Token Ring Arbiters: An Exercise in Asynchronous Logic Design with Petri Nets

Kia Seng Low [*]and Alexandre Yakovlev [†]
University of Newcastle upon Tyne, NE1 7RU, UK

Abstract

Designing an asynchronous communication architecture in a VLSI system, we have the choice of either using a token ring or a bus. The token ring structure is often more reliable than the bus structure because of its point-to-point interconnection. In this paper, we study two alternative token ring arbitration protocols which we call *Busy Ring Protocol* (BRP) and *Lazy Ring Protocol* (LRP). Their performance evaluation shows that BRP allows better response time under higher request rates, while its major disadvantage is waste of activity, and hence power, if the request traffic is relatively low. We describe the design of speed-independent control circuits for these two ring protocols. The initial specification of the protocol made in a Petri net is refined to a Signal Transition Graph, which is further implemented into a logical circuit by recently developed methods. The logical circuit involves, as a standard component, a two-way mutual exclusion (ME) element. The arbiter designs have been verified at all levels, using different Petri net interpretations. The final check has been performed using Cadence HSPICE simulation tool. We conclude with the idea of a "hybrid" approach, combining the advantages of both BRP and LRP.

Index terms: arbiters, asynchronous circuits, design process, mutual exclusion, Petri nets, Signal Transition Graphs, system level design, token ring.

1 Introduction

As the demand for ever smaller devices and lower power consumption increases, asynchronous designs become more appealing [26]. Asynchronous designs do not use a global clock. This simplifies the overall chip routing and eliminates malfunctions that are caused by clock skew [1]. Without the global clock, an asynchronous chip is allowed to achieve near-zero standby power at idle state. Asynchronous digital systems have a potential of operating at the highest possible

[*]Work supported by Nuffield Foundation UndergraduateResearch Bursary 1994 AT/100/94/0299

[†]Work supported by EPSRC grant GR/J52327

speed based on the causal order of signal transitions rather than on the worst case delay that synchronous design uses.

Asynchronous designs are however more difficult to implement if specific constraints are imposed by component libraries and implementation technologies. Existing formal techniques and tools [27, 28], when applied to behavioural specifications of a broad class, still cannot produce automatically an implementation which is hazard-free for a given element library. Alternatively, if such an implementation was possible by using some syntax-direct translation techniques, it would most likely be inefficient in area and time. The designer should therefore be equipped with additional verification tools which could be used for checking if a particular logic gate decomposition does not introduce hazards [41, 37, 38]. Often, the designer may wish to compromise with total speed-independence [1] in order to optimise the overall design. Timing-based verification can be applied for that purpose [39]. Best results in the use of formal techniques and tools for synthesis and verification have recently been achieved for labelled Petri nets [20], Signal Transition Graphs (STGs) [5, 13, 14] and Change Diagrams [15], all these models being with close syntax and semantics. Existing software tools, such as SIS (from Berkeley, USA), ASSASSIN (from IMEC, Belgium) and FORCAGE (from St.Petersburg, Russia and Aizu, Japan), can relatively easily cope with moderate-size specifications of circuits without internal nondeterminism. Recent work [17] has demonstrated possibilities for extending their power to models of arbitration circuits, which have so far been designed manually at the logic or transistor level.

Enhancing design techniques in that latter way seems quite crucial for the overall success of asynchronous design approach. Indeed, many asynchronous designs which are deterministic can usually have a straightforward synchronous equivalent. The latter often beats the asynchronous design in performance and power, e.g. in the case of asynchronous versus synchronous adders [31, 30]. Where the asynchronous approach is fundamental and has no alternative are the applications in which a system is activated by means of a nondeterministic, really "asynchronous", signal source. System-level designs, communication channels and interface circuits are the applications which seem to be better candidates for asynchronous approach than computational structures. The best example for that is probably arbitration and resource allocation schemes and protocols. This paper thus focuses on the design of asynchronous circuits for distributed arbitration.

When designing an asynchronous interface, one has two alternatives of either using an open-ended bus or a ring structure. For our distributed arbitration protocols, we choose the ring structure that is more robust because of its *point-to-point* interconnections. The aim of our work has been to design and evaluate the asynchronous control circuits for two ring protocols that we

[1]According to the common classification, circuits whose behaviour is independent of gate delays (and wire delays) are called *speed-independent* (*delay-insensitive*) [43].

have named as Busy Ring Protocol and Lazy Ring Protocol. The results obtained in this work can be used in (re)designing an asynchronous token ring interface [32]. The latter has been the first design example of a totally speed-independent communication channel, which we hope will open up a series of successful applications for asynchronous design principle. The adjacent goal of the paper is to demonstrate the usefulness of the formal language of Petri nets and STGs in designing asynchronous circuits with internal nondeterminism. It is crucial that such circuits are produced correct by construction and their logical part is synthesized separately from the mutex elements. Another recent example of the use of this technique was developing a control circuit for pipeline interstage synchronisation in the Sproull Counterflow Pipeline Processor [33].

The structure of the paper is as follows. Section 2 is a background section. Here, we identify the objectives of our design, outline the design procedure and briefly review the Petri net and STG modelling approach. In Sections 3 and 4, we describe the behavioural specifications of the two protocols and demonstrate their performance evaluation results. In Section 5 and 6, we model these protocols, firstly at a relatively high level, by means of a labelled Petri net, and then at a more detailed level by an STG. In Section 7, we describe the logic synthesis of circuits from the STG refinement. This method has been applied with the aid of SIS. In Section 8, we describe circuit verification using the so-called Circuit Petri net, which is a special type of STGs, describing the logic level implementations. In Section 9, we show some of the simulation results of the control circuits using the HSPICE tool of Cadence. Finally, we have the Summary and Conclusion in Section 10, where we present an idea of a "hybrid" arbitration protocol, combining the advantages of the Busy and Lazy Ring versions.

Before passing to the next section, we need to clarify the meaning of the word "token", which is used quite often throught the text. There are two kinds of tokens. The *privilege token* is the token that permits the service to be granted if there is a request pending. A *net token* is the token that the places in a Petri net or STG hold. The latter thus means the presence of a system in a particular substate. Any mention of a token without prefix "privilege" refers to the Petri net token.

2 Background

2.1 Overview of Ring Structure

Figure 1 shows a distributed system with a ring architecture. In this structure, we assume that there is a finite number of User Subsystems, which are connected to their respective Ring Adaptors via the User Links. The Ring Adaptors are connected together to form the overall ring communication channel, through which User Subsystems can send packets of data. An asynchronous design for a ring channel of that type has been described in [32], where the channel's implementation was based on a speed-independent pipeline with delay-insensitive data path encoding.

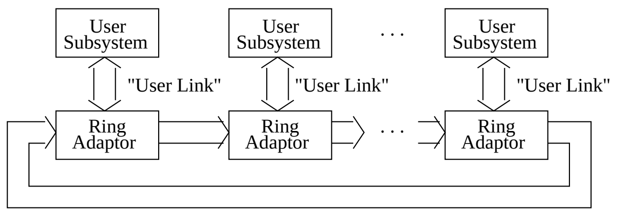


Figure 1: Ring Channel structure

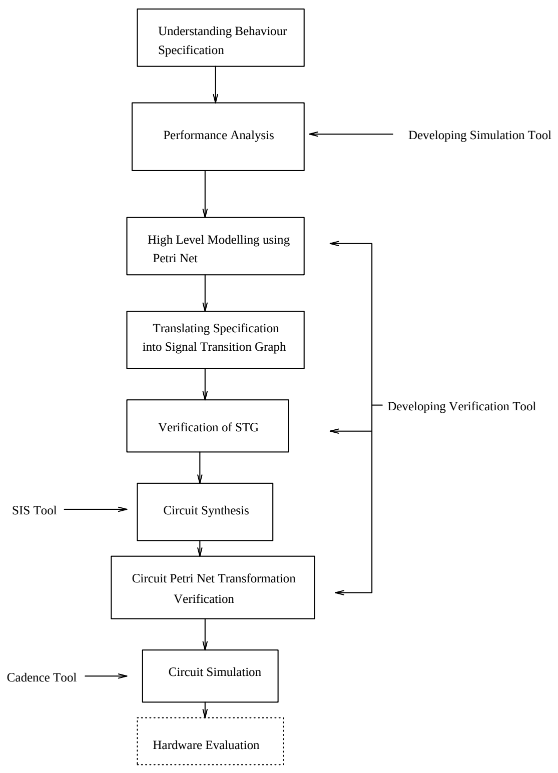
In order to transmit data to the pipeline channel, each User Subsystem makes independent requests for exclusive access to the channel's data path. At any one time, only one particular User Subsystem is allowed to transmit data. The User Subsystems are physically separate from each other; and they only communicate via their respective Ring Adaptors. When a User Subsystem wants to transmit data, it issues a request to its Ring Adaptor. When request is granted, the User Subsystem can start transmitting data (a specially structured packet [32]) to the channel. After a finite period of time, when the User Subsystem has finished its data transmission, it informs its Ring Adaptor about that.

In this paper, we are only interested in the asynchronous control circuits that implement the channel acquisition or arbitration protocol. We mainly consider two protocols for that: the Busy Ring and the Lazy Ring Protocol. Data path design, which refers to the design of how data is transmitted between the User Subsystems, may be based on two-phase micropipline structures [26] or on four-phase pipeline elements [32], and is not discussed here.

2.2 Design Procedure

The final goal in this design exercise is the CMOS implementation of the ring arbitration circuits. The main correctness requirement is that the circuits must be speed-independent at the logic level. That is, the behaviour of circuits must satisfy their specification and be free from hazards under any delays in the logic gates and interconnections between individual cells (adaptors) of the ring structure and between the ring and its users.

Figure 2 illustrates the major steps undertaken in the design process. Some of these steps have involved using existing CAD tools for circuit synthesis and simulation. They were mostly used at later stages of the design, when a correct and complete STG specification of the circuits has been available. The SIS tools were used for synthesis of the logic equations from STGs and obtaining corresponding gate netlists for the circuits. The CADENCE tools were used for



Understanding Behaviour Specification
Performance Analysis
Developing Simulation Tool
High Level Modelling using Petri Net
Translating Specification into Signal Transition Graph
Developing Verification Tool
Verification of STG
SIS Tool
Circuit Synthesis
Circuit Petri Net Transformation Verification
Cadence Tool
Circuit Simulation
Hardware Evaluation

Figure 2: The Design Process

converting the logic gate netlists to CMOS implementations and simulating them through the HSPICE simulator.

Additional software has been built to evaluate the performance of the Busy Ring and Lazy Ring Protocols and for the analysis and verification of Petri net models. Such models exist at the following three levels: (i) a labelled Petri net description of protocols, (ii) an STG circuit specification level and (iii) circuit Petri net level. We assume that at each of these levels the designer may introduce some changes, in the form of either behavioural refinement or structural decomposition. The former involves the signalling and handshake expansion of abstract labelled actions, changes between two-phase and four-phase signalling disciplines etc. The latter applies to boolean gate decomposition, adding mutual exclusion elements into the logic circuits obtained through SIS and so on. Additionally, the designer may apply some "last minute" re-orderings into the STG, especially when solving the Complete State Coding problem [2]. In the future we would like to add to these tools timing analysis software based on time Petri nets [35], to facilitate designs that are less conservative than purely speed-independent circuits.

[2]Unfortunately, none of the existing STG-based software tools, including SIS, is capable of dealing with a sufficiently wide class of STGs as yet; the tool usually needs an active role played by the designer, e.g., when adding encoding state signals. This is especially the case for the STG for arbitration circuits, which are not free-choice Petri nets.

2.3 Modelling with Petri nets and Signal Transition Graph

2.3.1 Petri nets

Petri nets are well-known for their use in describing concurrent systems of various types, including protocols and arbitration systems. This subsection is included for readers with little or no experience in Petri net concepts, who might also like to refer to a more comprehensive Petri net text, e.g. [25, 11].

A Petri net is a triple $N = \langle P, T, F\rangle$, where P is a set of *places*, T is a set of *transitions* (sometimes also called *events*) and F is the *flow relation.* A place $p \in P$ is a *predecessor* of a transition $t \in T$ (hence, t is a *successor* of p) if $(p,t) \in F$. Likewise, a transition $t \in T$ is a predecessor of a place $p \in P$ (p is a successor of t) if $(t,p) \in F$. In informal discussions, words "predecessor" and "successor" are often replaced by "input" and "output", respectively. A simple Petri net example is shown in Figure 3, where $p1, \ldots, p6$ are the places, and $t1, \ldots, t6$ are the transitions. E.g., $p3$ and $p4$ are the predecessors of transition $t3$; $p3$ is the successor of transition $t1$.

Petri nets form a number of structural subclasses by putting some restructions on the flow relation. A net which allows places to have at most one predecessor transition and at most one successor transition is called a *Marked Graph.* Similarly, if a transition is allowed to have at most one predecessor place and one successor place, the net is called a *Finite State Machine.* If in every subset of transitions which share some predecessor places all transitions have the *same* set of predecessors, the net is called an *Extended Free-Choice* net (it is called a free-choice net if every such shared set has cardinality 1).

A *marking* of a Petri net represents the marked places in the net. In Figure 3(a), we have marking $M = \{p1, p2\}$. A transition is *enabled* whenever all its predecessor places are marked with at least one token. Referring to Figure 3(a), transition $t1$ is enabled since $p1$ is marked with a token. Similar refers to $t2$ and its predecessor place $p2$. Any enabled transition can *fire* – when it fires, a token is removed from every predecessor place, and a token is added to every successor place. Transition $t3$ in Figure 3(b) is not enabled at this marking since not all its predecessor places have at least one token. When one of the enabled transitions, say $t1$, fires the marking of the net becomes $M' = \{p3, p2\}$. A marking M'' is *reachable* from another marking M' if there exists a sequence of enabled transition firings that produces M'' starting from M'. The following properties are crucial for verifying correctness of systems modelled by Petri nets.

A marking M' is *live* if for all markings M'' reachable from M', every transition can be enabled through some sequence of firings from M''. A Petri net is live if its initial marking is live. A marking M' is *bounded* (k-bounded) if the number of tokens that any place can hold after

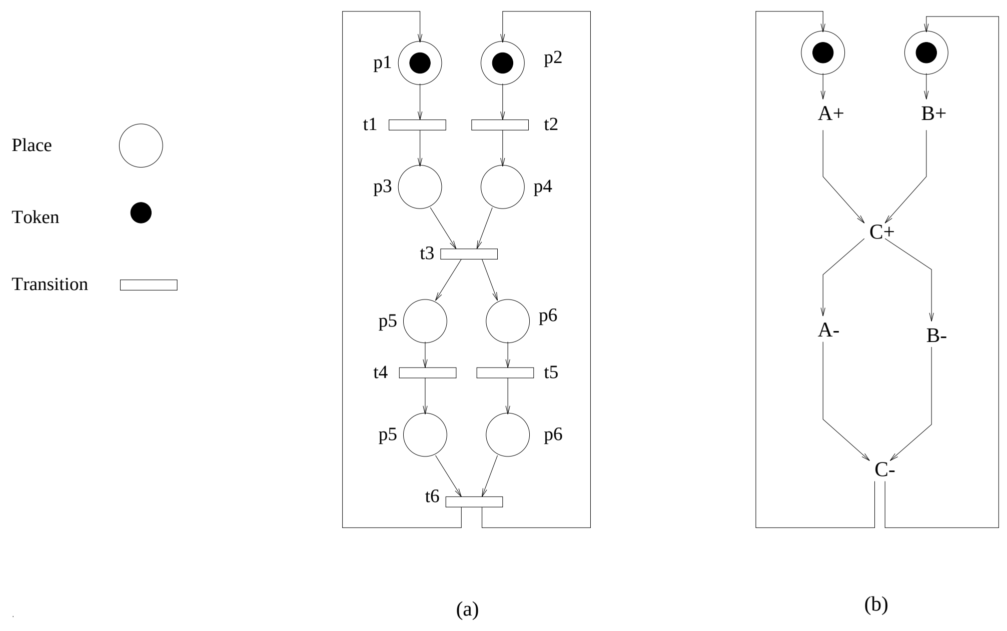


Figure 3: Petri net (a) and Signal Transition Graph (STG) (b)

any sequence of firings from M' is bounded (by k, which must be finite). A marking M' is *safe* if it is 1-bounded. A net is safe if all reachable markings in the net are safe.

A Petri net is *persistent* with respect to a transition t (and t is called *persistent transition*) if for all markings M' reachable from the initial marking M, such that t is enabled in M' along with some other transition t', t remains enabled in the marking M'' reached from M' by firing t'. Otherwise, t is said to be non-persistent. If the above-mentioned transition t' brings the net to M'' where t is not enabled, t is said to be *disabled by* t'. A Petri net is *persistent* if it is persistent with respect to all its transitions. When talking about STG-interpretation of a Petri net, an STG is called *output-persistent* if its underlying Petri net is persistent with respect to all transitions which are labelled with *output* signals.

2.3.2 Signal Transition Graph

A Signal Transition Graph (STG) is an *interpreted Petri net*[3] whose events are associated with *rising and falling edges* of binary signals or boolean variables. STGs were independently introduced in [5] and [13] (in the latter, the model was called Signal Petri Net and its subclass Signal Graph) for formal specification of asynchronous speed-independent circuits. An STG describes the causal ordering between signal transitions in a circuit, thus being a formal capture of the information often presented by timing diagrams. Figure 3,(b) shows an STG built on the net shown in Figure 3,(a) by means of signal transition labelling. The positive or negative sign on

[3]In general, an *interpreted Petri net* is a Petri net whose events are associated with certain system model interpretation. Another term often used for interpreted Petri nets is *labelled Petri nets*.

the right of the signal name indicates the rising or falling signal transition respectively. Note that the places with a single predecessor and successor transitions are usually omitted in an STG for clarity. A place between two events in a Petri net is implied in the arc connecting two signal transitions in STG. The token marking is therefore transferred from places to the corresponding arcs.

Being an interpreted Petri net, STG inherits all properties of the underlying Petri net, e.g. structural properties, such as marked graph and free-choice, and behavioural properties, such as (freedom from) deadlocks, boundedness, safeness and liveness. However, some of the properties (they are explained in Section 7), e.g. consistency and the Complete State Coding property are entirely dependent on the STG interpretation.

3 Behavioural Description of Protocols

The overall idea of a ring access protocol is similar to what is conventionally known as a Token Ring Protocol in Local Area Network terminology [8]. Symbolically, one could imagine that there is a *single* token in the ring channel. We call this token as a *privilege token*. It is only when some Ring Adaptor has this privilege token it is allowed to grant permission to its User Subsystem to start transmitting data to the channel data pipe. In the Busy Ring Protocol, the privilege token goes round the ring continuously except when it is held to allow service of user requests, i.e., the privilege token moves from one Ring Adaptor to another in a cyclic sequential fashion. In the case of the Lazy Ring Protocol, the privilege token remains stationary at a particular Ring Adaptor which it has last served until such time that it is summoned to serve some other Ring Adaptor. At all times, only the particular User Subsystem which has the privilege token, is allowed to transmit data.

The main difference between the two protocols is that the privilege token is stationary in the Lazy Ring Protocol if there is no job to be served. On the other hand, in the Busy Ring Protocol, the privilege token does not stop polling each station even if there are no jobs in the system.[4] In the following subsections, we detail the behaviour of both protocols.

3.1 Busy Ring Protocol

In the Busy Ring Protocol, the privilege token in the ring moves from one Ring Adaptor to another in a clockwise fashion. When a User Subsystem wants to transmit data, it sends a request to the ring adaptor. If the privilege token is not at the Ring Adaptor, the request is put on hold. The Ring Adaptor waits for the privilege token to arrive before it grants permission

[4] *There is no job in the system* refers to the situation when none of the User Subsystem wants to transmit data.

to its User Subsystem. A request at any Ring Adaptor is served only if it comes before the privilege token arrival. After the permission is granted to the User Subsystem to transmit data, the privilege token remains at that Ring Adaptor until it is told by the User Subsystem that it is no longer required. The privilege token moves on to the next ring adaptor, and so forth.

3.2 Lazy Ring Protocol

The Lazy Ring Protocol has been inspired by the *Distributed Mutual Exclusion* circuit designed by Alain Martin[24], using CSP. However, we further use the STG to model its behaviour, and based on the STG, we synthesize the specification into a circuit. In the Lazy Ring Protocol, there is one Ring Adaptor that holds the privilege token at any one time. Only the Ring Adaptor that has the privilege token may grant the permission to its User Subsystem to transmit data. This ensures the mutual exclusion on the data transmission, i.e. no more than one User Subsystem is allowed to transmit data at a time. If the privilege token is not at the Ring Adaptor when its User Subsystem requests to transmit data, the Ring Adaptor transmits a request to the next Ring Adaptor on its right. The request circulates to the right (clockwise) until it reaches a Ring Adaptor which keeps the privilege token. The privilege token then starts moving to the left (anti-clockwise) until it reaches the Ring Adaptor whose User Subsystem has generated the request. This Ring Adaptor then grants its User Subsystem permission to transmit data.

Any Ring Adaptor that has decided to relay the request message from its left neighbour to the right neighbour is not allowed to transmit its own request for the privilege token if its User Subsystem has also requested to transmit data. It is allowed to send its own request for the privilege token only after the privilege token has passed by.

As the request for the privilege token is going from one Ring Adaptor to another, searching for the privilege token, it may encounter a Ring Adaptor which has also transmitted the request for the privilege token. If this is the situation, the former's request is put on hold until the latter has received the privilege token and has been served; and then the privilege token again moves to the the left. During the time when there is no request generated from any of the User Subsystems, the privilege token remains stationary at the Ring Adaptor it has last served.

Intuitively, both protocols fulfil their main mission, namely to select a *unique* User Subsystem which can become the owner of the privilege token. Furthermore, both protocols do it fairly, that is, no User Subsystem is left to "starve" infinitely without the privilege token provided that any User Subsystem releases the privilege token in a finite period of time and does not capture it again until after some reasonable time delay[5]. Petri net modelling and verification allow the

[5]This delay should be equal to the maximum delay of the response of a two-way mutual exclusion element, that is just a few nanoseconds in its worst case. This seems like a reasonable assumption for any physical design.

designer to formally check that this intuitive expectation is valid. What remains less clear which (if any) of these two protocols is better in terms of performance (say, in terms of the average response time).

4 Performance and Power Estimation

Before we start discussing how to design and implement the control circuits, we would like to look into the performance evaluation of the two protocols. Recently, there has been a growing interest in the Petri net community in the use of Stochastic Petri nets[42] to study average case performance of system models. We looked at a particular tool called *UltraSan*[44, 45]. However, as any variant of a Petri net has an inherent limitation due to potential state space explosion, *UltraSan* can only be used with a limited fixed-size queues of requests. Since our model of a system is partial and we have no complete models of the User Subsystems, their effects are better represented by possibly infinite queues. Hence, the decision to use *UltraSan* has been deferred.

There are two approaches that we could take to study the performance of the protocols. They are the analytical method and simulation. Doing a truly analytical performance study of the two protocols is outside the scope of this work. An event-driven simulation program has been written (in C++) to study the performance of both protocols. Further details about the program and comparison with some analysis results can be found in [10].

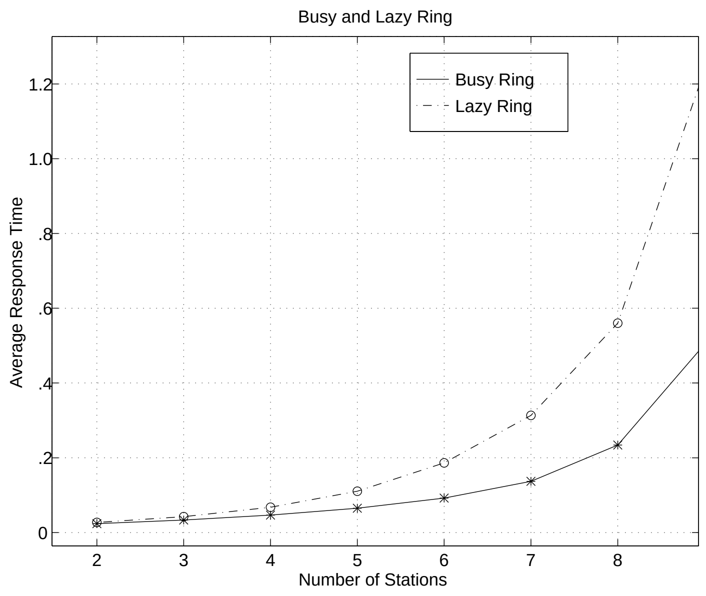


Figure 4: Plot of Average Response Time against Number of Stations

In this simulation, the term "Station" is applied to any Ring Adaptor. The main parameters

Normally, the delay between two adjacent requests coming from the same User Subsystem would be at least one order of magnitude large since it will involve operating through a cycle of preparing a packet of data for sending.

chosen for simulation are as follows:

- the mean request (of the privilege token) interarrival time is 1 unit;
- the mean service (holding the privilege token) time is 0.1 units;
- the mean switch (between two adjacent stations) time is 0.01 units;
- the total simulation time bound is 100,000 units (this number is scaled down by the number of stations to determine the number of iterations per station).

We considered that the ratio of 10 between service and request rates would realistically reflect the situation in the channel. The results of the performance simulation are shown in Figure 4. They indicate to us that the Busy Ring Protocol has a faster response time compared to that of the Lazy Ring Protocol. Both curves have an exponential rise in their average response time against the number of stations in the ring.

The average number of signal transitions per unit of time has been commonly accepted as a determining factor of any estimation of power[29, 30] consumption in circuits. In Figure 5, we have plotted the estimated number of signal transitions against the number of stations in the ring. Only "crucial" signal transitions have been taken into account, namely those that take place on interconnections connecting each Ring Adaptor with the remaining units. We have assumed four signal transitions per privilege token switch from one ring adaptor to another. However, this is only a rough estimation for a four-phase (Return-to-Zero) handshaking. Thus, we have not taken into account the effect of possible difference in the number of transitions internal for the circuits of the Ring Adaptors. This could be done after the design has been complete. However, the difference in the activity between protocols can be already clearly seen at this high abstraction level.

The graph in Figure 5 has indicated to us the far greater amount of activity occurring in the Busy Ring Protocol compared to that of the Lazy Ring Protocol. The amount of activity in Lazy Ring Protocol is relatively constant. However, it is interesting to note that the amount of activity reduces in the Busy Ring Protocol as the number of Ring Adaptors in the ring structure increases. This may be explained by the fact that the privilege token slows down as the number of service requests queued increases.

5 Petri Net Modelling of Protocols

In this section, we discuss the Petri net modelling of the Busy Ring and Lazy Ring Protocol. First, we describe the use of the high level modeling using Petri nets to model the protocols in

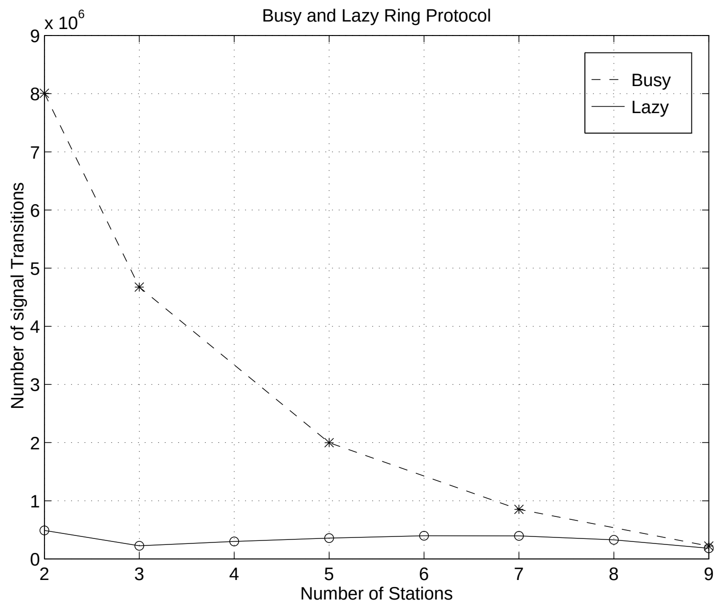


Figure 5: Plot of Average Number of Signal Transitions against Number of Stations

Section 5.1. Then, in Section 6.2, we relate the framework set by the high level modeling using Petri nets to describe the protocols in Signal Transition Graph.

5.1 High level modeling using Petri net

In this section, we discuss the use of high level modeling using Petri nets to model the behavioural specification of the two protocols. The purpose of the high level modeling using Petri nets is to maximally explore possible events that can happen in the protocol execution. This is needed for better understanding of the protocols and their verification, e.g. checking their deadlock-freedom.

5.2 Busy Ring Protocol

Figure 6 shows a labelled Petri net for the Busy Ring Protocol, depicting only a generic part of it that models a single Ring Adaptor. The example marking indicates that the privilege token is present at the Ring Adaptor but that no request is pending. The dotted lines refer to the external environment.

The places *TOKENARRIVE* and *TOKENDEPART* are complementary, that is when one is marked with a token the other is empty. When the privilege token arrives at the Ring Adaptor, there is a token at the place *TOKENARRIVE*. When the privilege token leaves the Ring Adaptor, the token at the place *TOKENARRIVE* is removed and is put in the place *TOKENDEPART*.

Referring to Figure 6, when a request comes into the Ring Adaptor, the tokens in the places *NEXT* and *ME* are removed and a token is inserted at the place *WAIT* after the transition *REQUEST* has fired. The transition *SERVICE* does not fire if no token is available at the place

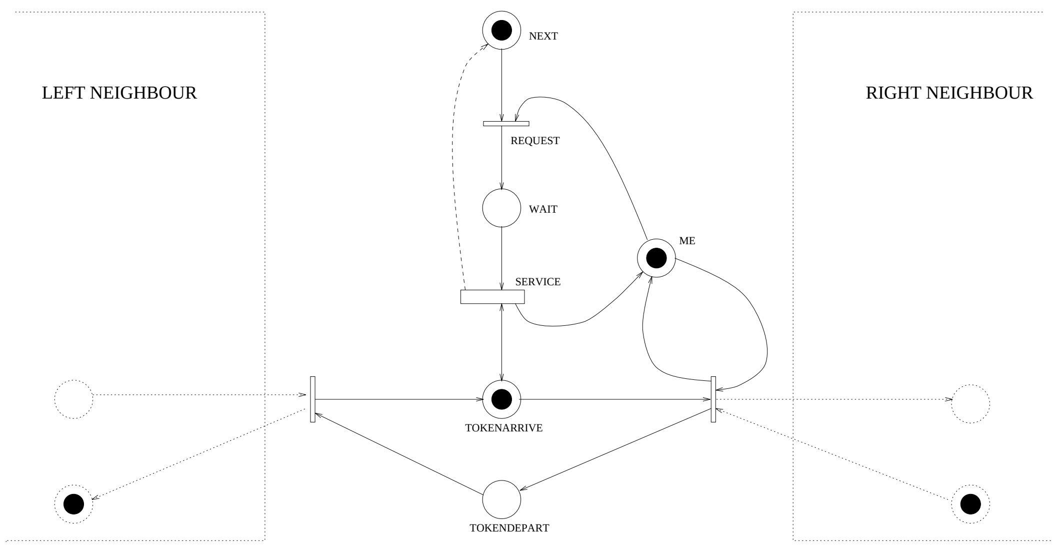


Figure 6: High level modeling of Busy Ring Protocol in Petri net

TOKENARRIVE.

When the privilege token arrives at the Ring Adaptor, the token is removed from the place *TOKENDEPART* and a token is inserted into the place *TOKENARRIVE.* The privilege token remains at this Ring Adaptor as long as the place *ME* remains empty. The privilege token is not allowed to move on until the service has been done.

5.3 Lazy Ring Protocol

Describing the Lazy Ring Protocol with a labelled Petri net is a bit more complicated. Figure 7 also shows a net model for a single Ring Adaptor. The dotted lines again refer to the external environment.

The places *TOKENFULL* and *TOKENEMPTY* are complementary. When the privilege token arrives at the Ring Adaptor, there is a token at the place *TOKENFULL.* When the privilege token leaves the Ring Adaptor, token at the place *TOKENFULL* is removed and is placed in the place *TOKENEMPTY.*

The place *PROBE* is used to probe if the privilege token is at the Ring Adaptor. If not, a request is sent "clockwise" by passing the token from the place *PROBE* to the subnet corresponding to the Ring Adaptor on the right.

The place *LEFTWON* is used to indicate that the Ring Adaptor is being used to relay the request for the privilege token. The token at the place *LEFTWON* is removed as the privilege token moves from one Ring Adaptor to another Ring Adaptor, looking for the one which has requested it. Once the privilege token sees no token at the place *LEFTWON* of one of the Ring Adaptors, it knows that that this Ring Adaptor is the one which sent for it as it could not move

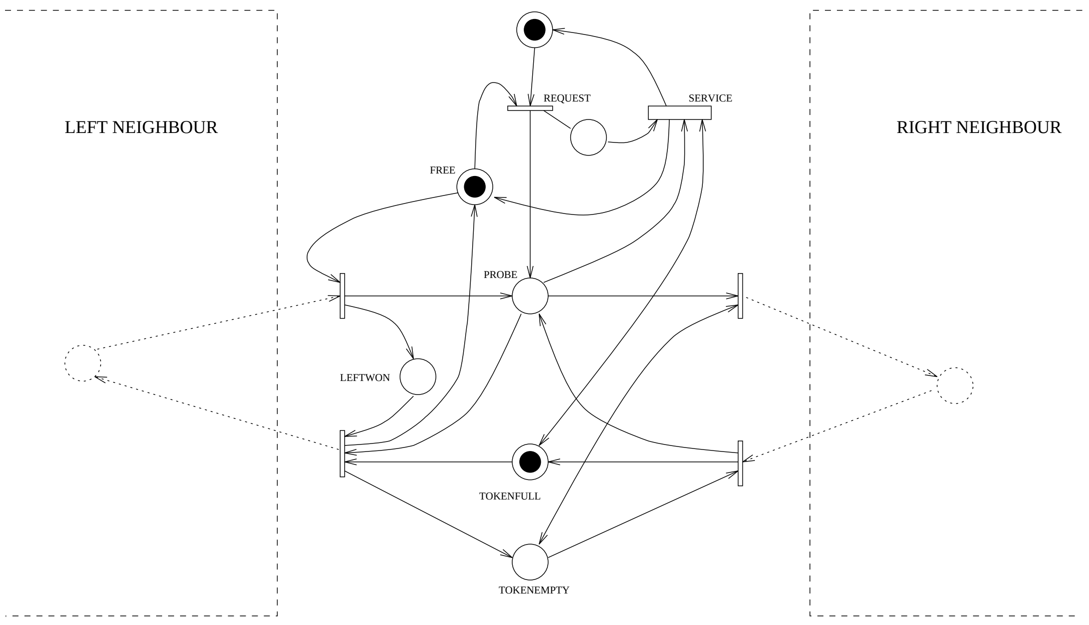


Figure 7: High level modeling of Lazy Ring Protocol in Petri net

any further to the left.

The above high-level descriptions have been formally verified using our Petri net verification tool PN_SOFT [10]. This tool operates with reachability graph analysis, checking the net for deadlocks, liveness, safeness and persistency. For these relatively concise models (real advantage of the high-level modelling approach), the reader could try, with a fairly moderate effort, to compose those Petri nets for a ring of three Ring Adapters and build their reachability graphs by hand. For more detailed models, one certainly needs an automatic tool to generate and verify the nets. In many cases, even a tool which is based on the construction of the full reachability graph cannot help due to the exponential complexity of the graph. Other techniques such as the reduced reachability graph (the so called *stubborn sets* technique) [34] or the net undolding [36] can be applied. The latter operates with the partial order representation of the net behaviour and can be more efficient for checking some properties, such as safeness and persistency.

Having performed the high level modelling and verification of the protocol for the entire system, we can now proceed to the refinement of the generic part of the system, namely its Ring Adaptor model. The refinement should take into account a specific structural organisation of both main interfaces of the Ring Adaptor. One is with its the User Subsystem, and the other is with its adjacent Ring Adaptors.

6 Ring Adaptor Model Refinement

In this section we refine our high level descriptions in terms of the events associated with signals involved in the protocols of the Ring Adaptor interfaces. Let us first consider those interface options that can be found amongst the common types of self-timed signalling disciplines.

6.1 Interface signalling options

There are two main self-timed signalling methods, *two-phase*, or Non-Return-to-Zero (NRZ), and *four-phase*, or Return-to-Zero (RZ) [12]. In the NRZ method, both rising and falling edges of a signal waveform are equally significant for the logic of interaction. Sometimes this signalling is also called transition signalling. Such type of signalling has been used in Sutherland's micropipeline devices [26]. In the RZ method, only one (say, rising [6]) of the edges is significant, the other is used purely for resetting the signal to its initial state.

Combined with a particular wiring convention, such as single wire (without an acknowledgement), two wire handshake (request and acknowledgement) etc., those methods form a basis for a specific intermodular interface protocol. The choice between the use of the RZ and NRZ method depends on a number of factors, such as the transmission line delays and the size and delays of the implementation logic. It is often the case that the time (and power) saving expected with the NRZ method does not in fact take place due to the additional delays introduced by the conversion logic if the internal devices (such as latches or mutex elements) operate in the NRZ mode. Generally, the transition signalling requires more complex elements than the RZ method, which normally uses conventional logic gates.

6.1.1 Interface with User Subsystem

Let us first look at the interface between the Ring Adaptor and its User Subsystem. The main three actions taking place at this interface in both protocols are: *Request* of the privilege token (produced by the User Subsystem), *Grant* of the privilege token (produced by the Ring Adaptor), and *Release* of the privilege token (produced by the User Subsystem). Depending on whether NRZ or RZ signalling method is used, we either have a 3-wire or 2-wire handshake at this interface. For the NRZ case, each action is realised as a transition on its own dedicated wire. For the RZ method, the rising (falling) edge of the request part of the handshake can be used for the *Request* (*Release*) action, and the rising edge of the acknowledge signal for the *Grant* action. The falling edge of the acknowledge signal does not have its specific significance and is simply

[6]However, usually microcomputer buses utilise the so-called "low active" logic, in which case the signal assertion is manifested by the falling edge of a waveform.

used for confirmation of the acknowledgement of the *Release* action and ordering with the next activation of the request signal.

6.1.2 Interface between adjacent Ring Adaptors

This interface also depends on whether we want to use RZ or NRZ method to propagate privilege tokens, in the Busy Ring Protocol, or both tokens and requests for them as in the Lazy Ring Protocol. In the Busy Ring case, the propagation of the token from one Adaptor to its neighbour can be manifested in one of the two ways:

1. The *NRZ/Propagate* protocol, where a single wire (labelled as *Token-in* at the receiving end and as *Token-out* at the sending end of the Ring Adaptor) is used and each signal transition on this wire is significant.

2. The *RZ/Propagate* protocol, also using one wire but only one edge of the signal is significant; the system must thus execute a releasing phase around the ring to reset the privilege token front.

In the Lazy Ring case, we can have two options:

1. The *NRZ/Handshake* protocol, where the two-wire two-phase handshake (labelled as *Lr, La* at the receiving end and *Rr, Ra* at the sending end of the Ring Adaptor) is used in such a way that a transition on the request signal manifests the passing of the request from the Ring Adaptor to its right neighbour, and a transition on the acknowledgement part means the passing of the privilege token travelling in the opposite direction.

2. The *RZ/Handshake* protocol, with two-wire four-phase handshake, where the rising edges of the request and acknowledgement signals are significant; they stand for the request and privilege token propagation action; and the falling edges of these signals are for resetting the wavefronts.

All examples of asynchronous ring arbiters known to us from the literature fall into the above classification. For example, M.Kishinevsky and V.Varshavsky [41] implement the Busy Ring and use RZ/Propagate protocol between Ring Adaptors and RZ method to connect the User Subsystem. A.J. Martin's DME [24], which implements the Lazy Ring Protocol, uses RZ/Handshake method at both interfaces. J.Ebergen et al [6], in their Busy Ring version, use NRZ/Propagate between Ring Adaptors and NRZ to connect with the User Subsystem (in fact, they have a five-wire interface here because they implement the so called "arbiter with reject"). Finally, G. Gopalakrishnan [7] uses NRZ/Propagate between Ring Adaptors and RZ for the User Subsystem interface.

6.2 Signal transition graph modelling

In this section, we describe the Signal Transition Graph (STG) models for some of the lower level refinements of the Busy Ring and Lazy Ring Protocol.

6.2.1 Busy Ring models

We shall start with a fairly simple case shown in Figure 8,a. This is a Busy Ring which has an RZ/Propagate protocol between Ring Adaptors and RZ method for the User Subsystem interface. It is clear from the STG that the active token propagation phase is between transitions *Tin+* and *Tout+*. The actual request polling takes place here. The other two transitions, *Tin-* and *Tout-* are used to reset the ring connections to their original low state. We should be aware of the presence of a special component in the ring which first generates the rising edge of the *Tin* signal at one of the adaptors. This is easily implemented at the circuit level by an invertor. Alternatively, we might reflect this in the STG for one of the Ring Adaptors, by swapping the *Tout+* and *Tout-* labels on the corresponding STG actions.

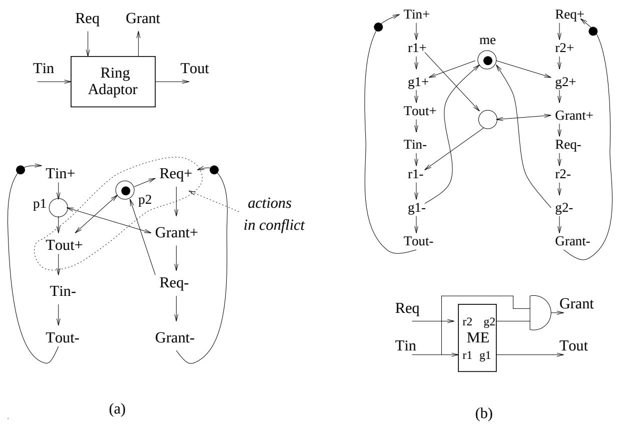


Figure 8: Signal Transition Graph model for Busy Ring Adaptor with RZ signalling (a) and its implementation (b)

Another important detail to be drawn from this specification refers to the presence of a conflict between transitions *Tout+* and *Req+*, which manifests actual local arbitration condition between the arriving privilege token and the user request. The *Tout+* transition (output for the circuit) is enabled and can fire unless the token in its right input place has been removed by transition *Req+* (input for the circuit). Since this conflict involves an output signal, which has to be implemented in the circuit, it must be resolved by a special transformation of the STG before this STG can be used for the logic synthesis by tools like SIS. We shall explain this transformation in Section 7.

Another STG model for the Busy Ring Adaptor is shown in Figure 9. This model corresponds to the use of NRZ methods at both interfaces with a single wire connection for privilege token transmission and three wire link with the User Subsystem. Note that in the STG for this model we have used a notational extension applied to STG modelling the transition signalling. It is called the *toggle* transition labelling, denoted by a tilde placed near the signal name. Basically, this is just a shorthand form for an STG with single wire signal transitions where the direction of the signal edge is unimportant. The software tools like SIS accept such toggle form and "untoggle" it to the normal STG format (with + and − transitions) before performing logic synthesis.

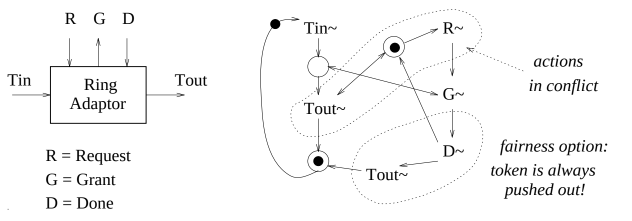


Figure 9: Signal Transition Graph model for Busy Ring Adaptor with NRZ signalling

The Busy Ring Adaptor model shown in Figure 10 combines the NRZ method at the User Subsystem link and the RZ method for privilege token transmission. Here, we do not use the toggle form for *Tin* and *Tout* to avoid cluttering of notations.

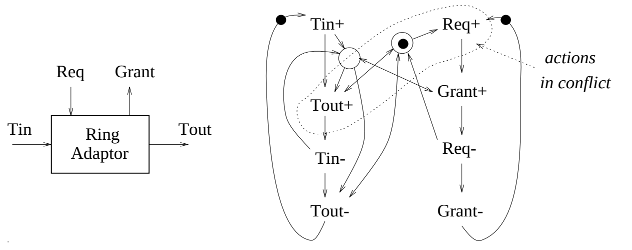


Figure 10: Signal Transition Graph model for Busy Ring Adaptor with combined RZ and NRZ signalling

It is easy to observe the behavioural correspondence between all three STGs and their original high-level prototype, the Petri net shown in Figure 6. The only difference between them is in the way their refine the original model, 0according to the chosen local interface methods.

6.2.2 Lazy Ring models

Modelling the Lazy Ring Adaptor produces slightly more complex Signal Transition Graphs. This is mainly concerned with the fact that we need at least two wires in the handshake between adjacent Ring Adaptors in the ring.

Our first model uses the NRZ method at both interfaces of a Ring Adaptor, with the User Subsystem and with the adjacent Adaptor. It is shown in Figure 11. Note that this STG, depicted in the toggle format (see the previous section), differs very little from the high-level model shown in Figure 7. It basically "authorises" protocol events in Figure 7 to become signal transitions. Note that two actions are labelled *dum*. These are dummy transitions – they do not carry any signal transition interpretation. Changing the marking of the underlying Petri net, their firing does not change the binary state (signal state vector) of the system. Dummy transitions are often used in STGs to avoid unnecessary cluttering in graphics. As we shall see later, discussing circuit synthesis, these transitions can help in introducing the internal signal transitions of a mutex element.

In the second model, shown in Figure 12, the RZ method is used at both interfaces. Note that we have explicitly represented the state of the flag "privilege token is full" by signal t. The reader may easily trace the STG to find out that all four main behavioural cases, discussed earlier, are adequately represented here. Indeed, let for example the privilege token be absent ($t = 0$) and request from the User Subsystem win the local arbitration. The following sequence of transitions will take place: *R+* $\Rightarrow$ *Rr+* $\Rightarrow$*Ra+* $\Rightarrow$ *t+* $\Rightarrow$ *Rr-* $\Rightarrow$ *Ra-* $\Rightarrow$ *G+* $\Rightarrow$ *R-* $\Rightarrow$ *G-*. This sequence however hides the fact that the releasing phase of the handshake (*Rr, Ra*), i.e. the subsequence *Rr-* $\Rightarrow$ *Ra-* is actually done in parallel with the subsequence *G+* $\Rightarrow$ *R-*; these two subsequences being synchronised on event *G-*.

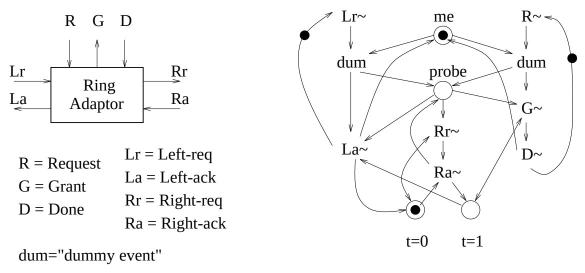


Figure 11: Signal Transition Graph model for Lazy Ring Adaptor with NRZ signalling

The freedom with which we can put the releasing phases of handshakes in parallel with other actions reflects the fact that these events are not signficant for the specification. The major requirement which must be inherited from the original specification is the order of events that propagate the privilege token and produce the grant signal high. The other constraint placed on such reordering (cf., "reshuffling" in [24]) of the release phase is the consistency of the order of the four-phase handshake signalling, i.e. making sure for example that sequencing *Rr+* $\Rightarrow$*Ra+* $\Rightarrow$ *Rr-* $\Rightarrow$ *Ra-* is preserved in the STG. Release phase reordering is an important source of optimisation

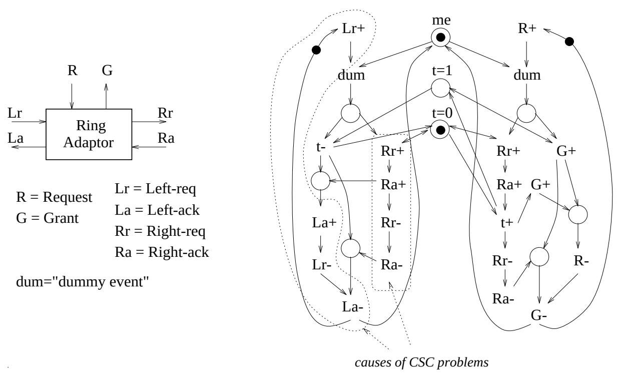


Figure 12: Signal Transition Graph model for Lazy Ring Adaptor with RZ signalling

in the design, which can be efficiently if logic synthesis tools are engaged in this procedure. However surprising, adding more parallelism between the release phases does not necessarily means better performance. An STG with more concurrency typically results in a more complex logic, with extra latches in the critical path. Trading off their delays against the handshake wire delays is another optimisation task. These issues of refining Petri net specifications into STGs are intimately related to the task of logic synthesis, which is discussed in the next section.

7 Synthesis of circuits from Signal Transition Graphs

In this section, we first briefly discuss the major aspects of circuit synthesis from STGs and then show some logic implementations for our Busy and Lazy Ring Adaptors. The method is based on the technique recently formalised in a number of publications, e.g. [13, 3, 5, 14].

7.1 Factoring out conflicts in STGs

Both synthesis methods are effective if the STG is presented in such a form that all conflicts (if any) between transitions involving output signals are “protected” by special mutex elements. Therefore before we proceed with these methods we need to focus our attention on the transformations that must be applied to STGs with conflicts. All our STGs built in the previous section fall into such a class. Their reachability graphs exhibit conflicts in the form of disabling of some transitions by others. The property of an STG in which none of the transitions of non-input signals is disabled (the underlying Petri net is persistent with respect to such transitions; see Appendix 2.3.1) is called *output-persistency*. For instance, the STG shown in Figure 8,a is output-nonpersistent with respect to transition *Tout+*. This transition can be disabled in the marking (*p1,p2,* ⟨ *Grant-,Req+* ⟩) by the firing of *Req+*, which removes token from place *p2*. An STG with such a problem cannot be synthesised through SIS, as its logic implementation would be hazardous. The method developed in [17] proposes to treat such signals separately, by "factoring them out" of the specification and associating them with a standard arbitration component, a two-way *mutex (ME) element*.

The overall procedure [17] for implementing STGs with output non-persistency can be summarised as follows:

1. determine a set of non-input signals whose transitions make the STG non-persistent;
2. insert *semaphore* actions, making semantic-preserving transformation at the STG level;
3. associate each semaphore with an ME element (if semaphores are multi-way, use decomposition of multi-way mutex components to 2-way ME's, e.g. using a cascaded structure);
4. factor the semaphore implementations (the "mutex part") from the circuit, making their outputs be additional inputs to the "logical part" of the circuit, which should now be output-persistent;
5. synthesise the "logical part" by the existing methods and tools (e.g., SIS);
6. interconnect the "mutex part" with the "logical part" through the signals of ME's.

The semaphore actions used in the above procedure effectively mean what is shown in Figure 13. These actions are refined in terms of input-ouput signal transitions of ME components. The STG description and possible implementation of two-way ME components is shown in Figure 14.

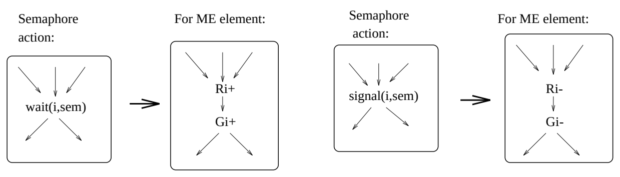


Figure 13: Illustration of semaphore actions

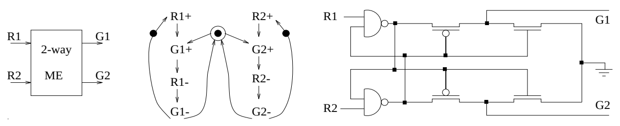


Figure 14: A two-way ME component

7.2 Logic Synthesis from Signal Transition Graphs

An STG with a standard signal transition $(x+, x-)$ labelling of its underlying Petri net was defined earlier in Section 2.3. This model gives rise to the state graph based on the reachability graph of the underlying Petri net. The circuit synthesis from an STG is based on deriving logic equations from its state graph. This process, called logic implementation of an STG, is summarised for in Figure 15. The fact that an STG with bounded underlying Petri net can be implemented into a logic circuit hinges on the following two important properties of the STG and its state graph [13]:

1. the transitions labelled with the same signal interleave in their signs (+ and −) for any firing sequence (this property is also called *STG consistency*), and
2. all the states that are labelled with the same vector have the same set of enabled non-input signal transitions; this is sometimes called *Complete State Coding* (CSC).

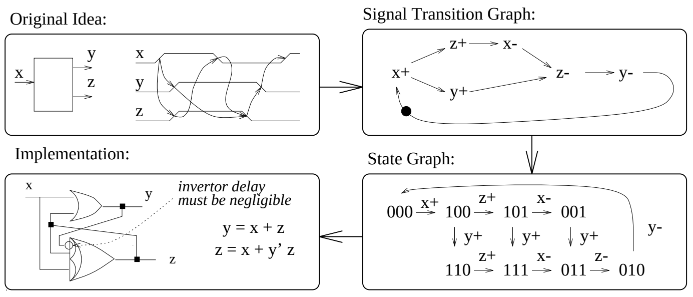


Figure 15: STG implementation process

The first condition guarantees that a boolean vector consistent with the firing can be assigned to each reachable state (marking). The second condition ensures that each non-input signal can be implemented with a combinational logical circuit computing its *implied value*. The implied value of a signal in a marking is the same as the value in the marking label if no transition for that signal is enabled in the marking; the complement otherwise. For example, the implied value of output z in the initial state 000 of the STG shown in Figure 15 is 0, because no transition of z is enabled in that state. Its implied value in the state labelled 100 is 1, the complement of the value of z in the label, because transition $z+$ is enabled. Intuitively, the implied value is the value each non-input signal "tends to" in each state, according to the enabled transitions. The logic equations for non-input signals are derived from the truth tables built for the implied values on all boolean vectors domain. Existing tools like SIS perform logic minimisation during such derivation. SIS also verifies for consistency and CSC property but does it on the basis of reachability analysis. Recently, more efficient techniques for STG implementability have been reported [40]. They are based on symbolic (using binary decision diagrams for boolean representations) traversal of its reachability set.

For example, the STG description which was obtained in Figure 12 satisfies only the consistency condition. It has the CSC problem which can be solved in one of two possible ways. One way, which can be slightly aggressive, is to change the ordering in the STG, for example, by reducing concurrency or reshuffling the insignificant (e.g., releasing) signal transitions. The other way, which can actually be combined with the first one is to add new internal state signals. Existing tools like SIS have some facilities to resolve the CSC problem but they are quite limited, so in many cases the tool actually resorts to the help of the designer. Note that the CSC condition also implies that the STG must be persistent with respect to non-input signals for which we want to derive boolean equations. The persistency can be ensured by the technique of introducing special-purpose mutex signals, described earlier.

We are now ready to apply the above synthesis techniques to our STG models of the Busy and Lazy Ring Adaptors.

7.3 Deriving Circuits for Ring Adaptors

7.3.1 Busy Ring Circuit Solutions

We now look at the circuit solutions obtainable for the Busy Ring Adaptor models. We synthesise logic circuit for the first model shown in Figure 8,a. The first step is to insert semaphore (2-way mutex) actions to protect output signals, *Tout* and *Grant*. Note that, although the participants of the conflict are transitions of *Tout* and *Req*, the latter is input signal and cannot be preconditioned by the semaphore. The role of place $p1$ in the original STG is played by the place which carries the meaning of $r1 = 1$. It preconditions (through a self-loop arc) the transition *Grant+*, which may only be produced when the input privilege token has arrived. The SIS tool generates the logic for non-input signals $r1, r2$, *Tout* and *Grant*, treating the two outputs of the ME element, $g1$ and $g2$, as inputs to the circuit. The final implementation is very simple; it is shown in Figure 8,b.

The overall Busy Ring consisting of such arbitration cells works very fast, but it requires two rounds of signal transitions of *Tin* and *Tout* through the ring. During the rising edge phase the privilege token actually performs polling of requests. The second phase, the releasing of all local ME elements and signals *Tin* and *Tout* corresponds to the resetting round. It is easy to estimate the delay of this solution – two ME delays per Ring Adaptor cell per arbitration cycle. An additional invertor is required in one of the cells to re-establish the rising edge on the *Tin*-*Tout* channel.

Let us now consider the Busy Ring model shown in Figure 10, which is similar to the first as far as the RZ interface with the User Subsystem is concerned. The signalling in the ring interface is NRZ, thus this model does not distinguish between the rising and falling edge phases. This saves time on resetting the ring but only at the cost of circuit complexity, as can be seen from the implementation shown in Figure 16. This figure shows that in order to synthesise logic the STG requires an additional variable s, which resolves the CSC problem. The extra circuitry, compared to the previous solution, consists of two *transparent latches* (a transparent latch with *data* input D, "latch" input L, and output Q is described by the equation $Q = D\ L + (D + \ \overline{L})\ Q)$ and one XOR gate. This circuitry effectively converts the two-phase protocol on the external signals *Tin* and *Tout* to the four-phase signalling accepted by the ME element. The delay introduced by this circuitry amounts to the sum of two latches, XOR and ME element per Ring Adaptor per arbitration cycle. Note that the resetting of the ME element is done in parallel with the propagation of the privelege token further along the ring. If we are to choose between these two designs, we have to trade off the total delay of two latches and XOR in the Figure 16 solution against the total delay of ME element and additional wire delays, concerned with the resetting round, in the Figure 8 solution. If the distance between Adaptors is large, and special, relatively slow, line drivers are used for ring connections, the Figure 16 solution would be better. However, if the ring is for example an on-chip interface, then the first protocol and its circuit has better performance.

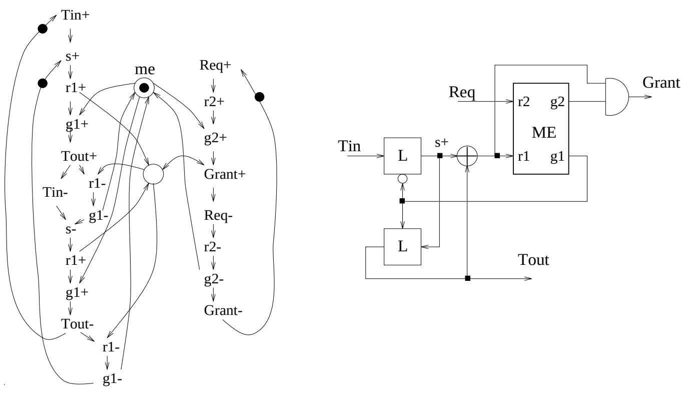


Figure 16: Circuit implementation for model of Figure 10

Let us now proceed to the model with two-phase protocols at both interfaces, shown in Figure 9. As before, we first insert the semaphore actions. Then we refine the original STG in such a way (see Figure 17,a) that the idea of "strong fairness", expressed in it (always pushing the privilege token out) is preserved. This requirement, which has not been put to our previous designs, guarantees that after using the privilege token the User Subsystem cannot acquire it again (regardless of its speed!) until the token travels at least once around the ring. To implement this requirement, our refined STG works in the following way. Depending on the resolution of the race between *Tin* and *R*, we have two possible sequences of actions. If *Tin* arrives before *R* has arrived, then the flag t (this flag indicates that the User Subsystem's request is pending in the Adaptor) is in state 0, and we simply generate *Tout* releasing the ME element. If *R* has won the local mutex condition, then it toggles the t flag to state 1, and releases the ME element.

Now, since another request cannot arrive on R until the previous has been fully processed, the adaptor awaits the arrival of *Tin* unconditionally. At this point, after $r1$, $g1$ is produced without actual arbitration and since $t = 1$ the other branch following $g1$ is chosen. Here, t toggles back to 0, then Grant G is produced and after the arrival of the Done signal (D) the *Tout* is generated, followed by the release of the arbiter.

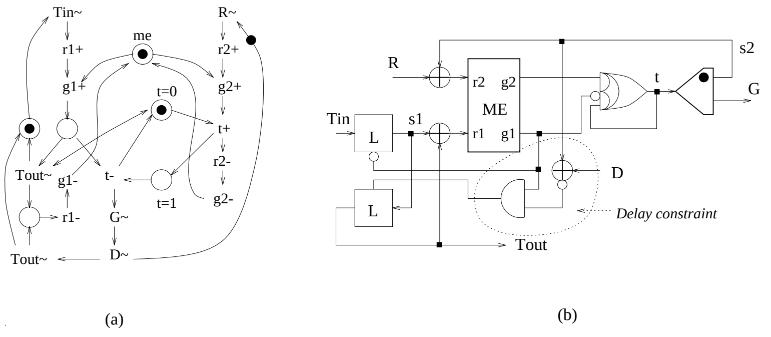


Figure 17: Circuit implementation for model of Figure 9

It is easy to see that this behaviour is equivalent to the one produced by the original STG model with respect to the original interface signals. The circuit synthesised with the SIS tool is shown in Figure 17,(b). During this implemetation we had to insert two additional signals, labelled $s1$ and $s2$, to solve the CSC problem. Note that the actual equation produced by SIS for $Tout$ included the logic of the NXOR and AND inside one complex latch. The decomposition shown in the figure strongly relies on the timing constraint imposed on the composite delay of NXOR and AND, which must be sufficiently short to prevent hazards in the latch. In the circuit diagram we have used a "shorthand notation" for the logic implementation of signals s and G – a Toggle element. Its Petri net description and possible implementation is shown in Figure 18. Note that the speed-independent circuit shown in Figure 18,(c) uses an auxiliary C-element (a *C-element* is described by the equation $y = x1\ x2 + (x1 + x2)\ y$) at the front to match the arrival of the new input change with the completion of the response to the previous change.

7.3.2 Lazy Ring Circuit Solutions

Now let us turn to the logic synthesis for the RZ option of the Lazy Ring Adaptor, whose STG was shown in Figure 12. Our first step should be the insertion of the semaphore (ME element) actions into the STG. Then, our analysis shows that the refined STG, shown in Figure 19,a, is consistent but has a number of CSC problems. These are concerned with the "decoupled" execution of the three handshakes in all their four actions (see the subgraphs within the dotted areas in Figure 12), e.g. for the "right handshake": *Rr+* $\Rightarrow$ *Ra+* $\Rightarrow$ *Rr-* $\Rightarrow$ *Ra-*. In order to assist SIS in deriving logic we added one additional variable, $s+$ after *Rr+* $\Rightarrow$ *Ra+* and $s-$ after

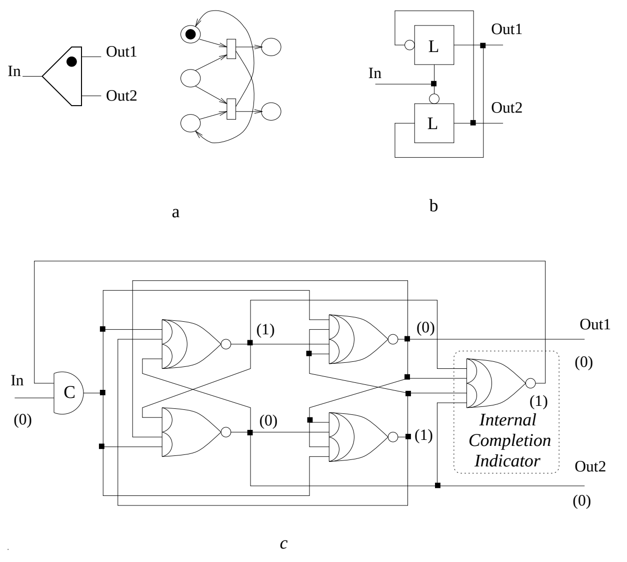


Figure 18: Toggle module: Petri net fragment and circuits

$g2-$. In the other branch, where the local request wins the arbitration, the role of a "separating variable" is played by the privilege token flag t.

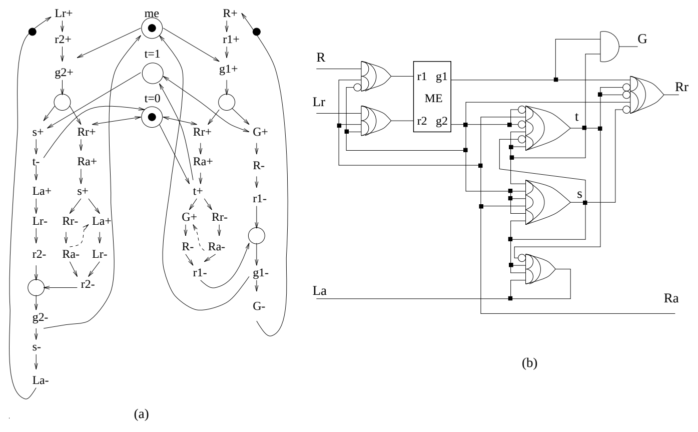


Figure 19: Circuit implementation for model of Figure 12: refined STG (a) (dashed arcs are used in a version with less concurrency) and logic circuit(b)

Synthesis by SIS gives the solution shown in Figure 19,b. The cycle time of action of this circuit can be easily estimated from its STG in Figure 19,a. Here we estimate only the response for the rising edge phase of the handshake signals. The complete cycle time should also include the falling edge phase. The latter is however done in parallel in adjacent adaptors, when the privilege token is propagated through them. This is illustrated in the fragment of the partial order diagram shown in Figure 20. The diagram is built for the case of three adaptors and when the winning request has to travel from the leftmost adaptor to the rightmost one, initially holding the privilege token. The overall delay is the maximum of the delays of the three paths in this acyclic graph: *R1+* $\Rightarrow$ *Ra2-* $\Rightarrow$ *t1+* $\Rightarrow$ *G1-*, *R1+* $\Rightarrow$ *Ra2-* $\Rightarrow$ *t1+* $\Rightarrow$ *La2-* $\Rightarrow$ *G1-*, and

R1+ $\Rightarrow$ *La2-* $\Rightarrow$ *G1-*.

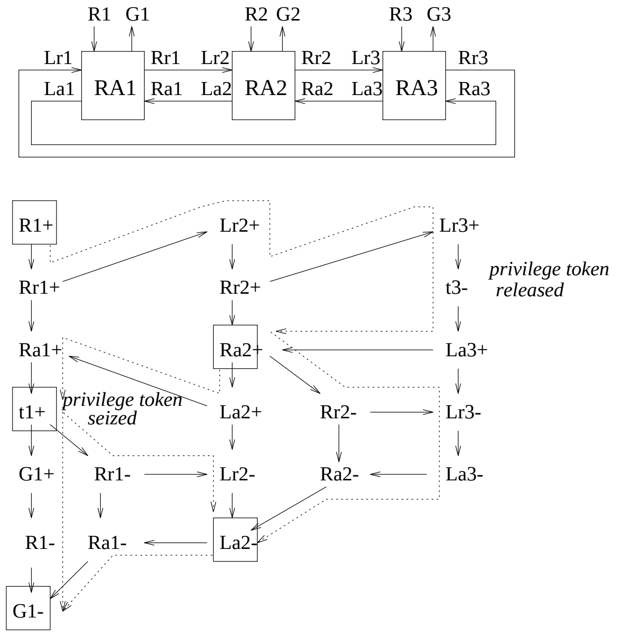


Figure 20: Partial order fragment for delay estimation

For example, the delay in responding to the request arriving on line *Lr* for the case when the privilege token is within the given cell can be estimated through the signal transition path *Lr+* $\Rightarrow$ *La+* when $t = 1$. It is equal to $4t_{ANDOR} + t_{ME}$, where t_{ANDOR} is an average delay of an ANDOR gate and t_{ME} is an average delay of a ME element.

The delay in responding to the request which appears on *Lr* when the privilege token is not in the cell can be estimated through the same signal path but for $t = 0$. This delay is added from the local elements' delay $4t_{ANDOR} + t_{ME}$ and the delay of the "right" handshake. The latter can be estimated in its worst case (whene the whole ring is traversed for the privilege token) as $(4t_{ANDOR} + t_{ME})(n-2)$, where n is the number of Ring Adaptors. Hence the overall response can be as long as $(4t_{ANDOR} + t_{ME})(n-1)$.

The responses to the request coming on signal *R* can be found by examining the signal transition path *R+* $\Rightarrow$ *G+*. For the case of the $t = 1$, the response is very fast, $t_{ANDOR} + t_{AND} + t_{ME}$. If $t = 0$, then the delay is $3t_{ANDOR} + t_{AND} + t_{ME}$ plus the the delay of the "right" handshake.

This circuit operates on the whole faster than the one by Martin in [24] for the same Lazy Ring protocol. Martin's circuit does not allow parallel release of inter-adaptor handhsakes (it effectively operates as if we used the dashed arcs in Figure 19,a). In our circuit the extra delay introduced by the release phase is constant, it does not depend on the size of the ring. In average, it is equal to the delay of the path *Rr-* $\Rightarrow$ *G-*, which is $5t_{ANDOR} + t_{AND} + 2t_{ME}$.

8 Verification at the Circuit Level

In this section, we discuss some aspects of the verification that is done after circuit synthesis. We also summarise all verification work throughout the whole design process. Why do we need to verify circuits which are synthesised mechanically, by STG-based logic synthesis, and therefore must be correct by construction? Theoretically, we needn't verify them. If the circuit is built from those, so-called complex, gates (some of which may be latches), then this circuit is speed-independent with respect to the delays of such gates – it is indeed correct by construction. In practice, the situation may be different. The designer, having got the boolean equations for the circuit may find that some of them are far too complex to be realised in one gate. The use of input inversions may also be inadequate to the technology mapping requirements. The existing formal techniques, which again theoretically produce correct-by-construction decompositions [2, 9], are not quite efficient in practice, nor do they work for widest possible class of output-persistent STGs. Therefore, the designer may introduce some modifications by hand; thus a check must be done whether they bring any hazards to the circuit or not. In our case, it would also be desirable to verify the whole design consisting of logic elements and ME elements.

There are some efficient methods for checking semi-modularity [15, 38]. These are however not suited for circuits with internal ME components. Petri nets, which are used at all stages of the design process, can be applied for circuit verification, too. Their model of logic circuits is called Circuit Petri net [41]. Let us briefly describe this method.

A Circuit Petri net is in fact a specific type of STG, in which each signal y is associated with two places, representing its two logical states. The groups of transitions labelled with $y+$ and $y-$ are connected to these places in such a way that the enabling/firing AND semantics of Petri net transitions, "corrected" through the appropriate labelling mechanism, adequately represents either AND or OR conditions in logic. The actual input "guards" for these transitions are formed by using *self-loop* Petri net arcs from the places associated with the state of the input signals to the gate. The use of self-loops, rather than "normal" input arcs is essential to this modelling method. It only allows tokens to be moved from the state-holding places associated with signals by firing transitions of the elements, whose outputs are modelled by these inputs. Therefore, if one models a circuit with inputs and outputs, the Petri net model of the circuit can only change the state of the places associated with its outputs. The marking of the places for the input signals can only be changed by the part of the net representing the circuit's environment.

As an example of the model of a level-based circuit, consider the one shown in Figure 21.a.

This circuit is a closed one, i.e. it is autonomous and has no interconnections with its environment. Its behaviour can be analysed using the reachability graph of its Petri net, which

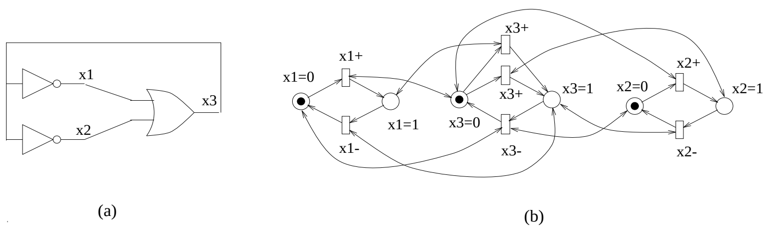


Figure 21: Example of a level-based circuit model

is depicted in terms of the binary states corresponding to the markings of the Petri net as shown in Figure 21.b. It is easy to see that this circuit has hazards if the delay of one of the invertors (say, for x_1) is greater or equal to the sum of the delays of the other invertor and the OR gate. In this case, the Petri net may start in state 000, in which both transitions x_1+ and x_2+ are enabled, then fire x_2+ and x_3+ in sequence, and finally enter state 011, with x_1+ disabled without firing. In the physical circuit, this corresponds to a potential hazard on signal x_2, while in Petri net terms, this is called *non-persistency* of the transition. Modelling possibility of a hazard by net non-persistency (with the exception of output signals of ME elements) automatically assumes that the circuit has *inertial* delays in its gates. Indeed, the idea of disabling a transition in a net is the same. Bearing this in mind, it would not be possible to adequately represent the behaviour of the circuit after it has manifested a hazard if the circuit's gates have *pure* delays. At this point the model's action diverges from that of the circuit [16].

Trying to simplify the task of converting an arbitrary logic circuit into an equivalent Circuit Petri net model, we have found a convenient block-diagram notation for Circuit Petri nets, which is shown in Figure 22. The connections between any block diagram are made using lines without double arrows in the Circuit Petri net. It is understood that no token can be captured by any of the successor transitions in the connected blocks. All the internal arrows of the Circuit Petri net are encapsulated in the block diagram. Intuitively, we know that if the token is resided at the place D=*1*, it is removed and inserted in its complementary place D=*0* if any of the transition D- is enabled.

As an experiment we have converted our circuit design shown in Figure 19,(b) into a Circuit Petri net. Its fragment is shown in Figure 23. The environment can also be modelled in the same way. For example, the Circuit Petri net model of an inverter adequately represents the User Subsystem, which effectively inverts signal G into R with some finite delay. Other handshakes, "left" and "right", are modelled as an inverter and a delay element, respectively. We have verified the net model of the circuit, built of the complex gates produced by the synthesis tool, together with the environment model. The check has confirmed our expectations the circuit is hazard-free under the assumptions about its environment behaviour given by its STG specification. We have

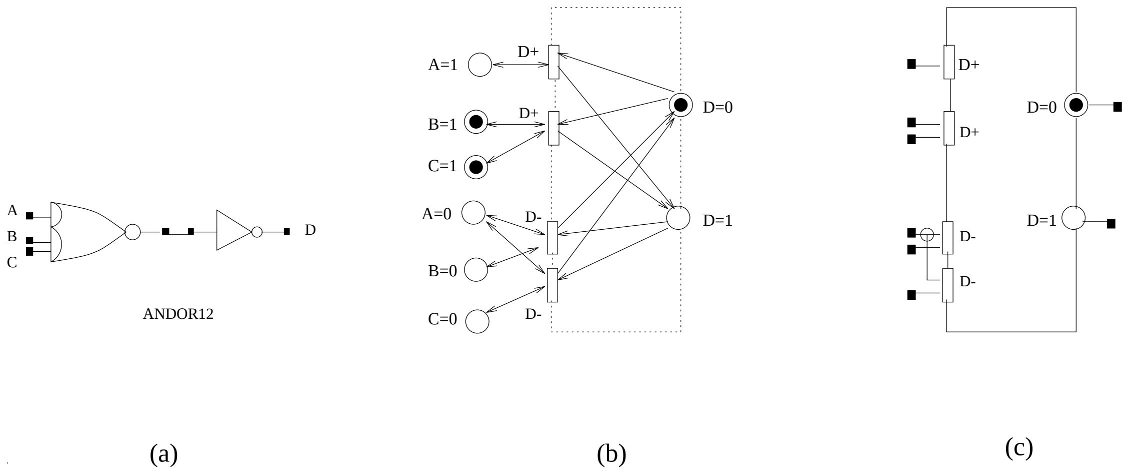


Figure 22: (a) Logic gate, (b) Circuit Petri net, and (c) block diagram

also carried an experiment of connecting three such models and verified a system consisting of three Ring Adaptors, one of which was put in the state with $t = 1$ while the other two had $t = 0$. Again the system behaves as required.

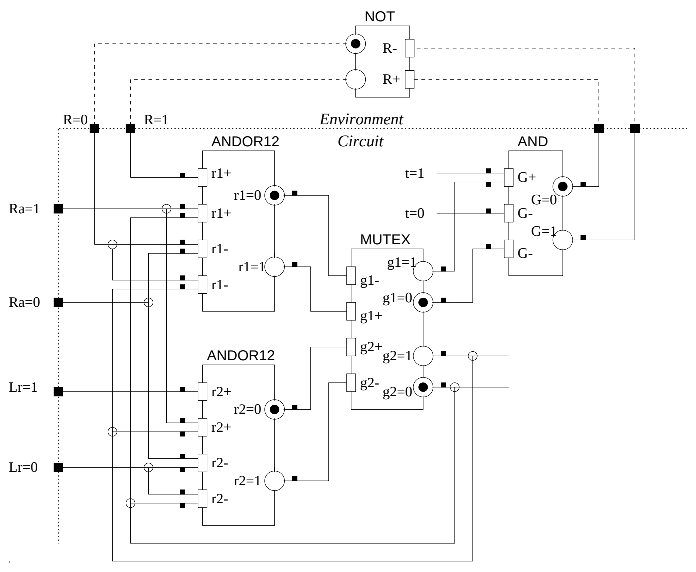


Figure 23: Fragment of Circuit Petri net model of Lazy Ring adaptor

We have then verified this circuit by making some input inversions explicit, by introducing Circuit Petri net models of invertors. Here our results were as follows. Introducing an explicit invertor for signal t creates no problems. The circuit remains hazard-free. On the other hand, using an explicit invertor either for $g2$ or s creates the possibility of a hazard under the unbounded delay assumptions. However, if we insert an invertor only for s and add its output as an aditional input to the function for t, thus making it $t = \overline{s}\ \overline{g2}\ Ra + (\overline{s} + \overline{g2})\ t$, then the circuit would still be hazard-free under the unbounded delay assumptions.

To summarise, during the design process, we have used Petri nets to verify our designs on at least three levels of abstraction: (1) the initial high-level description of the protocol; (2) the Signal Transition Graph refinement, and (3) the logic circuit level. Level (2) required us to make sure that our refinement of the semantically insignificant actions, such as the release of the handshakes does not create deadlocks, even though the STG the specification of a single adaptor can be correct. We verified the interconnection of three identical STGs which form the STG model of the three-adaptor system, with one of them initialised in the state of the privilege token ownership.

9 Circuit Simulation Using Cadence

In the previous sections, we have described how verification is carried out on the Petri net models at three different levels of abstractions. We have shown the reasons why a circuit that has been produced by synthesis methods may need to be checked for presence of hazards. The designer may also wish to check the functional correctness of the circuit behaviour at the low level, by means of simulation. In this section, we describe some of the results we have obtained from the simulation of the circuits based on the netlists obtained using Cadence. The simulation results presented here have been obtained from the transistor level using the CMOS implementation of our Lazy Ring adaptors.

This simulation has been carried out on a ring structure with three Ring Adaptors as shown in Figure 24. One of these adaptors, called *LazyInit*, is initialised with privilege token in it (t set to logic high). The waveforms obtained for the simulation are shown in Figures 25 and 26. *t1, t2* and *t3* are the latch signals in the ring adaptor 1,2 and 3. These signals are mutually exclusive, i.e. there must be only one of the signals at logic high at any time. Whichever signal is at logic high indicates the presence of the privilege token at that ring adaptor. *Lr1, Lr2* and *Lr3* are the request signals that are either generated when the request arrived at a ring adaptor, or relayed from the ring adaptor on the left. Notation *Ur/Ua* stands for Request/Grant signals coming from the corresponding User Subsystems (cf., pairs labelled R/G in Figure 19).

The waveforms shown in Figures 25 and 26 illustrate the following cases. Initially, the privilege token is in *LazyInit*, which is Ring Adaptor 2, thus $t2 = 1$. First, a request is produced in Ribg Adaptor 1, which translates it to Adaptor 2 on *Lr2.* the length of the pulse on *Lr2* is very small, it corresponds to the delay of **only one** handshake cycle between Adaptors 1 and 2 (this delay is approximately 10 ns). Then, Adaptor 1 obtains the privilege token ($t2 = 0, t1 = 1$) and produces a grant to its User Subsystem. Subsequently, Adaptor 2 receives its own request and translates it on *Lr3*, though it takes a bit longer time until *Lr3* is reset. This is because in this case the privilege token transfer passes through **two** handshake cycles, between Adaptors 2

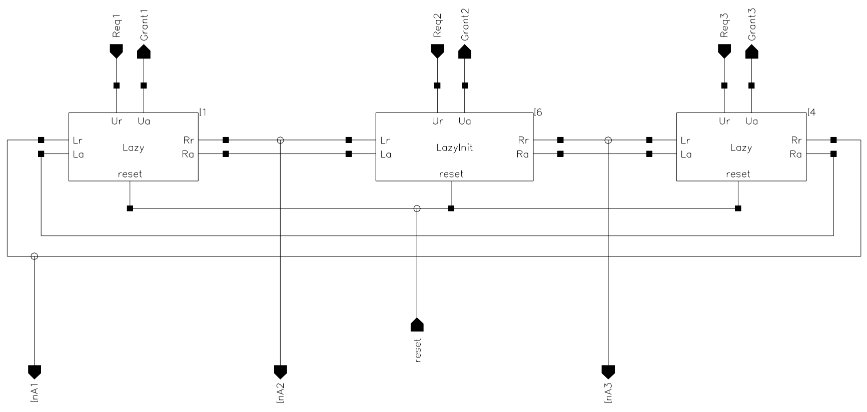
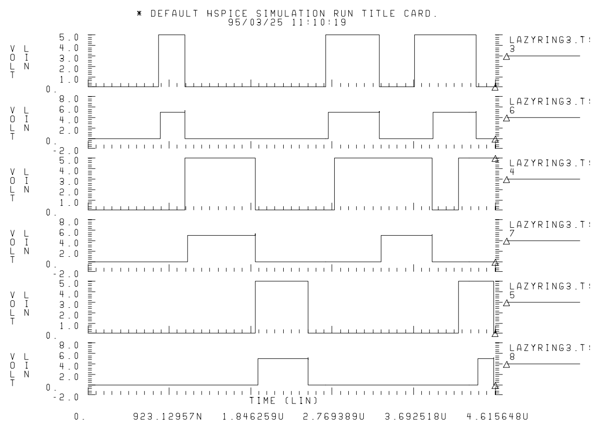


Figure 24: Ring Structure built with 3 ring adaptors using Lazy Ring Protocol

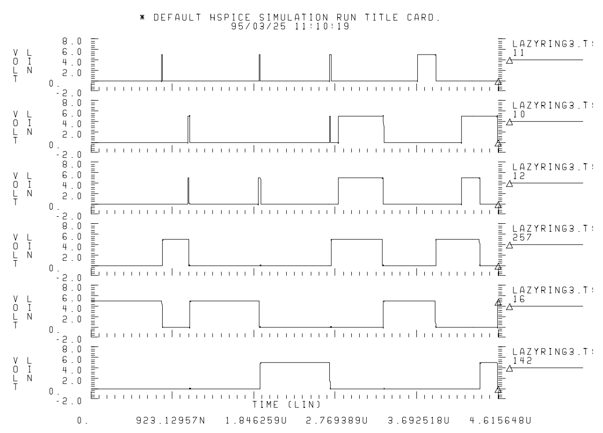
and 1 (via Adaptor 3). The following operation should be clear from these waveforms.



Line	3	6	4	7	5	8
Signal	Ur1	Ua1	Ur2	Ua2	Ur3	Ua3

Figure 25: Requests and Grants in Lazy Ring

An interesting situation happens when the privilege token is in Adaptor 3 ($t3 = 1$) and two requests arrive close to each other (though Adaptor 1 is slightly earlier) in Adaptors 1 and 2. The waveform clearly shows how Adaptor 1 first sends its request on *Lr2*, which is translated by Adaptor 2 on *Lr3*, which loses arbitration. Then, after the privilege has been delivered from Adaptor 3, via Adaptor 2, to Adaptor 1, the request inside Adaptor 2 is finally being put forward on *Lr3*, which now takes relatively long time – the time that the User System 1 holds the resource.



Line	11	10	12	257	16	142
Signal	Lr2	Lr3	Lr1	t1	t2	t3

Figure 26: Token Request and Privilege Token in the Lazy Ring

10 Summary and Conclusion

10.1 Summary

We have described a Petri net-based methodology for designing asynchronous control circuits. This has been done for a communication architecture based on Busy Ring or Lazy Ring Protocols. We began with the textual description of the behavioural specification of both protocols. The performance of the two protocols was measured by a simulation program. The high level modeling using Petri net was carried out to explore all possible event occurrences that could happen in each protocol. From the Petri net model, we described the protocol behaviours in terms of Signal Transition Graphs. After their verification, the STGs were synthesized into circuits using logic synthesis with the SIS tool. The netlists were mapped into circuit Petri nets and verified. using our software Petri net. Lastly, we simulated the netlist using Cadence HSPICE package.

10.2 Conclusion

One of the aims of this work has been to design and evaluate two asynchronous ring protocols. We have evaluated the performance of the two protocols at two levels, i.e. high-level event-driven simulation using C++ programming and circuit level using Cadence. Based on these results, we conclude that the Busy Ring Protocol has a better response time performance compared to that of the Lazy Ring Protocol. In terms of work load, the Busy Ring Protocol has a better performance when the load is heavy due to its ability to process at a faster rate than that of the Lazy Ring. As the number of stations increases the ratio between response times for the Lazy and Busy Ring also grows. On the other hand, the Lazy Ring Protocol proved to be more power-efficient than the Busy Ring Protocol. When the work load is lighter, the Lazy Ring can be a better option. At the circuit level, the Busy Ring Adaptor is more area-efficient than the Lazy Ring. This can also be noticed at the specification level because the number of states of the Busy Ring is less than that of the Lazy Ring. The size and complexity of the Lazy Ring proved to be twice as costly as that of the Busy Ring.

The suitability of the application of either of the two protocols depends on the type of environment that the communication architecture works in. For a light work load environment, a communication architecture that employs a Lazy Ring Protocol is a sensible choice. In a heavy work load environment, the power saving in the Lazy Ring Protocol no longer applies[7]. So, then, choosing a communication architecture that employs Busy Ring Protocol is a wiser choice.

Let us try to explain, at a more intuitive level, why the Lazy Ring Protocol has longer response time. Indeed what is the worst case delay between the arrival of a request in a Ring Adaptor and the arrival of the privilege token in it? Assume that, when a particular request has been released the same Adaptor does not seize the privilege immediately again. That is, we always guarantee that any pending request on any individual ME element is granted, once the other request has been released. In this case, the Busy Ring Protocol makes the worst case delay when a request arrives in the given Adaptor when it is the closest anti-clockwise (assuming that the privilege token travels clockwise) neighbour of the Adaptor which has currently seized the token. Then, if the traffic is active and all the Adaptors between the initial token holder and the given one acquire the token, the time when the privilege token reaches the given Adaptor is proportional to the total number of Adaptors in the ring.

In the Lazy Ring case, the worst case happens when a given Adaptor has received its request at the time when its second closest anti-clockwise neighbour holds the privilege token, and its closest anti-clockwise neighbour has issued its request which wins arbitration in the given Adaptor against the local request. Now, recall the our Lazy Ring Protocol applies a "no pre-emption rule", which forbids an Adaptor to take over the privilege token on its way to one of its anti-clockwise neighbours if the request from the latter has won arbitration in the Adaptor's mutex element. The overall time in this case, again, if the traffic is active and everyone on the way of the token acquires it for one utilisation act, becomes proportional to nearly twice the length of the ring. One length takes the closest anti-clockwise neighbour of the given Adaptor to obtain the token, and then one more length for the given Adaptor.

This intuitive analysis suggests an idea of creating a "hybrid" protocol, the one in which the "no pre-emption rule" is not applied. In other words, the request for the privilege token is produced and propagated as in the Lazy Ring, clockwise. This request propagation does not however carry out its "polling mission" as it has had in the normal Lazy Ring Protocol. Thus,

[7]The privilege token has no time to rest

when this propagating request reaches the current location of the token, the token begins to travel anti-clockwise and poll the actual requests in the Adaptors. It thus behaves like a Busy Ring polling token. With this discipline the overall utilisation of the token movements in the ring becomes better than in the Lazy Ring, and the worst case delay is again proportional to the one ring length.

Such an improvement would however be at the cost of the two factors. One is the factor of fairness. That is, the abolition of the "no pre-emption rule", may allow an Adaptor which is the closest anti-clockwise neighbour of the current token holder to acquire the token even though its request has arrived the last amongst all other Adaptors. This should not, however, be a problem since the same thing effectively happens in the Busy Ring, when the token does its polling in the clockwise direction. Provided that the above-mentioned fairness requirement for the ME element, the token never gets stalled in any one Adaptor. The other negative factor is complexity of the protocol and its circuit implementation.

To describe this "hybrid" protocol in a Petri net form let us refer to Figure 27. The protocol is defined by Petri net at the high level. It can be refined to an STG model in the way described in Section 6. Note that, the request propagation is done using the so-called *OR causality* [18]. That is, regardless which of the two requests, *Lr* or *R* arrives first, it produces a request on *Rr*. In the high-level model this effect of OR causality is represented in the place called *probe* which is allowed to be 2-safe (two tokens may arrive in it concurrently). Similar sort of effect was used for a low latency arbiter in [19]. Using the STG-based logic synthesis approach, described in Section 7.2, a speed-independent implementation can be obtained for the Ring Adaptor of this protocol.

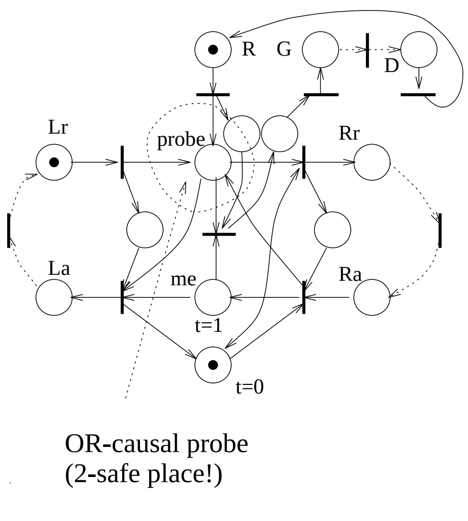


Figure 27: Illustration for the idea of "hybrid" protocol

References

[1] Seitz, C.L., "System timing," Chapter 7 of *Introduction to VLSI Systems (C.Mead & Conway),* Addison-Wesley (1980).

[2] P. Beerel, and T. Meng, "Automatic gate-level synthesis of speed-independent circuits," Proc. ICCAD'92, IEEE Comp. Soc. Press, November 1992, pp. 581-586.

[3] Chu, T.A.,"Synthesis of self-timed control circuits from graphs: An example," *Proc. ICCD,* pp.565-571 (Oct.1986).

[4] Chu, T.A.,"Synthesis of self-timed VLSI circuits from graph-theoretic specifications," *Ph.D Thesis,* MIT (June 1987).

[5] Chu, T.A.,"On the Models for Designing VLSI Asynchronous Digital Systems," *Integration: The VLSI Journal,* Vol.4 No.2, June 1986, pp.99-113.

[6] J.C. Ebergen, P.F. Bertrand,. and S. Gingras, "Solving a mutual exclusion problem with the RGD arbiter," Asynchronous Design Methodologies, S. Furber and M.Edwards (Eds), IFIP Transations A-28, North-Holland, 1993, (Proc. IFIP WG10.5 Working Conference, Manchester, March-April 1993).

[7] G. Gopalakrishnan, "Developing Micropipeline Wavefront Arbiters," IEEE Design and Test of Computers, Winter 1994, pp. 55-64.

[8] A. Hopper. S. Temple, and R. Williamson, "Local Area Network Design," Addison Wesley, 1986.

[9] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen and A. Yakovlev, "Basic gate implementation of speed-independent circuits,", Proc. DAC'94, San Diego, June 1994, IEEE Comp. Society Press, N.Y., pp. 56-62.

[10] K.S. Low, "Asynchronous Communciation Architecture", B.Eng. Thesis, Department of Electrical and Electronic Enginering, University of Newcastle upon Tyne, June 1995.

[11] T. Murata, "Petri Nets: Properties, Analysis and Applications," Proc. of IEEE, April 1989, pp. 541-580.

[12] C.L. Seitz, "Ideas about arbiters," *Lambda*, Vol.1(1, First Quarter), 1980, pp. 10-14.

[13] L.Rosenblum and A.Yakovlev, "Signal Graphs: from self-timed to timed ones," *Proc. Int. Workshop on Timed Petri Nets,* Torino, Italy, IEEE Comput. Soc. Press, July 1985, pp.199-207.

[14] L. Lavagno and A. Sangiovanni-Vincentelli, "Algorithms for synthesis and testing of asynchronous circuits," Kluwer Academic Publishers, 1993.

[15] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky, "Concurrent Hardware: The Theory and Practice of Self-Timed Design," Wiley and Sons, London, 1993.

[16] Yakovlev, A.V., Lavagno, L., and A. Sangiovanni-Vincentelli, "A unified signal transition graph model for asynchronous control circuit synthesis, " Proc. Int. Conf. on CAD (ICCAD'92), Santa Clara, CA, November 1992, IEEE Comp. Society Press, N.Y., pp. 104-111.

[17] J.Cortadella, L. Lavagno, P. Vanbekbergen, and A. Yakovlev, "Designing Asynchronous Circuits from Behavioural Specifications with Internal Conflicts," Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems, Salt Lake City, Utah, November 1994, IEEE Comp. Soc. Press, pp. 106-115.

[18] Yakovlev, A., Kishinevsky, M., Kondratyev, A. and Lavagno, L., "OR causality: modelling and hardware implementation," Proc. Int. Conference on Application and Theory of Petri Nets (ed. R. Valette), Zaragoza, June 1994, Lecture Notes in Computer Science, vol. 815, Springer, Berlin, 1994, pp. 568-587.

[19] Yakovlev, A., Petrov, A., Lavagno, L., "A Low Latency Asynchronous Arbitration Circuit," *IEEE Trans. on VLSI Systems*, vol. 2, No. 3, Sept. 1994, pp. 372-377.

[20] A.V.Yakovlev, A.M.Koelmans, and L.Lavagno,"High-Level Modeling and Design of Asynchronous Interface Logic," *IEEE Design and Test of Computers,* Spring 1995,pp.32-40.

[21] N.C.Paver, "The Design and Implementation of an Asynchronous Microprocessor," *PhD Thesis,* The University of Manchester, Comp Sci Dept, 1994.

[22] Coates B., Davis A., Stevens K.S., "The Post Office experience: designing a large asynchronous chip," *Integration: the VLSI journal,* Vol.15, No.3, Oct.1993, pp.341-366.

[23] Stevens K.S., "Practical Verification and Synthesis of Low Latency Asynchronous Systems," *PhD Thesis,* The University of Calgary, Calgary, Alberta, Sept.1994.

[24] A.J. Martin, "Synthesis of Asynchronous VLSI Circuits," *Formal Methods for VLSI Design,* J.Staunstrup, ed., North Holland, Amsterdam, 1990, pp.237-283.

[25] Peterson,J.L., "Petri Net Theory and the Modeling of Systems," *Prentice-Hall,* Englewood Cliffs, NJ, 1981.

[26] Sutherland, I.E, "Micropipelines," *Communication of the ACM,* Vol.32,No.6,January 1989, pp.226-231.

[27] E.M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R.Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R. Brayton and A. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," *Memorandum No. UCB/ERL M92/41* Electronics Research Laboratory, Department of Electrical Engineering and Computer Science, University of Californica, Berkeley, CA 94720, 4 May 1992.

[28] C. Ykman-Couvreur, B. Lin, H. De Man, "Assassin: A synthesis system for asynchronous control circuits," Technical Report, IMEC, Sept. 1994, User and Tutorial manual.

[29] Lyon, R.F., "Cost,Power and Parallellism in Speech Signal Processing." *IEEE 1993 Custom Integrated Circuits Conference,* May 1993.

[30] D.J.Kinniment, J.D.Garside, B.Gao, "A Comparison of Power Consumption in Some CMOS Adder Circuits," accepted for Fifth International Workshop on POWER - TIMING - MODELING - OPTIMIZATION - SIMULATION, Oldenburg, Germany, October 4 - 6, 1995.

[31] D.J.Kinniment, "Evaluation of Asynchronous Addition," *IEEE Trans. on VLSI Systems*, to appear in December 1995.

[32] A. Yakovlev, V. Varshavsky, V. Marakhovsky and A. Semenov, "Designing an Asynchronous Pipeline Token Ring Interface," Proc. Second Working Conf. on Asynchronous Design Methodologies, London, May 1995, IEEE Computer Society Press, pp. 32-41.

[33] A. Yakovlev, "Designing Control Logic for Counterflow Pipeline Processor Using Petri nets," Technical Report Series No. 522, Department of Computing Science, University of Newcastle upon Tyne, May 1995.

[34] A. Valmari, "Stubborn attack on state explosion," *Formal Methods in System Design*, Vol.1, 1991, pp. 297-322.

[35] A. Semenov and A. Yakovlev, "Verification of asynchronous circuits using Time Petri Net unfolding, " Unpublished manuscript, June 1995.

[36] K.L. Mcmillan, "Using unfolding to avoid the state explosion problem in asynchronous circuits," *Formal Methods in System Design*, 1995, to appear.

[37] D.L. Dill, "Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits," The MIT Press, Cambridge, MA, 1988, An ACM Distinguished Dissertation, 1988.

[38] O. Roig, J. Cortadella, E. Pastor, "Hierarchical verification of speed-independent circuits," Proc. Second Working Conf. on Asynchronous Design Methodologies, London, May 1995, IEEE Computer Society Press.

[39] T. Yoneda, Y. Tohma, and Y. Kondo, "Acceleration of timing verification method based on time Petri nets," *Systems and Computers in Japan*, Vol. 22(12), 1991, pp. 37-52.

[40] Kondratyev, A., Cortadella, J., Kishinevsky, M., Pastor, E., Roig, O., and A. Yakovlev, "Checking Signal Transition Graph Implementability by Symbolic BDD Traversal," European Design and Test Conference, Paris, March 1995, IEEE Comp. Society Press, N.Y., pp. 325-332.

[41] V. Varshavsky, M. Kishinevsky, V. Marakhovsky, V. Peschansky, L. Rosenblum, A. Taubin, and B. Tzirlin, *Self-timed Control of Concurrent Processes,* Kluwer Academic Publishers, 1990.

[42] M.Ajmone Marsan, "Stochastic Petri Nets: An Elementary Introduction," *Advances in Petri Nets 1989,* pp 1-29.

[43] S. Hauck, "Asynchronous Design Methodologies," *Proceedings of the IEEE*, Volume 83(1), 1995.

[44] J.Couvillion,R.Freire,R.Johnson, "Performability Modelling with UltraSan," *IEEE Software,* vol.8, no.5, Sept.1991, pp.69-80.

[45] W.H.Sanders and J.F.Meyer, "Reduced Base Model Construction Methods for Stochastic Activity Networks," *Computer-Aided Modeling, Analysis, and Design of Communication Networks,* vol.9, no.1, Jan.1991, pp 25-36.