Modeling and Verification of NoC Communication Interfaces

Vinitha Palaniveloo[1]   Arcot Sowmya[1]   Sridevan Parameswaran[1]

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

Technical Report
UNSW-CSE-TR-0921
December 2009

Abstract

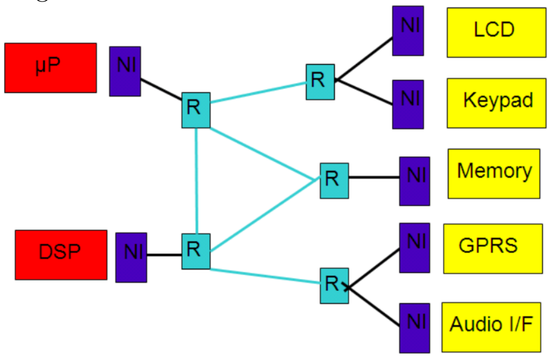
The concept of Network on Chip (NoC) addresses the communication requirements on-chip and decouples communication from computation. One of the challenges faced by designers of NoC is verifying the correctness of communication scheme for a NoC Architecture. The NoCs borrow the networking concept from computer networks to interconnect complex Intellectual Property (IP) on chip. 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. The NoC communication architecture is heterogeneous as it consists of both synchronous and asynchronous nodes and communication pipelines. In this project, we propose a heterogeneous formalism for modeling and verification of NoC. 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 [1–6]. 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



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 [21–32]. 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 was made to define generic model [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, finites state machines were used for modeling synchronous NoC [21–26]. CHP and ASC an extension of System C is used for modeling asynchronous circuits in asynchronous 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 Other Heterogeneous Formalism

The synchronous modeling languages is a well developed discipline of languages used to model synchronous system. A synchronous system consists of multiple processes where each process is modeled as Automata as states and transitions where the system changes state every clock tick. The synchronous models are easy to verify they are deterministic with finite number of states and are reactive hence the state changes are predictable. The inter-process communication is modeled in the automata using state variables or control variables on the transition. The protocol of inter-process communication depends on whether sender and receiver are blocking or non-blocking. The synchronous modeling enables one or multiple protocols of interaction between automata. There synchronous models that transition on clock tick are basic forms of synchronous language. There are discrete time models, dense time models and continuous time models. The examples of synchronous modeling language are LUSTURE, ESTEREL, and STATECHART.

Asynchronous modeling languages are still a developing area of research. The asynchronous system consists of multiple process where each process executes sequentially. The inter-process communication is always synchronized exchange of messages as sender and receiver wait and exchange messages by handshake. The asynchronous modeling languages such as CHP, CRP that communicate use CSP-style rendezvous communication. The languages require the receiver and sender to synchronous before exchanging communication and atmost one data is exchanged at a given time.

Yet another system of modeling language exists it is called Heterogeneous modeling language. Heterogeneous system consists of multiple process where each process executes on independent clock ticks. The clock ticks of the processed are not synchronized. Hence the interaction between the processes is asynchronous. Hence it requires a sequential inter-process communication modeling techniques. There are languages such as CRSM to model GAL interface that can model both clocked and un-clocked states but it still has CSP style rendezvous communication that enables exchange of atmost one data item. Here, we are proposing a heterogeneous modeling language that can model all types of communication in the GALS pipeline between nodes operating with clock or clocklessly. Some of the heterogeneous modeling languages are meta-model frameworks that derive actual system from a high level of abstraction like time tagged model, ForSyde, Meteropolis, Ptolemy. Some of the heterogeneous modeling languages are ACFSM, CRSM and CRP which is formalisms proposed to model both synchronous and asynchronous specifications in the same language.

System level design frameworks and languages such as SystemC, SpecC, Ptolemy II, expresses heterogeneous models by modeling each sub-system using a model of computation based on the inherent computational nature. The most renowned classification of MoCs includes Finite State Machine (FSM), Discrete-Time (DE), Synchronous Dataflow (SDF), Communicating Sequential Processes (CSP), Kahn Process Network (KPN).

Heterogeneous models are modeled using a framework that can model different models of computation in a single formalism like SML-Sys, Tagged Signal Model .These are a meta model of computation, which allows one to model various communication and computation semantics in a uniform way with a high level of abstraction. These meta-models are generic models of computation with high level of abstraction and the other models of computation are obtained by some type of transformation.

Our approach uses different models of computation modeled in a same formal framework with a low level of abstraction without requiring any transformation from synchronous model to other sub-domains. We use bottom-up approach in achieving system level modeling language for NoC, which is to obtain a basic formalism that is capable of modeling the heterogeneous communication interfaces of the components of NoC. In this paper, we propose our heterogeneous modeling language for modeling synchronous and asynchronous modules of on-chip interfaces. This language is based on Synchronous Protocol Automata, a synchronous formalism proposed for modeling synchronous bus interfaces on-chip. This is a simple formalism based on combing synchronous and asynchronous FSMs.

5 Heterogeneous Protocol Automata

Heterogeneous Protocol Automata (HPA) 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 behaviour 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 layers 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 communication 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 communication 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 before 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.

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 Heterogeneous 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.

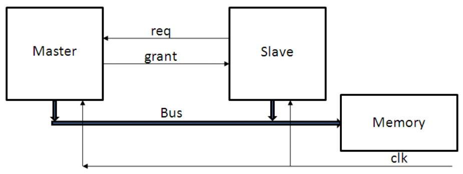


Figure 5.1: Master/Slave Controller with Shared Memory

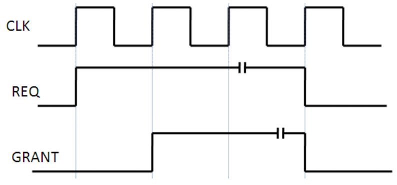


Figure 5.2: Synchronous Bus Arbitration Protocol

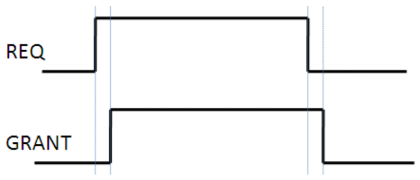


Figure 5.3: Asynchronous Bus Arbitration Protocol

The example demonstrates handshake between microprocessor and slave devices to request bus control. The handshake protocol can be implemented synchronously or asynchronously. In the synchronous protocol the actions are performed at the clock tick whereas, in asynchronous protocols the actions are performed on edge transitions of the signal. With the existing framework different formalisms are required to verify these protocols. Using GALS languages such as CRSM, CRP are not suitable for modeling a completely synchronous or asynchronous systems as they focus on message passing between asynchronous processes. Existing heterogeneous modeling languages such as SYS-sml and LSV time tagged formalism model generically at a high level of abstraction which is refined to synchronous or asynchronous models for implementation. Hence we propose a language capable of handling message passing between different models of computation in a single framework.

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 writes 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.

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 execute in locked step at every clock tick. The asynchronous automata are 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 have 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 $n >= 1$ 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 $s_{suspend}$. 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 channels with virtual channel interfaces require FIFO only at the source. A simple point to point data channel that transmits and receives 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

$$s \xrightarrow{B1;C;B2} s'$$

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.

$$s \xrightarrow{B1;C} s'$$

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 happens in the order they appears.

Control Channel Properties

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

Table 5.1: Operations on control channel

Operation	Input Channel	Output Channel
Instantaneous Write		$a!$, $(\#a!)$
Instantaneous Read	$a?$, $(\#a?)$	
Delayed Read	$a??$, $(\#a??)$	
Read Previous	$\$a$, $\$\#a$	

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 instantaneously (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 Properties

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.

Hence the model of synchronous and asynchronous handshake protocol shown in figure 2 and 3 is shown in HPA on figure 4 and 5. The synchronous protocol is modeled as clocked finite state machines (FSMs) in Fig.4. The application schedules the communication interface of the master to receive the bus arbitration request. The master may or may not service the request immediately depending on the scheduling of the application. The result of master and slave state machines operating synchronously in parallel ($Master||_{sync}Slave$) for all the three scenarios is shown in Fig.4.

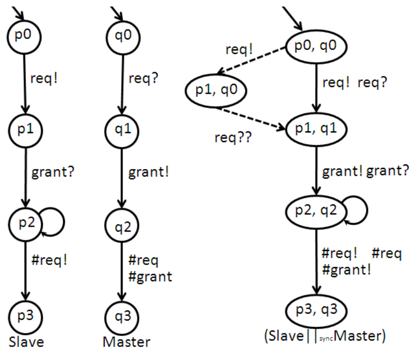


Figure 5.4: Model of Synchronous Bus Arbitration Protocol

The asynchronous are implemented using sequential logic that are edge/level triggered. Therefore, signal assertions/deassertions are detected instantaneously except for latency on the wires. The protocol may be modeled as an event triggered state machine (Fig.5). The format of the transition label in the asynchronous FSM is (pre-guard; communication action;post-guard). The pre-guard is the condition that must be true for the communication action to take place, and the post-guard is the condition that must be true for the transition to move to the next state.

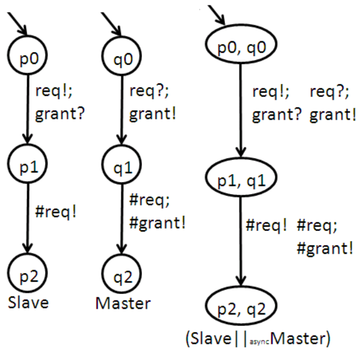


Figure 5.5: Model of Asynchronous Bus Arbitration Protocol

6 Formal Definition of HPA

An Heterogeneous Protocol Automaton (HPA) is a finite state machine with bounded counters that models clocked and clock less systems. The definition allows modeling of event and clock based transitions.

HPA is a tuple $\mathbf{A} = (Q, clk, C, D, V, T, q_0, q_f)$, 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 $(C_I \cup C_O)$
- D is a set of input and output data channels $(D_I \cup D_O)$. Channel properties such as $width(d), slot(d), type(d)$ are defined during initialization.
- V is a set of counters. For $v \in V$, v is the counter associated with a data channel or automaton $\mathbf{A}$. Capacity $capacity(v)$ and initial value $init(v)$ of the counter are defined during initialization. The counter can be incremented $(v{+}{+})$ or decremented $(v{-}{-})$. The current value can obtained with $len(v)$.
- $T \subseteq Q \times A(C) \times A(D) \times A(D_c) \times Q$ is the transition relation, where
  - $A(C)$ is the set of actions on the control channels, $A(D)$ the set of actions on the data channels, $A(D_c)$ the set of actions on the counters.
  - $A(C) = \{a!, \#a!, \#a, a?, a??, \#a??, a_{suspend}, \$a\}$ for $a \in C$
  - $A(D) = \{d!, d?\}$ for $d \in D$.
  - $A(D_c) = \{(v{+}{+}), (v{-}{-}), len(v)\}$ for $v \in V$

- q_0 is the initial state and q_f the final state

The generic transition is t: $= s \xrightarrow{l} s'$ where l is the label of the transition t. l contains both blocking and non-blocking actions, and $l \equiv B1; C; B2$ where, $B_1 \subseteq A(C) \cup A(D_c)$ and $B_2 \subseteq A(C) \cup A(D_c)$ are the pre- and post- guards, while $C \subseteq A(C) \cup A(D)$ are communication actions. The post-guard B_2 is not used in synchronous FSMs. When two HPAs execute in parallel, the correct communication subset can be obtained using the *may* predicates defined below.

6.1 Definition of Path

A path in an HPA is a sequence of alternating states and transitions. A path in HPA **A** from state q_j to state q_k is given as $\pi^A_{jk} = q_j \xrightarrow{l_j} q_{j+1} \xrightarrow{l_{j+1}} q_{j+2} \cdots \xrightarrow{l_{k-1}} q_k$ where $q_{m-1} \xrightarrow{l_{m-1}} q_m \in T$, for $j \leq m < k$.

- $|\pi^A_{jk}|$ denotes the number of transitions in π_{jk}, also known as length of π_{jk}
- $Paths(A, q_j, q_k)$ denotes the (possibly infinite) set of paths in A from q_j to q_k
- $Write(\pi^A_{jk}, a!) = \{i \in \mathcal{N}, 0 < i \leq |\pi_{jk}| \wedge a! \in l_i\}$ is the set of indices on path π^A_{jk} where $a!$ is on control channel a
- $Write(\pi^A_{jk}, \#a!) = \{i \in \mathcal{N}, 0 < i \leq |\pi_{jk}| \wedge \#a! \in l_i\}$ is the set of indices on path π^A_{jk} where $\#a!$ is on control channel a

7 Rules for Message Passing

For transitions t_1 and t_2

$$t_1 := q_1 \xrightarrow{B1_{q1};C_{q1};B2_{q1}} q_2 \tag{7.1}$$

$$t_2 := s_1 \xrightarrow{B1_{s1};C_{s1};B2_{s1}} s_2 \tag{7.2}$$

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 communicates 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 the same data signal on data channel. The interpretation of this rule is similar to the above rule.
- The order of appearance is preserved or checked as a union at the end of transition - to be finalized.

7.1 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.

$$q_1 \xrightarrow{Q_1} q_2 \in A \tag{7.3}$$
$$s_1 \xrightarrow{S_1} s_2 \in B \tag{7.4}$$
$$s! \in Q_1 \land s? \in S_1 \tag{7.5}$$
$$(q_1, s_1) \xrightarrow{Q_1, S_1} (q_2, s_2) \in A \| B \tag{7.6}$$

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

$$q_1 \xrightarrow{Q_1} q_2 \in A \tag{7.7}$$
$$s_1 \xrightarrow{S_1} s_2 \in B \tag{7.8}$$
$$s! \notin Q_1 \land s? \in S_1 \tag{7.9}$$
$$(q_1, s_1) \xrightarrow{Q_1, S_1} (q_2, s_2) \notin A \| B \tag{7.10}$$

7.2 Rule for 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.

$$q_1 \xrightarrow{Q_1} q_1 \land q_1 \xrightarrow{Q_2} q_2 \in A \tag{7.11}$$
$$s_1 \xrightarrow{S_1} s_2 \in B \tag{7.12}$$
$$s_{suspend} \in Q_1 \land s! \in Q_2 \land s? \in S_1 \tag{7.13}$$
$$(q_1, s_1) \xrightarrow{Q_1} (q_1, s_1) \in A \| B \tag{7.14}$$
$$(q_1, s_1) \xrightarrow{Q_2, S_2} (q_2, s_2) \in A \| B \tag{7.15}$$

7.3 Rule for delayed read

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 wails 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

$$q_i \xrightarrow{Q_i} q_k \cdots q_{k+n} \xrightarrow{Q_{k+n}} q_f \in A \text{ where i is initial state, f is final state} \tag{7.16}$$

$$s_m \xrightarrow{S_m} s_{m+1} \in B \wedge s?? \in S_m \tag{7.17}$$

$$\pi_n = Paths(A\|B, (q_i, s_i), (q_k, s_m)),\text{paths from } (q_i, s_i) \text{ to } (q_k, s_m) \tag{7.18}$$

$\forall$ paths π_n ,

$$Writepresences(\pi_n, s) = \{x \in N|0 < x \leq |k| \wedge s! \in Q_k\} \tag{7.19}$$

$$Writeabsences(\pi_n, s) = \{y \in N|0 < y \leq |m| \wedge \#s! \in Q_k\} \tag{7.20}$$

Check if the last action on control channel s was a write action in atleast one path of π_n. $\forall n$, check if write present is on channel s at the last index l

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

$$\text{if so,}(q_k, s_m) \xrightarrow{Q_k, S_m} (q_{k+1}, s_{m+1}) \in A\|B \tag{7.22}$$

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.

$$Writepresences(\pi_j, s)[l] > Writepresences(\pi_j, s)[l], j \in n \tag{7.23}$$

$$\text{if}\#a! \in Q_k \tag{7.24}$$

$$\text{then}(q_k, s_m) \xrightarrow{Q_k, S_m} (q_{k+1}, s_{m+1}) \notin A\|B \tag{7.25}$$

$$\tag{7.26}$$

exception, if there is a presence of write absence and write presence in more than one path the delayed read is preserved to be decided during verification.

$$Writepresences(\pi_a, s)[l] > Writepresences(\pi_j, s)[l], forpatha \tag{7.27}$$

$$Writeabsences(\pi_b, s)[l] > Writeabsences(\pi_j, s)[l], forpathb \tag{7.28}$$

$$\text{if so,}(q_k, s_m) \xrightarrow{Q_k, S_m} (q_{k+1}, s_{m+1}) \in A\|B \tag{7.29}$$

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.

$$Writepresences(\pi_j, s)[l] > Writepresences(\pi_j, s)[l], j \in n \tag{7.30}$$

$$\text{if so,}(q_k, s_m) \xrightarrow{Q_k, S_m} (q_k, s_{m+1}) \cdots \xrightarrow{Q_k, S_m + r} (q_{k+1}, s_{m+r}) \in A \| B \tag{7.31}$$

Delayed read happening before write

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

$$\pi_{n1} = Paths(A\|B, (q_k, s_m), (q_k, s_f)),\text{paths from } (q_k, s_m) \text{ to } (q_k, s_f) \tag{7.32}$$

$\forall$ paths π_{n1} ,

$$Writepresences(\pi_n 1, s) = \{x \in N | 0 < x \leq |k| \wedge s! \in Q_k\} \tag{7.33}$$

$$Writeabsences(\pi_n 1, s) = \{y \in N | 0 < y \leq |m| \wedge \#s! \in Q_k\} \tag{7.34}$$

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

$$Writepresences(\pi_j, s)[1] > Writepresences(\pi_j, s)[1], j \in n \tag{7.35}$$

$$\text{if so,}(q_k, s_m) \xrightarrow{Q_k, S_m} (q_{k+1}, s_{m+1}) \in A \| B \tag{7.36}$$

if there is no past or future writes then,

$$(q_k, s_m) \xrightarrow{Q_k, S_m} (q_{k+1}, s_{m+1}) \notin A \| B \tag{7.37}$$

7.4 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.

$$q_i \xrightarrow{Q_i} q_{i+1}(reg(a_i)) \xrightarrow{Q_i + 1} q_{i+2}(updatereg(a_{i+1})) \tag{7.38}$$

$$q_{i+2} \xrightarrow{Q_i + 2} q_{i+3}(updatereg(a_{i+2})) \cdots \tag{7.39}$$

$$q_k \xrightarrow{Q_k} q_{k+1}(updatereg(a_k)) \cdots \tag{7.40}$$

$$q_{k+n} \xrightarrow{Q_{k+n}} q_f \in A \text{ where i is initial state, f is final state} \tag{7.41}$$

$$s_m \xrightarrow{S_m} s_{m+1} \in B \wedge \$s \in S_m \tag{7.42}$$

$$(q_k, s_m) \xrightarrow{Q_k, S_m} (q_{k+1}, s_{m+1}) \in A \| B \text{iff} reg(a_{k-1}) = \text{ presence of } a \text{ even if } \#a \in Q_k \tag{7.43}$$

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 to model loosely synchronous systems.

7.5 Rules for Synchronous Operation

The synchronous 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. Cyclic dependency between the control channels of the transition is not permitted. If $q_1 \xrightarrow{B1_{q1};C_{q1}} q_2$ and $s_1 \xrightarrow{B1_{s1};C_{s1}} s_2$ then B_{q1} must not be dependent on C_{s1} and B_{s1} must not be dependent on C_{q1}

7.6 Rules for Asynchronous Operation

The asynchronous FSMs operate independently as the transition is based on actions. Each transition takes place when the guard becomes true irrespective of the state of other FSMs.

If $q_1 \xrightarrow{B1_{q1};C_{q1};B2_{q1}} q_2$ and $s_1 \xrightarrow{B1_{s1};C_{s1};B2_{q1}} s_2$ then

- $B1_{q1}; C_{q1}$ communicates with $C_{s1}; B2_{s1}$
- $C_{q1}; B2_{q1}$ communicates with $B1_{s1}; C_{s1}$
- Cyclic redundancy between guards and communication channels are permitted
- $B1; C; B2$ communication with $C; B1; C$ 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 Parallel Composition Rules

8.1 Definition of $may(t_1, t_2)$ predicates

For HPAs **A** and **B**, and transitions t_1: $= q_m \xrightarrow{l_1} q_{m+1} \in \mathbf{A}$ and t_2: $= s_m \xrightarrow{l_2} s_{m+1} \in \mathbf{B}$, the predicate $may(t_1, t_2)$ holds, iff for every control channel a:

- $a! \in l_1$ and no actions on channel a in l_2
- $a_{suspend} \in l_1$ and no actions in l_2
- $a? \in l_1$ and $a! \in l_2$

- $a?? \in l_1$ then the last action on control channel a was $a!$ in the path to the transition t_2 from initial state(s_0) in B, including t_2, and $\#a!$ does not occur in the path after the last $a!$. i.e., For $\pi^B_{0m+1} = Paths(B, s_0, s_{m+1})$, $\exists i \in Write(\pi^B_{im+1}, a!)$, $0 < i \leq m+1$ and $Last[Write(\pi^B_{0m+1}, a!)] > Last[Write(\pi^B_{0m+1}, \#a!)]$, where $Last$ is the last index in the set.

- $\$a \in l_1$, then the last action in B on channel a was an assertion that is sustained in the path to transition t_2, excluding t_2. i.e., For $\pi^B_{0m} = Paths(B, s_0, s_m)$, $\exists i \in Write(\pi^B_{im}, a!)$, $0 < i \leq m$ and $Last[Write(\pi^B_{im}, a!)] > Last[Write(\pi^B_{im}, \#a!)]$, where $Last$ is the last index of the set.

- $a? \in l_1$ and $a? \in l_2$, the high priority signal has precedence

- $a! \in l_1$ and $a! \in l_2$, the high priority signal has precedence

- data lines act as guards in data flow domain

The rules are similar for deassertion.

8.2 Definition of Synchronous Product

For two HPA automata $\mathbf{A} = (Q, clk_1, C_1, D_1, V_1, T_1, q_0, q_f)$ and $\mathbf{B} = (S, clk_2, C_2, D_2, V_2, T_2, s_0, s_f)$, the synchronous product is defined when they operate synchronously, i.e., only if $clk_1 = clk_2 \neq$ NULL and $clk1, clk2$ are isochronous and derived from the same global source.

We define synchronous product automaton as $\mathbf{A} \|_{sync} \mathbf{B} = (Q \times S, clk_1, C_1 \cup C_2, D_1 \cup D_2, V_1 \cup V_2, \rightarrow, (q_0, s_0), (q_f, s_f))$ where, $(q_m, s_m) \xrightarrow{l_1 \cup l_2} (q_{m+1}, s_{m+1}) \in \rightarrow$ for transitions t_1: $= q_m \xrightarrow{l_1} q_{m+1} \in T_1$ and t_2: $= s_m \xrightarrow{l_2} s_{m+1} \in T_2$ iff $may(t_1, t_2)$ is true.

8.3 Definition of Asynchronous Product

For two HPA automata $\mathbf{A} = (Q, clk_1, C_1, D_1, V_1, T_1, q_0, q_f)$ and $\mathbf{B} = (S, clk_2, C_2, D_2, V_2, T_2, s_0, s_f)$, the asynchronous product is defined when they operate asynchronously, i.e., only if clk_1 and clk_2 are independent clocks or at least one of clk_1 and clk_2 is absent.

We define asynchronous product automaton as $\mathbf{A} \|_{async} \mathbf{B} = (Q \times S, NULL, C_1 \cup C_2, D_1 \cup D_2, V_1 \cup V_2, \xrightarrow{,} (q_0, s_0), (q_f, s_f))$ where, for transitions t_1: $= q_m \xrightarrow{l_1} q_{m+1} \in T_1$ and t_2: $= s_m \xrightarrow{l_2} s_{m+1} \in T_2$ the transitions in $\rightarrow$ are:

- $(q_m, s_m) \xrightarrow{l_1 \cup l_2} (q_{m+1}, s_{m+1})$ if $may(t_1, t_2)$ is true

- $(q_m, s_m) \xrightarrow{l_1} (q_{m+1}, s_1)$ iff $may(t_1, \phi_t)$ is true, where ϕ_t is an empty transition.

- $(q_m, s_m) \xrightarrow{l_2} (q_1, s_{m+1})$ iff $may(\phi_t, t_2)$ is true.

9 Modeling GALs interface

Two synchronous islands communicating through asynchronous FIFO, shown in Fig.6, is an example of a GALS interface used in Asynchronous NOC [14]. Asynchronous FIFOs buffer data between devices operating at different speeds. The asynchronous FIFO is accessed by Devices A and B on separate read/write interfaces with distinct clocks for read and write $(rclk, wclk)$. The FIFO is modeled as an event triggered state machine with a counter that keeps track of buffer capacity.

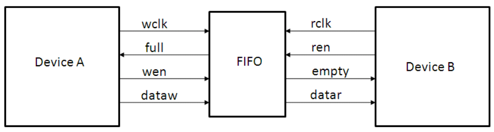


Figure 9.1: Example of GALS interface with Asynchronous FIFO

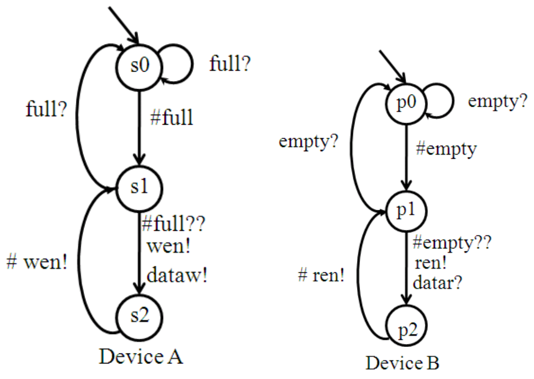


Figure 9.2: Model of GALS interface devices

The state machines of Devices A and B (Fig.7) are clocked on $rclk$ and $wclk$ respectively. A and B enter initial states $[s0]$,$[p0]$ respectively to write/read from FIFO. In the initial state FIFO status is checked before reading/writing into FIFO. If the status is full $(full?)$ device A stays in $[s0]$ until FIFO status is not full $(\#full)$ and writes data using $wen!$ and $dataw!$. If the FIFO status is empty $(empty?)$ device B stays in $[p0]$ until FIFO status is not empty $(\#empty)$ and reads data using $ren!$ and $datar!$. If the write clock $wclk$ is fast the FIFO 'overruns', i.e., it gets 'full' fast. If the read clock $rclk$ is fast FIFO 'underruns', i.e., it gets empty fast. Since, the read/write devices operate on independent clocks the combined operation is controlled by events on FIFO.

The FIFO is an event-based state machine (Fig.8). It is modeled with a counter v, which is incremented/decremented after new data is written/read. In the initial state $[init(v)]$, FIFO status is set to empty $(empty!)$ and not full $(\#full)$. When new data is written the FIFO increments the counter and moves to $[v\ {+}{+}]$ and the status is updated to not empty $(\#empty)$. In this state the FIFO is modeled to receive multiple data or wait for data to be read from it. When data is read the FIFO decrements the counter and moves to $[v\ {-}{-}]$, else remains in $[v\ {+}{+}]$ until FIFO capacity is reached, when it moves to $[q1]$ updating status to $full!$. In $q1$, FIFO does not allow writing until at least one existing data is read. When that happens FIFO moves to $[v\,--]$ updating status to not full. In $[v\,--]$ multiple data can be read, or new data written and FIFO moves to $[v\,++]$. If multiple data is read and FIFO becomes empty, it moves to initial state $[init(v)]$ updating status to *empty*!.

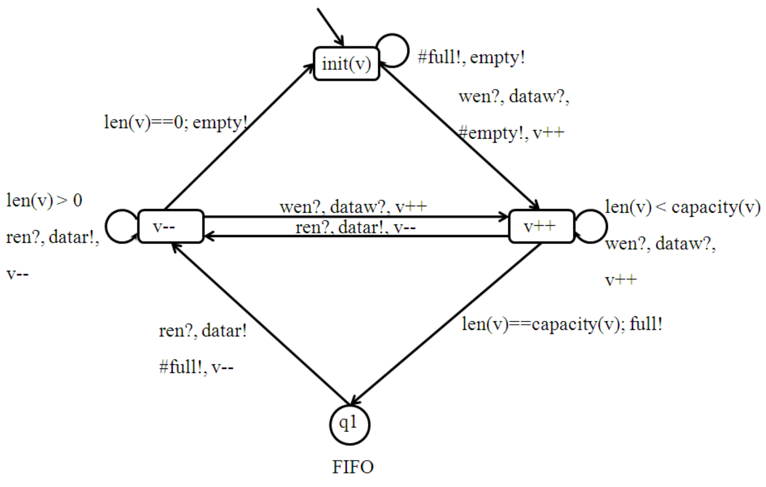


Figure 9.3: Model of FIFO

The execution state space of Device A||FIFO||Device B operating in parallel is obtained using asynchronous parallel composition on the FSMs and the resulting correct communication subset is shown in Fig.9. Although Devices A and B are clocked, the resulting state machine is based on events in FIFO.

10 Verification

The verification of HPA model can be done in two ways. First methods in using rules for message passing and obtaining correct communication subset as defined and appendix E and developing our own model checking tool for verification. The second method is translating HPA model to one of the existing model checking tools for verification. Since, the first method is time consuming our initial approach will be using third party model checking tools.

HPA contains clocked and clock less automata that operate synchronously or asynchronous. As there is no single model checking tool that can support the verification requirements of HPA. Hence, we have chosen SPIN for verification of GALS process and NuSMV to verify synchronous process to verify distributed synchronous and asynchronous process. The verification procedure involves translating HPA to SPIN for GALS models and to NUSMV or synchronous models to performing verification as shown in the following figure.

The verification is carried out by translating HPA model to kripke structure by the model checking tools. Hence, we will define the rules for translating HPA to Kripke structure.

10.1 Definition of Kripke Structure

Kripke structure is a tuple $\mathbf{K} = (P, p_i, \Theta, AP, L)$, where

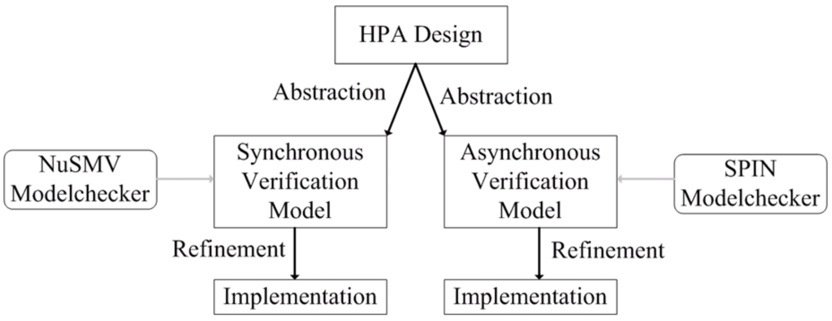


Figure 10.1: Model of FIFO

- P is set of finite states
- p_i is the initial state
- Θ is the transition relation between $P \times P$
- AP is the set of atomic propositions
- L is the labels of each states with set of atomic propositions true in each state

10.2 Translation to Kripke Structure

The rules for translating HPA to Kripke structure is:

- $AP = Q \times Clk \times A(2^C) \times A(2^D) \times A(2^V)$ where Q is the set of states in HPA, C is the set of control channels, D is the set of data channels and V is the set of counters.
- L is the labels of each states with set of atomic propositions true in each state $L : P \rightarrow AP$
- P states are denoted with atomic propositions true in the state. L(p) is the proposition true in state 'p'

The translation can be explained with a simple example shown in the following figure. The set of control channels in the example is $C = \{a, b\}$. There is no data channel or counter. The complete set of possible actions on control channel a is $A(C_a)$= $\{a!, \#a!, \#a, a?, a??, \#a??, a_{suspend}, \$a\}$ for $a \in C$. Similarly the set of actions in control channel b is given as $A(C_b)$. The set of actions on control channels are $A(C) = A(C_a) \cup A(C_b)$. The set of valid actions on the control channel in this given example is $A(C) = \{a!, b?\}$. The given HPA model is a clocked automata hence clk is present during translation.

The set of atomic propositions are $AP = \{(s_0), (s_1), (s_2), (clk), (a!), (b?), (s_0, s_1), (s_0, s_2), (s_0, clk), (s_0, a!), ($
The labels of states where atomic propositions are true are $L = \{k_{s0} = (s_0, clk, a!), k_{s1} = (s_1, clk, b?), k_{s2} = (s_2, clk)\}$. The set of states in Kripke structure are $P = \{k_{s0}, k_{s1}, k_{s2}\}$.

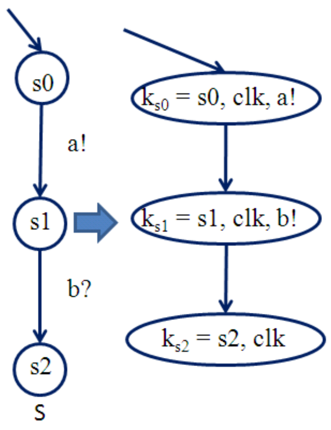


Figure 10.2: Model of FIFO

10.3 Parallel Composition of Kripke structure

Synchronous Parallel Composition

Synchronous parallel composition of two Kripke structures $K_1 = (P, p_i, \Theta_1, AP_1, L_1)$ and $K_2 = (Q, q_i, \Theta_2, AP_2, L_2)$ is a Kripke Structure $K = K_1 \|_{sync} K_2 = (R, r_i, \Theta, AP, L)$ where:

- $R = P \times Q$
- $r_i = p_i, q_i$
- transition $(p_1, q_1) \rightarrow (p_2, q_2) \in \Theta$ iff $p_1 \rightarrow p_2 \in \Theta_1$ and $q_1 \rightarrow q_2 \in \Theta_2$
- $AP = AP_1 \times AP_2$
- $L : R \rightarrow AP = ((p_1, q_1), (l_{p1}, l_{q1}))$ iff for $(p_1, l_{p1}) \in L_1$ and $(q_1, l_{q1}) \in L_2$ $may_k(l_{p1}, l_{q1})$ is true.

Asynchronous Parallel Composition

Asynchronous parallel composition of two Kripke structures $K_1 = (P, p_i, \Theta_1, AP_1, L_1)$ and $K_2 = (Q, q_i, \Theta_2, AP_2, L_2)$ is a Kripke Structure $K = K_1 \|_{async} K_2 = (R, r_i, \Theta, AP, L)$ where:

- $R = P \times Q$
- $r_i = p_i, q_i$
- transitions are
  - $(p_1, q_1) \rightarrow (p_2, q_2) \in \Theta$ iff $p_1 \rightarrow p_2 \in \Theta_1$ and $q_1 \rightarrow q_2 \in \Theta_2$
  - $(p_1, q_1) \rightarrow (p_2, q_1) \in \Theta$ iff $p_1 \rightarrow p_2 \in \Theta_1$ and $q_1 \in \Theta_2$
  - $(p_1, q_1) \rightarrow (p_1, q_2) \in \Theta$ iff $q_1 \rightarrow q_2 \in \Theta_2$ and $q_2 \in \Theta_1$
- $AP = AP_1 \times AP_2$

- $L : R \rightarrow AP = ((p_1, q_1), (l_{p1}, l_{q1}))$ iff for $(p_1, l_{p1}) \in L_1$ and $(q_1, l_{q1}) \in L_2$ $may_k(l_{p1}, l_{q1})$ is true.

They may definition of $may_k(l_{p1}, l_{q1})$ is similar to may definition of HPA, it will defined later.

10.4 Equivalence between HPA and Kripke structure

The equivalence between Kripke and HPA will be proved later.

Given two HPA H_1 and H_2 is translated to Kripke structure K_1 and K_2 respectively. It will be proved that $H_1 \equiv K_1$, $H_2 \equiv K_2$ and $H_1 \| H_2 \equiv K_1 \| K_2$.

11 Translation of HPA to SPIN

The models in SPIN are specified in PROMELA, programming language of SPIN. SPIN has two types of message channels for message passing which is rendezvous and bounded asynchronous buffered message passing. PROMELA also has integer and arrays for variables. The operation of each process is specified in different procedure and instantiated later. In rendezvous style message passing the sender and receiver are synchronized for message passing but sender can write multiple times before receiver accepts data. In asynchronous message passing the sender can write finite number of data into buffer which can be read any time later. The buffers can be programmed to block or lose manage after buffer is full. The message is consumed after the message is read. The message passing techniques of HPA is mapped to SPIN for verification as follows.

Mapping instantaneous read/write channel

The rendezvous of HPA is similar to SPIN where the sender and receiver synchronize for message passing except that the message is consumed after read in SPIN but not in HPA. But, instantaneous read/write can be mapped directly to SPIN and the channel can be traced in the programmed to rewrite it again where required. The rendezvous channel cannot be verified using 'never' claim. Hence, for the purpose of verification the instantaneous read/write channel is mapped to buffered read/write. The problem with this technique is that sender can write and proceed with execution but receiver can read it anytime later. This is handled by using a another rendezvous channel for synchronizing at that instant.

Mapping instantaneous writes and delayed read channel

The instantaneous read with delayed read is similar to asynchronous bounded buffer message passing of buffer length 1. Here, the message is consumed after every read, the model gets complex to overwrite after read and consume overwritten data. Therefore, this message passing is handled using variable.

Mapping read previous channel

There is no equivalent of reading previous status of channel, this is handled using variables in SPIN.

Mapping delayed write

Delayed write involves suspending write on a channel for specific time or clock cycles before writing, the writing is stalled using a counter for time. Depending on whether it is a instantaneous read/ delayed read it is written into asynchronous message channel or variable.

Mapping of counter

Counter is translated to variable.

Mapping of data channel

The data channel is mapped to bounded asynchronous buffered message channel.

12 Verification in SPIN

The HPA model of GALS interface shown in figure 9.2 and 9.3 is translated to spin as shown in appendix I and key FIFO properties such as overflow and deadlock are verified using SPIN.

13 Modeling and Verification of Asynchronous No

In this paper we consider modeling a GALS NOC called Asynchronous FIFO based NoC called ANOC proposed by F.Clermidy etal[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 clock less 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.

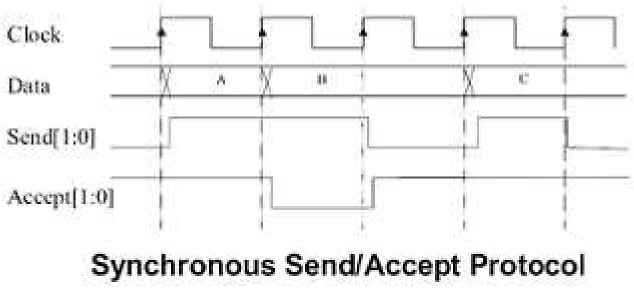
13.1 Synchronous handshake Protocol

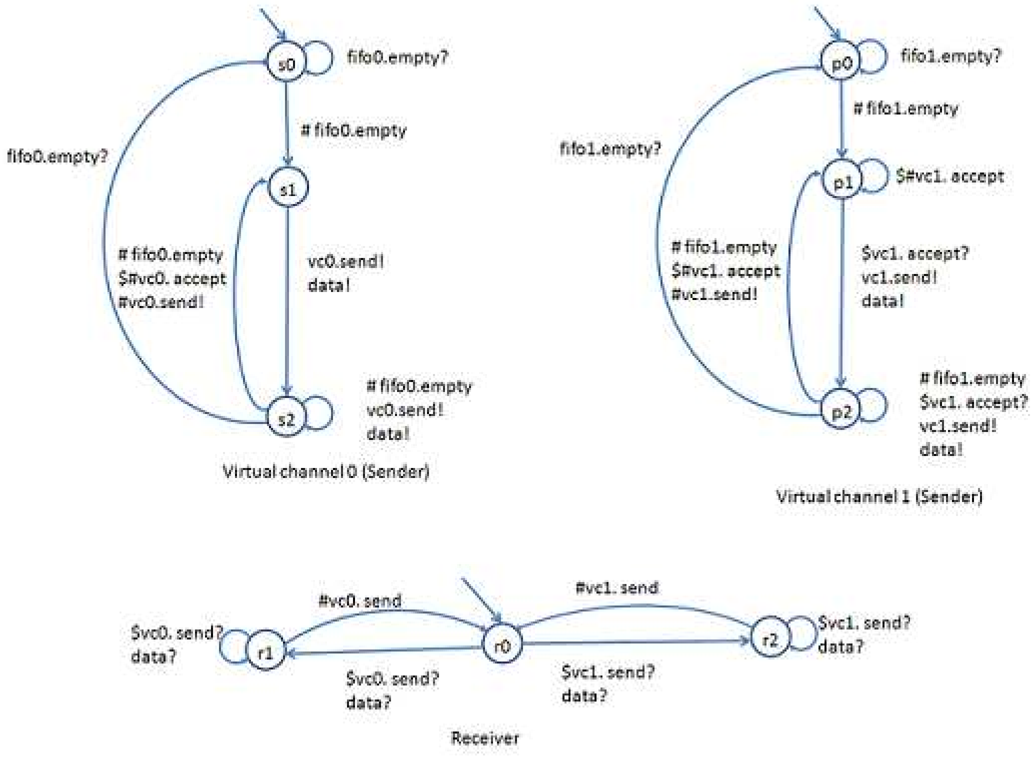
The synchronous handshake protocol specified for Asynchronous NoC [31] is based on virtual channel multiplexing. The handshake protocol with message sequence is shown in figure1.

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.

Figure 13.1: Synchronous Handshake Protocol

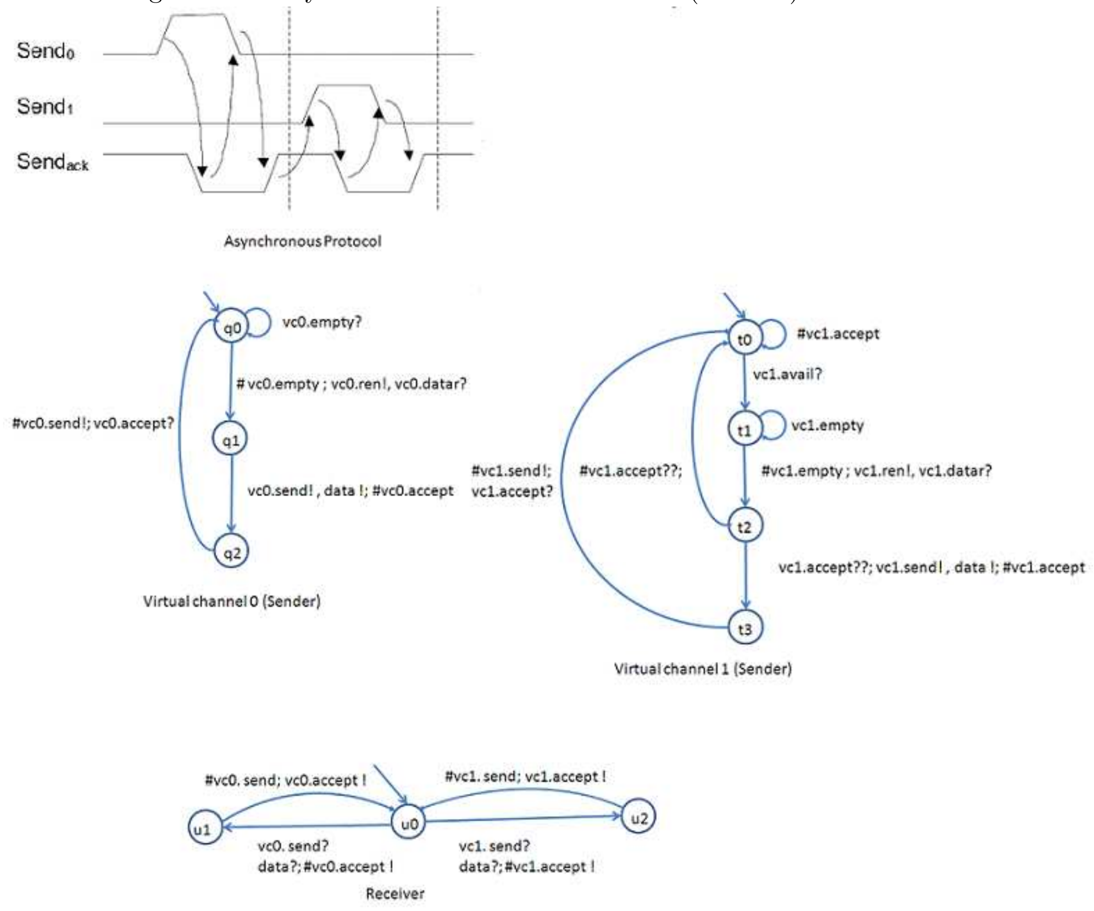




13.2 Asynchronous handshake Protocol

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 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.

Figure 13.2: Asynchronous Handshake Protocol (4-Phase)


Send0
Send1
Sendack
Asynchronous Protocol
q0
vc0.empty?
# vc0.empty ; vc0.ren!, vc0.datar?
#vc0.send!; vc0.accept?
q1
vc0.send! , data !; #vc0.accept
q2
Virtual channel 0 (Sender)
t0
#vc1.accept
vc1.avail?
t1
vc1.empty
#vc1.send!; vc1.accept?
#vc1.accept??;
#vc1.empty ; vc1.ren!, vc1.datar?
t2
vc1.accept??; vc1.send! , data !; #vc1.accept
t3
Virtual channel 1 (Sender)
#vc0. send; vc0.accept !
#vc1. send; vc1.accept !
u1
u0
u2
vc0. send? data?; #vc0.accept !
vc1. send? data?; #vc1.accept !
Receiver

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_s and may_a 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

[1] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1):70–78, Jan 2002.

[2] P. Wielage and K. Goossens. Networks on silicon: blessing or nightmare? *Digital System Design, 2002. Proceedings. Euromicro Symposium on*, pages 196–200, 2002.

[3] Hannu (Eds.) Jantsch, Axel; Tenhunen. *Networks on Chip*. Springer, 2003, VIII, 303 p., Hardcover.

[4] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M.Millberg, and D. Lindqvist. Network on chip: An architecture for billion transistor era. *Proceeding of the IEEE NorChip Conference*, November 2000.

[5] Jörg Henkel, Wayne Wolf, and Srimat Chakradhar. On-chip networks: A scalable, communication-centric embedded system design paradigm. *IEEE Computer Society,VLSID '04: Proceedings of the 17th International Conference on VLSI Design*, page 845, 2004.

[6] William J. Dally and Brian Towles. Route packets, not wires: on-chip inteconnection networks. *ACM,DAC '01: Proceedings of the 38th conference on Design automation*, pages 684–689, 2001.

[7] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. *ACM, DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 250–256, 2000.

[8] K. Goossens J. Dielissen, A. Radulescu and E. Rijpkema. Concepts and implementation of the phillips network-on-chip. *In Proceedings of the IP based SOC (IPSOC)*, Nov 2003.

[9] L. Benini D. Bertozzi, G. Biccari, M. Dallosso, and L. Giovannini. xpipes: a latency insensitive parameterized network-on-chip architecture for multi-processor socs. *IEEE, ICCD 03: Proceedings of the 21st International Conference on Computer Design*, 2003.

[10] R. Thid M. Millberg, E. Nilsson and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip. *IEEE, DATE 04: Proceedings of the conference on Design, automation and test in Europe*, 2004.

[11] F. Moraes A. Mello, N. Calazans, L. Moller, and L. Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *VLSi Journal*, 38(1), 2004.

[12] J. Sparso T. Bjerregaard. A router architecture for connection-oriented service guarantees in the mango clockless network-on-chip. *IEEE, DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, 2005.

[13] F. Clermidy A. Clouard, E. Beigne, P. Vivet, and M. Renaudin. An asynchronous noc architecture providing low latency service and its multi-level design framework. *IEEE, ASYNC 05: Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 54–63, 2005.

[14] E. Friedman D. Rostislav, V. Vishnyakov and R. Ginosar. An asynchronous router for multiple service levels networks on chip. *IEEE, ASYNC '05: Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 44–53, 2005.

[15] J. Pontes, M. Moreira, R. Soares, and N. Calazans. Hermes-glp: A gals network on chip router with power control techniques. *Symposium on VLSI, 2008. ISVLSI '08. IEEE Computer Society Annual*, pages 347–352, April 2008.

[16] D.M. Chapiro. Globally asynchronous locally synchronous systems. *PhD Thesis,Stanford University*, Oct 1984.

[17] J. Sparsø. Future networks-on-chip; will they be synchronous or asynchronous? (invited talk). *SSoCC'04 (Swedish System on Chip Conference, Båstad)*, 2004.

[18] K Goossens. Formal methods for networks on chips. *IEEE, Application of Concurrency to System Design, ACSD 2005*, pages 188–189, 2005.

[19] M.D. Grammatikakis M. Coppola, S. Curaba, R. Locatelli, G. Maruccia, and F. Papariello. Occn: a noc modeling framework for design exploration. *Elsevier, J. Syst. Archit.*, 50(2-3):129–163, 2004.

[20] J. Joven D. Castells-Rufas and J. Carrabina. A validation and performance evaluation tool for protonoc. *System-on-Chip, 2006. International Symposium on*, (1-4), 2006.

[21] F.W. Vaandrager B. Gebremichael and M. Zhang. Formal models of guaranteed and best-effort services for networks on chip. *Technical Report,ICIS-R05016,Radboud University Nijmegen*, March 2005.

[22] M. Zhang B. Gebremichael, F. Vaandrager, K. Goossens, E. Rijpkema, and A. Radulescu. *Deadlock Prevention in the thereal Protocol.* Correct Hardware Design and Verification Methods,Springer, Book Chapter, 2005.

[23] J. Schmaltz G.A. Sammane and D. Borrione. Formal design and verification of on chip networking. *IEEE, Information and Communication Technologies: From Theory to Applications*, pages 657–658, March 2004.

[24] J. Schmaltz and D.Borrione. A generic network on chip model. *LNCS, Book TPHOLs*, 3603:310–325, 2005.

[25] L.Pierre D.Borrione, A.Helmy. Acl2-based verification of the communications in the hermes network on chip. *Proc. International Workshop on Symbolic Methods and Applications to Circuit Design (SMACD'06)*, 2006.

[26] R. Shankar A. Agarwal. Modeling concurrency in noc for embedded systems. *Proc. High Performance Embedded Computing,Massachusetts Institute of Technology*, 2006.

[27] S. Kumar R. Holsmark, M. Hgberg. Modeling and evaluation of a network on chip architecture using sdl. *LNCS,11th International SDL Forum, Stuttgart, Germany*, 2708, July 2003.

[28] O. Bringmann A. Siebenborn and W. Rosenstiel. Communication analysis for network-on-chip design. *IEEE,PARELEC '04: Proceedings of the international conference on Parallel Computing in Electrical Engineering*, pages 315–320, 2004.

[29] L. Tsiopoulos and M. Waldn. Formal development of noc systems in b. *Nordic Journal of Computing*, 13(1-2):127 – 145, 2006.

[30] Y. Thonnart C. Koch-Hofer, M. Renaudin and P. Vivet. Asc, a systemc extension for modeling asynchronous systems, and its application to an asynchronous noc. *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, pages 295–306, May 2007.

[31] Y. Thonnart G. Salaun, W. Serwe and P. Vivet. Formal verification of chp specifications with cadp illustration on an asynchronous network-on-chip. *IEEE,ASYNC '07: Proceedings of the 13th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 73–82, 2007.

[32] I. Sander Z. Lu and A. Jantsch. Towards performance-oriented pattern-based refinement of synchronous models onto noc communication. *IEEE,DSD '06: Proceedings of the 9th EUROMICRO Conference on Digital System Design*, pages 37–44, 2006.

[33] K. Goossens A. Rădulescu. Communication services for networks on chip. *In Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation, Book Chapter*, pages 193–231, 2004.

A Kernel of HPA

The kernel of HPA is the primitive operations on automata. These kernel operations are represented as process calculi.

Table A.1: HPA process calculi

Sustained write	Write presence control	c!
	Write absence control	#c!
Instantaneous read	Presence control	c?
	absence control	#c
Delayed read	Presence control	c??
	Absence control	#c??
Read Previous pre(?S)	Presence control	$c
	Absence control	$#c
Data Channel	Write data	d!
	Read data	d?
Suspend/Delayed write	selfloop	$s_{Suspend}$
Operation	Parallel operation	$P\|\|Q$
	Sequential operation	P;Q

B Intuitive Semantics of HPA

This section describes how signals are emitted and how control is transmitted between states. The status of signal in the input channel is determined by the event on input channel at that instant or the past activity in the channel. The intuitive semantics gives the default behaviour. The status of signal written into the output channel is determined per-instant basis, the status of signal before writing can be either high or low. Overwriting is permitted on both data and control output channels. The input channels decided on instantaneous or delayed read. When read and write happen at the same time it is called handshake.

The intuitive semantics of the operations on automata are:

- Write on control channel c!, #c! : This writes presence (c!) or absence (#c!) of signal on output control channel. This operation terminates instantaneously. The signal sustains on output channel unless deasserted in the next cycle. This is similar to 'Sustain' in Esterel. This is not equivalent to 'Emit' in Esterel where the signal is starts and terminates in the same instance unless specified explicitly to sustain. For the operation to behave as emit, the presence and absence must be written in the same transition. This is because the output channels does not need to be typed, so that it knows before whether the write is sustain or entity type, but it is specified by the combination of actions. The write is broadcast to all other processes similar to Esterel.

- Delayed write on control channel $c_{suspend}$ c!: The self loop in the state represents suspend . Hence the write is done after completing computation in each state.

- Instantaneous Read on control channel c?, #c : As opposed to calling it a synchronous/synchronized read we call it a instantaneous read inorder to avoid confusion by the hardware engineers, where synchronous interface refers to interfaces operating with respect to clock as opposed to the meaning of formal methods where it refers that read and write happen at the same instant. Here the operation terminates instantaneously. This requires write on the input channel at the instant it reads. The read on control channel is blocking, means the process pauses or waits or idles in the state doing nothing till it receives the input. Even if the operation on the state is incomplete, when it receives the input it transitions to the next state. Pause do nothing is a microstep of this operation. The AND and OR logical operations are allowed in instantaneous read control operations.

- Delayed Read on control channel c??, #c?? : This is similar to the 'Presence' in Esterel, where the current status of the signal in input channel is checked. It reads the current status, does not require signal to be written in the current cycle. Any write in the path since the start of initial state can be read, unless it was overwritten. The electrical line is considered to buffer the value of input channel. This operation also reads instantaneously. This completes the computation in the current status and checks the input channel, does not exit on the occurrence of event. The AND and OR logical operations are allowed in delayed read control operations.

- Read Previous on control channel \$c, \$#c : This is similar to 'pre(?S) in Esterel, where the state of the signal in the previous clock cycle can be read. The status of the signal in previous cycle is constantly updates. In a clockless system this refers to the past write on the channel before the current write. The AND and OR logical operations are allowed in read Previous control operations.

- Write on data channel d! : The write on input data channel is always sustain. The data written can be overwritten before it is read.

- Read on data channel d? : The read on input data channel is always delayed. The data read can have a write in the current cycle or any of the previous cycle. It may not have a write as well, because the value of data is read is checked and necessary operations are performed in the state. But its a delayed read to mean that the read will be done only after all the computation in the current state are complete although it was written before.

- Process : The process can be clocked or unclocked. In a clocked process all the transitions occur at clock tick and the waits, self loops are multiples of clock tick. The processes working at different clock rates cannot interact directly but through GALS interface like Asynchronous FIFO, bi-synchronous FIFO or clock stopping. But from the perspective of communication interfaces by abstracting the FIFOs they can interact on input and output channels. Processes working with unrelated clock is said to work asynchronously. The processes interact on control and data channels. The control channels have necessary protocols to enable correct communication between processes.

C Logical Behavioural in HPA

This logical semantics describes logically correct execution of programs. Logical correctness is the requirement that there exists exactly one status for each signal after execution. The logical correctness ensures that it is logically reactivity and logically deterministic. Consider the following example

Let Process A

$$p1 \xrightarrow{I?S1!} p2$$

Let Process B

$$q1 \xrightarrow{\#S1S2!} q2$$

Let Process C

$$x1 \xrightarrow{S2?O!} x2$$

The system is $A\|B\|C$. For which input signal is I and output is O. The signals S1 and S2 are internal signals. Hence there are two possible execution paths,

C.1 Execution 1

- Step1: Assume I is absent when the system starts. Process A stays in p1 and S1 is not emitted. Process B transitions to q2 and S2 is emitted. Process C transitions to x2 and emits O.
- Step2: As long as I is absent Process A stays in p1, Process B stays in q2 and Process C stays in x2.
- Step3:When I becomes present. Process A transitions to p2 and emits S1. Process B and C continue to stay in q2 and x2.

C.2 Execution 2

- Step1: Assume I is present when the system starts. Process A transitions to p1 and S1 is emitted. Process B and C stays in q1 and x1 respectively.
- Step2: Even if I changes states subsequently Process A, B, C continues to stay in p2,q1,x1 respectively.

The logically incorrect behaviour is non-determinism and non-reactivity. Example of logically incorrect behaviour is:

Let Process A

$$q1 \xrightarrow{O?O!} q2$$

Process A is non-deterministic

Let Process D

$$y1 \xrightarrow{O1?O2!} y2$$

Let Process E

$$z1 \xrightarrow{\#O2O1!} z2$$

$D\|E$ is non-reactive

The logical semantics results in indeterminism.

C.3 Logical Behavioural Semantics of HPA

The logical behaviour rules are of the form

$$p \xrightarrow[I]{O} p' \qquad p \xrightarrow[I \bigcup O]{O} p'$$

The format is $I \bigcup O$ inorder to satisfy logical coherence.

Since we focus on state level modeling, we focus on state level logical semantics. Although each state is considered atomic, to take n clock cycles depending on the completion of computation in each state. For a signal s, the behaviour can be written as

$$p \xrightarrow[I(s)]{O(s')} p' \qquad s \xrightarrow[\text{E}]{E',k} s'$$

Here s denotes the logical operation and E , E' denotes the status of signals before and after execution of operation. s' is the resulting operation and k is the completion code. The presence is denoted by s^+ and absence is denoted by s^-. The unknown status is denoted by $s^\perp$ and retaining previous state is denoted by $s^\dashv$

$$\frac{p \xrightarrow{s!} p'}{s! \xrightarrow[\text{E}]{s^+,0} 0} \quad \text{(write presence)}$$

$$\frac{p \xrightarrow{\#s!} p'}{s! \xrightarrow[\text{E}]{s^-,0} 0} \quad \text{(write absence)}$$

$$\frac{p \xrightarrow{s_{suspend}} p \xrightarrow{s!} p'}{s! \xrightarrow[\text{E}]{suspend,k} s!} \quad \text{(delayed write presence)}$$

$$\frac{p \xrightarrow{s?} p'}{s? \xrightarrow[s^+]{E,0} 0\ s? \xrightarrow[s^-]{E,k} s? \Rightarrow s?p';p \xrightarrow[s^+]{E,0} p'\ s?p';p \xrightarrow[s^-]{E,k} p} \text{(instantaneous read)}$$

$$\frac{p \xrightarrow{\#s} p'}{\#s \xrightarrow[s^-]{E,0} 0\ \#s \xrightarrow[s^+]{E,k} \#s \Rightarrow \#s?p';p \xrightarrow[s^-]{E,0} p'\ \#s?p';p \xrightarrow[s^+]{E,k} p} \text{(instantaneous read)}$$

$$\frac{p \xrightarrow{s??} p'}{s?? \xrightarrow[s^+ \vee s^-]{E,k} s??\ s?? \xrightarrow[s^+]{E,0} 0 \Rightarrow s??p';p \xrightarrow[s^+]{E,0} p'\ s??p';p \xrightarrow[s^+ \vee s^-]{E,k} p} \text{(delayed read)}$$

$$\frac{p \xrightarrow{\#s??} p'}{\#s?? \xrightarrow[s^+ \vee s^-]{E,k} \#s??\ \#s?? \xrightarrow[s^-]{E,0} 0 \Rightarrow \#s??p';p \xrightarrow[s^-]{E,0} p'\ \#s??p';p \xrightarrow[s^+ \vee s^-]{E,k} p} \text{(delayed read)}$$

$$\frac{p \xrightarrow[\text{E}]{E',k} p' q \xrightarrow[\text{E}]{F',l} q' k, l \neq 0}{p \| q \xrightarrow[\text{E}]{E' \cup F', max(k,l)} p' \| q'} \qquad \text{(Parallel composition)}$$

The parallel rule performs synchronization using max(k,l).

D State Based Semantics of HPA

The state based semantics is particularly helpful in obtaining detailed computing reactions of parallel communication. Similar to Esterel we use extended syntax to denote state after a reaction. Here p denotes states, $\widehat{p}$ denotes current active state where the execution is passing. p' denotes the next state and $\overline{p}$ denotes the pausing state.

$$\frac{p \xrightarrow{s!} p'}{\widehat{p} \xrightarrow{s^{+}} p'} \qquad \text{(write presence)}$$

$$\frac{p \xrightarrow{\#s!} p'}{\widehat{p} \xrightarrow{s^{-}} p'} \qquad \text{(write presence)}$$

$$\frac{p \xrightarrow{s_{suspend}} p \xrightarrow{s!} p'}{\widehat{p} \xrightarrow{s^{\dashv}} \overline{p} \xrightarrow{s^{+}} p'} \qquad \text{(delayed write presence)}$$

$$\frac{p \xrightarrow{s?} p'}{\widehat{p} \xrightarrow[s^{-}]{s^{\dashv}} \overline{p} \xrightarrow[s^{+}]{s^{\dashv}} p'} \qquad \text{(instantaneous read)}$$

$$\frac{p \xrightarrow{\#s} p'}{\widehat{p} \xrightarrow[s^{+}]{s^{\dashv}} \overline{p} \xrightarrow[s^{-}]{s^{\dashv}} p'} \qquad \text{(instantaneous read)}$$

$$\frac{p \xrightarrow{s??} p'}{\widehat{p} \xrightarrow[s^{-} \vee s^{+}]{s^{\dashv}} \overline{p} \xrightarrow[s^{+}]{s^{\dashv}} p'} \qquad \text{(instantaneous read)}$$

$$\frac{p \xrightarrow{\#s??} p'}{\widehat{p} \xrightarrow[s^{-} \vee s^{+}]{s^{\dashv}} \overline{p} \xrightarrow[s^{-}]{s^{\dashv}} p'} \qquad \text{(instantaneous read)}$$

E 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 are 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

F 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 $A \mid B = (Q \times S, C_1 \cup C_2, D_1 \cup D_2, \rightarrow, (q_0, s_0), (q_f, s_f))$, where $(q_1, s_1) \xrightarrow{Q_1,S_1} (q_2, s_2)$ is a transition of $A \mid B$ 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 $A \mid_B$ 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 π_{A_n} and π_{B_m} in A and B. such that

$$\pi_{A_n} = q_0 \xrightarrow{Q_1} q_1 \xrightarrow{Q_2} q_2 \cdots \xrightarrow{Q_k} q_k$$
$$\pi_{B_n} = s_0 \xrightarrow{S_1} s_1 \xrightarrow{S_2} s_2 \cdots \xrightarrow{S_k} s_k$$

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: $C = A \mid B$, 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 $(q_1, s_2) \in P_c$ do
5:   for all transitions $q_1 \xrightarrow{Q_1} q_2 \in A$ do
6:     for all transitions $s_1 \xrightarrow{S_1} s_2 \in B$ do
7:       Add transitions $(q_1, s_1) \xrightarrow{Q_1 \cup S_1} (q_2, s_2)$
8:       if $(q_2, s_2) \notin P_c$ then
9:         Add $(q_2, s_2)$ to $P_c$
10:      end if
11:    end for
12:  end for
13: end for
```

G 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 $A \parallel B = (Q \times S, C_1 \cup C_2, D_1 \cup D_2, \rightarrow, (q_0, s_0), (q_f, s_f))$, 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 $q_1 \xrightarrow{Q_1} q_2$ and $s_1 \xrightarrow{S_1} s_2$ then $(q_1, s_1) \xrightarrow{Q_1, S_1} (q_2, s_2) \in A \parallel B$ 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 $B1 \subset \{s?, \#s, s??, \#s??, \$s, s_{suspend}\}$ and communication actions are $C \subset \{s!, d!, d?\}$. The guards are blocking actions, where as communication actions are non-blocking.

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 state 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:     for all transitions $s_1 \xrightarrow{S_1} s_2 \in B$ do
7:       if $Rel_a(Q_1, S_1)$ then
8:         CheckAdd($P_c, (q_2, s_2)$)
9:       else if $Rel_b(Q_1, S_1)$ then
10:        CheckAdd($P_c, (q_2, s_1)$$\vee$ CheckAdd($P_c, (q_1, s_2)$)
11:      else if $Rel_c(Q_1, Q_2, S_1)$ then
12:        [CheckAdd($P_c, (q_1, s_1)$$\wedge$ CheckAdd($P_c, (q_2, s_2)$] $\bigvee$
           [CheckAdd($P_c, (q_1, s_2)$$\wedge$ CheckAdd($P_c, (q_1, s_2)$])
13:      else if $Rel_d(Q_1, S_1)$ then
14:        CheckAdd($P_c, (q_2, s_2)$)
15:      else if $Rel_e(Q_1, S_1)$ then
16:        CheckRemove($P_c, (q_2, s_2)$)
17:      else if $Rel_f(Q_1, S_1)$ then
18:        CheckAdd($P_c, (q_1, s_1)$$\wedge$ CheckAdd($P_c, (q_1, s_2)$$\wedge$
           CheckAdd($P_c, (q_1, s_2)$)
19:      end if
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
```

Algorithm 4 CheckRemove(P,s)

```
1: Input:FSM P, state s
2: Output: Remove state s from P, if present already
3: if $s \in P$ then
4:     Remove $(s)$ to $P$
5: end if
```

Algorithm 5 $Rel_a(Q_1, S_1)$ Instantaneous Write and Instantaneous Read

```
1: Input:$Q_1, S_1$ actions on transition
2: Output: True or False
3: $Rel_a(Q_1, S_1)$ is true for sets of actions on control channels of $Q_1, S_1$, when
   one action is blocking and other is non-blocking.
```

$$\text{if} a! \in Q_1 \text{and} a? \in S_1 \tag{G.1}$$

$$\text{if} \# a! \in Q_1 \text{and} \# a \in S_1 \tag{G.2}$$

$$\text{if} a! \in S_1 \text{and} a? \in Q_1 \tag{G.3}$$

$$\text{if} \# a! \in S_1 \text{and} \# a \in Q_1 \tag{G.4}$$

$$\text{then} (q_1, s_1) \xrightarrow{Q_1, S_1} (q_2, s_2) \in A \parallel B \tag{G.5}$$

$$\tag{G.6}$$

```
4: if $a! \in Q_1 \wedge a? \in S_1$ then
5:     CheckAdd($P_c, (q_2, s_2)$)
6:     return True
7: else if $\#a! \in Q_1 \wedge \#a? \in S_1$ then
8:     CheckAdd($P_c, (q_2, s_2)$)
9:     return True
10: else if $a! \in S_1 \wedge a? \in Q_1$ then
11:     CheckAdd($P_c, (q_2, s_2)$)
12:     return True
13: else if $\#a! \in S_1 \wedge \#a? \in Q_1$ then
14:     CheckAdd($P_c, (q_2, s_2)$)
15:     return True
16: else
17:     return False
18: end if
```

Algorithm 6 $Rel_b(Q_1, S_1)$ Instantaneous write and Delayed read

```
1: Input:$Q_1, S_1$ actions on transition
2: Output: True or False
3: $Rel_b(Q_1, S_1)$ is true for sets of actions on control channels of $Q_1, S_1$, when
   one action is blocking and other is non-blocking.
```

$$\text{if} a! \in Q_1 \text{and} a?? \in S_1 \tag{G.7}$$
$$\text{if} \# a! \in Q_1 \text{and} \# a?? \in S_1 \tag{G.8}$$
$$\text{then} (q_1, s_1) \xrightarrow{Q_1, S_1} (q_2, s_1) \in A \parallel B \tag{G.9}$$
$$\text{if} a! \in S_1 \text{and} a?? \in Q_1 \tag{G.10}$$
$$\text{if} \# a! \in S_1 \text{and} \# a?? \in Q_1 \tag{G.11}$$
$$\text{then} (q_1, s_1) \xrightarrow{Q_1, S_1} (q_1, s_2) \in A \parallel B \tag{G.12}$$
$$\tag{G.13}$$

```
4: if $a! \in Q_1 \land a?? \in S_1$ then
5:     CheckAdd($P_c, (q_2, s_1)$)
6:     return True
7: else if $\# a! \in Q_1 \land \# a?? \in S_1$ then
8:     CheckAdd($P_c, (q_2, s_1)$)
9:     return True
10: else if $a! \in S_1 \land a?? \in Q_1$ then
11:     CheckAdd($P_c, (q_1, s_2)$)
12:     return True
13: else if $\# a! \in S_1 \land \# a?? \in Q_1$ then
14:     CheckAdd($P_c, (q_1, s_2)$)
15:     return True
16: else
17:     return False
18: end if
```

Algorithm 7 $Rel_c(Q_1, Q_2, S_1)$ Suspended/Delayed write(self-loop)

```
1: Input:$Q_1, Q_2, S_1$ actions on transition
2: Output: True or False
3: $Rel_c(Q_1, S_1)$ is true, when one action is self-loop with non-blocking outgoing
   transition and other is blocking.
```

$$\text{When} q_1 \xrightarrow{Q_1} q_1 \tag{G.14}$$

$$q_1 \xrightarrow{Q_2} q_2 \tag{G.15}$$

$$s_1 \xrightarrow{S_1} s_2 \tag{G.16}$$

$$\text{if} a_{suspend} \in Q_1, a! \in Q_2 \text{and} a? \in S_1 \tag{G.17}$$

$$\text{then}(q_1, s_1) \xrightarrow{Q_1} (q_1, s_1) \in A \parallel B \tag{G.18}$$

$$\text{then}(q_1, s_1) \xrightarrow{Q_2, S_1} (q_2, s_2) \in A \parallel B \tag{G.19}$$

$$\tag{G.20}$$

When one action is self-loop with non-blocking outgoing transition and other is non-blocking.

$$\text{if} a_{suspend} \in Q_1, a! \in Q_2 \text{and} b! \in S_1 \tag{G.21}$$

$$\text{then}(q_1, s_1) \xrightarrow{Q_1} (q_1, s_1) \in A \parallel B \tag{G.22}$$

$$\text{then}(q_1, s_1) \xrightarrow{Q_1, S_1} (q_1, s_2) \in A \parallel B \tag{G.23}$$

$$\text{then}(q_1, s_2) \xrightarrow{Q_1} (q_1, s_2) \in A \parallel B \tag{G.24}$$

```
4: if $a_{suspend} \in Q_1 \land a! \in Q_2 \land a? \in S_1$ then
5:     CheckAdd($P_c, (q_1, s_1)$)
6:     CheckAdd($P_c, (q_2, s_2)$)
7:     return True
8: else if $a_{suspend} \in Q_1 \land a! \in Q_2 \land b! \in S_1$ then
9:     CheckAdd($P_c, (q_1, s_1)$)
10:    CheckAdd($P_c, (q_1, s_2)$)
11:    CheckAdd($P_c, (q_2, s_1)$)
12:    return True
13: else
14:    return False
15: end if
```

Algorithm 8 $Rel_d(Q_1, S_1)$ Instantaneous write and Instantaneous write

```
1: Input:$Q_1, S_1$ actions on transition
2: Output: True or False
3: $Rel_d(Q_1, S_1)$ is true for sets of actions on control channels of $Q_1, S_1$, when
   both actions are non-blocking.
```

$$\text{if} a! \in Q_1 \text{then} b! \in S_1 \text{then} (q_1, s_1) \xrightarrow{Q_1, S_1} (q_2, s_2) \in A \parallel B \tag{G.25}$$

```
4: if $a! \in Q_1 \land b! \in S_1$ then
5:     CheckAdd($P_c, (q_2, s_2)$)
6:     return True
7: else
8:     return False
9: end if
```

Algorithm 9 $Rel_e(Q_1, S_1)$ Instantaneous read and Instantaneous read

```
1: Input:$Q_1, S_1$ actions on transition
2: Output: True or False
3: $Rel_e(Q_1, S_1)$ is false for sets of actions on control channels of $Q_1, S_1$,when
   both actions are blocking.
```

$$\text{if} a? \in Q_1 \text{then} b? \in S_1 \tag{G.26}$$

$$\text{then} (q_1, s_1) \xrightarrow{Q_1, S_1} (q_2, s_2) \notin A \parallel B \tag{G.27}$$

```
4: if $a? \in Q_1 \land b? \in S_1$ then
5:     CheckRemove($P_c, (q_2, s_2)$)
6:     return True
7: else
8:     return False
9: end if
```

Algorithm 10 $Rel_f(Q_1, S_1)$ Delayed read and Delayed read

```
1: Input:$Q_1, S_1$ actions on transition
2: Output: True or False
3: $Rel_f(Q_1, S_1)$ is true for sets of actions on control channels of $Q_1, S_1$,when
   both actions are blocking.
```

$$\text{if} a?? \in Q_1 \text{then} b?? \in S_1 \tag{G.28}$$

$$\text{then}(q_1, s_1) \xrightarrow{Q_1,S_1} (q_1, s_2) \in A \parallel B \tag{G.29}$$

$$\text{then}(q_1, s_1) \xrightarrow{Q_1,S_1} (q_2, s_1) \in A \parallel B \tag{G.30}$$

$$\text{then}(q_1, s_1) \xrightarrow{Q_1,S_1} (q_2, s_2) \in A \parallel B \tag{G.31}$$

```
4: if $a?? \in Q_1 \wedge b?? \in S_1$ then
5:     CheckAdd($P_c, (q_1, s_2)$)
6:     CheckAdd($P_c, (q_2, s_1)$)
7:     CheckAdd($P_c, (q_2, s_2)$)
8:     return True
9: else
10:    return False
11: end if
```

Algorithm 11 $Rel_g(Q_1, S_1)$ Read Previous

```
1: Input:$Q_1, S_1$ actions on transition
2: Output: True or False
3: $Rel_g(Q_1, S_1)$ is true for sets of actions on control channels of $Q_1, S_1$,when
   both actions are blocking.
```

$$\text{if}\$a \in Q_1 \text{then} a! \in S_1 \tag{G.32}$$

$$\text{then}(q_1, s_1) \xrightarrow{Q_1,S_1} (q_2, s_2) \in A \parallel B \tag{G.33}$$

$$\text{if}\$a \in Q_1 \text{then}\#a! \in S_1 \tag{G.34}$$

$$\text{then}(q_1, s_1) \xrightarrow{Q_1,S_1} (q_2, s_2) \in A \parallel B \tag{G.35}$$

$$\text{if}\$a \in Q_1 \text{then} a?? \in S_1 \tag{G.36}$$

$$\text{then}(q_1, s_1) \xrightarrow{Q_1,S_1} (q_2, s_1) \in A \parallel B \tag{G.37}$$

```
4: if $\$a \in Q_1 \wedge a! \in S_1$ then
5:     CheckAdd($P_c, (q_2, s_2)$)
6:     return True
7: else if $\$a \in Q_1 \wedge \#a! \in S_1$ then
8:     CheckAdd($P_c, (q_2, s_2)$)
9:     return True
10: else if $\$a \in Q_1 \wedge a?? \in S_1$ then
11:    CheckAdd($P_c, (q_2, s_1)$)
12:    return True
13: else
14:    return False
15: end if
```

H Algorithms for correct communication subset

The algorithms employed to obtain desired communication subset from complete synchronous product and gross product are:

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 vi-
    olating the ruke
 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 $s_{suspend}$ 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 ruke
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:       WriteSuspends$(\pi, C_i)$ = Writepresences$(\pi, s_{suspend} \in C_i) = i_1 < i_2 \cdots i_n$
8:       Writes$(\pi, C_i)$ = Writepresences$(\pi, s! \in C_i) = j_1 < j_2 \cdots j_m$
9:       for all $len = 1 :: len \leq Writes[\pi, C_i]$ do
10:        if $WriteSuspends[len] = Writes[len] - 1$ 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
```

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 vi-
   olating the ruke
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:       Writepresent$(\pi, C_i)$ = Writepresences$(\pi, s! \in C_i) = i_1 < i_2 \cdots i_n$
8:       ReadDelayed$(\pi, C_i)$ = Writepresences$(\pi, s?? \in C_i) = k_1 < k_2 \cdots k_l$
9:       Writeabsent$(\pi, C_i)$ = Writeabsences$(\pi, \#s! \in C_i) = j_1 < j_2 \cdots j_m$
10:      for all $len = 1 :: len \leq ReadDelayed[\pi, C_i]$ do
11:        if $(ReadDelayed[len] \geq Writepresent[len-1]) \bigvee$
           $((ReadDelayed[len] \geq Writeabsent[len-1]) \wedge (Writepresent[len-$
           $1] \geq Writeabsent[len-1]))$ then
12:          Keep the transition
13:        else if $(ReadDelayed[len] \geq Writepresent[len+1])$ 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 ruke
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:          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
```

I Translation of HPA to SPIN for verification

```
mtype = { msg1,high,low };
mtype = {hi,lo}

bool afull = true;
bool bempty = true;

chan wen = [1] of {mtype};
chan wensync = [0] of {mtype};

chan ren = [1] of {mtype};
chan rensync = [0] of {mtype};

int clka = 0;
int clkb = 0;
int flen = 0;

int datar = 0;
int dataw = 0;

int adatar = 0;
int adataw = 0;

proctype clocka()
     {
clockahigh:
progressCLKA1:
        if
           ::(clka==0)->  atomic{clka =1; goto clockalow; }
         fi;

clockalow:
progressCLKA2:
          if
           :: (clka==1)->  atomic{clka  = 0; goto clockahigh;}
          fi;
          }

proctype clockb()
     {
clockbhigh:
progressCLKB1:
            if
              ::(clkb==0)->  atomic{clkb =1; goto clockblow;}
             fi;


clockblow:
progressCLKB2:
```

```
             if
             ::(clkb==1)->  atomic{clkb  = 0; goto clockbhigh;}
            fi;
       }

active proctype devicea()
       {
astate0:
progressA1:
     if
        :: (clka == 1) ->
           if
              :: afull == false ->  goto astate1;
              :: afull == true -> goto astate0;
          fi;
     fi;

astate1:
           if
             ::(clka == 1)  ->
                     if
                        :: afull == false ->  atomic{wensync!high; wen!high;
                                      dataw ++; goto astate2; }
                        :: afull == true ->  goto astate0;
                     fi;
           fi;

astate2:
            if
              ::(clka == 1)  -> goto astate1; acceptL4:skip;
            fi;
       }

active proctype deviceb()
       {
bstate0:
progressB1:
    if
       :: (clkb == 1)  ->
          if
            :: bempty == false ->  goto bstate1;
            :: bempty == true -> goto bstate0;
         fi;
     fi;

bstate1:
       if
        :: (clkb == 1) ->
                 if
                    :: bempty == false ->  atomic{rensync!high; ren!high;
```

```
                            datar = adatar; goto bstate2;}
                     :: bempty == true -> goto bstate0;
                 fi;
        fi;

bstate2:
         if
           :: (clkb == 1)  -> goto bstate1;
         fi;
       }

active proctype afifo()
       {
fstate0:
progressL1:
          if
           ::wensync?high->  atomic{wen?high; adataw = dataw; flen++;
                           bempty = false; goto fstate2; }
           ::afull = false; bempty = true; goto fstate0;
           fi;

fstate2:
      if
        ::(flen < 10) ->
               if
                   :: wensync?high ->   atomic{wen?high; adataw = dataw;
                             flen++; goto fstate2;}
              fi;
        ::(flen == 10 )->  atomic{afull = true; goto fstate3;}
        ::rensync? high ->  atomic{ren?high; adatar = adataw;
                          flen--; goto fstate4;}
      fi;

fstate3:
         if
            :: rensync? high ->  atomic{ren?high; adatar = adataw;
                      afull = false; flen--; goto fstate4; }
         fi;

fstate4:
   if
        ::(flen >0) ->
                 if
                       :: rensync? high ->  atomic{ren?high; adatar = adataw;
                       flen--; goto fstate4; }
                  fi;
       :: (flen == 0) ->  atomic{bempty = true; goto fstate0;}
       :: wensync?high ->   atomic{wen?high; adataw = dataw; flen++;
                             goto fstate2;}
   fi;
```

```
}

init     {
         atomic{ run clocka() ; run clockb()}
   }
```

FIFO overflow

```
define        p            (flen !=11)
LTL property - [] p, always FIFO never overflows
A counter example is generated to check if FIFO over flows.

never {    /* !([] p) */
T0init:
if
:: (! ((p))) -> goto acceptall
:: (1) -> goto T0init
fi;
acceptall:
skip
}

Result: FIFO overflow does not happen.

State-vector 92 byte, depth reached 9999, errors: 0
  1423679 states, stored
  4041085 states, matched
  5464764 transitions (= stored+matched)
   147042 atomic steps
hash conflicts:   3328166 (resolved)

Stats on memory usage (in Megabytes):
  146.634 equivalent memory usage for states (stored*(State-vector + overhead))
  103.432 actual memory usage for states (compression: 70.54%)
          state-vector as stored = 60 byte + 16 byte overhead
    2.000 memory used for hash table (-w19)
    0.305 memory used for DFS stack (-m10000)
  105.626 total actual memory usage
```

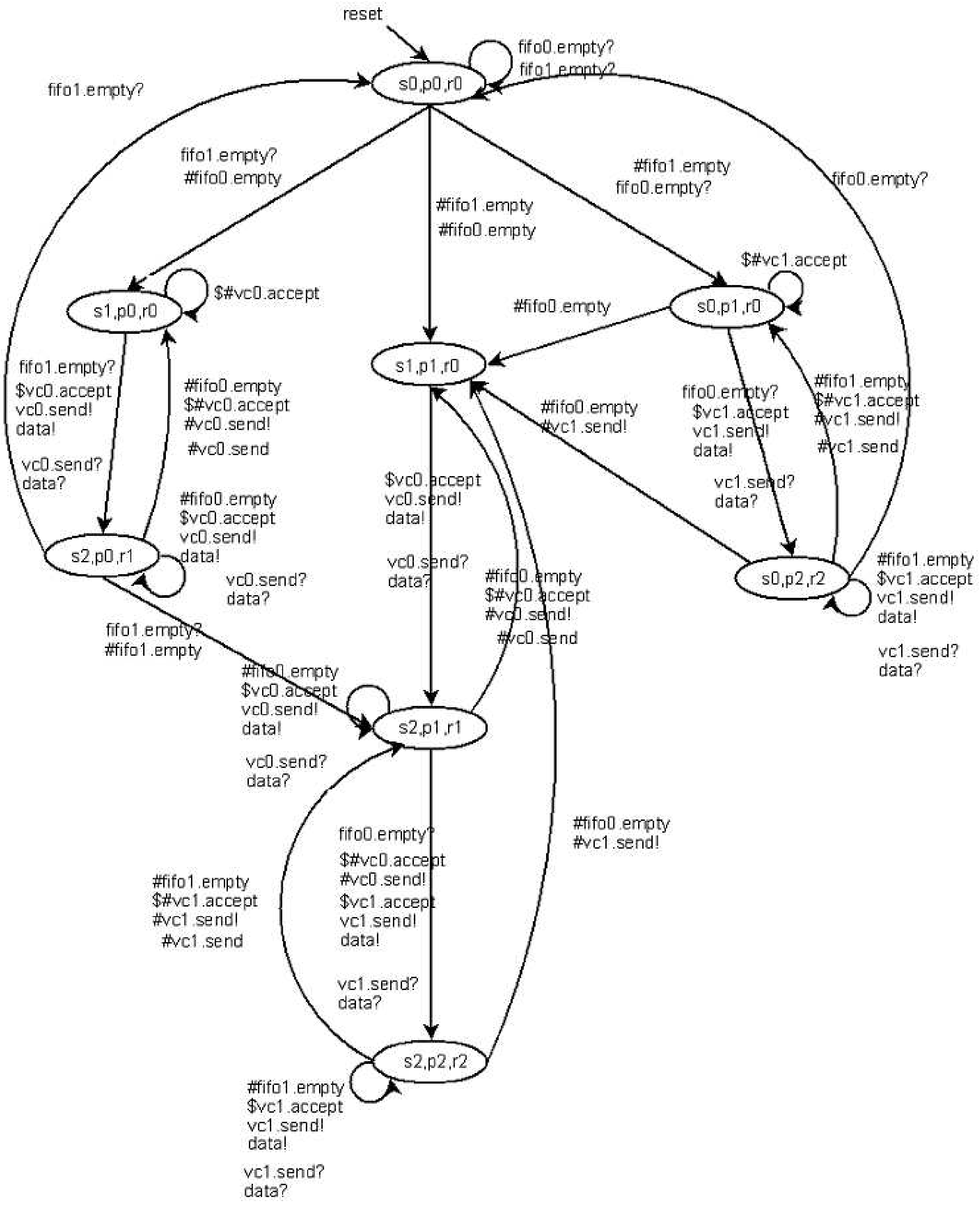
Absence of deadlock after overflow

```
FIFO does not go to deadlock with no further read/write possible after it
gets full. If FIFO capacity is reached it has to recover by having non-more
writes but just reads in the state,

define p (flen ==10)
define r (ren?[high])
LTL property: <>p -> r , always when FIFO is full the next event is ren.
```

```
never {    /* !(<> p -> r) */
T0_init:
if
:: (! ((r)) && (p)) -> goto accept_all
:: (! ((r))) -> goto T0_S3
fi;
T0_S3:
if
:: ((p)) -> goto accept_all
:: (1) -> goto T0_S3
fi;
accept_all:
skip
}

Result: This is verified to be true

State-vector 92 byte, depth reached 9999, errors: 0
  1423679 states, stored
  4041085 states, matched
  5464764 transitions (= stored+matched)
   147042 atomic steps
hash conflicts:   3334971 (resolved)

Stats on memory usage (in Megabytes):
  146.634 equivalent memory usage for states (stored*(State-vector + overhead))
  103.432 actual memory usage for states (compression: 70.54%)
          state-vector as stored = 60 byte + 16 byte overhead
    2.000 memory used for hash table (-w19)
    0.305 memory used for DFS stack (-m10000)
  105.626 total actual memory usage
```

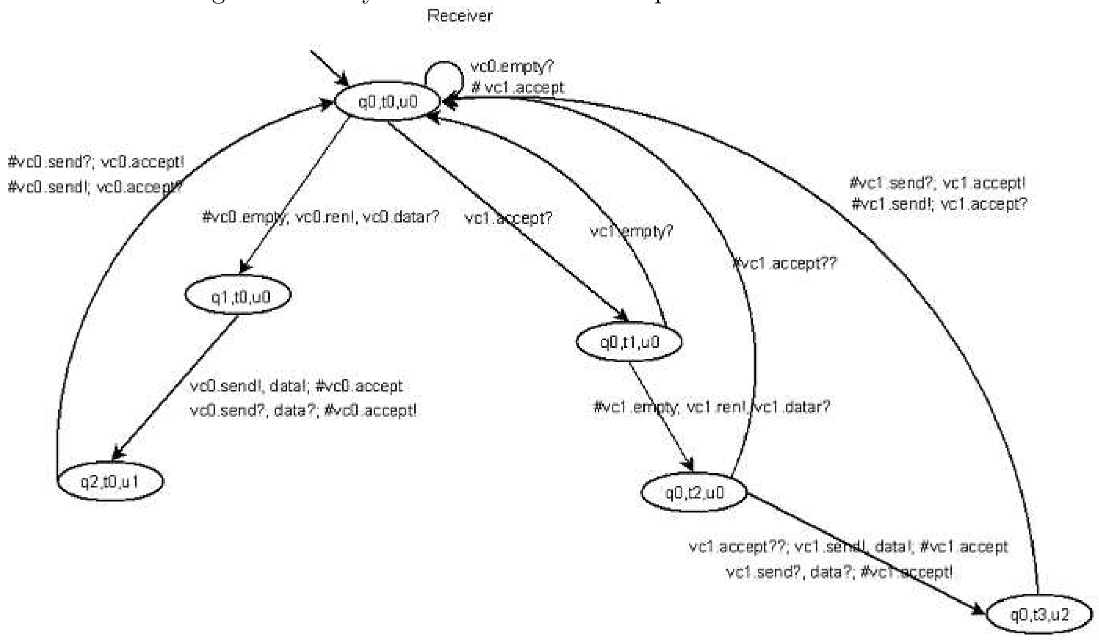
J Synchronous Parallel Composition

Figure J.1: Synchronous Parallel Composition



K Asynchronous Parallel Composition

Figure K.1: Asynchronous Parallel Composition



Receiver
vc0.empty?
# vc1.accept
q0,t0,u0
#vc0.send?; vc0.accept!
#vc0.send!; vc0.accept?
#vc0.empty; vc0.ren!, vc0.datar?
vc1.accept?
vc1.empty?
#vc1.accept??
#vc1.send?; vc1.accept!
#vc1.send!; vc1.accept?
q1,t0,u0
q0,t1,u0
vc0.send!, data!; #vc0.accept
vc0.send?, data?; #vc0.accept!
#vc1.empty, vc1.ren!, vc1.datar?
q2,t0,u1
q0,t2,u0
vc1.accept??; vc1.send!, data!; #vc1.accept
vc1.send?, data?; #vc1.accept!
q0,t3,u2