International Journal of Modelling and Simulation

ISSN: 0228-6203 (Print) 1925-7082 (Online) Journal homepage: http://www.tandfonline.com/loi/tjms20

Concurrency Model for Network-On-Chip Design Architecture

A. Agarwal & R. Shankar

To cite this article: A. Agarwal & R. Shankar (2009) Concurrency Model for Network-On-Chip Design Architecture, International Journal of Modelling and Simulation, 29:3, 238-247

To link to this article: http://dx.doi.org/10.1080/02286203.2009.11442529

International Journal of Modelling and Simulation, Vol. 29, No. 3, 2009

CONCURRENCY MODEL FOR NETWORK-ON-CHIP DESIGN ARCHITECTURE

A. Agarwal* and R. Shankar*

Abstract

Designers exploit design reuse to enhance system design productivity. Integration of pre-designed reusable blocks may fail if these blocks execute in parallel, share resources, and/or interact with each other. Such concurrency issues, if not addressed, may be detrimental to normal functioning of the system. Multiprocessor architectures, recently introduced to extend the applicability of the Moore's law, depend upon concurrency and synchronization in both software and hardware to achieve that goal. Concurrency issues, if not addressed, may lead the system into a deadlock or a livelock state. System design integration and verification approaches will not be cost-effective in exposing concurrency failures as they are intermittent; this can be costly (significantly increased time to market and field failures). One would have to develop abstract concurrency models and do exhaustive analysis on these models to test for concurrency problems. In this paper we present a systematic approach that models concurrency by using an abstract symbolic modelling language to build a systematic high level concurrent system model; we also use an exhaustive analysis and verification tool to confirm that the system is devoid of concurrency failures. We present results for the NOC (network-on-chip) multiprocessor platform.

Key Words

Concurrency modelling, network-on-chip architecture

1. Introduction

System complexity, driven by both increasing transistor count and customer need for increasingly savvy applications, has increased so dramatically that system design and integration can no longer be an afterthought. SoC (system-on-chip) design methodology is based on the principle of "divide and conquer." Such a system may integrate one or two general processors, with a few slaves as application specific processors (ASPs). The system is divided into several independent functional blocks which are then designed, most likely as dedicated hardware engines (ASPs) or as tasks on one or two processors. These functional blocks synchronize their activities by means of local operating system on the processor(s) and an on-chip communication bus such as AMBA, to form a functional system [1]. With increasing user demands for computationally extensive applications on an embedded system, such as, multimedia, real-time video communication, and 3-D video gaming, this model is no longer feasible. Thus the industry trend has shifted toward multiprocessor-based systems. This has led to introduction of the multiprocessor-system-on-chip (MPSoC) platform [2]. The only addition in MPSoC as compared to SoC is the use of multiple processors for processing a single application [2].

As the technology scaling works better for transistors than for interconnecting wires [3], there is an increasing disparity between the wires and the transistors in terms of power consumption and latency. Thus bus-based communication becomes a bottleneck [4]. In traditional systems all the resources share one or more common buses and thus the same bandwidth. As a consequence, as the number of the processors on the bus is increased, the system performance decreases significantly [5]. Many innovations have been introduced in bus architectures to counteract this. These include pipelining, split-and-retry techniques, removal of tri-state buffers and multiphase clocks, and various efforts to define standard communication sockets [6]. However, in many cases introducing new bus architectures, such as AMBA and ASB to AHB2.0, AMBA-lite and AMBA AXI, has required many changes in bus implementation, and more importantly bus interfaces, thus impacting Intellectual Property (IP) reusability. Another reason that the buses are not scalable is that they cannot decouple the activities of the transaction, transport and physical layers [7]. On a billion transistor chip, it would not be possible to send a global signal across the chip in real-time. Moreover, a bus-based SoC or MPSoC does not offer the required amount of reuse to meet the time-to-market requirements. This will continue to impact the productivity of system architects and designers [8].

Network-on-chip (NOC) can improve design productivity by supporting modularity and reuse of complex

* Department of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL 33431, USA; e-mail: {ankur, ravi}@cse.fau.edu

Recommended by Prof. Omar Al-Jarrah
(paper no. 205-4809)

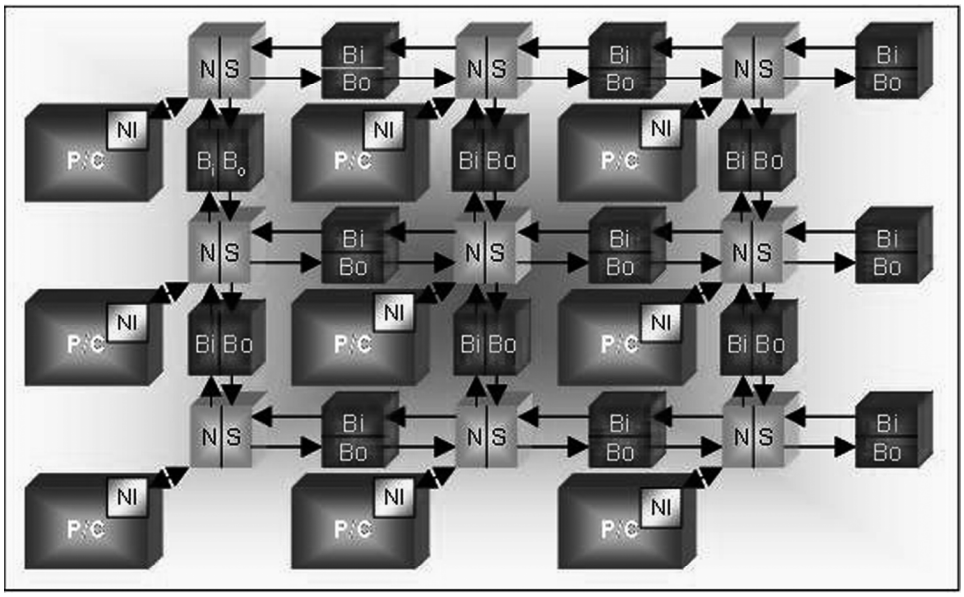

Figure 1. NOC architecture.

cores, thus enabling a higher level of abstraction in the architectural modelling of future systems [9–11]. A NOC is designed as a layered architecture and mainly comprises of two layers: communication protocol and communication backbone. Communication protocol layer consists of the network interface (NI) and is responsible for decoupling communication from computation and packetization/depacketization. The communication backbone layer, on the other hand, is instantiated with three components: routers, buffers, and links. Figure 1 shows a 3×3 mesh-based NOC architecture. In Fig. 1, Bi, Bo, P, C, N, S, and NI represent input buffer, output buffer, producer, consumer, node, scheduler, and network interface respectively. Links provide connection to various buffers, node/scheduler, and buffer. The routers are connected to each other and to NI, as per a specific topology. The communication backbone layer supports routing, switching priority-based communication, and flow control logic. A resource such as processor, memory, field programmable gate array (FPGA), application-specific integrated circuit (ASIC), or any other hardware element must be connected to routers or switches through an NI. The consequences of NOC architecture imply a shift in concern from computation and sequential algorithms to modelling concurrency, synchronization, and communication in every aspect of hardware–software co-design and development.

In a multiprocessing environment there are various processes executing simultaneously; therefore, it is very important to address synchronization and concurrency issues. In NOC, at a given time, every resource may either be producing or consuming some data packets. Thus it can be said that almost every element (node, link, buffer, router, scheduler, resource, etc.) might be communicating with another element at some point perhaps concurrently on the NOC platform. Therefore, there are several interprocesses communications in such a model. If these interprocess communications are not modelled properly then the system may fail. Such a failure may not be detected at the system integration phase. This is due to the fact that these failures are intermittent, which occur only under certain condition, but nevertheless may be catastrophic. A system may be more prone to intermittent failures if concurrency concerns are not addressed properly. These intermittent failures may finally result into a deadlock or a livelock state. Thus it is very important to model concurrency issues in such NOC systems. In this paper we propose to model concurrent processes in an NOC.

2. Background

During the previous decade one could enhance the system performance by simply increasing the clock speeds. International Technology Roadmap of Semiconductors (ITRS) predicts that the saturation point for these clock speeds is nearing [12]. Thus, we need to find other innovative ways of further enhancing performance. One way that could help in this direction is by exploiting concurrency [13]. The concept of concurrency is essential to for multiprocessing and distributed system environment and, at the same time, allows us to see a set of interacting objects as a collection of concurrent processes whose behavior can be observed by means of all their possible interactions. Concurrency may also help in reducing the overall system power consumption. Software developers have realized the need for concurrency and have started using multithreaded programming models for embedded system designs. Multithreaded JAVA and pthreads are two examples for incorporating concurrency [14]. However, in most cases concurrent processing has been overshadowed by a failure to achieve synchronization. Unified system design frameworks, such as Ptolemy, use JAVA as backbone to address concurrency issues [15]. But the main issue is to be able to analyze whether after following all the steps for designing a concurrent system, does the new system design still has any concurrency concerns? There are various models of computation, which can achieve concurrency [16]. Some of these

Models of Computations (MOCs) include communicating sequential processes (CSP) [17], pi-calculus [18], lambda-calculus [19], and finite state machines (FSM) [20]. CSP, pi-calculus, and lambda-calculus offer an effective mechanism of specification and verification [21]. Pi-calculus is a well-defined process algebra that can be used to describe and analyze process systems. It allows mobility of communication channels, includes an operational semantics, and can be extended to a higher-order calculus, where not only channels but whole processes can be transferred [22]. However, they are based on mathematical models; they are not only hard to understand for a software–hardware designer but also hard to apply practically. FSM have been extensively used in the past, especially in the domain of modelling control applications. But these FSM are not able to address the increasing size of the software content in the systems. Hierarchical FSM have replaced FSM in modelling concurrent applications. Several tools and methodologies have also been introduced which address these concurrency concerns. Unified modelling language (UML) addresses these concurrency concerns in the form of state diagrams and sequence charts [23]. But UML may not be useful for analyzing the system exhaustively for concurrency failures.

Finite state processes (FSPs) and labelled transition system analyzer (LTSA) provide a framework for modelling concurrency and analyzing it exhaustively [24]. FSP is a language based on the CSP. But unlike CSP a system designer will not have to analyze the system specification in the form of a mathematical model to expose concurrency issues in the system. We can develop a simplified concurrency model in FSP and analyze the concurrency issues with LTSA graphically. LTSA also provides a capability of exhaustive analysis to uncover synchronization issues such as deadlocks and livelocks.

3. Introduction to FSP

In FSP, we define a system in terms of its processes. These processes may be hierarchical processes. The final system composition is obtained by composing all the processes and its sub-processes. Life cycle of a process and its interaction with other processes can be described based on some basic constructs in FSP.

3.1 Process Definition and Action Prefix

The life cycle of a process Q is described by (1).

$$Q = (a \rightarrow Q). \tag{1}$$

A dot operator "." in the end represents end of the process definition. (1) also means that a process Q engages in an action a. After the completion of an action a it returns back to its normal state Q. The statement further represents that this process is a never-ending process and always performs the actions a before retuning to its initial state. Note that the actions in the process description are denoted by the small case letters or words and process names with the capital letter or words. A process that ends at some point may be represented by (2). A STOP state represents that the process Q terminates after completing the action a.

$$\mathrm{Q} = (\mathrm{a} \rightarrow \mathrm{STOP}). \tag{2}$$

3.2 Choice

A choice of possible actions of a process is denoted as shown in (3). It can be realized from (3) that a process P can result in either a process L or M depending upon the choice of actions a or b. If the process P engages in an action a then it will result in a process L, otherwise if it engages in action b then it will result in a process M. The symbol "|" represents a deterministic choice.

$$P = (a \rightarrow L | b \rightarrow M). \tag{3}$$

If the two actions are same then it will represent a non-deterministic choice as can be seen from (4).

$$P = (a \rightarrow L | a \rightarrow M). \tag{4}$$

3.3 Conditions

Condition operator behaves in a way similar to the choice operator, with the difference that condition operators will never represent a non-deterministic behavior of choice. The available condition operations available are *when* and *if*. Both the condition operators can perform similar operations. (5) shows a condition with *when* as a condition operator.

$$P = (when(i = 0)a \rightarrow P | when(i > 0)b \rightarrow P) \tag{5}$$

It can be realized from (5) that when the value of the local variable i is equal to zero; action a takes place, whereas when the value of i is greater than zero, action b occurs.

3.4 Parallel Composition

In any system, a set of processes may have their actions interleaved at any time instant. This interleaving of actions is defined by parallel composition. A parallel composition is represented by (6).

$$P = (L||M...X). \tag{6}$$

It can be seen from (6) that the processes $L, M, \ldots, X$ are combined together by a parallel composition operator. The parallel composition is commutative and associative. Therefore the order of composition of various processes is immaterial. By parallel composition various processes communicate with each other by synchronizing their common actions and interleaving all the possible remaining set of actions. It should be noted that the simulation of parallel composition would cover all possible scenarios of interleaved actions of various processes and synchronized common activities.

3.5 Shared Actions

Shared action is the way to synchronize the actions of the processes. It probably represents the most important aspect of concurrent programming. Two processes synchronize when they share at least one action. For example in (7) and (8), processes P and Q share an action b. It can be implied from the two processes P and Q that they synchronize with each other while they perform action b .

$$P = (a \rightarrow b \rightarrow c \rightarrow P). \tag{7}$$

$$Q = (d \rightarrow b \rightarrow l \rightarrow Q). \tag{8}$$

3.6 Process Labelling

In a real world, any number of processes may be sharing a common resource. For example, we may think of a situation where we have two users as a and b and both users share a resource: printer. The $USER$ processes acquire the resource, use the resource and finally release it before returning to its normal state. User process in this example can be described by (9).

$$USER = (acquire \rightarrow use \rightarrow release \rightarrow USER). \tag{9}$$

A RES process may be acquired and released by any user. Thus the process definition for a resource process can be given as in (10).

$$RESOURCE = (acquire \rightarrow release \rightarrow RESOURCE). \tag{10}$$

In this case if we have two instances of $USER$ process, then we represent it using process labelling. Let us assume that there are two $USER$ processes as a and b. Then the composition process depicting the process labels should be represented as in (11).

$$||COMPOSTION = (a : USER||b : USER||\{a, b\} :: RESOURCE). \tag{11}$$

3.7 Re-labelling

In a development environment, all the components are never built by the same person. The specifications are often reused and redesigned. The components of a system are built separately, which are combined in the integration phase. Thus, there will always be some actions in different processes that were suppose to synchronize but are labelled differently. So there is a need for mechanism to re-label these actions and synchronize them. (12) shows that in a composition process P action a has been re-labelled as actions b. It can also be referred as data-matching. The re-labelling operator "/" assigns a common name to specified actions allowing them to synchronize with each other.

$$||P = (Q||R)/\{a/b\}. \tag{12}$$

3.8 Hiding

In any system when we define the processes of the system, we define the processes for all its entire set of actions. But for a particular scenario some of its actions may be irrelevant from the simulation perspective. Thus we should be able to hide this subset of actions. This hiding of actions reduces the number of states thus allowing us to simulate only relevant set of actions and processes. This data-hiding is done using "@" operator. It can be seen that in (13) we are hiding action b of process P.

$$P = (a \rightarrow b \rightarrow c \rightarrow P)@\{b\}. \tag{13}$$

3.9 Priority

Actions of some process may have either higher or lower priority than other actions in the same process. This can be represented by either assigning high or low priority to the actions. (14) shows that action b in process P has a higher priority than action a. For assigning action b a lower priority than action a we just need to change "$\gg$" operator to "$\ll$" operator.

$$P = (a \rightarrow b \rightarrow P)/b >> a \tag{14}$$

3.10 Alphabet Extension

Alphabet extension is used to extend the set of actions to a process that are not defined in the process definition. In (15), process P extends its alphabet with the action $a[1]$.

$$P = (a[0] \rightarrow a[2] \rightarrow P) + \{a[0..2]\}. \tag{15}$$

4. Concurrency Model for NOC

In this paper, we have used FSP language for concurrency modelling and LTSA for concurrency verification. For developing a concurrent model, we first write a high-level specification. This should not be platform or data dependent. We then identify the concurrent processes in our model. As the concurrency issues arise only among interacting processes, not in the (sequential) internal data processing, we model only the (external) interaction among various processes. This reduces the model complexity significantly; models execute faster due to reduced state exploration. In this section, we present our NOC concurrency model for the communication backbone layer.

4.1 High-Level Specification

These high-level specifications should not contain any details of system specification. Abbreviated specifications follow: (1) Data will be received in serialized packet format. (2) There will be several paths available arranged in a matrix fashion for the data packet to travel. (3) Data may be buffered at each intersection. (4) Further routing is available based on the availability and congestion of links

at the destination. (5) Packet will contain destination address and priority. (6) Links may be unidirectional (2 links for each direction) or bi-directional. We further make some assumptions to simplify our concurrency model: links are unidirectional and nodes are arranged in 2-D mesh.

4.2 Identification of Concurrency Processes

We identified concurrent processes from the above specifications. These concurrent processes identified were links, buffers, schedulers, nodes, producers, and consumers. We then define detailed specification for each of these processes: (1) Link specification: link collects the data from the source; and link forwards the data to the buffer. (2) Buffer specification: store & input data: data sent by link; forward the data to the output: data sent to the node; inform about the buffer status: buffer is empty, buffer is full and forward the high priority data first. (3) Scheduler specification: receives the request from the buffer for forwarding the data to the node; forwards the request for transmitting the higher priority data first; and checks the availability of data path for the data packet. (4) Node specification: determines the route information; gets the data from buffer; and forwards the data to a buffer. (5) Producer/Consumer specification: producer forwards the data packet to the buffer; data packet will be forwarded based on the availability of the buffer; and consumer accepts the data packet from the buffer.

4.3 Develop Model Incrementally and Hide Internal Details

While modelling our concurrent NOC model we followed an incremental approach. We first developed the producer and link processes. We then checked for concurrency issues in these two processes before including other processes in the model. As link is attached to buffer, we then added a buffer process to the model. Figure 2 shows the FSP code for the simplified producer process. Figure 3 shows the model of producer process.

$$\text{PRODUCER} = (\text{buffAvail} \rightarrow (\text{hiPriDataOut} \rightarrow \text{PRODUCER} | \text{midPriDataOut} \rightarrow \text{PRODUCER})).$$

Figure 2. FSP code for producer process.

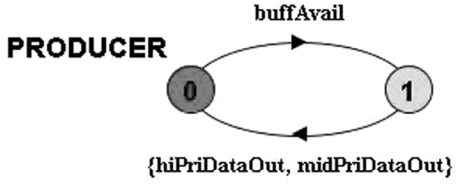

Figure 3. State diagram for producer process.

It is evident from Fig. 2 that we are modelling only the interaction among the producer and buffer processes. If the buffer is available then producer outputs either a high-priority data (hiPriDataOut) or a mid-priority data (midPriDataOut). It does not reveal any details so as to the source or contents of the data. The FSP code with two processes: buffer and producer, is shown in Fig. 4. In the buffer process, we declare two count variables of size 2, for storing the data values corresponding to each priority type: hihg-priority and mid-priority. We assume that the short data such as interrupt signals, and read/write signals consisting of a smaller data packet, will be modelled as a high-priority data (thus being represented as shortDataInBuff). While all the remaining data are represented as dataInBuff. Whenever we receive a hiPriDataIn or a midPriDataIn, we increment their corresponding counter values.

The buffAvail signal is enabled when the sum of the two counts is less than the total size of the buffer ($a + b < N$). Thus, there is a signalling protocol being modelled between buffer and producer processes. Once a buffer receives either a hiPriDataIn or midPriDataIn, it informs the next process that it has data ready to be transmitted further by enabling shortDataInBuff or dataInBuff signals. As a hiPriData needs to be transmitted before a midPriData, dataInBuff signal will not be enabled until all the hiPriData have been transmitted. A scheduler, which will interact with the buffer process, accepts the data packet from the buffer and routes it further. As the scheduler is connected to four buffer processes, a buffer needs to get access to a scheduler process before sending the data packet. We have modelled this interaction based on request–grant (handshaking) protocol. On receipt of a nodeGrantFast (for hiPriData) or nodeGrantSlow (for midPriData) buffer sends out its data (hiPriOut or midPriOut). The data packet transmitted may not get routed further due to network congestion and may get dropped at some point. We have modelled two interactions "confirm" and "notConfirm" to avoid such scenario. We delete the data from the buffer only after it has been transmitted and stored into another buffer or delivered to a consumer process.

State diagram for a PRODUCER_BUFFER (producer and buffer combined process) is large; thus, in Fig. 5, we show the interaction among these two processes with only one data priority level. However, we have run the simulation with three priority levels. Figure 6 shows the simulation result for PRODUCER_BUFFER with three different priority levels. It can be seen from the figure that there are no deadlock or livelock in the system.

In Fig. 7 we show the state diagram of the scheduler process. Scheduler process handles the control and status signals for the node process. It interacts with the buffer process and the node process. Its main functionality is to facilitate the data transfer from a buffer process to the next node. A handshaking protocol has been implemented between buffer and scheduler. The buffer shows the availability of data to be transferred by raising the "shortDataInBuff" (for high-priority data packets) or "dataInBuff" (for low-priority data packets) signals depending upon the priority of the data packets. The scheduler responds to these signals by "nodeGrantFast" for transferring high-priority data packets and "nodeGrantSlow" for transferring low-priority data packets. Upon the receipt of grant signal, buffer sends the data to the node process. Node process then calculates the output path for the data packet and then passes this information to the node process. The node

```
PRODUCER = (buffAvail → (hiPriDataOut → PRODUCER
                        |midPriDataOut → PRODUCER)).
const N     = 2
range Data_Range = 0..N
BUFFER = STATE[0][0],
STATE[a:Data_Range][b:Data_Range]
   = (when ((a+b)<N) buffAvail → (hiPriDataIn  →  STATE[a+1][b]
                                 |midPriDataIn → STATE[a][b+1])
     |when (a>0) shortDataInBuff → nodeGrantFast → hiPriOut → (confirm → STATE[a-1][b]
                                                              |notConfirm → STATE[a][b])
     |when((a==0)&& b>0) dataInBuff → nodeGrantSlow → midPriOut → (confirm → STATE[a][b-1]
                                                                  |notConfirm → STATE[a][b])).
        ||PRODUCER_BUFFER = (p1:PRODUCER||b1:BUFFER)/{p1.buffAvail/b1.buffAvail,
        p1.hiPriDataOut/b1.hiPriDataIn, p1.midPriDataOut/b1.midPriDataIn}.
```

Figure 4. FSP code for the combined buffer-producer processes.

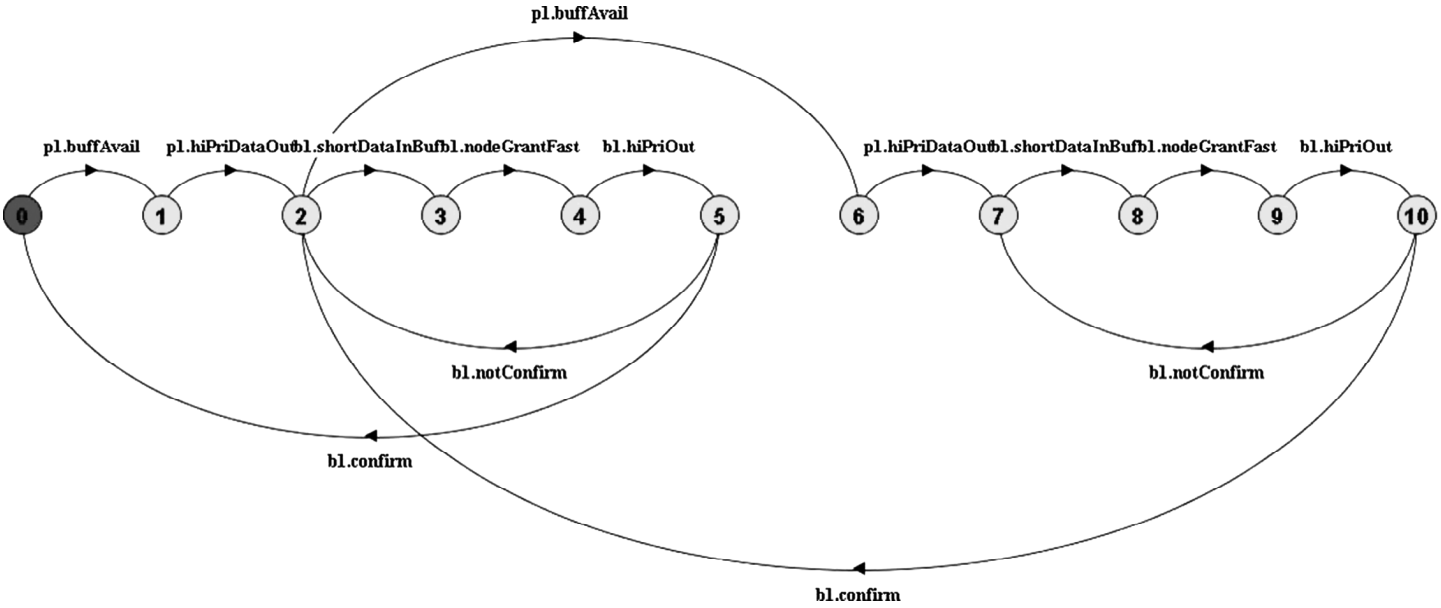

Figure 5. Interaction between producer and buffer process.

process then checks the availability of the output buffer. The status information of output buffer is then passed to the input buffer. In case the output buffer is full, the input buffer does not remove the copy of the data packet. If the output buffer is ready to accept the data packet the input buffer removes the copy of the data packet. Node is also responsible for prioritizing the data-transfer requests. As per the mesh topology, a node is connected to five input buffers: four for forwarding the data packet in North, South, East and East direction and fifth for sending the data packet to NI. Thus node serves these input buffers in a priority-based round-robin scheduling criteria. This allows the node process to transfer the high-priority data packet before serving low-priority data packets. Upon the simultaneous request for transferring two similar priority data packets, round-robin scheduling is employed.

4.4 Abstraction of Specification

As mentioned earlier, the concurrency issues are caused only when two or more processes interact with each other. The internal operations of a process will never cause any concurrency violations. Thus, we abstracted the internal details of a process and only modelled the interactions among processes to determine concurrency concerns. While modelling the system for checking concurrency issues we need to make sure that the model is an abstracted model not a detailed one. Each modelled process represents the actions that lead the process into different states. When different processes are composed together, the system is analyzed exhaustively through each possible path through different states. Thus, we must limit the number of states in the model. To limit the number of states, it is important to hide unnecessary details from the model. An NOC model with nine nodes and schedulers along with 36 buffers, nine producers and consumers, and more than 100

```
Compiled: BUFFER; Compiled: PRODUCER
Composition: FINAL1 = b1:BUFFER || p1:PRODUCER
State Space:  24 * 2 = 2 ** 6
Composing...
-- States: 24 Transitions: 34 Memory used: 2626K
Composed in 110ms; No deadlocks/errors
Progress Check...-- States: 24 Transitions: 34 Memory used: 2705K
No progress violations detected. Progress Check in: 0ms
```

Figure 6. Simulation result for PRODUCER_BUFFER with three priority levels.

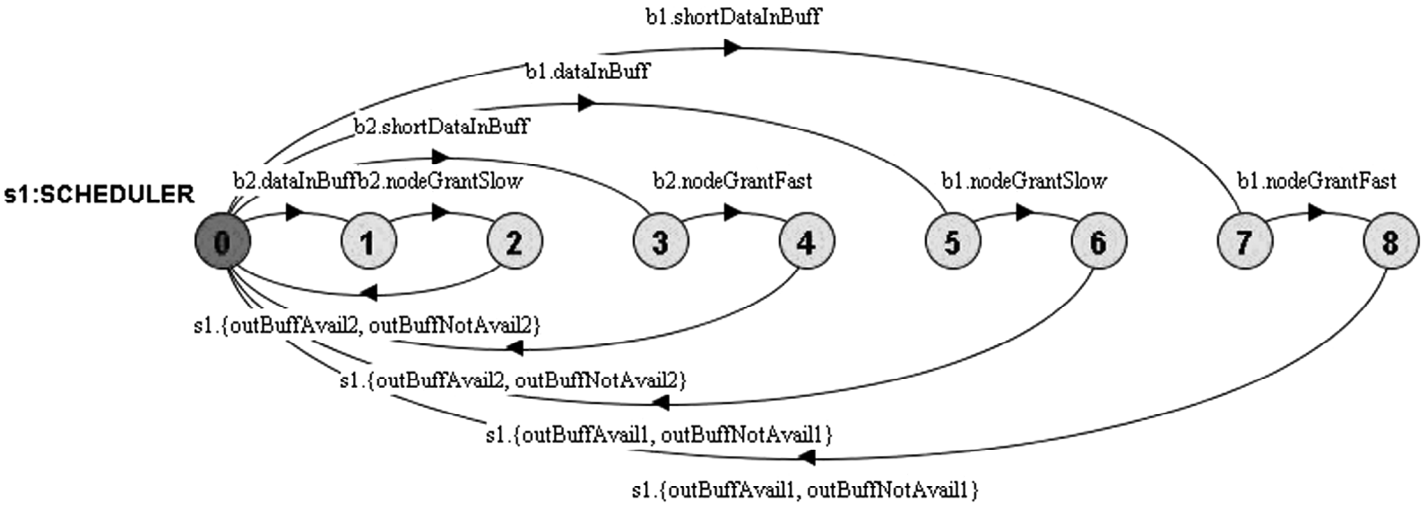

Figure 7. State diagram for scheduler process.

links will comprise of tens of thousands of states. Therefore, it will not only be difficult but almost impossible to model with FSP and analyze it with LTSA. Hence we abstracted NOC model to represent only one representative scenario of interaction meaning, if have analyzed one complete path from a producer process to a consumer process. If the interactions in this path do not have any concurrency concerns, then other interactions in the similar other paths may be replicated to avoid any deadlock or livelocks in the system. This abstracted model with all the paths from one producer to a consumer is represented in Fig. 8.

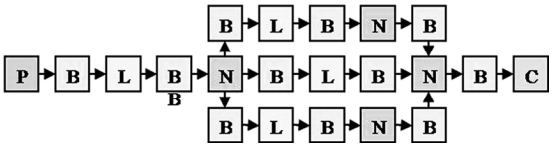

Figure 8. NOC model for interaction between one producer and one consumer process.

To prove the above statement we have considered a system with two producers instead of one producer. We have further assumed that the 1st producer (p1) is sending the data to 1st buffer (b1) and 2nd producer (p2) is sending data to 2nd buffer (b2). We have analyzed such system for deadlock conditions. Fig. 9 represents only the partial FSP code that shows two producers and two buffers interacting with producers. It can be seen from the simulation results in Fig. 10 than the final result will not change. Thus, it verifies that abstraction of the system will not alter the system properties.

Further, from the implementation of the link process we realized that a link is responsible for just forwarding the data to the next process. In this it does not play any role that may cause any concurrency concerns. Thus, we eliminated the link process from the model. We further reduced this model shown in Fig. 8, into a more simplified model by removing unnecessary data paths. The synchronization issues will arise when more than one process tries to interact

```
||FINAL1 = ({b1, b2}:BUFFER||s1:SCHEDULER||{p1,p2}:PRODUCER)/{
        b1.hiPriDataIn/p1.hiPriDataOut,b1.midPriDataIn/p1.midPriDataOut,b1.buffAvail/p1.buffAvail,
        b1.shortDataInBuff/s1.shortDataInBuff1,b1.dataInBuff/s1.dataInBuff1,
        b1.nodeGrantFast/s1.nodeGrantFast1,b1.nodeGrantSlow/s1.nodeGrantSlow1,
        b2.hiPriDataIn/p2.hiPriDataOut,b2.midPriDataIn/p2.midPriDataOut,b2.buffAvail/p2.buffAvail,
        b2.shortDataInBuff/s1.shortDataInBuff2,b2.dataInBuff/s1.dataInBuff2,
        b2.nodeGrantFast/s1.nodeGrantFast2,b2.nodeGrantSlow/s1.nodeGrantSlow2,
        s1.outBuffAvail1/b1.confirm,s1.outBuffNotAvail1/b1.notConfirm,
        s1.outBuffAvail2/b2.confirm,s1.outBuffNotAvail2/b2.notConfirm}.
```

Figure 9. Partial FSP implementation with two producers.

```
Compiled: BUFFER; Compiled: SCHEDULER; Compiled: PRODUCER
Composition:
FINAL1 = b1:BUFFER || b2:BUFFER || s1:SCHEDULER || p1:PRODUCER|| p2:PRODUCER
State Space:  24 * 24 * 9 * 2 = 2 ** 15
Composing...
-- States: 351 Transitions: 882 Memory used: 3076K
Composed in 120ms; FINAL2 minimising.....
Minimised States: 351 in 49ms; No deadlocks/errors; Progress Check...
-- States: 351 Transitions: 882 Memory used: 3343K
No progress violations detected.; Progress Check in: 33ms
```

Figure 10. Simulation result for two producer model.

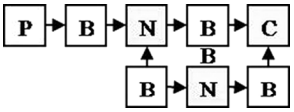

Figure 11. NOC model for interaction between one producer and one consumer process with link process eliminated.

with another process. But as long as the number of similar processes is more than one, we can have a general model to represent these interactions as long as the processes do not change the way they interact. That is, if a node (N) does not cause any concurrent violations while interacting with two buffers (B) then it will not pose any concerns while communication with three as well. This is due to the fact

```
Composition: FINAL1 = b1:BUFFER || p1:PRODUCER
State Space:  30 * 2 = 2 ** 6
Analysing...Depth 4 -- States: 11 Transitions: 17 Memory used: 6095K
Trace to property violation in b1:BUFFER:
        b1.dataInBuff;  b1.nodeGrantSlow;  b1.midPriOut;  b1.confirm
Analysed in: 15ms
---------------------------------------------------------------------------------------
Composition: FINAL1 = b1:BUFFER || p1:PRODUCER
State Space:  30 * 2 = 2 ** 6
Progress Check...Safety property violation detected - check safety.
Progress Check in: 16ms
```

Figure 12. Simulation result of model with artificial error introduced.

```
PRODUCER = (buffAvail → (hiPriDataOut → PRODUCER | midPriDataOut → PRODUCER)).

const N     = 2
range Data_Range = 0..N
BUFFER = STATE[0][0],
STATE[a:Data_Range][b:Data_Range]
   = (when ((a+b)<N) buffAvail → (hiPriDataIn  → STATE[a+1][b] |midPriDataIn → STATE[a][b+1])
    |when (a>0) shortDataInBuff → nodeGrantFast → hiPriOut → (confirm → STATE[a-1][b]
                                                              |notConfirm → STATE[a][b])
    |when((a==0)&& b>0) dataInBuff → nodeGrantSlow → midPriOut → (confirm → STATE[a][b-1]
                                                                  |notConfirm → STATE[a][b])).
const M = 1
range Node_Requests = 0..M

SCHEDULER = STATE[0][0][0][0],
STATE[h1:Node_Requests][l1:Node_Requests][h2:Node_Requests][l2:Node_Requests]
                    = (when (((h1+l1+h2+l2)<M)) shortDataInBuff1          → STATE[h1+1][l1][h2][l2]
                      |when (((h1+l1+h2+l2)<M)) dataInBuff1               → STATE[h1][l1+1][h2][l2]
                      |when (((h1+l1+h2+l2)<M)) shortDataInBuff2          → STATE[h1][l1][h2+1][l2]
                      |when (((h1+l1+h2+l2)<M)) dataInBuff2               → STATE[h1][l1][h2][l2+1]
                      |when (h1>0) nodeGrantFast1 → (outBuffAvail1        → STATE[h1-1][l1][h2][l2]
                                                    |outBuffNotAvail1     → STATE[h1-1][l1][h2][l2])
                      |when (h2>0) nodeGrantFast2 → (outBuffAvail2        → STATE[h1][l1][h2-1][l2]
                                                    |outBuffNotAvail2     → STATE[h1][l1][h2-1][l2])
                      |when ((h1==0)&&(h2==0)&&(l1>0)) nodeGrantSlow1     → (outBuffAvail1
                                                                          → STATE[h1][l1-1][h2][l2]
                                    |outBuffNotAvail1                     → STATE[h1][l1-1][h2][l2])
                      |when ((h1==0)&&(h2==0)&&(l2>0)) nodeGrantSlow2     → (outBuffAvail2
                                                                          → STATE[h2][l1][h2][l2-1]
                                    |outBuffNotAvail2                     → STATE[h2][l1][h2][l2-1])).

||FINAL1 = ({b1, b2}:BUFFER||s1:SCHEDULER||p1:PRODUCER)/{
          b1.hiPriDataIn/p1.hiPriDataOut,b1.midPriDataIn/p1.midPriDataOut,b1.buffAvail/p1.buffAvail,
          b1.shortDataInBuff/s1.shortDataInBuff1,b1.dataInBuff/s1.dataInBuff1,
          b1.nodeGrantFast/s1.nodeGrantFast1,b1.nodeGrantSlow/s1.nodeGrantSlow1,
          b2.shortDataInBuff/s1.shortDataInBuff2,b2.dataInBuff/s1.dataInBuff2,
          b2.nodeGrantFast/s1.nodeGrantFast2,b2.nodeGrantSlow/s1.nodeGrantSlow2,
          s1.outBuffAvail1/b1.confirm,s1.outBuffNotAvail1/b1.notConfirm,
          s1.outBuffAvail2/b2.confirm,s1.outBuffNotAvail2/b2.notConfirm}.
```

Figure 13. FSP implementation for abstracted NOC model.

```
Compiled: BUFFER; Compiled: SCHEDULER; Compiled: PRODUCER
Composition:
FINAL1 = b1:BUFFER || b2:BUFFER || s1:SCHEDULER || p1:PRODUCER
State Space:  24 * 24 * 9 * 2 = 2 ** 15
Composing...-- States: 351 Transitions: 882 Memory used: 3043K
Composed in 110ms; FINAL1 minimising.....
Minimised States: 351 in 46ms
No deadlocks/errors; Progress Check...
-- States: 351 Transitions: 882 Memory used: 3397K
No progress violations detected. Progress Check in: 31ms
```

Figure 14. Simulation result for abstracted NOC model.

that all the buffers have same implementation. Additionally, any protocol does not change. Therefore, instead of showing three data paths (three buffers connected to node) we represented the model with two data paths (two buffers with one node).

It may be concluded that the number of buffers (as long as it is more than one) will not make a difference in addressing concurrency issues. However, it should be noted that a node with one buffer will not have the same implementation as a node with two buffers. Fig. 11 represents a further simplified model with a single data path for NOC.

4.5 Verifying a Deadlock/Livelock Situation

To prove that we can identify deadlock/livelock in a system with FSP modelling, we have introduced an artificial error in our FSP code shown in Fig. 4. While checking for the last condition in the buffer process of Fig. 4, we change the AND condition to a OR condition. Therefore, the system simulation will read this model as, if $(a = 0)$ or $(b > 0)$, i.e. there is a mid-priority data in the buffer and therefore the system will grant the node to store this data.

However, this process will be executed even when $(b = 0)$ and $(a = 0)$ i.e. there is no mid-priority data packet, but the system will still try to store this data. This must cause system violations. We verified this model. The simulation result of this model is shown in Fig. 12. It is clearly evident from the result that the system is in a deadlock/livelock situation.

5. Final Simulation and Results

Final abstracted model as shown in Fig. 11 has been described in Fig. 13. We can realize from the model implementation that there are one producer process – p1, two buffer processes – b1 and b2, one scheduler process – s1. The final composition process includes all the above processes. These processes have been exhaustively tested. Figure 11 shows the result for exhaustive simulation. It can be seen from the simulation result that there are 351 different possible states and 882 possible transitions among these states in the system. All these states are not involved in any deadlock and livelock.

6. Conclusion

We have proposed the use of a high-level modelling language "FSP" for exposing, evaluating, and modelling concurrency issues, at an abstract level. This methodology has been depicted by packet-based multiprocessing environment "NOC architecture" model. We verified our designed model using LTSA for any concurrency violations. This methodology may be adopted for use in other multiprocessor/multicore architectures. It is our experience that this abstract modelling will reduce undesirable interactions among concurrently executing components. It will further provide proper interfaces for components so that they can be integrated into a subsystem (and similarly, subsystems into a system) enabling substantial reuse at component, subsystem, and system levels. Therefore, it will enhance the system design productivity significantly.

References

[1] G. Desoli & E. Filippi, An outlook on the evolution of mobile terminals: From monolithic to modular multi-radio, multi-application platforms, IEEE *Magazine CAS*, *6*(2), 2006, 17–29.

[2] J. Ahmed Meine & W. Wayne, *Multiprocessor system-on-chips* (Amsterdam, Boston, London, New York, Tokyo: Morgan Kaufamann Publisher, 2005).

[3] L. Benini & G.D. Micheli, Networks on chip: A new SOC paradigm, *IEEE Computer*, *35*(1), 2002, 70–78.

[4] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Öberg, M. Millberg, & D. Lindqvist, Network on chip: An architecture for billion transistor era, *Proc. of IEEE NorChip Conference*, 2000, 8–12.

[5] D. Bertozzi & L. Benini, Xpipes: A network-on-chip architecture for gigascale system-on-chip, *IEEE Circuits and Systems*, *4*(1), 2004, 18–31.

[6] E. Cota, M. Kreutz, C.A. Zeferino, L. Carro, M. Lubaszewski, & A. Susin, The impact of NoC reuse on the testing of core-based systems, 21st *Proceedings of VLSI*, 2003, 128–133.

[7] A. Jantsch & H. Tenhunen. *Networks on Chip* (Boston, Dordrecht, London: Kluwer Academic Publisher, 2003).

[8] S. Kumar, A. Jantsch, J-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, & A. Hemani, A network on chip architecture and design methodology, *IEEE Computer Society Annual Symposium on VLSI*, 2002, 117–124.

[9] X. Jiang, W. Wolf, J. Hankel, & S. Charkdhar, A methodology for design, modelling and analysis for networks-on-chip, *IEEE International Symposium on Circuits and Systems*, 2005, 1778–1781.

[10] A. Agarwal & R. Shankar, A layered architecture for NOC design methodology, *IASTED ICPDCS*, 2005, 659–666.

[11] P.P. Pande, C. Grecu, M. Jones, A. Ivanov, & R. Saleh, Performance evaluation and design trade-offs for network-on-chip interconnect architectures, *IEEE Transactions on Computers*, *54*(8), 2005, 1025–1040.

[12] Semiconductor Industry Association, The international technology roadmap for semiconductors (ITRS) 2001, http://public.itrs.net/Files/2001ITRS/Home.htm.

[13] H. Sutter, The free lunch is over: A fundamental turn toward concurrency in software, *Dr. Dobb's Journal*, *30*(3), 2005.

[14] D.R. Butenhof, *Programming with POSIX threads*, (Boston, London, Paris, New York: Addition-Wesley, 1997).

[15] M. Guler, S. Clements, N. Kejriwal, L. Wills, B. Heck, & B.G. Vachtsevanos, Rapid prototyping of transition management code for reconfigurable control systems, *13th IEEE International Workshop on Rapid System Prototyping*, 2002, 76–83.

[16] J. Burch, R. Passerone, & A.L. Sangiovanni-Vincentelli, Overcoming heterophobia: modelling concurrency in heterogeneous systems, *IEEE International Conference on Application of Concurrency to System Design*, 2001, 13–32.

[17] G.H. Hilderink, Graphical modelling language for specifying concurrency based on CSP, *IEEE Proceedings on Software Engineering*, *150*(2), 2003, 108–120.

[18] S. Chrobot, Modelling communication in distributed systems, *IEEE International Proceeding in Parallel Computing in Electrical Engineering*, 2002, 55–60

[19] T. Murphy, K. Crary, R. Harper, & F. Pfenning, A symmetric modal lambda calculus for distributed computing, *Annual IEEE Symposium on Logic in Computer Science*, 2004, 286–295
[20] A. Girault, B. Lee, & E.A. Lee, Hierarchical finite state machines with multiple concurrency models, *IEEE Transaction on CAD of Integrated Circuits and Systems*, *18*(6), 1999, 742–760.
[21] M. Barrio & P.D.L. Fuente, A Formal Model of Concurrency for Distributed Object-Oriented Systems, *IEEE International Computer Science Conference on Software Engineering*, 1997, 466–474.
[22] D. Sangiorgi., *Expressing mobility in process algebras: First-order and higher-order paradigms*. Ph.D. thesis, Computer Science Department, University of Edinburgh, May 1993.
[23] V.D. Bianco, L. Lavazza, & M. Mauri, Model checking UML specifications of real time software, *8th IEEE International Conference on Complex Computer Systems*, 2002, 203–212.
[24] J. Magee & J. Kramer, *Concurrency state models and Java programs*, (West Sussex England: John Wiley & Sons, 1999).

Biographies

Ankur Agarwal is working as assistant professor and assistant director for the "Center for Systems Integration" in Department of Computer Science and Engineering at Florida Atlantic University, USA. He had pursued his Ph.D. and MS in Computer Engineering from Florida Atlantic University, USA and BE in electrical engineering from Pune University, India. He also holds postgraduate diplomas in VLSI design and embedded system designs. His main research interests include Network-on-Chip, system level designs and issues, embedded system design, VLSI design, FPGA design and mathematical modelling of real-time-operating systems.

Ankur Agarwal is the vice president of IEEE, and the chair of IEEE Computer Society, Palm Beach chapter. He is also a reviewer and an editor for International Journal of Computer Science and Engineering, International Journal of Electronics, Circuits and Systems, International Journal of Computer System Science and Engineering and Scientific International Journal of Engineering, Computing and Architectures. He has reviewed papers for IEEE portable conference, IEEE multimedia conference, and IEEE conference on embedded systems. Ankur Agarwal has published several journals and conference papers in low power processor designs, low power VLSI implementations of ALU, Network-on-Chips architecture, and power management at operating system level. He is an IEEE member. He is also Co-PI of two research grants (One Pass to Production granted by Motorola and HSM project granted by FAU) at Florida Atlantic University.

Ravi Shankar is working as a professor and the director for the "Center for Systems Integration" in Department of Computer Science and Engineering at Florida Atlantic University, USA. He has a BS in Telecommunications Engineering from Karnataka University, Dharwar, India, and MS and Ph.D. degrees in Electrical and Computer Engineering from the University of Wisconsin, Madison, WI. His areas of specialization for the Ph.D. were Computer Engineering and Biomedical Engineering. He joined his Electrical and Computer Engineering Department of Florida Atlantic University (FAU), Boca Raton, FL, in 1982, and has been a professor with the Computer Science and Engineering Department at FAU since 1991. He is a senior member of IEEE. Dr. Shankar holds Professional Engineering (PE) license in the State of Florida. He has co-authored a book and has published several journal and conference papers. He is the founder of center of system integration at Florida Atlantic University. He has received various prestigious grants. He has been a consultant to IBM, Aptek, and Motorola, since 1985. In 1993, Motorola recognized the relevance of Dr. Shankar's efforts in EDA and helped him establish the Center for VLSI and Systems Integration (CVSI) with generous grants, EDA Vendor tools, NSF matching funds, and access to Motorola resources. Dr. Shankar's related research expertise is in mixed mode system design, structured design, and EDA tool integration.