FIFOs. The data channels that are associated with a FIFO are modeled as counters in FSM with actions to increment or decrement counter. When the
data is read from the input data channel into FIFO the counter increments, when the data is read from FIFO the counter decrements. When the data is written on the output data channel the counter decrements and when data is written in FIFO the counter increments.

Finally, the synchronous and asynchronous FSM have different formats of signal transition. Generically, the actions on a transition are defined as

\[ s \xrightarrow{B_1;C;B_2} s' \]

where, B1 is a pre-guard and C is the communication action and B2 is a post-guard. In SFMS B2 is not used, the transition are of the format.

\[ s \xrightarrow{B_1;C} s' \]

The blocking read actions on input control channel appears as pre-guard in B1 and non-blocking write actions on output control and data channels belong to communication action in C. In AFMS B1, B2 are used for modeling asynchronous systems, where B1 and B2 are optional. The system waits in ‘s’ till it receives B2 to transition to s’. The execution of actions happen in the order they appears.

<table>
<thead>
<tr>
<th>Table 5.1: Channel Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
</tr>
<tr>
<td>Instantaneous Write (control signal)</td>
</tr>
<tr>
<td>Instantaneous Read (control signal)</td>
</tr>
<tr>
<td>Delayed Read (control signal)</td>
</tr>
<tr>
<td>Read Previous (control signal)</td>
</tr>
<tr>
<td>Self-loop on the state</td>
</tr>
<tr>
<td>Write Data (data signal)</td>
</tr>
<tr>
<td>Read Data (data signal)</td>
</tr>
</tbody>
</table>

5.3 Assumptions on Channel Properties

Control Channel

The control channels represent the hardware electrical wires. The control channels are not pipelined and hence are one word bounded buffers. Since it is a one word bounded buffer, it does not need any buffer management. The signal can be overwritten by the sender. The receiver must be designed to read instantaneously if the message is not to be lost. There are no centralized processes such as semaphores in hardware to co-ordinate message transfer. Hence the sender is always non-blocking. This means that the sender writes the control
signal whenever it is ready to write. The receiver process reads it instantaneous (equivalent to synchronized handshake in formal language) if it has been waiting to read. The receiver can perform a delayed read (equivalent to asynchronous communication in formal language).

**Data Channel**

The data channels represent the hardware electrical wires. The data channels can be timed-division multiplexed (TDM) or re-used as virtual channels. The TDM data-channels are used in synchronous pipeline. Virtual channels are used in asynchronous pipeline. The receiver can perform a delayed or instantaneous read. Hence it is represented as read on data channel.

---

### 6 Definition of Heterogeneous Protocol Automata

A Heterogeneous protocol Automaton (HPA) is a finite state machine with bounded counters to model clocked and clockless systems. The definition allows modeling of event based transitions and clock based transitions.

HPA is a tuple $A=(Q,clk,C, D,V,T,q_i,q_f)$, where

- $Q$ is a set of protocol states
- $clk$ defines if the automaton works on clock ticks or not
- $C$ is a set of input and output control channels ($C_I \cup C_O$)
- $V$ is a set of internal counters that can be associated data channels, buffers and fifos. During initialization $V(d)$ and $\text{Capacity}(V)$ is defined. In $V(d)$, $d$ defines the data channel or buffer or fifo associated with the counter and $\text{Capacity}(V)$ defines the capacity of counter. The counter can be incremented or decremented using $V++$ or $V--$ respectively. The increment and decrement action on the counters happen at the states not at the transition.
- $D$ is a set of input and output Data channels ($D_I \cup D_O$)
- $T$ is the transition relation $T \subseteq Q \times A(C) \times A(D) \times A(D_c) \times Q$, where
  - $A(C)$ is the set of actions on the control channels, $A(D)$ is the set of actions on the data channels, $A(D_c)$ is the set of actions of the counters (if any) of data channels.
  - $A(C) = \{s!, #s!, #s, s?, s??, #s??, s_{\text{suspend}}, \$s\}$ for $s \in C$
  - $A(D) = \{d!, d?\}$ for $d \in D$. During initialization $\text{width}(d)$, $\text{slot}(d)$, $\text{type}(d)$, $v(d)$ are defined.
- $q_i$ is the initial state
- $q_f$ is the final state
Definition of Path

A path in a HPA automaton is a sequence of alternating states and transitions. Hence a path of $A$ is given as

$$\pi_A^n = q_0 \xrightarrow{Q_1} q_1 \xrightarrow{Q_2} q_2 \cdots \xrightarrow{Q_k} q_k$$

such that for all $0 \leq m < k, q_m \xrightarrow{Q_n} q_{m+1}$ is a transition in the automaton and $n$ is the number of paths in automaton.

- $|\pi|$ denotes the number of transitions in $\pi$, also known as length of $\pi$
- $\text{Paths}(A, q_j, q_k)$ denotes the (possibly infinite) set of paths in $A$ from $q_j$ to $q_k$
- $\text{Writepresences}(\pi, s) = \{i \mid 0 < i \leq |\pi| \land s! \in Q_k\}$ is the set of indices on the path $\pi$ where there is a write action $s!$ on control channel $s$
- $\text{Writeabsences}(\pi, s) = \{i \mid 0 < i \leq |\pi| \land \#s! \in Q_k\}$ is the set of indices on the path $\pi$ where there is a write action $\#s!$ on control channel $s$
- For a path $\pi = (q_0, s_0) \xrightarrow{Q_0, s_0} \cdots \xrightarrow{Q_k, s_k} (q_{k+1}, s_{k+1}) \in A \parallel B$, where $q_j \in A$ and $s_j \in B$, the projection of $\pi$ on $A, \pi \parallel A = q_0 \xrightarrow{Q_0} q_1 \cdots \xrightarrow{Q_k} q_k + 1$. The projection $\pi \parallel B$ is defined similarly.

7 Rules for communication on control channels

7.1 Rules for synchronous transition

The basic rules for correct communication between synchronous FSMs on control and data channels are:

- The write action of control signal in output control channel communicate only with the read actions of the same control signal on input control channel. That is if control signal $a \in C$, $C$ being control channel. The write actions $a!$ of signal $a$ can be read only using read actions $\#a!, a!!, a??, \#a??$ of control signal $a$.

- The write actions of data signals in data channel communicate only with the read actions of then same data signal on data channel. The interpretation of this rule is similar to the above rule.

- Cyclic dependency between the control signals is not permitted, if $q_1 \xrightarrow{B_1; C_1} q_2$ and $s_1 \xrightarrow{B_2; C_2} s_2$ then $B_1$ must not be dependant on $C_2$ and $B_2$ must not be dependant on $C_1$

- The FSMs operate in locked steps, at each clock tick both the FSMs check the transition that can be taken and move to the next state.

- The order of appearance is preserved or checked as a union at the end of transition - to be finalized.
7.2 Rule for instantaneous read and write

The write \((a!)\) and read \((a?)\) must happen in the same transition. If the write has not occurred the automaton waits in the previous state to make a synchronized transition after write occurs.

\[
q_1 \xrightarrow{Q_1} q_2 \in A \quad (7.1)
\]
\[
s_1 \xrightarrow{S_1} s_2 \in B \quad (7.2)
\]
\[
s! \in Q_1 \land s? \in S_1 \quad (7.3)
\]
\[
(q_1, s_1) \xrightarrow{Q_1, S_1} (q_2, s_2) \in A \parallel B \quad (7.4)
\]

If the instantaneous read is occurring without a instantaneous write it must be excluded in the correct communication subset.

\[
q_1 \xrightarrow{Q_1} q_2 \in A \quad (7.5)
\]
\[
s_1 \xrightarrow{S_1} s_2 \in B \quad (7.6)
\]
\[
s! \notin Q_1 \land s? \in S_1 \quad (7.7)
\]
\[
(q_1, s_1) \xrightarrow{Q_1, S_1} (q_2, s_2) \notin A \parallel B \quad (7.8)
\]

7.3 Rule for Suspended/Delayed Write

The number of cycles the automaton waits in the previous cycle depends on the time writing is suspended to complete the operation in the current state.

\[
q_1 \xrightarrow{Q_1} q_1 \land q_1 \xrightarrow{Q_2} q_2 \in A \quad (7.9)
\]
\[
s_1 \xrightarrow{S_1} s_2 \in B \quad (7.10)
\]
\[
s_{suspend} \in Q_1 \land s! \in Q_2 \land s? \in S_1 \quad (7.11)
\]
\[
(q_1, s_1) \xrightarrow{Q_1} (q_1, s_1) \in A \parallel B \quad (7.12)
\]
\[
(q_1, s_1) \xrightarrow{Q_2, S_2} (q_2, s_2) \in A \parallel B \quad (7.13)
\]

7.4 Rule for delayed read on control channel

The read \((a??)\) requires a write \((a!)\) in the any of the previous transitions in the path, without being overwritten by another write. The value on signal \(a\) is read after the operation is previous cycle is complete and does not need to take a transition when the status is true. Therefore, the number cycles the automata remains in the current state depends on the operation complexity in the state. Similar to the instantaneous read, the automaton remains in the state checking status of \(a\) if it has not be written previously. Therefore, the automaton remains in the state after the computation is complete until the guard on the outgoing transition becomes true. The rule for delayed read consists of three sub-rules depending on when read is happening.

Delayed read happening after write

First, it is checked if there is any valid write in the path to the present state. The valid write one where the value of the signal is not being written over by
other action. If there is a presence of valid write in the previous path the state transitions to next state.

**Delayed read happening before write**

First, it is checked if there is any valid write in the path to the present state. If there is no valid write, it waits for write to occur in the future and the delayed read synchronizes with future write.

**Postponed Read**

Here, the checking of the value is postponed even after a valid write is on the path. This option is given to enable process to remain in the present state for n clock cycles to complete its task. The process then checks the status of postponed read to relative state of the other process.

**Delayed read happening after write**

\[
q_i \xrightarrow{Q_k} q_k \cdots q_{k+n} \xrightarrow{Q_{k+n}} q_f \in A \text{ where } i \text{ is initial state, } f \text{ is final state} \quad (7.14)
\]

\[
s_m \xrightarrow{S_m} s_{m+1} \in B \land s?? \in S_m \quad (7.15)
\]

\[
\pi_n = \text{Paths}(A\Vert B, (q_i, s_i), (q_k, s_m)), \text{paths from } (q_i, s_i) \text{ to } (q_k, s_m) \quad (7.16)
\]

∀ paths \(\pi_n\),

\[
\text{Writepresences}(\pi_n, s) = \{x \in N | 0 < x \leq |k| \land s! \in Q_k\} \quad (7.17)
\]

\[
\text{Writeabsences}(\pi_n, s) = \{y \in N | 0 < y \leq |m| \land \#s! \in Q_k\} \quad (7.18)
\]

Check if the last action on control channel \(s\) was a write action in in atleast one path of \(\pi_n\). ∀n, check if write present is on channel \(s\) at the last index \(l\)

\[
\text{Writepresences}(\pi_i, s)[l] > \text{Writepresences}(\pi_j, s)[l], j \in n \quad (7.19)
\]

if so, \((q_k, s_m) \xrightarrow{Q_k, S_m} (q_{k+1}, s_{m+1}) \in A\Vert B \quad (7.20)
\]

The order of appearance of actions on transition is not considered, it is sampled just before making a transition to next state. If there was a write action in atleast one path of in the previous states as well as a write in the present state. The status of signal written in the present cycle overrides the past status as the signals are sampled at the end of clock cycle not in micro ticks.

\[
\text{Writepresences}(\pi_j, s)[l] > \text{Writepresences}(\pi_j, s)[l], j \in n \quad (7.21)
\]

if \#a! \in Q_k

\[
\text{then}(q_k, s_m) \xrightarrow{Q_k, S_m} (q_{k+1}, s_{m+1}) \notin A\Vert B \quad (7.22)
\]

\[
\text{Writepresences}(\pi_a, s)[l] > \text{Writepresences}(\pi_j, s)[l], \text{for patha} \quad (7.25)
\]

\[
\text{Writeabsences}(\pi_b, s)[l] > \text{Writeabsences}(\pi_j, s)[l], \text{for pathb} \quad (7.26)
\]

if so, \((q_k, s_m) \xrightarrow{Q_k, S_m} (q_{k+1}, s_{m+1}) \in A\Vert B \quad (7.27)
\]

\[
\text{exception, if there is a presence of write absence and write presence in more than one path the delayed read is preserved to be decide during verification.}
\]

\[
\text{Writepresences}(\pi_a, s)[l] > \text{Writepresences}(\pi_j, s)[l], \text{for patha} \quad (7.25)
\]

\[
\text{Writeabsences}(\pi_b, s)[l] > \text{Writeabsences}(\pi_j, s)[l], \text{for pathb} \quad (7.26)
\]

if so, \((q_k, s_m) \xrightarrow{Q_k, S_m} (q_{k+1}, s_{m+1}) \in A\Vert B \quad (7.27)\]
**Postponed Read**

Even if the previous write presences are true, the state can decide not to take the transition and postpone taking a transition. The number of postpone cycle is decide by the state.

\[ \text{Writepresences}(\pi_j, s)[l] > \text{Writepresences}(\pi_j, s)[l], j \in n \]  \hspace{1cm} (7.28)

\[ \text{if so}, (q_k, s_m) \xrightarrow{Q_k, S_m} (q_k, s_{m+1}) \cdots \xrightarrow{Q_k, S_{m+r}} (q_{k+1}, s_{m+r}) \in A \parallel B \]  \hspace{1cm} (7.29)

**Delayed read happening before write**

If the above is not true then check for future write presence on channel \( s \)

\[ \pi_{n1} = \text{Paths}(A \parallel B, (q_k, s_m), (q_k, s_f)) \], paths from \((q_k, s_m)\) to \((q_k, s_f)\)  \hspace{1cm} (7.30)

\[ \forall \text{ paths } \pi_{n1}, \]

\[ \text{Writepresences}(\pi_{n1}, s) = \{x \in N | 0 < x \leq |k| \land s! \in Q_k\} \]  \hspace{1cm} (7.31)

\[ \text{Writeabsences}(\pi_{n1}, s) = \{y \in N | 0 < y \leq |m| \land \# s! \in Q_k\} \]  \hspace{1cm} (7.32)

Check if the first action on control channel \( s \) in future is a write action in atleast one path of \( \pi_{n1} \). \( \forall n1, \) check if write present is on channel \( s \) at the first index 1

\[ \text{Writepresences}(\pi_j, s)[1] > \text{Writepresences}(\pi_j, s)[1], j \in n \]  \hspace{1cm} (7.33)

\[ \text{if so}, (q_k, s_m) \xrightarrow{Q_k, S_m} (q_k, s_{m+1}) \in A \parallel B \]  \hspace{1cm} (7.34)

if there is no past or future writes then,

\[ (q_k, s_m) \xrightarrow{Q_k, S_m} (q_k, s_{m+1}) \notin A \parallel B \]  \hspace{1cm} (7.35)

### 7.5 Rule for read previous on control channel

The read previous \((a)\) action updates the status of the signal in the previous clock cycle is constantly. It is therefore not dependent on the present value of the signal. The value of \( a \) is updated in a virtual register \( reg(s) \) constantly in the background by a FSM before (or) at the end of transitioning to next state.

\[ q_i \xrightarrow{Q_i} q_{i+1}(\text{reg}(a_i)) \xrightarrow{Q_{i+1}} q_{i+2}(\text{updatereg}(a_{i+1})) \]  \hspace{1cm} (7.36)

\[ q_{i+2} \xrightarrow{Q_{i+2}} q_{i+3}(\text{updatereg}(a_{i+2})) \cdots \]  \hspace{1cm} (7.37)

\[ q_k \xrightarrow{Q_k} q_{k+1}(\text{updatereg}(a_k)) \cdots \]  \hspace{1cm} (7.38)

\[ q_{k+n} \xrightarrow{Q_{k+n}} q_f \in A \text{ where } i \text{ is initial state, } f \text{ is final state} \]  \hspace{1cm} (7.39)

\[ s_m \xrightarrow{S_m} s_{m+1} \in B \land \# s \in S_m \]  \hspace{1cm} (7.40)

\[ (q_k, s_m) \xrightarrow{Q_k, S_m} (q_{k+1}, s_{m+1}) \in A \parallel B(freg(a_{k-1})= \text{ presence of } a \text{ even if } \# a \in Q_k) \]  \hspace{1cm} (7.41)
Difference between delayed read and read previous

The delayed read (a ??) reads previously written present status of the signal. Therefore delayed read cannot be used instead of read previous. The use of read previous ensures that it is in that state for one clock cycle, but using delayed read the transition takes times which is dependent on the state that does delayed read. The delayed read can be used for loosely synchronous models, that can resynchronize after read operation.

7.6 Additional rules for asynchronous transition

The rule for actions are same as the above sections. The transition takes place on clock tick in synchronous FSM but in asynchronous FSM the transition is based on actions, not on time. The rules for asynchronous transition are:

\[
q_1 \xrightarrow{B_1; C_q; B_2} q_2 \quad (7.42)
\]
\[
s_1 \xrightarrow{B_1; C_s; B_2} s_2 \quad (7.43)
\]

then \(B_1\) must not be dependant on \(C_2\) and \(B_2\) must not be dependant on \(C_1\)

- \(B_{q1}; C_q; B_{q2}\), where \(B_{q1}\) and \(B_{q2}\) are guards (optional) and \(C_q\) can be data or control channel non-blocking actions
- \(B_{q1}; C_q\) communicates with \(C_s; B_{s2}\) or \(C_q; B_{q2}\) communicates with \(B_{s1}; C_s\)
- Cyclic redundancy between guards and communication channels are permitted
- \(B_1; C; B_2\) communication with \(C; B_1; C\) is not permitted now.

7.7 Rule for counters

The increment \((v + +)\) and decrement \((v - -)\) actions that happen on counter \((v)\) are preserved at the states during parallel composition.

8 Synchronous Product

Given two HPA automata, \(A = (Q, clk_i, C_i, D_i, T_i, q_0, q_f)\) and \(B = (S, clk_i, C_i, D_i, T_i, s_0, s_f)\), \(i = 1, 2\).

The synchronous product is derived when two automata operate synchronously, that is clk1 and clk2 are frequency and phase locked (isochronous) derived from the same global source. That is \(clk_1 = clk_2\).

There is absence of cyclic redundancy on control lines. \(C_{t1} \cap C_{t2} = \phi\) and \(C_{o1} \cap C_{o2} = \phi\) and \(D_{t1} \cap D_{t2} = \phi\) and \(D_{o1} \cap D_{o2} = \phi\).

We define synchronous product automaton as \(A_{sync} \mid B = (QXS, C_1 \in C_2, D_1 \in D_2, q_0, s_0, q_f, s_f)\), where \((q_1, s_1) \rightarrow Q_1, S_1(q_2, s_2)\) is a transition of \(A_{sync} \mid B\) if \(q_1 \rightarrow Q_1q_2\) and \(s_1 \rightarrow S_1s_2\), such that may\((Q_1, S_1)\) is true and set of actions is \(Q_1 \cup S_1\).
The pruning rules can be used to derive the actual system from complete product composition using \( \text{may}(Q_1, S_1) \) predicates. The synchronous transitions \( Q_1 \) and \( S_1 \) are of the form \( B_1, C \), where \( B_1 \) is guard and \( C \) is communication. The guard operations \( B_1 = \{ s?, \# s, \# s? ?, \# s? ?, s, s_{suspend} \} \) and communication actions are \( C = \{ s!, s_l, d_l, d? \} \). The guards are blocking actions, where as communication actions are non-blocking.

9 Asynchronous Product

Given two HPA automata, \( A = (Q, clk_i, C_i, D_i, T_i, q_{i0}, q_f) \) and \( B = (S, clk_i, C_i, D_i, T_i, s_{i0}, s_f) \), \( i = 1, 2 \).

The asynchronous product is obtained two automata operate asynchronously, that is, if \( clk_1 \) and \( clk_2 \) are absent (or) \( clk_1 \) is present but \( clk_2 \) is absent (or) \( clk_2 \) is present but \( clk_1 \) is absent (or) \( clk_1 \) and \( clk_2 \) are present but they are not frequency and phase locked (mesochronous) although derived from the same global source (or) \( clk_1 \) and \( clk_2 \) are present but they are not frequency and phase locked derived from independent source

There is absence of cyclic redundancy on control lines. \( C_{i1} \cap C_{i2} = \phi \) and \( C_{i1} \cap C_{i2} = \phi = \phi \) and \( D_{i1} \cap D_{i2} = \phi \) and \( D_{i1} \cap D_{i2} = \phi \).

We define asynchronous product automaton as \( A_{async} = (QXS, C_1 \in C_2, D_1 \in D_2, (q_{i0}, s_{i0}), (q_{f}, s_f)) \), where \( (q_1, s_1) \rightarrow Q_1(q_2, s_2), (q_1, s_1) \rightarrow S_1(q_1, s_2), (q_1, s_1) \rightarrow Q_1, S_1(q_2, s_2) \) is a transition of \( A_{async} \) if \( q_1 \rightarrow Q_1q_2 \) and \( s_1 \rightarrow S_1s_2 \), such that \( \text{may}(Q_1, S_1) \) is true and set of actions is \( Q_1 \cup S_1 \).

The pruning rules can be used to derive the actual system from complete product composition using \( \text{may}(Q_1, S_1) \) predicates. The asynchronous transitions \( Q_1 \) and \( S_1 \) are of the form \( B_1, C, B_2 \), where \( B_1, B_2 \) are guards and \( C \) is communication. The guard operations \( B_1, B_2 = \{ s?, \# s, s??, \# s??, s, s_{suspend} \} \) and communication actions are \( C = \{ s!, s_l, d_l, d? \} \). The guards are blocking actions, where as communication actions are non-blocking.

**Definition of \( \text{may}(Q_1, S_1) \) predicates**

The predicate \( \text{may}(Q_1, S_1) \) is true for two actions in \( Q_1 \) and \( S_1 \). The transitions \( Q_1 \) and \( S_1 \) are of the form \( B_1, C \) for synchronous product and \( B_1, C, B_2 \) for asynchronous product. The cyclic dependency is not permitted between actions on control channel in \( Q_1 \) and \( S_1 \) for synchronous product. But, the cyclic dependency between \( Q_1 \) and \( S_1 \) actions are permitted for asynchronous product as they are not based on clock tick but edge transitions.

The predicate is true iff for every control channel for \( s \in C_1, C_2 \), the actions in \( B_1 \) and \( B_2 \) have communication action on \( C \):

- if \( s? \in Q_1 \), then \( s! \in S_1 \)
- if \( \# s \in Q_1 \), then \( \# s! \in S_1 \)
- if \( s?? \in Q_1 \), then \( s! \) in the path of \( S_1 \) and \( \# s! \) is not in the path after \( s! \)
- if \( \# s?? \in Q_1 \), then \( \# s! \) in the path of \( S_1 \) and \( s! \) is not in the path after \( \# s! \)
- if \( \$ s \in Q_1 \), then \( s! \) in the previous cycle of \( S_1 \) and \( \# s! \) is not in the path after \( s! \)
10 Correct communication subset

The algorithm to generate communication subset from complete parallel product or complete synchronous product will be derived in this section. From the complete product the rules for various actions on control and data channel can be implemented successively to obtain correct.

If two synchronous FSMs are combined, the complete synchronous product is used to obtain a synchronous FSM based on clock tick. If two asynchronous FSMs or one synchronous and one asynchronous FSM is combined the desired complete parallel product will be used to obtain an asynchronous FSM based on signal transition. The communication rules for control and data channel action will be used to obtain correct communication subset, which will be used for model checking or simulation based verification.

The series of algorithms required to extract correct communication are:

- Correct instantaneous read and instantaneous write - keep the transitions where \(a! \) and \(a?\) appear in pair. Remove the transitions where \(a?\) appears without pair. Keep the transitions with \(a!\) as they will be required for delayed read.
- Correct instantaneous read and suspended write - keep the self-loops on the state as they preserve the time delay.
- Correct delayed read - check if there is write in the previous paths without overwritten value, if so keep the \(a??\) transition else remove the transition with \(a??\).
- Correct previous read
- Give extraction rule when one is sync FSM and other is async FSM or both are async FSM where cyclic dependency is allowed on the transition.
- For different types of data channels and counters

11 Complete Parallel Product

Given two HPA automata, \(A = (Q, clk_i, C_i, D_i, T_i, q_0, q_f)\) and \(B = (S, clk_i, C_i, D_i, T_i, s_0, s_f)\), \(i = 1, 2\).

The complete parallel product is nothing but gross product. It is defined as \(A \parallel B = (Q \times S, C_1 \cup C_2, D_1 \cup D_2, (q_0, s_0), (q_f, s_f))\), where \((q_1, s_1) \xrightarrow{Q_1, S_1} (q_2, s_2)\) is a transition of \(A \parallel B\) iff \(q_1 \xrightarrow{Q_1} q_2\) or \(s_1 \xrightarrow{S_1} s_2\). Here the where the FSMs progress asynchronously and they will be pruned further based on rules on control channel.

The path in a parallel composition of two HPA automata is \(A \parallel B\) is defined as \(\pi_{A,B} = (q_0, s_0) \xrightarrow{Q_1, S_1} (q_1, s_1) \xrightarrow{Q_2, S_2} (q_2, s_2) \cdots \xrightarrow{Q_k, S_k} (q_k, s_k)\) such that there exists matching paths \(\pi_{A,n}\) and \(\pi_{B,m}\) in A and B, such that

\[
\begin{align*}
\pi_{A,n} &= q_0 \xrightarrow{Q_1} q_1 \xrightarrow{Q_2} \cdots \xrightarrow{Q_k} q_k \\
\pi_{B,m} &= s_0 \xrightarrow{S_1} s_1 \xrightarrow{S_2} \cdots \xrightarrow{S_k} s_k
\end{align*}
\]
Algorithm for obtaining gross product is shown below:

Algorithm 1 CompleteParallelProduct(A,B)
1: Input:Two FSMs. This is used if both or one of the two FSMs are asynchronous FSM
2: Output: C = A | B, complete parallel composition of two FSMS with all the states and transitions from initial state.
3: P_c = (q_i, s_j) // list of state in the complete product starting from initial states of A and B
4: for all states (q_1, s_2) ∈ P_c do
5: for all transitions q_1 Q_1 \rightarrow q_2 ∈ A do
6: for all transitions s_1 S_1 \rightarrow s_2 ∈ B do
7: Add transitions (q_1, s_1) Q_1∪S_1 \rightarrow (q_2, s_2)
8: if (q_2, s_2) ∉ P_c then
9: Add (q_2, s_2) to P_c
10: end if
11: end for
12: end for
13: end for

12 Complete Synchronous Product

Given two HPA automata, A = (Q, clk_i, C_i, D_i, T_i, q_0, q_f) and B = (S, clk_i, C_i, D_i, T_i, s_0, s_f), i = 1, 2.

The complete synchronous product is defined as A ∥ B = (Q ∗ S, C_1 ∪ C_2, D_1 ∪ D_2, \rightarrow, (q_0, s_0), (q_f, s_f)), where the FSMS progress in locked step. The two automata operate in locked step when clk_1 and clk_2 are frequency and phase locked (isochronous) derived from the same global source. For this rule, clk_1 = clk_2. There must be absence of cyclic redundancy on control lines. C_{I1} \cap C_{I2} = \emptyset and C_{O1} \cap C_{O2} = \emptyset and D_{I1} \cap D_{I2} = \emptyset and D_{O1} \cap D_{O2} = \emptyset.

If q_1 Q_1 \rightarrow q_2 and s_1 S_1 \rightarrow s_2 then (q_1, s_1) Q_1∪S_1 \rightarrow (q_2, s_2) ∈ A ∥ B if Rel(Q_1, S_1) is true. The Rel(Q_1, S_1) are pruning rules based on rules for communication on control channel. The synchronous transitions S1 and S2 are of the form B1,C, where B1 is guard and C is communication. The guard operations B1 ⊂ \{s?, #s, st?, #st?, s suspend\} and communication actions are C ⊂ \{s!, d!, d?\}. The guards are blocking actions, where as communication actions are non-blocking.

The following Rel(Q_1, S_1) rules are used to obtain the communication subset of the system.
Instantaneous Write and Instantaneous Read

\( Rel_a(Q_1, S_1) \) is true for sets of actions on control channels of \( Q_1, S_1 \), when one action is blocking and other is non-blocking.

\[
\begin{align*}
\text{if} a! \in Q_1 \text{ and } a? \in S_1 & \quad (12.1) \\
\text{if} \#a! \in Q_1 \text{ and } \#a \in S_1 & \quad (12.2) \\
\text{if} a! \in S_1 \text{ and } a? \in Q_1 & \quad (12.3) \\
\text{if} \#a! \in S_1 \text{ and } \#a \in Q_1 & \quad (12.4) \\
\text{then} (q_1, s_1) \xrightarrow{Q_1, S_1} (q_2, s_2) & \in A \parallel B \quad (12.5)
\end{align*}
\]

Instantaneous write and Delayed read

\( Rel_b(Q_1, S_1) \) is true for sets of actions on control channels of \( Q_1, S_1 \), when one action is blocking and other is non-blocking.

\[
\begin{align*}
\text{if} a! \in Q_1 \text{ and } a?? \in S_1 & \quad (12.7) \\
\text{if} \#a! \in Q_1 \text{ and } \#a?? \in S_1 & \quad (12.8) \\
\text{then} (q_1, s_1) \xrightarrow{Q_1, S_1} (q_2, s_1) & \in A \parallel B \quad (12.9) \\
\text{if} a! \in S_1 \text{ and } a?? \in Q_1 & \quad (12.10) \\
\text{if} \#a! \in S_1 \text{ and } \#a?? \in Q_1 & \quad (12.11) \\
\text{then} (q_1, s_1) \xrightarrow{Q_1, S_1} (q_1, s_2) & \in A \parallel B \quad (12.12)
\end{align*}
\]

Suspended/Delayed write(self-loop)

\( Rel_c(Q_1, S_1) \) is true, when one action is self-loop with non-blocking outgoing transition and other is blocking.

\[
\begin{align*}
\text{When } q_1 \xrightarrow{Q_1} q_1 & \quad (12.14) \\
q_1 \xrightarrow{Q_2} q_2 & \quad (12.15) \\
s_1 \xrightarrow{S_1} s_2 & \quad (12.16) \\
\text{if } a_{\text{suspend}} \in Q_1, a! \in Q_2 \text{ and } a? \in S_1 & \quad (12.17) \\
\text{then} (q_1, s_1) \xrightarrow{Q_1} (q_1, s_1) & \in A \parallel B \quad (12.18) \\
\text{then} (q_1, s_1) \xrightarrow{Q_2, S_1} (q_2, s_2) & \in A \parallel B \quad (12.19)
\end{align*}
\]
When one action is self-loop with non-blocking outgoing transition and other is non-blocking.

\[
\text{if } a_{\text{suspend}} \in Q_1, a! \in Q_2 \text{ and } b! \in S_1
\]
then \((q_1, s_1) \xrightarrow{Q_1} (q_1, s_1) \in A \parallel B \quad (12.21)\)
\[
\text{then} (q_1, s_1) \xrightarrow{Q_1; S_1} (q_1, s_2) \in A \parallel B
\]
then \((q_1, s_2) \xrightarrow{Q_1} (q_1, s_2) \in A \parallel B \quad (12.24)\)

**Instantaneous write and Instantaneous write**

\(\text{Rel}_d(Q_1, S_1)\) is true for sets of actions on control channels of \(Q_1, S_1\), when both actions are non-blocking.

\[
\text{if } a! \in Q_1 \text{ then } b! \in S_1 \text{ then } (q_1, s_1) \xrightarrow{Q_1; S_1} (q_2, s_2) \in A \parallel B
\]

**Instantaneous read and Instantaneous read**

\(\text{Rel}_r(Q_1, S_1)\) is false for sets of actions on control channels of \(Q_1, S_1\), when both actions are blocking.

\[
\text{if } a? \in Q_1 \text{ then } b? \in S_1
\]
then \((q_1, s_1) \xrightarrow{Q_1; S_1} (q_2, s_2) \notin A \parallel B \quad (12.27)\)

**Delayed read and Delayed read**

\(\text{Rel}_f(Q_1, S_1)\) is true for sets of actions on control channels of \(Q_1, S_1\), when both actions are blocking.

\[
\text{if } a?? \in Q_1 \text{ then } b?? \in S_1
\]
then \((q_1, s_1) \xrightarrow{Q_1; S_1} (q_2, s_2) \in A \parallel B \quad (12.31)\)

**Read Previous**

\(\text{Rel}_p(Q_1, S_1)\) is true for sets of actions on control channels of \(Q_1, S_1\), when both actions are blocking.

\[
\text{if } a \in Q_1 \text{ then } a! \in S_1
\]
then \((q_1, s_1) \xrightarrow{Q_1; S_1} (q_2, s_2) \in A \parallel B \quad (12.32)\)
\[
\text{if } a \in Q_1 \text{ then } \#a! \in S_1
\]
then \((q_1, s_1) \xrightarrow{Q_1; S_1} (q_2, s_2) \in A \parallel B \quad (12.34)\)
\[
\text{if } a \in Q_1 \text{ then } a?? \in S_1
\]
then \((q_1, s_1) \xrightarrow{Q_1; S_1} (q_2, s_2) \in A \parallel B \quad (12.36)\)
\[
\text{if } a \in Q_1 \text{ then } a! \in S_1
\]
then \((q_1, s_1) \xrightarrow{Q_1; S_1} (q_2, s_2) \in A \parallel B \quad (12.37)\)
Using rules for message passing between automaton, the correct communication automaton can be obtained from the complete product. The rules are defined in the next subsection which can be used to extract the desired communication paths.

Algorithm 2 CompleteSynchronousProduct(A,B)
1: Input: Two FSMs. This is used if both FSMs are synchronous FSM
2: Output: $C = A \parallel B$, complete parallel composition of two synchronous FSMs with all the states and transitions from initial state.
3: $P_c = (q_i, s_j)$ // list of states in the complete product starting from initial states of A and B
4: for all states $(q_1, s_2) \in P_c$ do
5:   for all transitions $q_1 \xrightarrow{Q_1} q_2 \in A$ do
6:     if $\text{Rel}_a(Q_1, S_1)$ then
7:       CheckAdd($P_c, (q_2, s_2)$)
8:     else if $\text{Rel}_b(Q_1, S_1)$ then
9:       CheckAdd($P_c, (q_2, s_1)$) \lor CheckAdd($P_c, (q_1, s_2)$)
10:    else if $\text{Rel}_c(Q_1, Q_2, S_1)$ then
11:      CheckAdd($P_c, (q_1, s_1)$) \lor CheckAdd($P_c, (q_1, s_2)$) \lor CheckAdd($P_c, (q_2, s_2)$)
12:    else if $\text{Rel}_d(Q_1, S_1)$ then
13:      CheckAdd($P_c, (q_2, s_2)$)
14:    else if $\text{Rel}_e(Q_1, S_1)$ then
15:      CheckRemove($P_c, (q_2, s_2)$)
16:    else if $\text{Rel}_f(Q_1, S_1)$ then
17:      CheckAdd($P_c, (q_1, s_1)$) \lor CheckAdd($P_c, (q_1, s_2)$) \lor CheckAdd($P_c, (q_1, s_2)$)
18:  end if
19: end for
20: end for
21: end for
22: end for

Algorithm 3 CheckAdd($P, s$)
1: Input: FSM $P$, state $s$
2: Output: Add state $s$ in $P$, if not present already
3: if $s \notin P$ then
4:   Add ($s$) to $P$
5: end if

The above algorithms are employed on complete synchronous product and gross product to obtain desired communication subset.

12.1 Model of GALs interface

The following section shows the example of communication between devices operation on different clocks, that communicate through an asynchronous FIFO.
Algorithm 4 CheckRemove(P,s)
1: Input: FSM P, state s
2: Output: Remove state s from P, if present already
3: if s ∈ P then
4: Remove (s) to P
5: end if

Algorithm 5 Rela(Q₁,S₁)
1: Input: Q₁,S₁ actions on transition
2: Output: True or False
3: if a! ∈ Q₁ ∧ a? ∈ S₁ then
4: CheckAdd(Pₜ,(q₂,s₂))
5: return True
6: else if #a! ∈ Q₁ ∧ #a? ∈ S₁ then
7: CheckAdd(Pₜ,(q₂,s₂))
8: return True
9: else if a! ∈ S₁ ∧ a? ∈ Q₁ then
10: CheckAdd(Pₜ,(q₂,s₂))
11: return True
12: else
13: return False
14: end if

Algorithm 6 Relb(Q₁,S₁)
1: Input: Q₁,S₁ actions on transition
2: Output: True or False
3: if a! ∈ Q₁ ∧ a?? ∈ S₁ then
4: CheckAdd(Pₜ,(q₂,s₁))
5: return True
6: else if #a! ∈ Q₁ ∧ #a?? ∈ S₁ then
7: CheckAdd(Pₜ,(q₂,s₁))
8: return True
9: else if a! ∈ S₁ ∧ a?? ∈ Q₁ then
10: CheckAdd(Pₜ,(q₁,s₂))
11: return True
12: else if #a! ∈ S₁ ∧ #a?? ∈ Q₁ then
13: CheckAdd(Pₜ,(q₁,s₂))
14: return True
15: else
16: return False
17: end if
Algorithm 7 $Rel_c(Q_1, Q_2, S_1)$
1: Input: $Q_1, Q_2, S_1$ actions on transition
2: Output: True or False
3: if $a_{\text{suspend}} \in Q_1 \land a! \in Q_2 \land a? \in S_1$ then
4: CheckAdd($P_c, (q_1, s_1)$)
5: CheckAdd($P_c, (q_2, s_2)$)
6: return True
7: else if $a_{\text{suspend}} \in Q_1 \land a! \in Q_2 \land b! \in S_1$ then
8: CheckAdd($P_c, (q_1, s_1)$)
9: CheckAdd($P_c, (q_1, s_2)$)
10: CheckAdd($P_c, (q_2, s_1)$)
11: return True
12: else
13: return False
14: end if

Algorithm 8 $Rel_d(Q_1, S_1)$
1: Input: $Q_1, S_1$ actions on transition
2: Output: True or False
3: if $a! \in Q_1 \land b! \in S_1$ then
4: CheckAdd($P_c, (q_2, s_2)$)
5: return True
6: else
7: return False
8: end if

Algorithm 9 $Rel_e(Q_1, S_1)$
1: Input: $Q_1, S_1$ actions on transition
2: Output: True or False
3: if $a? \in Q_1 \land b? \in S_1$ then
4: CheckRemove($P_c, (q_2, s_2)$)
5: return True
6: else
7: return False
8: end if

Algorithm 10 $Rel_f(Q_1, S_1)$
1: Input: $Q_1, S_1$ actions on transition
2: Output: True or False
3: if $a?? \in Q_1 \land b?? \in S_1$ then
4: CheckAdd($P_c, (q_1, s_2)$)
5: CheckAdd($P_c, (q_2, s_1)$)
6: CheckAdd($P_c, (q_2, s_2)$)
7: return True
8: else
9: return False
10: end if
Algorithm 11 $Rel_g(Q_1,S_1)$
1: Input: $Q_1,S_1$ actions on transition
2: Output: True or False
3: if $a \in Q_1 \land a \not\in S_1$ then
4:     CheckAdd($P_c,(q_2,s_2)$)
5:     return True
6: else if $a \in Q_1 \land \#a \not\in S_1$ then
7:     CheckAdd($P_c,(q_2,s_2)$)
8:     return True
9: else if $a \in Q_1 \land a?? \not\in S_1$ then
10:     CheckAdd($P_c,(q_2,s_1)$)
11:     return True
12: else
13:     return False
14: end if

Algorithm 12 Instantaneous R/W rule
1: Input: $P_c$ FSM
2: Method: Traverse all the paths, keep the transitions where $s!$ and $s?$ occur together. Removes ones with $s?$ without $s!$
3: Output: FSM preserving states that follow the rule and eliminate states violating the rule
4: if $Paths(P_c,q_0,q_f) \neq \phi$ then
5:     for all $(\pi \in Paths(P_c,q_0,q_f))$ do
6:         for all $s \in C_i$ do
7:             Writes($\pi,C_i$) = Writepresences($\pi,s! \in C_i$) = $i_1 < i_2 \cdots i_n$
8:             Reads($\pi,C_i$) = Readpresences($\pi,s! \in C_i$) = $j_1 < j_2 \cdots j_m$
9:             for all len = 1 : len $\leq$ Reads[$\pi,C_i$] do
10:                if Reads[len] = Writes[len] then
11:                    Keep the transition
12:                else
13:                    Remove transitions with standalone read instantaneous
14:                Keep transitions with standalone write instantaneous
15:            end if
16:         end for
17:     end for
18: end for
19: end if
Algorithm 13 Suspended/Delayed Write rule

1: Input: \( P_c \) FSM

2: Method: Traverse all the paths, keep the transitions where \( s_{\text{suspend}} \) and and preserve the self-loops when the outgoing actions are non-blocking writes to preserve delay.

3: Output: FSM preserving states that follow the rule and eliminate states violating the rule

4: if \( \text{Paths}(P_c, q_0, q_f) \neq \phi \) then

5: for all \((\pi \in \text{Paths}(P_c, q_0, q_f))\) do

6: for all \( s \in C_i \) do

7: \( \text{WriteSuspends}(\pi, C_i) = \text{Writepresences}(\pi, s_{\text{suspend}} \in C_i) = i_1 < i_2 \cdots i_n \)

8: \( \text{Writes}(\pi, C_i) = \text{Writepresences}(\pi, s! \in C_i) = j_1 < j_2 \cdots j_m \)

9: for all \( \text{len} = 1 : \text{len} \leq \text{Writes}[\pi, C_i] \) do

10: if \( \text{WriteSuspends}[\text{len}] = \text{Writes}[\text{len}] - 1 \) then

11: Keep the self-loop

12: Keep the standalone non-blocking write

13: end if

14: end for

15: end for

16: end for

17: end if

Figure 12.1: GALS interface

13 Example

In this paper we consider modeling a GALS NOC called Asynchronous FIFO based NoC called ANOC proposed by F.Clermidy et al.[31]. The ANOC interfaces to resources through a GALS interface. The GALS interface takes care of clock domain cross over using an asynchronous FIFO that can be written to and read with separate clocks. The clockless router interfaces to GALS interface on handshake lines. There are a number of handshake protocols proposed to ensure delay insensitivity on the communication line. The routers perform routing and scheduling of incoming messages using input and output controllers. We assume routers to contain a static routing table generated from a routing algorithm. The NOC performs priority based scheduling and the priority is encoded in the packets.
Algorithm 14 Delayed Read rule

1: Input: $P_c$, FSM
2: Method: Traverse all the paths, check if there is a past or future write, with option for postponed read.
3: Output: FSM preserving states that follow the rule and eliminate states violating the rule
4: if $\text{Paths}(P_c, q_0, q_f) \neq \emptyset$ then
5:     for all $(\pi \in \text{Paths}(P_c, q_0, q_f))$ do
6:         for all $s \in C_i$ do
7:             $\text{Writepresent}(\pi, C_i) = \text{Writepresences}(\pi, s! \in C_i) = i_1 < i_2 \cdots i_n$
8:             $\text{ReadDelayed}(\pi, C_i) = \text{Writepresences}(\pi, s?? \in C_i) = k_1 < k_2 \cdots k_l$
9:             $\text{Writeabsent}(\pi, C_i) = \text{Writeabsences}(\pi, \#s! \in C_i) = j_1 < j_2 \cdots j_m$
10:     for all $\text{len} = 1 :: \text{len} \leq \text{ReadDelayed}[\pi, C_i]$ do
11:         if $(\text{ReadDelayed}[\text{len}] \geq \text{Writepresent}[\text{len} - 1]) \lor ((\text{ReadDelayed}[\text{len}] \geq \text{Writeabsent}[\text{len} - 1]) \land (\text{Writepresent}[\text{len} - 1] \geq \text{Writeabsent}[\text{len} - 1]))$ then
12:             Keep the transition
13:         else if $(\text{ReadDelayed}[\text{len}] \geq \text{Writepresent}[\text{len} + 1])$ then
14:             Keep the transition
15:         else if postpone read then
16:             Keep successive postpone read states of gross product, decides to make a transition
17:         else
18:             Remove transitions with delayed read
19:         end if
20:     end for
21: end for
22: end for
23: end if
Algorithm 15 Read Previous rule

1: Input: $P_c$ FSM
2: Method: Traverse all the paths, update read previous register, keep the transitions where $s$ if read previous is true.
3: Output: FSM preserving states that follow the rule and eliminate states violating the rule
4: if Paths($P_c, q_0, q_f$) $\neq \emptyset$ then
5:     for all ($\pi \in$ Paths($P_c, q_0, q_f$)) do
6:         for all $s \in C_i$ do
7:             Length of path $l = |\pi|$
8:             for all ($r = 0; r \leq l$) do
9:                 updatepresences($\pi, C_i$) = 1 if present and 0 if absent
10:             end for
11:         Readpreviouses($\pi, C_i$) = Readpresences($\pi, s \in C_i$) = $j_1 < j_2 \cdots j_m$
12:         for all len = 1 :: len $\leq$ Readpreviouses($\pi, C_i$) do
13:             if updatepresences[Readpreviouses(len) − 1] = 1 then
14:                 Keep the transition
15:             else
16:                 Remove transitions with read previous
17:             end if
18:         end for
19:     end for
20: end for
21: end if

13.1 Synchronous handshake Protocol

The synchronous handshake protocol specified for Asynchronous NoC [31] is based on virtual channel multiplexing. The handshake protocol with message sequence is shown in figure1.

The conditions for the sender to transmit new data on virtual channel are: presence of accept signal in the previous clock cycle and by asserting the send signal. Atmost one virtual channel can use the communication channel at a given time.

The parallel composition according to the given synchronous parallel composition rules are in Appendix 1.

13.2 Asynchronous handshake Protocol

The handshake communication protocol between the devices is shown. The data / send / accept signal is therefore implemented as an asynchronous handshake channel. The handshakes on the send and accept signals are sufficient to perform synchronization between the devices. The channel is idle when the accept signal is high. Since this is a 4-phase handshake protocol, the signal level of the handshake signals send and accept alternate 4 times before data is exchanged. The data transmitted on the asynchronous pipeline is modeled as communication action with send as pre-guard and accept as post-guard.

The automata of asynchronous protocols at the sender, receiver and the parallel composition using asynchronous parallel composition rules are given in
Appendix 2.

14 Verification

In the parallel composition of synchronous protocol between sender and receiver we perform verification of properties such as:

- Dead lock: Absence of states without next state
• Priority: high priority channels are not blocked by lower priority channels at a given time

The properties can be presented using temporal logic.

15 Road Map

• Tighten may and may rules
• Do more examples for async parallel composition as clocked with unclocked and asynchronous clocks
• Propose parallel composition rules for counters
• Verify properties on smaller models using existing model checking tool or develop model checking algorithms if the properties cannot be verified using existing tools
Model and verify complete NoC including Switch

16 Conclusions

The interface can be interchangeably modeled using any type of communication methodology. Each modeling methodology denotes different kinds of formal methods. But using different formal methods to model requires methods of integrating these methods. But we have proved here that same formalism can have different formal semantics to model different communication interfaces.

Bibliography


17 Appendix 1

18 Appendix 2
Figure 17.1: Synchronous Parallel Composition
Figure 18.1: Asynchronous Parallel Composition