Formal Models of Guaranteed and Best-Effort Services for Networks on Chip *

Biniam Gebremichael Frits Vaandrager

Miaomiao Zhang†

{B.Gebremichael,F.Vaandrager,miaomiao}@cs.ru.nl

Institute for Computing and Information Sciences

Radboud University Nijmegen

The Netherlands

March 15, 2005

Abstract

Networks on a chip (NoC) are emerging as a scalable, compositional and efficient alternative to existing on-chip interconnects (such as point to point networks or buses). ÆTHEREAL is a protocol that has been proposed by Philips to enable both guaranteed and best effort communication in an on-chip packet switching network. We present a formal specification of the ÆTHEREAL protocol and its underlying network. All components of the network and their behavior are specified in detail, using the PVS specification language. Using PVS we prove, for an abstract version of our model, absence of deadlock within the ÆTHEREAL protocol.

*This work was supported by PROGRESS project TES4199, Verification of Hard and Softly Timed Systems (HaaST).

†Currently affiliated with Tongji University, China, miaomiao@tongji.edu.cn.

Contents

1 Introduction

Within the next 5 years, silicon technology will allow chip complexities of up to 1 billion transistors and 1 megabyte of embedded software [HaSTC04]. With this technology, we can virtually build a whole system (processors, memories, communication infrastructure, input and output interfaces, etc) on a single chip. The traditional bus-architecture or point to point wiring (PPW) between intellectual-property blocks (IPs) has been the standard hardware architecture for embedded systems. In dedicated PPW or buses, IPs are connected to all possible communication partners, which leads to many global wires, which in turn leads to cross talk, wire spacing, congestion, non-scalability and non re-usability problems [GvMPW02, RG04]. This makes impossible to integrate as many transistors as future applications require.

Recently, an alternative architecture —*network on chip* (NoC) [HJK+00]— has been introduced that uses packet-switching for communication between IPs. The main advantage of NoC is that it reuses communication wires instead of dedicating a single wire for every two IPs in the system. NoC borrows its architecture from TCP/IP network where links between two nodes are shared, and routers are used to program the exclusive (time-wise) use of the links. Again similar to problems in TCP/IP, sharing communication infrastructure may lead to cyclic waiting and deadlock. The problem of deadlock is even more severe in NoC than in TCP/IP since NoC is all in hardware and it is required to provide guaranteed services. Therefore it is extremely important to have a protocol that realizes correct functionality and at the same time avoids any circumstances that lead to a deadlock situation. One such protocol is the ÆTHEREAL protocol, which has been designed to meet both the functionality and the correctness requirement of the network on chip [GRG+05].

Designing a network on chip and a protocol (such as ÆTHEREAL) which meets all the requirements for best-effort and guaranteed traffic is a difficult task. During the design process, the designers play around with thousands of design alternatives before they commit to one. It is difficult to keep track of all these design alternatives in a systematic way, and to make sure that the choices that have been made are consistent. The contribution of this paper is that for one of the numerous design alternatives we produced a detailled, precise formal model. Within this model we were able to establish a key correctness criterion for the absence of deadlock. It is very unlikely that the design as we have formalized it will be the one that is actually going to be implemented in hardware. However, since our specification is highly abstract and very modular, it is relatively easy to modify our specification to reflect variations of the design.

In fact, we believe that our work illustrates that formal specification languages, such as the typed higher-order logic supported by PVS, can be most useful to document complex designs, to help designers to clarify design choices and to resolve problematic inconsistencies in an early stage of the design process.

Related Work

Our work is based on the documents [RGAR+03, NPR+02, GvMPW02, RG04, GRG+05] provided to us by Philips in the context of PROGRESS project HaaST,

and personal and email communications with the designers from Philips Research.

There are numerous articles in the literature showing successful application of formal specification and analysis to complex protocols that are designed on top of complex architectures. For example, the IEEE 1394 architecture [IEE96] has been formally specified and/or verified in several articles [vLRG03, DGRV00, Gri00]. The futurebus+ cache coherence protocol was formally specified and verified by Clarke et al. [CGH$^+$93]; during this verification a previously undetected bug was detected. To the best of our knowledge, only very little work has been done on the formal specification and verification of protocols for networks on chip. A notable exception is the work of Julien Schmaltz [Sch04], who gave a functional specification of the Octagon network on chip (the name is derived from the specific octagon shaped topology of the network) and verified termination of the routing algorithm using the ACL2 theorem prover. Our work differs from that of Schmaltz in that the ÆTHEREAL protocol supports general topologies for packet switching networks for which the challenge is to prove absence of deadlock rather than termination of routing.

We have used the Cadence SMV model checker [McM93] in the early stage of our modeling process to simulate the normal operations of the network. The SMV model also allowed us to rediscover deadlock scenarios that occur in variations/simplifications of the protocol. Nevertheless, we believe that model checkers are of limited relevance for this type of case studies because (a) the design is highly parameterized (network topology, routing functions, choice of buffers) and with a model checker one can only analyze one model at a time, after fixing specific values for all the parameters, (b) for nontrivial instances of the protocol, the state space becomes so big that even for a state-of-the-art full state space analysis becomes very difficult, if not impossible. Our preference to use PVS [OSRSC99, COR$^+$95] is mainly due to the fact that its specification language, which is based on classical, typed higher-order logic, is extremely expressive.

Paper Organization

The paper is organized as follows. Section 2 presents an informal introduction to the network on chip architecture and the ÆTHEREAL protocol. The PVS specification language that we use to formally specify the architecture and the protocol is introduced in Section 3. Sections 4 and 5, respectively, describe the topology of the network and the data structures used in the network. Communication and computation operations that take place within the network are described in Section 6. Fragment of PVS code are presented to illustrate the data types and the operations. Section 7 models the ÆTHEREAL protocol as a synchronous state machine, where states are the configuration of the network (the value of the buffer and the time slot counter), and transitions correspond to the computation and communication operations of the network. Using the state machine approach a proof of absence of deadlock is presented in Section 8 for an abstract version of the model. In Appendix A all the PVS theories of our model are listed. The PVS sources are also available in electronic form at `http://www.cs.ru.nl/ita/publications/papers/biniam/noc/`.

2 Network on Chip

Network on Chip architecture is packet-switching network that provide a programmable and efficient communication infrastructure. NoC (as shown in Fig. 1) is composed of several nodes or intellectual-property blocks (IPs) and the link or wires connecting the IPs. NoC is connected to the upper layer (application layer) through network interface IPs. A network interface can either be Active network interface IP (ANIP) or Passive network interface IP (PNIP) depending whether the application with which it is connected to is an active (sender) or passive (receiver) respectively. NOC has also a number of intermediate IPs or routers that directly or indirectly connect ANIPs with PNIPs. The internal part of the IPs are described in detail in Section 4.

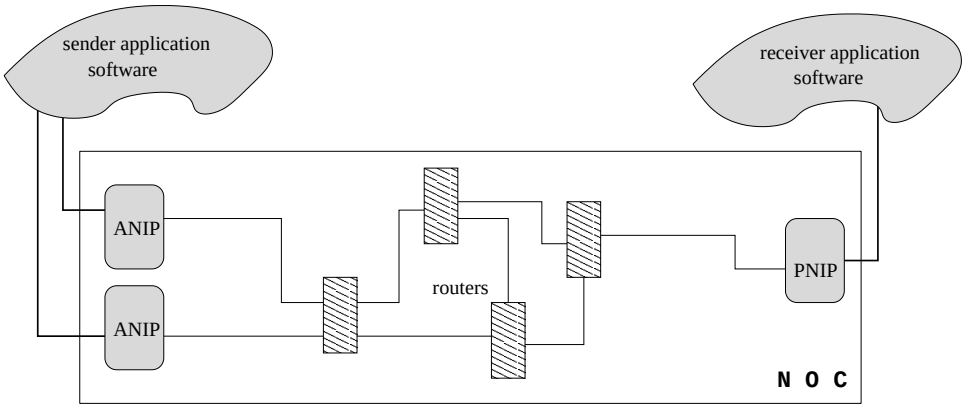

Figure 1: The network and application layer in a NOC integrated system

2.1 The ÆTHEREAL Protocol

It is easy to proved a guaranteed service in a point to point wiring network, where as in a packet-switching networks it is very difficult. The ÆTHEREAL protocol as described in [NPR+02] supports both guaranteed and best-effort services.

2.1.1 Guarantee Service Programming

Guaranteed service require reservation of resource as to insure data integrity, loss less and order preserving data delivery, while best-effort services does not require reservation of resources as no assurance are meant to be given. ÆTHEREAL protocol is designed for both best effort (BE) and guaranteed-throughput (GT) services. BE services are easy to use, while GT services require careful programming to reserve the required resources in the network. This section shows how GT connections are setup and torn down by means of BE packets.

To avoid contention during GT services, every router maintains a slot table with a slot number as a row and the routers output ports as a column. This table is used to keep track which output port is reserved at which time slot. Initially the slot table of every router is empty. There are two system packets called: `SETUP` and `TDOWN` to reserve resources using the slot table. There are

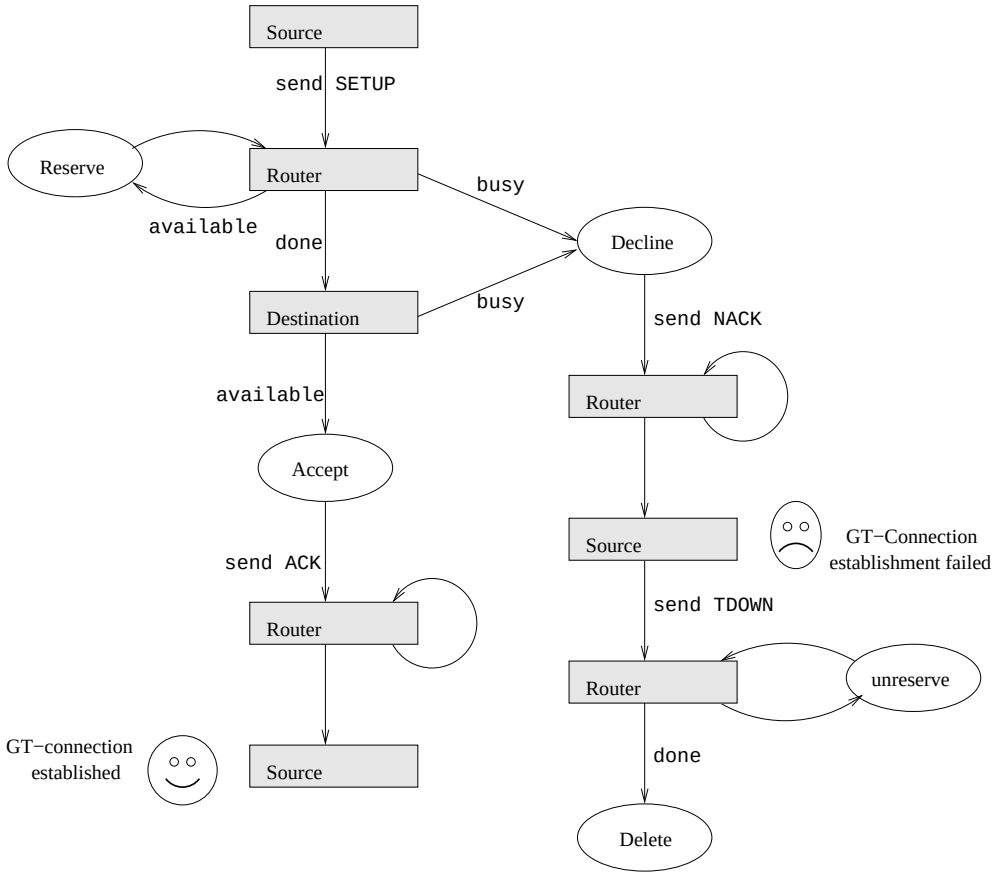

Figure 2: GT-connection establishment

two additional acknowledgment packets `ACK` (positive acknowledgment) and `NACK` (negative acknowledgment) used to acknowledge the success or failure of resource reservation.

2.1.2 Establishing a GT-connection

In order for an application software to establish a guaranteed through put connection (GT-connection) the application software should send a `SETUP` packet. A `SETUP` packet will then attempt to creates a GT-connection from the source to the destination. Every router along the path of the `SETUP` packet checks if the output port to the next node in the path is free in the slot indicated by the packet. If it is free the output is reserved in the slot table and the `SETUP` packet is forwarded to the next router. Otherwise, the `SETUP` packet is bounced as `NACK` packet back to its source. `NACK` packets may have to pass through a number of routers before they reach their destination, but they do not try to unreserve those output ports reserved earlier. When `SETUP` packet arrives to its destination, then packet is acknowledged by returning an `ACK` or `NACK` packet to the source. If it is `ACK` then the GT connection has been established successfully, otherwise we say that the attempt to establish a connection has been failed. the source node should undo all reservation upto the node where the `SETUP` packet bounced back. This unreser-

vation is done by sending `TDOWN` packet. No acknowledgement is necessary for `TDOWN` packets, because it is guaranteed that any packet arrives to its destination eventually [NPR+02].

Once the GT connection is established GT-packets can flow from the source to the destination along the reserved output ports without difficulty.

In the reminder of the paper we will only talk about BE-packets, and we mean BE-packets when we say packets, unless otherwise specified. Figure 2 summarizes the GT-connection establishment.

3 The PVS Specification and Verification System

PVS is a verification system with an interactive environment for writing formal specifications of systems and checking formal proofs. Only the relevant features of the tool is explained below. For detailed information about the tool we refer the reader to the PVS System Guide, the PVS Prover Guide and the PVS Language Reference available at `http://pvs.csl.sri.com`

3.1 Model Specification with PVS

The specification language of PVS is built on a higher-order logic, and it provides a rich set of built-in constructs for expressing a variety of notions.

3.1.1 PVS Theory

The PVS specification of a system is organized as a collection of theories. A theory is essentially made of declarations, which are used to introduce types, constants, variables, formulas etc.. A theory may have or may not have parameters as an input to instant with. It is possible for a theory to be imported by another theory using the key word `IMPORTING`. By importing a theory it will give the importer theory access to all declaration of the imported theory. The following code shows a typical PVS theory

```
port[P:TYPE]: Theory
Begin

% ... BODY

END port
```

Where `port` is the name of the theory and `Theory, Begin, End` are PVS key words to define theory. The theory `port` has one parameter `[P: TYPE]`. Any text that appears after the percentage sign (%) is comment.

3.1.2 Declaration

The PVS specification language is equipped with the usual base types such as `int`, `bool`, `nat`; and more complex types such as function type constructor (`[A -> B]`) record type constructor (`[# a:A, b:B #]`).

```
i: VAR int
point: VAR [#i:nat, j:nat#]
```

3.1.3 Types and sub types

PVS also allows new type name declaration using the key word `TYPE`. This declaration form the simplest type expression and can be used to construct more complex type expressions called *subtypes*. A distinctive feature of the PVS specification language are *predicate subtypes* – the subtype {`x:A | P(x)`} consists of exactly those elements of type `A` satisfying predicate `P`. Predicate subtypes are used, for instance, for explicitly constraining the domains and ranges of operations in a specification and to define partial functions. The following PVS code exemplifies the use of `TYPE` and subtype declaration

```
%new aninterpreted type.
INPORT: TYPE
OUTPORT: TYPE

% import the theory port
IMPORTING port[INPORT]
IMPORTING port[OUTPORT]

%new enumerated type
PortType: TYPE={DMY,APPLICATION,NORMAL,SPECIAL}

%new type name -  a record of two elements
node:= TYPE [# in:INPORT,
        out: OUTPORT #]

% a subtype of node
router: TYPE {n:NODE | is_router(n)}
```

Note that subtyping is a powerful specification concept, since a lot of information can be encoded during declaration. This concept has been used extensively in our model.

3.1.4 Axioms and Theorems

when it is not possible to encode the desired property of the type during type declaration. Or when there is a need for restricting the relationship between entries, then it can be represented as axioms. In PVS axioms are introduced using the keyword `AXIOM`, and they are assumed to be `TRUE`. Theorems can be introduced using many equivalent keywords such as `LEMMA` or `THEOREM`. These correspond to properties we want to prove.

For instance the following PVS code defines two mapping functions (`peer_oi` and `peer_io` and two axioms on the functions. The axioms state that `peer_io` is inverse of `peer_oi` and `peer_oi` is surjective. Based on these axioms we may want to prove `peer_oi` is injective, hence we write this claim as PVS `LEMMA`

```
%function declaration
peer_oi:[OUTPORT -> INPORT]
peer_io:[INPORT -> OUTPORT]

%facts about the functions
peer_io_ax: AXIOM peer_io(peer_oi(o1)) = o1
peer_io_ax1: AXIOM surjective?(peer_oi)

%Theorem ( lemma )
peer_lemma: LEMMA injective?(peer_oi)
```

3.1.5 Built-in prelude

The PVS tool also contain a number of built-in *prelude* and loadable *libraries*. These provide standard specifications and proved facts for a large number of theories including set theory, lists, sequences, finite sets and more. For instance the definition if `surjective?` and `injective?` predicates used in the above example are built-in predicates that reside in PVS prelude file under `functions` theory.

3.2 Proving with PVS

The PVS Prover tool provides a variety of commands to construct the proofs for a give theorem. Typically, the PVS Prover is used interactively to construct the proof of a theorem and it uses tree-like style to represent the theorems that are proven. The aim of the user will be to construct a proof tree that is complete, in the sense that all leafs are recognized as `TRUE`.

We will not use the PVS Prover in our model. Though we have some theorems proven using the PVS Prover, we will only give sketch of the prove and we will not use PVS syntax to explain the proof. We refer the reader to the PVS Prover Guide for more information.

4 Network Topology

Like any other network, NOC, is a directed graph with a set of vertices and edges. Typically the set of vertices in NOC are nodes: such as routers and network interfaces. Each node in the network has a set of ports that connect the node with its neighboring nodes.

4.1 Ports

Ports are interface of nodes through which a node communicate to the rest of the network. Ports are classified according to their use. These are application ports (interface to the application layer), normal ports (ordinary ports between two peer nodes), special ports (ports with special purpose) and a dummy port (a single port introduced for modeling convenience)[1].

A port is formally defined as a parameterized PVS theory as follows. Where `[P:TYPE]` is uninterpreted type as parameter to the theory , and (`PortType`) is an enumerated type for the four classification of ports listed above.

```
Port[P:TYPE]: Theory
Begin

%types of a port
PortType: TYPE={DMY,APPLICATION,NORMAL,SPECIAL}
porttype:[P->PortType]
```

We also introduce several predicates to test the type of a port such as:

[1] It will be clear soon in the subsequent sections why we have so many types of ports in our specification

```
%port type testing predicates
dmy(p:P):bool = porttype(p)=DMY
app(p:P):bool = porttype(p)=APPLICATION
sp(p:P):bool = porttype(p)=SPECIAL
normal(p:P):bool = porttype(p)=NORMAL
nport(p:P):bool = sp(p) OR normal(p)
notdmy(p:P):bool = nport(p) OR app(p)
```

The complete PVS specification of theory port is in Appendix A.1

4.2 Nodes

The IPs of the network (or nodes) are the vertices of the network which consists two types of ports. These ports are input and output ports. A preliminary definition of a node can be given as *a tuple of set of non dummy input ports, and set of non dummy output ports* And for two sets s_1 and s_2 of non dummy input ports or non dummy output ports, either $s_1 \cap s_2 = \emptyset$ or $s_1 = s_2$ and it belong to one node.

Using the PVS `TYPE+` keyword we can introduce `INPORT` and `OUTPORT` as a non empty and uninterpreted input and output ports respectively. By importing the theory port on both types we declare that `INPORT` and `OUTPORT` are of type port.

The following PVS code defines `PreNode` as stated above. A more specific definition of a node is given at the end of the section. We will use the built-in set functions to specify NOC nodes and the different types of nodes.

```
INPORT:TYPE+
OUTPORT:TYPE+

IMPORTING port[INPORT]
IMPORTING port[OUTPORT]

PreNode: TYPE = [#  inport: set[(notdmy?[INPORT])],
                 outport: set[(notdmy?[OUTPORT])]
        #]

n1,n2: VAR PreNode
is1,is2: VAR set[(notdmy?[INPORT])]
os1,os2: VAR set[(notdmy?[OUTPORT])]

disjoint_ax1: AXIOM
   FORALL is1,is2:
    (EXISTS n1,n2:     inport(n1) = is1 AND    inport(n2) = is2)
    IMPLIES    (n1 = n2 OR  disjoint?(is1,is2))

disjoint_ax2: AXIOM
   FORALL os1,os2:
    (EXISTS n1,n2:     outport(n1) = os1 AND    outport(n2) = os2)
    IMPLIES    (n1 = n2 OR  disjoint?(os1,os2))
```

4.3 Peer

Nodes are connected in the network by a link that connects one outport of a node with another input port of a node.

In NOC all ports, except the dummy and application ports, (called the `nport` ports) have a peer port in another node. The function peer is defined using two mapping function for both input and out put ports – namely `peer_oi` and `peer_io`. These functions are total and one function is the inverse of the other. one as follows.

```
peer_oi:[OUTPORT -> INPORT]
peer_io:[INPORT -> OUTPORT]

i1:VAR INPORT
o1:VAR OUTPORT

peer_io_ax: AXIOM peer_io(peer_oi(o1)) = o1
```

It is possible to prove interesting property of the peer function. We prove for instance that the peer functions are injective and the inverse of axiom `peer_io_ax` also holds.

```
peer_inj_lemma1: LEMMA injective?(peer_oi)
peer_inj_lemma2: LEMMA injective?(peer_io)

peer_oi_lemma: LEMMA peer_oi(peer_io(i1)) = i1
```

The peer theory is imported to our NOC for the `nport` types.

```
IMPORTING peering[(nport?[INPORT]),(nport?[OUTPORT])]
```

The complete PVS specification of peer is in Appendix A.2

4.4 Node Types

There are three types of nodes. Namely: network interface, `RCU` (reconfiguration unit) and router. The network interface is further classified into `ANIP` (Active Network interface Intellectual-Property block) and `PNIP` (Passive Network interface Intellectual-Property block).

4.4.1 Network Interface (NI)

Network Interfaces are nodes with one application and one special pairs of port. These node are the end points of the network from which data is enters and leafs the network. Formally NI is defined as:

```
appi: VAR (app[INPORT])
appo: VAR (app[OUTPORT])
spi: VAR (sp[INPORT])
spo: VAR (sp[OUTPORT])

NI: TYPE = {a:PreNode | (EXISTS appi,spi: inport(a) = add(appi,singleton(spi)))
            AND (EXISTS appo,spo: outport(a) = add(appo,singleton(spo)))
        }
```

- Active Network interface IP (ANIP): is a network interface, also known as *producer*, through which application software send messages to the network.

  ```
  appi: VAR (app?[INPORT])
  ANIPS: TYPE = set[NI]
  anips: ANIPS
  anip(a:PreNode):bool = member(a,anips)
  ```

- Passive Network interface IP (PNIP): is a network interface, also known as *consumer*, through which application software are connected to the network, and receive messages.

  ```
  PNIPS: TYPE = set[NI]
  pnips: PNIPS
  pnip(a:PreNode):bool = member(a,pnips)
  ```

4.4.2 Router

Routers are intermediate nodes that route packets toward their destination. A router is defined as a node with a set of normal ports and one special port.

```
rseti: VAR set[(normal[INPORT])]
rseto: VAR set[(normal[OUTPORT])]

ROUTER: TYPE = {a: PreNode | (EXISTS spi,rseti: inport(a) = add(spi,rseti))
                  AND (EXISTS spo,rseto: outport(a) = add(spo,rseto))
        }
```

4.4.3 Reconfiguration Unit (RCU)

RCU is a node coupled with a router, where the programming of the guaranteed service takes place. This node has single input and output port. They are of special type and they are also connected to the parent router through the parents special ports.

One reason for identifying this ports as special is to uniquely name router ports that are connected to the associated RCU. The following PVS definition captures the representation of the RCU in the network.

```
RCU: TYPE = {a:PreNode | (EXISTS spi: inport(a) = singleton(spi) AND
                  let spo2:OUTPORT = peer_io(spi) IN
               router(node(spo2)) AND sp(spo2))
           AND (EXISTS spo: outport(a) = singleton(spo) AND
                  let spi2:INPORT = peer_oi(spo) IN
              router(node(spi2)) AND sp(spi2)
             )
            AND (FORALL spi,spo: member(spi,inport(a)) AND
             member(spo,outport(a)) AND EXISTS n1:
             n1= node(peer_io(spi)) AND n1= node(peer_oi(spo)))
        }

RCUS: TYPE = set[RCU]
rcus: RCUS
rcu(a:PreNode):bool = member(a,rcus)
```

4.4.4 Node Definition

Finally we can give the definition of a node as a union of these four sets.

```
ni(n:PreNode):bool = anip(n) OR pnip(n)
IP(n:PreNode):bool = ni(n) OR router(n) OR rcu(n)
NODE: TYPE = (IP)
```

We also define a function that returns the node of a given input or output port. We assume that, there exists a node n for every port in the network p such that node of p is n. In PVS it is specified as axioms on the function `node` for both input and output ports separately, as shown below.

```
n3: VAR NODE

node(i0:(notdmy[INPORT])):NODE
node_ax_i: AXIOM node(i0) = n3 IMPLIES member(i0,inport(n3))
node_ax2_i: AXIOM FORALL i0: EXISTS n3: node(i0)=n3

node(o0:(notdmy[OUTPORT])):NODE
node_ax_o: AXIOM node(o0) = n3 IMPLIES member(o0,outport(n3))
node_ax2_o: AXIOM FORALL o0: EXISTS n3: node(o0)=n3
```

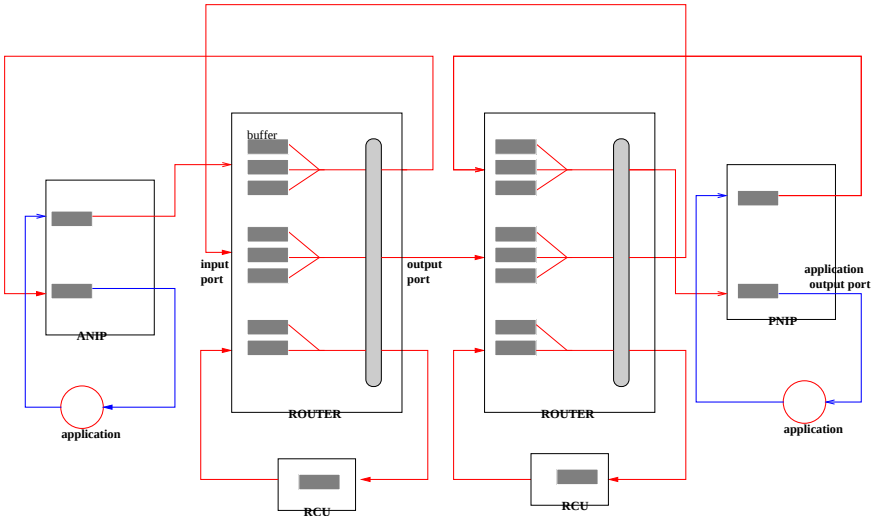

Figure 3: A typical network topology for an ANIP, PNIP, two routers, two RCUs and application software

Figure 3 shows an example of NOC topology with an ANIP, PNIP, two routers, two RCUs and application software connected to the network interfaces. The dark box inside the nodes are buffers explained in section 4.6. In Appendix A.3 there is a complete PVS specification of node and node types.

4.5 Slot Table

Another entry of the NOC architecture is the slot table. A slot is a time step in the progress of the network as explained in section 2.1.1. It is represented as a natural number. A slot table is, thus, a table that records which output port of a router is reserved by which input port of that router. The need for a slot table is to avoid multiple guaranteed service packets or best effort packets use the same port at the same time.

Each router has a slot table and it is maintained by the associated RCU node of the router. Slot table is defined as

```
SLOT_TABLE: TYPE = [SL_OUTPORT,nat -> SL_INPORT]
```

Where SL_OUTPORT and SL_INPORT are defined as

```
% ROUTER port types
r_port(p:P):bool = normal(p) OR sp(p)

%The output port is from the router
SL_OUTPORT: TYPE = {p:OUTPORT | router(node(p)) AND r_port(p)}

%The input port is from the router or dmy
% dmy (dummy input port) means the outport is not reserved
SL_INPORT: TYPE =  {p:INPORT  | dmy(p)
                OR ( router(node(p)) AND r_port(p))}
```

Given a slot table we can read whether an output port is reserved or unreserved as in the following PVS predicates

	o_1	o_2	o_3	o_4	...
s_1	i_1	dmy	dmy	i_2	...
s_2	i_3	dmy	i_1	i_2	...
s_3	i_3	dmy	dmy	i_2	...
s_4	dmy	dmy	dmy	dmy	...
...	...	...	...	...	...

Table 1: Example of a slot table

```
reserved_by(stb:SLOT_TABLE,o1:SL_OUTPORT,st:nat,i1:SL_INPORT):bool =
    notdmy(i1) AND i1=stb(o1,st)

unreserved(stb:SLOT_TABLE,o1:SL_OUTPORT,st:nat):bool =
    dmy(stb(o1,st))

already_reserved(stb:SLOT_TABLE,o1:SL_OUTPORT,st:nat):bool =
    notdmy(stb(o1,st))
```

Example 1 *A typical slot table in NOC would look like the one in table 1, where output ports are give as a row and time slots as columns. The* dmy *variable represents that the output port is not reserved during the given time slot, while* i_1 *means the outport is reserved by* i_1 *during the given time slot.*

The complete PVS specification od slot table can be found in Appendix A.4

4.6 Buffer Address

There are several buffers in a node. Each buffer in a node is associated with the input and output port of the node. A buffer can be uniquely identified by the ports it is associated. Thus we can define buffer as a record of input and output port it is associated with.

```
BUFFER: TYPE =
    [# inport:(notdmy[INPORT]),
        outport:{o1:(notdmy[OUTPORT]) | node(o1) = node(inport)} #]
```

To find the node of a buffer, it can easly derived from one of its port.

```
node(b):NODE = node(inport(b))
```

We will make use of several predicates in the higher theories of our model that identifies the type of buffers that are specific to specific types of node. These definition is given in table 2. The record type definition such as: `[#app,sp#]`[2] that appear in table 2 represent with which port type the buffer is made up of. Table 2 can also be coded as a PVS specification. For instance `anip_sys_buffer` can be defined as in the following predicate. The complete PVS definition for all buffer types is in Appendix A.5

```
% [app,sp] buffer - for NI
ni_sys_buffer(b):bool =  app(inport(b)) AND  sp(outport(b))

%anip system buffer
anip_sys_buffer(b):bool = ni_sys_buffer(b) AND anip(node(b))
```

[2]this buffer is made up of application input port (`app`) and special output port (`sp`)

Node	system buffers	acknowledgment buffers	both
ANIP	anip_sys_buffer [#app,sp#]	anip_ack_buffer [#sp,app#]	
PNIP	pnip_sys_buffer [#sp,app#]	pnip_ack_buffer [#app,sp#]	
RCU	rcu_buffer [#sp,sp#]		
ROUTER	router_sys_buffer [# normal ,sp#]	router_ack_buffer [# normal,normal#]	router_from_rcu_buffer [#sp,normal#]

Table 2: Buffer types

5 Network Data

In This section we define the different types of packets, buffer contents and operations on these data. We will not define the sequence of the operations, the next section is dedicated for explaining which operation executes when and which operation follows.

5.1 Packet

Communication in the network takes place by sending and receiving packets. There are two types of packets in the network – Guaranteed service packet (`GT packets`) and best effort (`BE packets`). As stated in the introduction, this document deals only with BE service. We mean BE packet whenever the word packet is used.

5.1.1 Packet Types

Depending on the purpose of the packet, a packet has one of the five possible types. These types are:

- **SETUP:** For establishing a connection. A setup packet reserves output ports (or links) along the path for the upcoming GT-connection.
- **TDOWN:** To tear down a connection. A TDOWN packet cancels all reservation done by the predecessor SETUP packet. TDOWN packets are sent whenever the SETUP packet fails to establish a complete connection.
- **ACK:** A positive acknowledgment for a successful connection establishment.
- **NACK:** A negative acknowledgment for a failed connection establishment
- **EMPTY:** The packet is empty. The easy way to remove a packet is to change its type to EMPTY.

Packet type is defined using PVS enumerated declaration

```
PACKET_TYPE: TYPE = {SETUP,TDOWN,ACK,NACK,EMPTY}
```

5.1.2 Packet fields

Besides the packet type, a packet contains four more fields. These are the source and destination nodes, a slot number that a packet want to reserve and a hub counter that counts the number of nodes the packet already passed though. The source and destination nodes are always network interfaces. Slot number and hub counters are of type natural numbers.

```
PACKET_TYPE: TYPE = {SETUP,TDOWN,ACK,NACK,EMPTY}
PACKET: TYPE = [# ptype: PACKET_TYPE,    % type of the packet
          psrc:  (ni?),        % source (ANIP or PNIP)
          pdes:  (ni?),        % destination (ANIP or PNIP)
          pslot: nat,        % initial slot to reserve
          phub:  nat         % hub counter
           #]
```

The packet theory also contains some predicates to test the type of packets and other functions that operate on packet. Operations on packets are explained later in the communication section where they are refereed. The complete PVS specification is in Appendix A.6

A packet is called *system packet* if it deals with reserving and unreserving output ports along the path to the destination. `SETUP` and `TDOWN` packets are system packets. `ACK` and `NACK` packets known as *acknowledgment packets* collectively, do not deal with reserving or unreserving output ports, instead they are simply sent to inform the source, that the desired connection has or has not been established.

System packets have `ANIP` as a source node and `PNIP` as a destination node. While acknowledgment packets have `PNIP` as a source node and `ANIP` as a destination node. Note that `NACK` packets can be sent from a router and `TDOWN` packets do not reach the destination `PNIP`, despite this fact the protocol excludes using routers as source or destination to avoid further complexity.

Formally system and acknowledgment packets are defined as:

```
%let p be any packet
p: VAR PACKET

%system packet
sys_packet(p):bool = (ptype(p)=SETUP OR ptype(p)=TDOWN ) AND
            pnip(pdes(p)) AND anip(psrc(p))

%acknowledgment packet
ack_packet(p):bool =  (ptype(p)=NACK OR ptype(p)=ACK ) AND
            anip(pdes(p)) AND pnip(psrc(p))
```

5.2 Buffer Content

Buffer address was defined in section 4.6, in this section we define the content of the buffer.

The content of a buffer (`BUFFERD`) is a queue of packets. The maximum number of packets a buffer can contain is not constant. The `capacity` function, defined below, gives the capacity of a given buffer.

```
BUFFERD: TYPE = Queue[PACKET]

CAPACITY:  TYPE = nat
capacity:[BUFFER->CAPACITY]
```

As a queue, it is possible to add, remove, check for empty or read the length, using the following predicates

```
Queue [T: TYPE+] : THEORY
BEGIN
    enqueue: [T, Queue -> (nonempty?)]
    dequeue: [(nonempty?) -> Queue]
    length: [Queue -> nat]
    empty: (empty?)
```

Appendix A.7

5.3 Links

When a packet is sent from one node to another there will be a delay in time, and the wire connecting the two nodes can hold the packet for the time delay. In another word the wire has a buffer (called a `link`) that can accommodate at most one packet (see fig 4). When there is no packet in a `link` we say that the link contains an empty packet. In PVS terminology link is a function from a `nport` (normal or special) input port to a packet.

```
link:         [(nport[INPORT]) -> PACKET]
```

5.4 Flow control

In the ÆTHEREAL network protocol, flow control is implemented in two parts – local flow control using local credit, and end to end flow control using end to end credit.

5.4.1 Local Credit and Flag

Local credit (`lcredit`) is situated in output ports of the sender node (see fig 4), and records how many space is left in the peer buffer of the receiver node.The advantage of local credit counter is that, whenever a packet is sent, the sender is sure that the destination buffer has a space to accommodate it. This frees the network from acknowledgment message overheads.

Physically a local credit is situated in out port of a node, but in out PVS definition local credit is defined using the buffer it refers too. This will not make any problem except that it will only make the PVS functions that deal with local credit shorter.

```
LOCAL_CREDIT: TYPE = [ BUFFER_A -> nat]
```

More over there is a flag associated with every buffer in the network (see fig 4). A flag is a boolean variable that is raised (`true`) when the buffer send a packet and lowered(`false`) otherwise.

```
flag: [BUFFER->bool]
```

Initially `lcredit` is equal to the length of the buffer it refers to. Updating a local credit of a buffer is done by two separate processes.

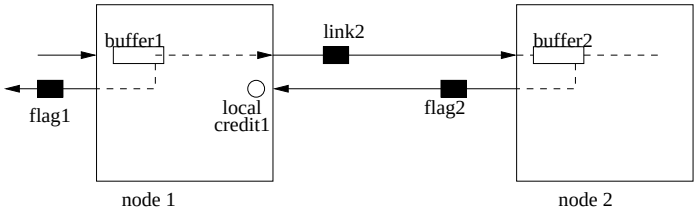

Figure 4: Local flow control, links and flag

- **Incrementing:** When a node (for example node 1 in fig 4) wants to update its local credit then it will read the flag of the corresponding buffer (flag 2) and if it flag is raised the local credit will be incremented by one, otherwise it remains unchanged.

```
credit_up_lcredit(lcredit:nat,flag:bool):nat =
         if(flag) then    lcredit+1
    else         lcredit
    endif
```

- **Decrementing:** If a node (or buffer) is sending a packet to the peer node (buffer) then the local credit associated with the peer buffer is decremented. (Note that the flag associated with the sender buffer will be raised as explained in the above.) Otherwise if nothing is sent the local credit remains unchanged.

```
credit_down_lcredit(lcredit:nat,receiving:bool):nat =
         if(receiving) then    lcredit-1
    else             lcredit
    endif
```

Example 2 *Assume A packet is sent from* $buffer_1$ *to* $buffer_2$ *via* $link_2$ *in fig 4, then* $flag_1$ *will be raised because* $buffer_1$ *has freed one space. Local* $credit_1$ *will be decremented by one since* $buffer_2$ *have lost one space.*

5.4.2 End to End Credit

The second credit, end to end credit, is only for `ANIP`. End to end credit is introduced to prohibit `ANIP` from sending more `SETUP` packets than the space it has to receive the acknowledgments.

The initial value of end to end credit is the capacity of the smallest buffer in the `ANIP`. Every time the `ANIP` sends a `SETUP` packet the credit is decremented, and every time the `ANIP` frees (consume) `ACK` or `NACK` packet from the input buffer the credit is incremented. See section 7.2 for formal PVS definition of end to end credit.

5.5 The Complete List of Network Data

The of the entire network is characterized by the values of the following variables.

```
DATA: TYPE = [#
    buffer:         [BUFFER -> Queue[PACKET]],
    seli:          [(nport[OUTPORT]) -> INPORT],
    flag:          [BUFFER->bool],
    link:          [(nport[INPORT]) -> PACKET],
    lcredit:    [ BUFFER -> nat],
    eecredit:      [(anip_sys_buffer) -> CAPACITY],
    slot_table:      SLOT_TABLE
    #]
```

1. **buffer:** The value of all buffers in the network. A buffer may contain zero or several packets queued in the order of their arrival. Initially all buffers are empty

2. **seli** (selected input port): When a node has several buffers (or input ports) sharing output port through which they can forward their packets, `seli` tells which input port is selected to send through which output port during a given time slot. Section 6.3 will discuss how the selection process is conducted. Initially `seli` will be `dmy` meaning no one has been selected.

3. **flag:** is a boolean variable for every buffer. It signals that the buffer has just freed one space or not.

4. **lcredit:** is a local credit counter on a sender outport side, and records the length of the receiver buffer.

5. **eecredit:** end to end credit is a counter associated with `ANIP` and controls packet generation in `ANIP`. Initially eecredit is equal to the size of the smallest buffer in the `ANIP`.

6. **slot_table:** is the slot table of every router in the network. Initially a slot table is empty (`dmy`) which means all output ports are not reserved during all time slots.

The initial value of these variables is given by the following PVS function.

```
initial_data(d:DATA):bool =
FORALL b: empty?(buffer(d)(b)) AND
      dmy(seli(d)(outport(b))) AND
      flag(d)(b) AND empty(link(d)(inport(b))) AND
      lcredit(d)(b) = 0 AND
      ( anip_sys_buffer(b) IMPLIES
        ( FORALL b2: anip_ack_buffer(b2) AND node(b2) = node(b) AND
            eecredit(d)(b) <= capacity(b) AND
            eecredit(d)(b) <= capacity(b2))) AND
      ( router_buffer(b) IMPLIES
        ( FORALL i: dmy(slot_table(d)(outport(b),i))))
```

Appendix A.9 shows the complete PVS code of network contents and operations on them.

6 Communication

6.1 Establishing a GT connection

Application software establish a GT connection among each other using the method described in section 2.1.2. That is an application software on an ANIP side sends

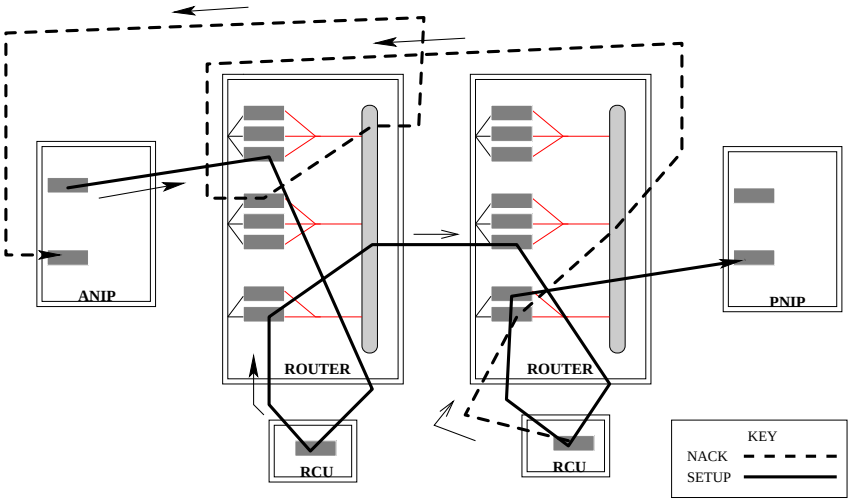

Figure 5: A typical SETUP and NACK packet paths

a SETUP packet and expects ACK or NACK packet as an acknowledgement from the peer application software or from a router along a path.

Depending on the general topology of the network, the path of the packet can be formalized as a list of buffers (more specific that the list of nodes) through which the packet travels to reach its destination.

```
lp: VAR list[BUFFER]
i: VAR nat

path(lp):bool =
length(lp) > 0 AND  anip?(node(nth(lp,0))) AND
     pnip?(node(nth(lp,length -1))) AND
     FORALL i: i > 0 AND i < length(lp) IMPLIES
      peer_oi(outport(nth(lp,i-1))) = inport(nth(lp,i))
```

Figure 5 shows an example of SETUP (thick continuous line) and NACK (thick dashed line) packet paths in a network of two routers an ANIP and a PNIP. The SETUP packet in fig 5 reaches to its destination PNIP. While the NACK packet starts from the second router (RCU) where a preceding SETUP packet (not shown in the figure) is assumed to fail. For TDOWN packet, the same path, as the SETUP packet's path, can be drown except that TDOWN packet does not travel upto PNIP. Similarly for ACK packet the path will look like NACK packet, except that the source of ACK packet is only PNIP.

In the remaining part of this section we define formally the operations on in the network, such us: routing, arbitration, reserving, receiving and sending.

6.2 Routing

When a packet arrives to the input port of a node, the routing function routes the packet, according to its destination, to an output port through which it has to leave the node. This function can be defined formally as:

```
route:[    i1:(notdmy?[INPORT]),p:PACKET
        -> {o1:(notdmy?[OUTPORT])| node(i1) = node(o1)} ]
```

6.2.1 Buffer Class

Table 2 classifies buffers as system buffers and acknowledgment buffers. This means, the routing function will make sure only system packets are routed to system buffers and only acknowledgement packets are routed to acknowledgement buffers. We also repeat here, this buffer classification as PVS predicates. note that, the buffers identified as `router_from_rcu_buffer` are both system and acknowledgement buffers because they are allowed under the NOC protocol to receive system and acknowledgement packets.

```
sys_buffer(b):bool = router_sys_buffer(b)  OR
             router_from_rcu_buffer(b) OR
             ni_sys_buffer(b) OR rcu_buffer(b)

%ack_buffer
ack_buffer(b):bool =     router_ack_buffer(b) OR
            router_from_rcu_buffer(b) OR
            ni_ack_buffer(b)
```

The following axiom defines that the route function obeys these classification. There is one exception though. In `PNIP` system packet can be routed to acknowledgment buffer. The packet will also change time on the process.

```
route_ax: AXIOM
    route(i1,p) = o1 IMPLIES
        LET n1:NODE = node(i1),
            b:BUFFER = (#inport:=i1,outport:=o1#) IN

        ( sys_packet(p) AND sys_buffer(b) ) OR
        ( ack_packet(p) AND ack_buffer(b) ) OR
        ( pnip?(n1) AND ni_ack_buffer(b) )
```

6.2.2 Dependency Graph

Based on the route function one can define buffer dependency graph, that represents the possible routing of packets from one buffer to another buffer. Dependency graph can be formally defined as a relation function between two buffers as:

```
dep_graph(b1,b2:BUFFER):bool =
    EXISTS p: route(peer_oi(outport(b1)),p) = outport(b2)
```

Figure 6 depicted dependency graph flow. The arrows between buffers shown in Figure 6 are possible routing links derived from the definition of the `route` function.

Note that, the reason why we do not have more arrows in the figure has to do with the definitions of buffers and nodes. For instance there is no direct arrow between router system buffer to PNIP system buffer because all router system buffers only connected to RCU. Thus all system buffers have to visit the RCU before they can be routed to the next node. Similar explanation can be found from the complete PVS specification in the appendix for all missing arrows.

6.2.3 Cycle on Dependency Graph

In this section we will prove an important theorem that the route function will not allow a packet to have a cycle on its path from source to destination. First we give

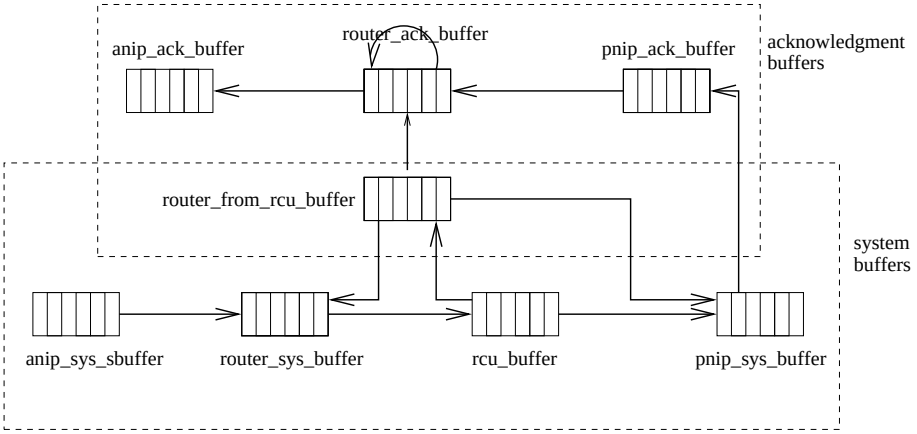

Figure 6: Routing between buffers

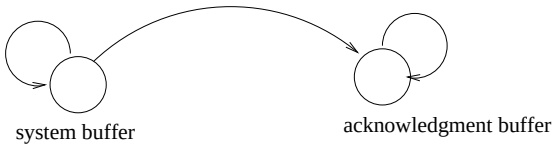

Figure 7: Routing between system and acknowledgment buffers

the definition of a modified buffer classes followed by path and cycle definition. Previously in section 6.2.1 we declare system and acknowledgment buffers. In this section we will give a slightly different version of buffer class definition inorder to have disjoint classes. That is the definition of system buffers remain the same while acknowledgment buffer is defined without `router_from_rcu_buffer`.

```
ack_buffer2(b):bool =     router_ack_buffer(b) OR
            ni_ack_buffer(b)
```

By grouping buffer classes into system and acknowledgment buffer the figure in 6 can be redrawn as 7. Similarly it can also be stated as PVS predicate as

```
t1,t2: VAR BUFFER
buffer_class_routing(t1,t2) = ack_buffer2?(t1) OR system_buffer(t2)
```

Definition 1 *Given a list of buffers* `lt`. `lt` *can be a valid path of a packet if the* i^{th} *buffer is the successor of* i-1^{th} *in the dependency graph.*

```
path?(lt):bool = length(lt) > 1 AND
       FORALL i: i > 0 AND i < length(lt) IMPLIES
         buffer_class_routing(nth(lt,i-1),nth(lt,i))
```

Definition 2 *A list of buffers* `lt` *is a cycle if the first and last of* `lt` *refers to the same buffer.*

```
cycle?(lt):bool = length(lt)>1 AND nth(lt,0) = nth(lt,length(lt)-1)
```

6.2.4 Absence of Cycle in Dependency Graph

The ÆTHEREAL protocol assumes that there will be no cycle between the same buffer class. This claim is assumed to be guaranteed at the hardware level of the network design. For our purpose we only state as axiom

```
%all buffers in the list are acknowledgment buffers
ackpath?(lt):bool =
    FORALL i: i >= 0 AND i < length(lt)
        IMPLIES ack_buffer?(nth(lt,i))

%all buffers in the list are system buffers
syspath?(lt):bool =
    FORALL i: i >= 0 AND i < length(lt)
        IMPLIES sys_buffer?(nth(lt,i))

%there is no cycle in ackpath
smcy: AXIOM ackpath?(lt) IMPLIES not cycle?(lt)

%there is no cycle in syspath
bgcy: AXIOM syspath?(lt) IMPLIES not cycle?(lt)
```

Theorem 1 *The dependency graph does not contain a cycle*

```
nocy:  THEOREM path?(lt) IMPLIES not cycle?(lt)
```

Proof *(sketch): Assume the last element of the path* `lt` *is a an acknowledgment buffer this implies* `ackpath?(lt)`*. Moreover, if the first element is a system buffer then* `syspath?(lt)` *is true. These claims can easily be proven by induction on the length of* `lt`*. Hence the first and the last element of a path* `lt` *can not be equal, which proves our theorem.*

In our PVS model we make a separate Theory for a routing definition as theory `route` (see appendix A.10) and cycle free-ness property of the dependency graph as theory `broute` (see appendix A.10.1). By importing the `broute` in `route` we show that theorem 1 holds for the route function.

```
IMPORTING broute[BUFFER,(sys_buffer), (dep_graph)]
```

6.3 Arbitration

6.3.1 Arbiter

Routers have multiple input and output ports, and thus multiple connections. As a consequence there is a need of arbitration among computing buffers (or input ports) that want to send packets via the same output port. For instance in fig 3 the output ports of the two routers are shared by buffers in three input ports. If all buffers have a packet to be sent via the same output port at the same time, then there will be a need for an arbitration. Moreover the same output port is also used to send GT-packets.

Arbitration in NOC is dealt by connecting the input ports and output ports of the node in a programmable architecture called virtual output queued architecture (VOQ) (which looks like as shown in fig 3)

Based on VOQ architecture an algorithm called matrix scheduler [RGAR$^+$03] is used to fairly resolve the output port sharing problem. We refer to the reader

to [RGAR+03] for extended explanation of the virtual output queued architecture and matrix scheduler. In this section we only state the required properties of the arbiter function as a PVS axiom.

The arbiter function in a router is function that requires the current content of the router (including buffer and slot table values) as an input and for every output port it returns an input port. The returned input port is then selected to use the output port in the next packet sending process.

```
arbiter:[DATA,(nport?[OUTPORT]),SLOT->INPORT]

schedule(old,new:DATA,cur_slot:SLOT):bool =
FORALL o1:  seli(new)(o1) = arbiter(old, o1,cur_slot)
```

6.3.2 Arbiter properties

Given the network's data `data1`, routers output port `o1` and a time slot `st` then `arbiter(data1,o1,st) = i1` if and only if:

1. `i1 = dmy?(i1)` (no input port is selected) if and only if

   - `o1` is already reserved during `st`. Thus no BE packet is served while a GT-packet connection is using `o1`.

     ```
     already_reserved(slot_table(data1),o1,st)
     ```

   - All destination buffers of the non empty source buffers are full. In other words, packets are not sent before making sure they will be received.

     ```
     (FORALL b1: outport(b1) = o1 AND
             nonempty?(buffer(data1)(b1)) IMPLIES
             LET i2 = peer_oi(o1),
                 p1 = first(buffer(data1)(b1)),
                    o2 = route(i2,p1),
                     b2 = (#inport := i2, outport:=o2#) IN
             lcredit(data1)(b2)=0
     ```

2. If the above condition is not met, then the arbiter chooses a BE buffer that satisfy the following conditions.

   - the selected buffer is not empty. The arbiter does not choose an empty buffer while there is another non empty buffer waiting to use the output port. If all buffers are empty then the first condition holds.

     ```
     LET b1 = (#inport := i1, outport:=o1#) IN
         NOT empty?(buffer(data1)(b1))
     ```

   - The destination of the first packet from the buffer queue ha a space to accommodate this packet. This way the selected buffer can send the packet and remove its copy without having to wait for any sort of confirmation for the arrival of the packet.

     ```
     LET i2 = peer_oi(o1),
             p1 = first(buffer(data1)(b1)),
             o2 = route(i2,p1),
             b2 = (#inport := i2, outport:=o2#) IN
             lcredit(data1)(b2) >=1
     ```

3. A maximum one packet is sent from one input port to any one of the output ports of the node. These restriction introduces fairness on the selection of buffers.

```
NOT o1 = o2 AND arbiter(data1,o1,st) = arbiter(data1,o2,st)
  IMPLIES dmy?(arbiter(data1,o1,st))
```

The complete PVS specification is in Appendix A.11

6.4 Receive

6.4.1 Receiving from Link

The receive process in NOC can be easily stated as three sequential actions. That is, for every buffer in the network:

1. read a packet from the link,
2. and if the link contains a packet, find the output port through which the packet will be routed
3. add the packet to the appropriate buffer.

The following PVS code segment defines the receive process as a relation function between `old` (before receiving) and `new` (after receiving) content of every buffer.

```
 read_from_link(old,new:DATA):bool =
FORALL b:
    LET    oldd:BUFFERD =  buffer(old)(b),
        newd:BUFFERD =  buffer(new)(b),
        linkp:PACKET =  link(old)(inport(b)) IN

    IF( NOT empty?(linkp) AND
        outport(b) = route(inport(b),linkp))

        THEN     newd = enqueue(linkp,oldd)
    ELSE
         newd = oldd
    ENDIF
```

6.4.2 Generating Packet

Another form of receiving a packet is when an application on the `ANIP` side generate a new packet for establishing a new GT-connection to a `PNIP`. This is the only way a new packet enters the network. The type of the packet and the means of generating a packet can be given by the PVS code given below.

1. The type of the packet is

```
new_packet(p:PACKET,a:(anip?),cur_slot:SLOT):bool =
    ptype(p) = SETUP     % type is SETUP

    AND phub(p) = 0     % hub counter is zero

    %its slot number should refer to future time (well ahead)
    AND pslot(p) > cur_slot + MINIMUM_SLOT_OVERHEAD

    % the source is the generating ANIP
    AND psrc(p) = a

    % destination is a PNIP
    AND pnip?(pdes(p))
```

2. The ANIP generating a packet should have enough credit in its `eecredit` counter.

3. The generated packet is saved to the system buffer of the ANIP.

4. generating a packet is optional. The `ANIP` may not generate new packet, even when the above preconditions are met.

```
generate_SETUP(old,new:DATA,cur_slot:SLOT):bool =
FORALL b: anip_sys_buffer(b)  AND

LET     %oldab = old value of the acknowledgment buffer
    oldab:BUFFERD = buffer(old)(b),
    newab:BUFFERD = buffer(new)(b)   IN

     ( buffer(new)(b) = buffer(old)(b)
    OR
    (eecredit(old)(b) > 0 AND
       ( EXISTS p: new_packet(p,node(b),cur_slot) AND
                buffer(new)(b) = enqueue(p,buffer(old)(b)))
    )
    ) AND
    eecredit(new)(b) =  up_eecredit(eecredit(old)(b),
                 length(buffer(old)(b)),
                 length(buffer(new)(b)))
```

The function`up_eecredit` update the `eecredit` as in

```
up_eecredit(eecredit:nat,old_length:nat,new_length:nat):nat =
    eecredit + (old_length - new_length)
```

6.5 Send

IN NOC packets are sent from the source buffer to the outgoing link before they arrive in the receiver buffer.

As explained is section 6.3 sending packet requires selecting a buffer that will use the outgoing link. Provided that a given buffer is selected to send its packet through a given link then the send process proceed as follows:

1. The first packet in the queue of the selected buffer is copied to the outgoing link, and

2. The packet is deleted from the source buffer

```
write_to_link(old,new:DATA):bool =
FORALL b:
    IF(inport(b)=seli(old)(outport(b))) THEN
        buffer(new)(b) = dequeue(buffer(old)(b)) AND
        link(new)(peer_oi(outport(b))) = first( buffer(old)(b))
    ELSE
        buffer(new)(b) = buffer(old)(b)
    ENDIF
```

6.6 Reservation

The reservation or unreservation of router output ports takes place in the peer `RCU`. This process starts by reading the packet in the `RCU` buffer. Of course if the `RCU` has no packet to process then there is nothing to do.

```
reserve(old,new:DATA):bool =
FORALL b: rcu_buffer(b) AND
    LET
        p:PACKET = last(buffer(old)(b)),
        o2:OUTPORT = peer_io(inport(b))        IN

    (( empty?(p) IMPLIES old = new) OR ...
```

Otherwise the following events happen depending on the type of the packet and the availability of the required output port in the required time slot. Let the required output port be `o1` and the requiring input port be `i1`, Then `o1` and `i1` can be determined as follows.

```
LET    o1:OUTPORT = route(peer_oi(outport(b)),p),
    i1:INPORT = seli(old)(o2)         IN
```

1. If the packet is a `SETUP` packet, and if the requested output port is already reserved during the slot number that the packet is interested in, then the establishment of the connection will fail and the packet is replaced by a negative acknowledgment (`NACK`) packet. This new packet will be used to announce to the sender `ANIP` that the attempt to establish a connection has failed. This phenomenon is labeled as `busy` in fig 2

   ```
   CASES ptype(p) OF

   SETUP: IF already_reserved(slot_table(old),o1,pslot(p)) THEN
           buffer(new)(b) = enqueue(fail(p),dequeue(buffer(old)(b)))
   ```

   `fail(p)` replace the packet `p` with `NACK` type, the source and destination are swapped as well.

   ```
   fail(p):PACKET =
       (# ptype:= NACK,
          psrc := pdes(p),
          pdes := psrc(p),
          pslot:= pslot(p),
          phub := phub(p)   #)
   ```

   If it is not reserved then the requiring input port is registered in the slot table, and the packet is promoted for the next node. Promoting a `SETUP` packet means increasing its hub counter and its time slot counter.

   ```
   buffer(new)(b) = enqueue(promote_SETUP(p),dequeue(buffer(old)(b)))
           AND reserved_by(slot_table(new),o1,pslot(p),i1)

   promote_SETUP(p):PACKET =
       p WITH  [ phub := phub(p) + 1, pslot:=pslot(p)+1]
   ```

2. in the case of `TDOWN` packet: the entry associated with this packet in the slot table is cleared, and then either the packet is destroyed or promoted to the next node. The packet is destroyed if this node is its last node (that is `phub(p) = 0`) or promoted otherwise.

   ```
   TDOWN: unreserved(slot_table(new),o1,pslot(p)) AND
      IF(phub(p)>0) THEN
       buffer(new)(b) = enqueue(promote_TDOWN(p),dequeue(buffer(old)(b)))
      ELSE     buffer(new)(b)= dequeue(buffer(old)(b))
      ENDIF
   ```

The use of the hub counter is thus, TDOWN packet only travels upto the last node where the predecessor SETUP packet failed.

```
promote_TDOWN(p):PACKET =
       p WITH  [ phub := phub(p) - 1, %  > 0 from "state.pvs"
              pslot:=pslot(p)+1]
```

6.7 PNIP Reply

When a SETUP packet reaches its destination PNIP, then the packet is queued in the buffer until the application responds. If it decided to respond to the request then the SETUP packet will be removed from the buffer and a new response (ACK or NACK) packet is generated to the second acknowledgment buffer (provided that the buffer is not full). The following PVS predicate defines the proper PNIP response using the old and new values of the system and acknowledgment buffers.

```
pnip_reply(old,new:DATA): bool =
FORALL b: pnip_ack_buffer(b) AND

    EXISTS b2: pnip_sys_buffer(b2) AND node(b2) = node(b) AND

    LET     %oldab = old value of the acknowledgment buffer
        oldab:BUFFERD = buffer(old)(b),
        newab:BUFFERD = buffer(new)(b),
        oldsb:BUFFERD = buffer(old)(b2),
        newsb:BUFFERD = buffer(new)(b2)   IN

    (newab = oldab AND newsb = oldsb)
    OR
    (
     length(oldsb)>0 and length(oldab)<capacity(b) AND
     newab = enqueue(reply(first(oldsb)),oldab) AND
            newsb = dequeue(oldsb)
    )
```

where the function reply swaps the source and destination address and the type of the packet will be either ACK or NACK depending on the acceptance or rejection of the request:

```
reply(p1) = p2 IMPLIES
    (ptype(p2) = ACK OR ptype(p2) = NACK) AND
    psrc(p2) = pdes(p1) AND
    pdes(p2) = psrc(p1) AND
    pslot(p2) = pslot(p1) AND
    phub(p2) = phub(p1)
```

6.8 Generating TDOWN packet

Similarly ANIP also respond whenever acknowledgment packets arrive. These acknowledgement packets are queued in the acknowledgement buffer until the applications responsible to them reads and consumes them. If the packet is NACK then there will be a need for generating a TDOWN packet to undo all the reservation done by the predecessor SETUP packet. Thus a TDOWN packet is generated to the system buffer with the following values in its field:

- The type of the packet is TDOWN.
- The source is the generating ANIP or the destination of the NACK. Note that it is also the source address of the predecessor SETUP packet.

- The destination is the source of the NACK. Note that it is the same destination of the predecessor SETUP packet.
- The time slot of this new TDOWN packet should be equal to the time slot of the predecessor SETUP packet. To calculate the value of the time slot we deduct phub(p) from the time slot of the NACKpacket. Because the hub counter value of the NACK packet is the number of the routers that the predecessor SETUP packet have traversed, also this counter tells how many times the slot time was incremented.

```
NEW_TDOWN_packet(p):PACKET =
    (# ptype := TDOWN,
       psrc  := pdes(p),
       pdes  := psrc(p),
       pslot := pslot(p)-phub(p),
       phub  := phub(p)    #)
```

Then this packet is queued to the system buffer and the NACK packet is marked as already processed. A packet is marked by setting its hub counter to zero.

```
MARK_packet(p):PACKET = p WITH [phub:=0]
```

Marking NACK packet will avoid the risk of generating multiple TDOWN packets from a single NACK packet. It should not be removed either because the application software that initiate the original SETUP packet needs to be informed that the attempt to establish a GT-connection has failed. This way the application can try again at different slot time.

The generating and marking process can formally be described as in the following PVS code.

```
generate_TDOWN(old,new:DATA): bool =
FORALL b: anip_ack_buffer(b) AND
EXISTS b2: anip_sys_buffer(b2) AND node(b2) = node(b) AND

LET     %oldab = old value of the acknowledgment buffer
    oldab:BUFFERD = buffer(old)(b),
    newab:BUFFERD = buffer(new)(b),
    oldsb:BUFFERD = buffer(old)(b2),
    newsb:BUFFERD = buffer(new)(b2)   IN

IF ( not empty?(oldab)) THEN

    LET  p:PACKET = first(oldab) IN

    IF(NEW_NACK(p)) THEN
                EXISTS intb: oldab = enqueue(p,intb) AND
                newab = enqueue(MARK_packet(p),intb) AND
                newsb = enqueue(NEW_TDOWN_packet(p),oldsb)
     ELSE
              newab = oldab AND newsb=oldsb
    ENDIF
ELSE
    newab = oldab AND newsb=oldsb
ENDIF
```

6.9 ANIP Consumption

ACK or marked NACK packets are deleted (consumed) from the acknowledgement buffer of an ANIPwhenever the application layer decides to do so. It is not guaranteed that packets will be consumed regularly but it is assumed that a packet will eventually be consumed.

The consumption of a packet will also result in freeing one space in the buffer, which in turn increases the end to end credit counter by one.

```
consume(old,new:DATA): bool =
FORALL b: ni_ack_buffer(b)  AND anip?(node(b)) AND
    (  buffer(new)(b) = buffer(old)(b) OR
       buffer(new)(b) = dequeue(buffer(old)(b)) )
    AND eecredit(new)(b) =
        up_eecredit( eecredit(old)(b),
        length(buffer(old)(b)),
        length(buffer(new)(b))    )
```

The complete PVS specifications of the send, receive, generating and consuming actions are provided in Appendix A.9

7 NOC as a State Machine

Communication in NOC is a synchronous [Lyn96] transmission of packets from one buffer to another buffer. Each transmission is signaled by the advancement of time slot [NPR+02]. Thus, the configuration of the network is determined by the content of the network at a given slot time. Further more, for modeling convenience, the communication of the network is modeled as three sequential transitions called *phases*. These transition phases are *read*, *execute* and *write*.

- in read phase the network reads packets from the incoming link.
- in execute buffers are selected for using the outgoing links and the reservation of links also take place in `RCU`s.
- in write phase the network sends packets to the outgoing links

Formally we can define phase data type as follows:

```
%The network operates in three phases
PHASE_TYPE: DATATYPE
   BEGIN    READ: READ?
            EXECUTE: EXECUTE?
            WRITE: WRITE?
   END PHASE_TYPE
```

Due to the additional classification of phases the state of the network is determined by the content of the network, time slot and phase.

```
%State variables
State: TYPE=[# data:DATA,
           phase: PHASE_TYPE,         % transition phase
           cur_slot: SLOT             % current time slot
    #]
```

7.1 Start State

The initial state of the network ism when the content of the network is in its initial value, and it is in the read phase and the current slot is zero. That is:

```
start_state(s):bool =
    inital_data(data(s)) AND
    phases(s) = READ AND
    cur_slot(s) = 0
```

7.2 Transition Phases

The three transition phases are defined as a boolean relation between two states as follows.

```
read:[State,State->bool]
execute:[State,State->bool]
write:[State,State->bool]
```

In the remaining part of this section we define axiomatically which actions take place in which phase and which state variable remains unaffected. Figure 8 summarizes the communication actions of NOC as a state machine.

7.2.1 Read Phase

Given two states `s` and `t` we say that `read(s,t)` holds if and only if the following actions take place.

1. Buffers read packets from the link as described in section 6.4.1
2. `ANIP` will also be given a chance to generate `SETUP` packet as described in section 6.4.2 and
3. Local credit counter is updated as shown in section 5.4.1.
4. the phase variable also changes from `READ` to `EXECUTE`
5. All other variables remain unchanged

```
read_ax: AXIOM read(s,t) IFF

    phase(s) = READ AND phase(t) = EXECUTE AND

    % in read phase
    % 1. new packets generated in anips and eecredit is updated
    % 2. local credit is updated
    % 3. buffers get data from the link(wire)
    read_from_link(data(s),data(t)) AND
    up_lcredit(data(s),data(t))  AND
    generate_SETUP(data(s),data(t),cur_slot(s)) AND

    % the following state variables will remain
    % unchanged in READ phase
    slot_table(data(t)) = slot_table(data(s)) AND
    seli(data(t)) = seli(data(s)) AND
    link(data(t)) = link(data(s)) AND
    cur_slot(t) = cur_slot(s) AND
    flag(data(t)) = flag(data(s))
```

7.2.2 Execute Phase

Given two states `s` and `t` we say that `execute(s,t)` holds if and only if the following actions take place.

1. The selection process of section 6.3 – choosing an input port that uses a given outport.
2. Reserving or unreserving 6.6
3. `ANIP` also generates tear down packets, see section 6.8

4. the phase variable changes from `EXECUTE` to `WRITE`

5. All other variables remain unchanged

```
execute_ax: AXIOM execute(s,t) IFF

    phase(s) = EXECUTE AND    phase(t) = WRITE AND

    schedule(data(s),data(t),cur_slot(s)) AND
    reserve(data(s),data(t)) AND
    generate_TDOWN(data(s),data(t)) AND

    % the following state variables will remain
    % unchanged in  this phase
    link(data(t)) = link(data(s)) AND
    cur_slot(t) = cur_slot(s) AND
    flag(data(t)) = flag(data(s)) AND
    eecredit(data(t)) = eecredit(data(s)) AND
    lcredit(data(t)) = lcredit(data(s)) AND
    FORALL b: NOT (anip_buffer(b) or rcu_buffer(b)) AND
    buffer(data(t))(b) = buffer(data(t))(b)
```

7.2.3 Write Phase

Given two states `s` and `t` we say that `execute(s,t)` holds if and only if the following actions take place.

1. Buffers write packets to the outgoing link, also parallelly the local credit is updated to reflect the sending of the packet. See section 6.5. At the same time All flags that are associated with the buffers that have sent a packet will be raised, while others remain down.

2. `ANIP` consumes acknowledgment packets as in section 6.9 and

3. `PNIP` replies as in section 6.7

4. the phase variable changes from `WRITE` to `READ`

5. the current slot is also incremented to the next value

6. All other variables remain unchanged

```
write_ax: AXIOM write(s,t) IFF

    phase(s) = WRITE AND    phase(t) = READ AND
    cur_slot(t) = cur_slot(s)+1 AND

    write_to_link(data(t),data(s)) AND
    update_flag(data(t),data(s)) AND
    consume(data(t),data(s)) AND
    pnip_reply(data(t),data(s)) AND
    down_lcredit(data(s),data(t))  AND
    % the following state variables will remain
    % unchanged in  this phase
    slot_table(data(t)) = slot_table(data(s)) AND
    seli(data(t)) = seli(data(s)) AND
    eecredit(data(t)) = eecredit(data(s))
```

Figure 8 summarizes the communication actions of NOC as a state machine. The complete PVS specification is in Appendix A.12

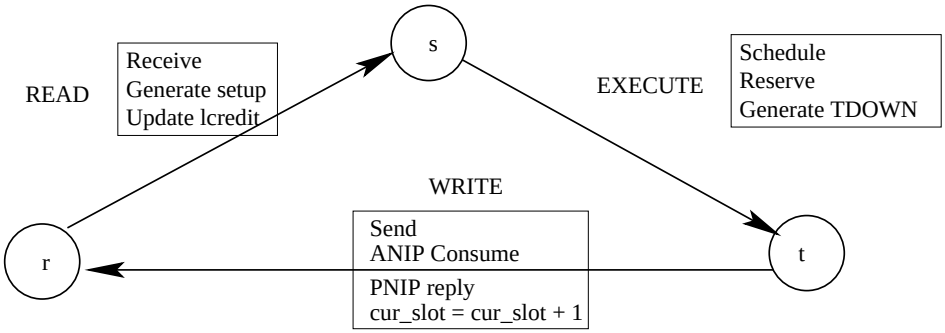

Figure 8: Communications actions of the network divided into three phases

7.3 Reachable States

The network starts from the initial state and the next state will be the result of the execution of the three phases described above.

```
steps(s:State,ph: PHASE_TYPE, t:State):bool =
CASES ph OF
    READ:      read(s,t),
    EXECUTE:  execute(s,t),
    WRITE:      write(s,t)
ENDCASES
```

The set of all reachable state is computed by the recursive function shown below.

```
reachable_n(A,s,(n:nat)) : RECURSIVE bool =
   IF n = 0 THEN starts(A)(s) ELSE
     (EXISTS (t:State),(a:Action) :
                 reachable_n(A,t,n-1) AND steps(A)(t,a,s))
   ENDIF
   MEASURE n
```

The complete PVS specification is in Appendix A.16

8 Absence of Deadlock

Deadlock is a property of global states of the network. We say that a certain state *s has a deadlock* iff there exists a finite, nonempty list *lb* which is a path, cyclic and all buffers in *lb* are full. We say a network *has a deadlock* if some reachable state of this network has a deadlock. Formally a deadlock can be defined as.

```
lb: VAR list[BUFFER]

deadlock?(s:State):bool =
EXISTS lb: path?(lb) AND cycle?(lb) AND
      FORALL i:    i>0 AND i<length(lb) AND
    LET    b1 = nth(lb,i-1),
        b2 = nth(lb,i)  IN
    length(buffer(data(s))(b1)) = capacity(b1)
```

The complete PVS specification is in Appendix A.14

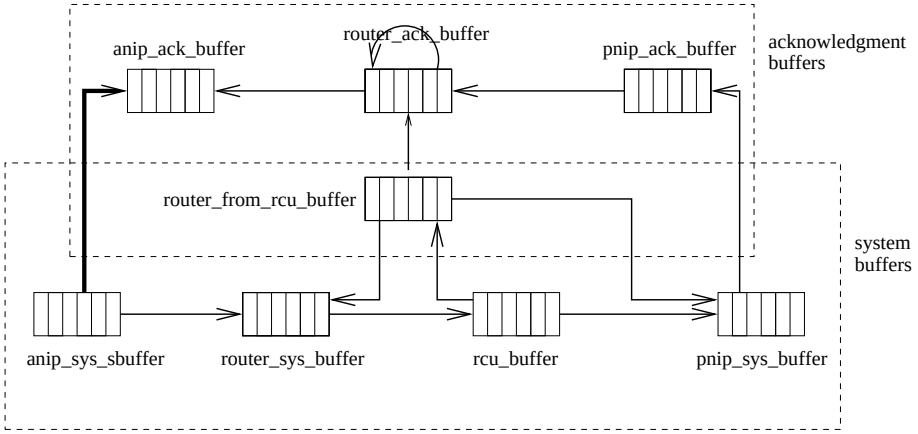

Figure 9: Routing between buffers including TDOWN generation in ANIP

8.1 Dependency Graph Revised

In section 6.2.2 we define the relation dep_graph as.

```
dep_graph(b1,b2:BUFFER):bool =
    EXISTS p: route(peer_oi(outport(b1)),p) = outport(b2)
```

We have also seen in section 6.8 that an ANIP can generate a TDOWN packet into the system buffer from an NACK packet in the acknowledgment buffer. This extra transfer of packet is not part of the dependency graph defined in section 6.2.2. During TDOWN generation, ANIP can route a packet from acknowledgment buffer to system buffer which means there will be an arrow anip_ack_buffer to anip_sys_buffer in fig 6, (see fig 9 for the change.) This change in the definition of dependency graph will also affect the definition of path? and cycle?, since path? and cycle? are defined as a function of dependency graph in section 6.2.2.

8.2 Proof of Absence of Deadlock

In this section we will prove that there is no deadlock in all reachable states.

```
s: VAR State
I1: LEMMA NOT deadlock?(s)
```

The absence of deadlock can not be deducted from theorem 1, since theorem 1 assumes there is no routing between acknowledgment and system buffers of ANIP. But if we assume that the path *lb* that results in a deadlock does not contain ANIP buffers then by theorem 1 we know that *lb* is not a cycle. Thus for any *lb* in a state without ANIP buffers in state *s*, deadlock?(s) is false.

The second case is when *lb* does contain a buffer from ANIP. The next section is dedicated to this case and it is proven that deadlock does not happen even if it is possible to route from ANIP acknowledgment buffer to ANIP system buffer.

8.3 Invariant Property of End to End Credit

Though we can not use the theorem of section 6.2.4 to prove absence of deadlock in ANIP. ANIP maintains the end to end credit counter to control packet flow, which

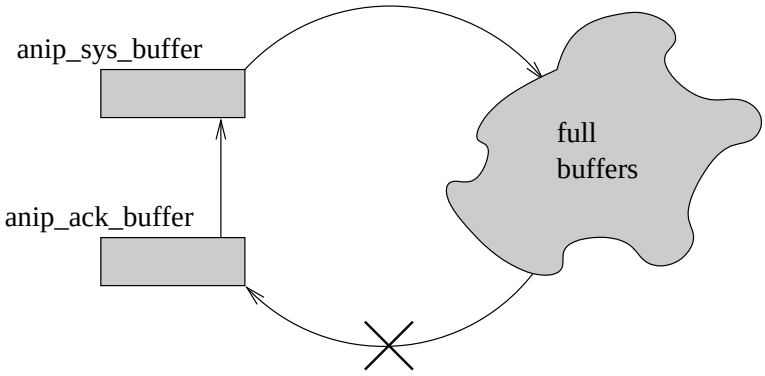

Figure 10: Deadlock associated with ANIP

will correct the problems emerging due to the new routing scheme. In this section we will prove an important invariant on the end to end credit counter, that will help us in proving the absence of deadlock property discussed in section 8.

8.3.1 End to End Credit Invariant

End to end credit counter (eecredit) is a natural number counter that is incremented when the ANIP generates new SETUP packet and it is decremented when ANIP consumes acknowledgment packet (or frees space in the buffer). Initially eecredit is equal to the minimum capacity of either of the two buffers in ANIP as shown in section 7.1.

Lemma 1 *For any ANIP in the system it is not possible that,* anip_ack_buffer *is full and there is a packet in the network whose destination is this ANIP*

Figure 10 shows a scenario where a list of buffers, including ANIP buffers make up a path which is cyclic and all buffers are full. Theorem 1 crosses out the link between the network and anip_ack_buffer as shown in fig 10.

8.3.2 Abstracted ANIP

In order to prove lemma 1 we have modeled ANIP separately as a PVS theory (see appendix A.15). This theory will have more variables to keep track of all the packets sent by an ANIP and all packets whose destination are this ANIP. Moreover, this theory abstracts buffer operations inorder to simplify and generalize the lemmas proven below. Our abstraction has several assumptions, to mention some of them:

- Adding and removing a packet from a buffer increases and decreases the length of the buffer queue by one respectively.
- ANIP receives packets that have this ANIP as their destination address.
- The number of SETUP packets sent is equal to the number of packets arriving in an ANIP.

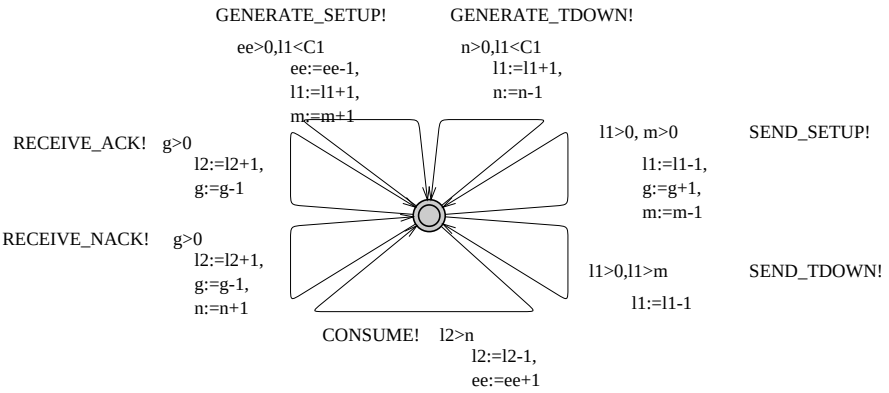

Figure 11: Abstracted ANIP operations

These assumption should follow from the definition of the route functions and other axioms already made, nevertheless it may require a lot of effort to formally prove them in PVS. Thus this document leaves the prove of this claim as a future work.

The ANIP operations described in the earlier sections can be abstracted as an automata of one location and eight distinct transitions. Each transition is labeled with a name, a precondition and effect style as shown in fig 11. The meaning of the variable is:

ee: end to end credit counter of the ANIP, counts who many credit is left for the ANIP to send a `SETUP`. Zero `ee` means the `ANIP` can not generate `SETUP` packet.

l1: the length of system buffer queue (`anip_sys_buffer`).

C1: the capacity of system buffer.

l2: the length of acknowledgment buffer queue (`anip_ack_buffer`).

g: the number of packets in the network that was originally sent by this ANIP.

m: the `SETUP` packets already generated but not yet sent.

n: `NACK` packets received but no `TDOWN` packet is generated for them.

All transitions run with arbitrary order. For example in fig 11 the transition `GENERATE_SETUP` can take place if `ee` is greater than zero and, the buffer is not full (`l1 < C1`). If this precondition holds, then the effect of taking this action will decrement `ee` by one, increment the length by one and increment the number of `SETUP` packets in `anip_sys_buffer` which are not yet sent (that is `m`) by one. Figure 11 can also be coded as PVS code. First we group all the variables as a state variable

```
state: TYPE=[#     ee: nat, l1: nat, l2: nat,
               n: nat,  g: nat,  m: nat #]
```

Then the transitions are defined as a boolean relation between two states. For instance for GENERATE_SETUP its PVS equivalence is

```
generate_setup(s1,s2):bool =
if(ee(s1)>0 and l1(s1)<C(l1(s1)))

    THEN (s2 = s1 WITH [ee := ee(s1)-1, m:=m(s1)+1, l1:=l1(s1) + 1]) OR
        (s2 = s1)
    ELSE    (s2 = s1)
ENDIF
```

The rest of the transitions are coded in the same style. See appendix A.15 for complete PVS code. A one step in the automata is equivalent to taking one of the eight transitions.

```
steps(s1,s2):bool =
    generate_setup(s1,s2) OR
    generate_tdown(s1,s2) OR
    send_setup(s1,s2) OR
    send_tdown(s1,s2) OR
    consume(s1,s2) OR
    receive_nack(s1,s2) OR
    receive_ack(s1,s2)
```

A reachable path is a list of states sl where the first element of the list is the initial state and $\forall i < length(sl) - 1$ the $(i+1)^{th}$ element of sl is the next step of i^{th} element.

```
reachable_path(sl):bool = length(sl) > 0 AND start_state(nth(sl,0)) AND
( FORALL i: i < length(sl)-1 IMPLIES step(nth(sl,i),nth(sl,i+1)))
```

This Abstracted ANIP automata exhibits the following invariant.

Lemma 2 *(***Invariant 1:***) In any reachable state s the sum of end to end credit, the packets in the network originally sent by this* ANIP *the* SETUP *packets in* anip_sys_buffer*, and the acknowledgment packets in* anip_ack_buffer *is less than or equal to the capacity of* anip_ack_buffer*.*

```
I1(s):bool = ee(s) + g(s) + m(s) + l2(s) <= C(l2(s))

p1: LEMMA  reachable_path(sl) IMPLIES
    (FORALL i: i< length(sl) IMPLIES I1(nth(sl,i)))
```

Lemma 2 can be proven by induction on the length of the list.

Lemma 3 *(***Invariant 2***) In any reachable state s* anip_ack_buffer *is full implies there is no packet in the network that has to be routed to the* ANIP*.*

```
I2(s):bool = l2(s) = C(l2(s)) IMPLIES g(s) = 0

p1: LEMMA  reachable_path(sl) IMPLIES
    (FORALL i: i< length(sl) IMPLIES I2(nth(sl,i)))
```

The proof for lemma 3 can be easily deducted from 2. Theorem 1 can also be proven using lemma 3. This completes the proof of the theorem of section 8.2.

9 Conclusions

We started our efforts to model the ÆTHEREAL protocol using the model checker SMV [McM93]. Our initial SMV model contained four nodes and three messages, which was enough to simulate the basic operations. It also allowed us to rediscover deadlock scenarios that occur in variations/simplifications of the protocol. Nevertheless, we believe that model checkers are of limited relevance for this type of case studies because (a) the design is highly parameterized (network topology, routing functions, choice of buffers) and with a model checker one can only analyze one model at a time, after fixing specific values for all the parameters, (b) for nontrivial instances of the protocol, the state space becomes so big that even for a state-of-the-art full state space analysis becomes very difficult, if not impossible.

For this reason, we decided to switch to PVS [COR+95], a tool that provides mechanized support for formal specification and verification. Writing the specification in the PVS input language, which is based on classical, typed higher-order logic, helped us a lot in getting a clear and consistent view on the design. In this phase, we did not use the PVS verifier for analysis. The input language of PVS is highly expressive but nevertheless still readable for anyone with some background in logic. This makes it useful for listing, in a precise and unambiguous manner, all relevant aspects of a design. Proving nontrivial properties using PVS is much more involved and requires specialist expertise.

We think that our model might potentially be interesting from a theoretical point of view since (to the best of our knowledge) nowhere else in the literature synchronous network models for systems with bounded buffering have been studied. For instance, half of Nancy Lynch's book on Distributed Algorithms [Lyn96] is devoted to synchronous algorithm, but these algorithms assume unbounded buffers between any pair of connected nodes.

A simple approach to solving the deadlock problem is to require absence of cycles in the routing graph. It may turn out that (for instance because of performance considerations) this approach is too simplistic, but at the moment this appears to be a workable solution. If there are cycles in the routing graph, proving absence of deadlock will become considerably more involved, since we then have to take into account the dynamic behavior of the network. In order to carry out such an analysis, more specific information about network topology, buffer structure and routing policies will be required. In the design of the ÆTHEREAL protocol, deadlocks had to be ruled out at (1) the basic data level, (2) the setup/teardown level, and (3) the application level. Absence of cycles in the routing graph takes care of deadlocks of type (1) and (2). To avoid deadlock at the application level, it suffices that each packet that arrives at an input port of a network interface is eventually consumed by the application level.

Acknowledgement

We very much appreciated the support from Kees Goossens, Andrei Rădulescu and Edwin Rijpkema, who cordially reserved a lot of time to explain the ÆTHEREAL protocol to us and to answer all our questions. We also thank Adriaan de Groot for helping us with PVS.

References

[CGH+93] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In *Proc. CHDL*, pages 15–30, 1993.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, , and Mandayam Srivas. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 1995.

[DGRV00] M.C.A. Devillers, W.O.D. Griffioen, J.M.T Romijn, and F.W. Vaandrager. Verification of a leader election protocol: Formal methods applied to IEEE 1394. *Formal Methods in System Design*, 16(3):307–320, June 2000.

[GRG+05] Om Prakash Gangwal, Andrei Rădulescu, Kees Goossens, Santiago González Pestana, and Edwin Rijpkema. Building predictable systems on chip: An analysis of guaranteed communication in the æthereal network on chip. In Peter van der Stok, editor, *Dynamic and Robust Streaming In And Between Connected Consumer-Electronics Devices*, chapter 1. Kluwer, 2005.

[Gri00] W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. PhD thesis, University of Nijmegen, May 2000. Postscript and PVS sources available via `http://www.cs.kun.nl/ita/former_members/davidg/`.

[GvMPW02] K. Goossens, J. van Meerbergen, A. Peeters, and P. Wielage. Networks on silicon: Combining best-effort and guaranteed services. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 423–425. IEEE, mar 2002.

[HaSTC04] Jörg Henkel and Wayne Wolf amd Srimat T. Chakradhar. On-chip networks: A scalable, communication-centric embedded system design paradigm. In 17^{th} *International Conference on VLSI Design*, page 845. IEEE, January 2004.

[HJK+00] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Öberg, M. Millberg, and D. Lindqvist. Network on chip: an architecture for billion transistor era. In *Proceedings of NorChip*, November 2000.

[IEE96] IEEE Computer Society. IEEE Standard for a High Performance Serial Bus. Std 1394-1995, August 1996.

[Lyn96] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Fransisco, California, 1996.

[McM93] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[NPR+02] K. Goossens A. Nieuwland, A. Peeters, A. Radulescu, E. Rijpkema, and P. Wielage. The ÆTHEREAL guaranteed-throughput router, draft version 1.2, 2002. Net.Lab. Technical Note xxx/01.

[OSRSC99] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[RG04] Andrei Rădulescu and Kees Goossens. Communication services for networks on chip. In Shuvra S. Bhattacharyya, Ed F. Deprettere, and Jŕgen Teich, editors, *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*, pages 193–213. Marcel Dekker, 2004.

[RGAR+03] E. Rijpkema, K. Goossens, J. Dielissen A. Rădulescu, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip. *IEE Proceedings: Computers and Digital Technique*, 150(5):294–302, September 2003.

[Sch04] J. Schmalz. Functional specification and validation of the Octagon network on chip using the ACL2 theorem prover. Technical Report TIMA-RR–04/01-02–FR, TIMA Laboratory, Grenoble, 2004.

[vLRG03] Izak van Langevelde, Judi Romijn, and Nicolae Goga. Founding firewire bridges through promela prototyping. In 8^{th} *International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA)*. IEEE Computer Society Press, April 2003.

A Appendix

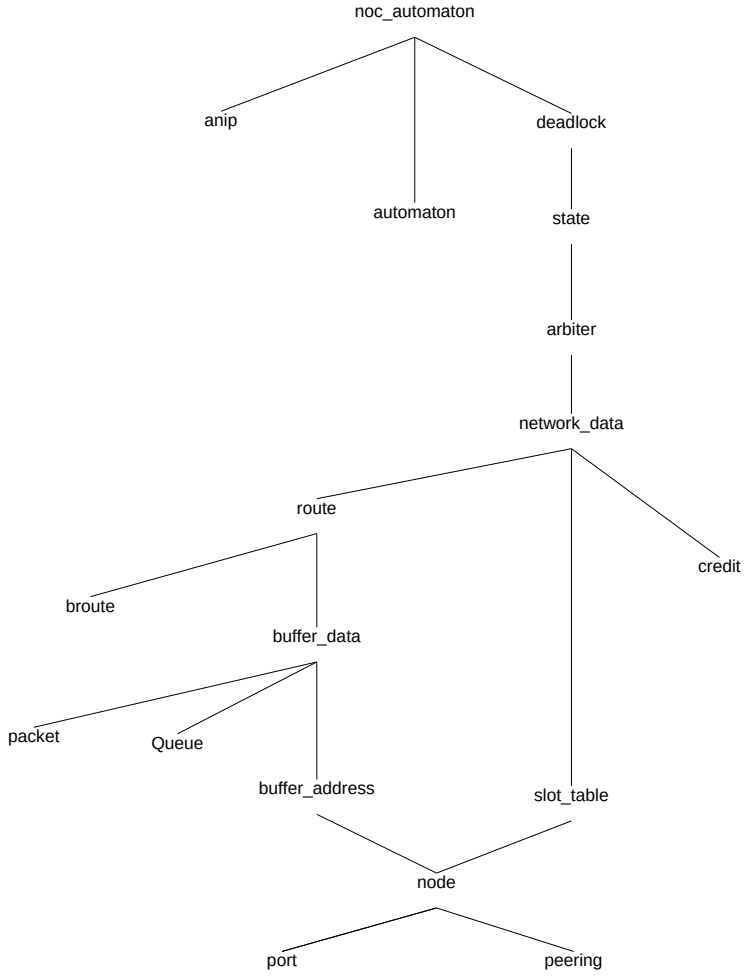

Figure 12: Theory Hierarchy of NOC specification

A.1 Port

Definition of port and type of ports in NOC

```
port[P:TYPE]: Theory
Begin

PortType: TYPE={DMY,APPLICATION,NORMAL,SPECIAL}

porttype:[P->PortType]

%port type testing predicates
dmy(p:P):bool = porttype(p)=DMY
app(p:P):bool = porttype(p)=APPLICATION
sp(p:P):bool = porttype(p)=SPECIAL
normal(p:P):bool = porttype(p)=NORMAL
nport(p:P):bool = sp(p) OR normal(p)
notdmy(p:P):bool = nport(p) OR app(p)

%
```

```
%node specific ports
%

% NI ports
ni_port(p:P):bool = app(p) OR sp(p)

% ROUTER ports
r_port(p:P):bool = normal(p) OR sp(p)

% RCU port
rcu_port(p:P):bool = sp(p)

END port
```

A.2 Peer

Definition of peer as a mapping between input ports and output ports

```
peering[INPORT:TYPE+,OUTPORT:TYPE+]: Theory
Begin

i1:VAR INPORT
o1:VAR OUTPORT

peer_oi:[OUTPORT -> INPORT]
peer_io:[INPORT -> OUTPORT]

peer_io_ax: AXIOM peer_io(peer_oi(o1)) = o1
%peer_io_ax1: AXIOM surjective?(peer_oi)

peer_inj_lemma1: LEMMA injective?(peer_oi)
peer_inj_lemma2: LEMMA injective?(peer_io)

peer_oi_lemma: LEMMA peer_oi(peer_io(i1)) = i1

peer(i1:INPORT,o1:OUTPORT):bool = peer_io(i1) = o1

END peering
```

A.3 Node

Nodes defined as a record of set of input ports and set of output port. ANIP, PNIP, router and RCU defined as a subtypes of a node

```
node: Theory
Begin

INPORT:TYPE
OUTPORT:TYPE

IMPORTING port[INPORT]
IMPORTING port[OUTPORT]

PreNode: TYPE = [#  inport: set[(notdmy[INPORT])],
                   outport: set[(notdmy[OUTPORT])]
        #]

n1,n2: VAR PreNode
is1,is2: VAR set[(notdmy[INPORT])]
os1,os2: VAR set[(notdmy[OUTPORT])]

disjoint_ax1: AXIOM
   FORALL is1,is2:
        (EXISTS n1,n2:         inport(n1) = is1 AND        inport(n2) = is2)
        IMPLIES        (n1 = n2 OR  disjoint?(is1,is2))

disjoint_ax2: AXIOM
   FORALL os1,os2:
        (EXISTS n1,n2:         outport(n1) = os1 AND        outport(n2) = os2)
```

```
        IMPLIES        (n1 = n2 OR  disjoint?(os1,os2))

o0: VAR (notdmy[OUTPORT])
i0: VAR (notdmy[INPORT])

node(i0:(notdmy[INPORT])):PreNode
node_ax_i: AXIOM node(i0) = n1 IMPLIES member(i0,inport(n1))
node_ax2_i: AXIOM FORALL i0: EXISTS n1: node(i0)=n1

node(o0:(notdmy[OUTPORT])):PreNode
node_ax_o: AXIOM node(o0) = n1 IMPLIES member(o0,outport(n1))
node_ax2_o: AXIOM FORALL o0: EXISTS n1: node(o0)=n1

IMPORTING peering[(nport[INPORT]),(nport[OUTPORT])]

i1: VAR (nport[INPORT])
o1: VAR (nport[OUTPORT])

peer_io_ax2: AXIOM NOT node(o1) = node(peer_oi(o1))
peer_oi_lemma2: LEMMA  NOT node(i1) = node(peer_io(i1))

%NI

appi: VAR (app[INPORT])
appo: VAR (app[OUTPORT])
spi: VAR (sp[INPORT])
spo: VAR (sp[OUTPORT])

NI: TYPE = {a:PreNode | (EXISTS appi,spi: inport(a) = add(appi,singleton(spi)))
                    AND (EXISTS appo,spo: outport(a) = add(appo,singleton(spo)))
           }

ANIPS: TYPE = set[NI]
PNIPS: TYPE = set[NI]

anips: ANIPS
pnips: PNIPS

anip(a:PreNode):bool = member(a,anips)
pnip(a:PreNode):bool = member(a,pnips)
ni(n:PreNode):bool = anip(n) OR pnip(n)

%router

rseti: VAR set[(normal[INPORT])]
rseto: VAR set[(normal[OUTPORT])]

ROUTER: TYPE = {a: PreNode | (EXISTS spi,rseti: inport(a) = add(spi,rseti))
                         AND (EXISTS spo,rseto: outport(a) = add(spo,rseto)) }

ROUTERS: TYPE = set[ROUTER]
routers: ROUTERS
router(a:PreNode):bool = member(a,routers)

%rcu

RCU: TYPE = {a:PreNode | (EXISTS spi: inport(a) = singleton(spi) AND
                          let spo2:OUTPORT = peer_io(spi) IN
                          router(node(spo2)) AND sp(spo2))
                   AND (EXISTS spo: outport(a) = singleton(spo) AND
                          let spi2:INPORT = peer_oi(spo) IN
                          router(node(spi2)) AND sp(spi2)
                         )
                    AND (FORALL spi,spo: member(spi,inport(a)) AND
                         member(spo,outport(a)) AND EXISTS n1:
                         n1= node(peer_io(spi)) AND n1= node(peer_oi(spo)))
                }

RCUS: TYPE = set[RCU]
rcus: RCUS
rcu(a:PreNode):bool = member(a,rcus)

IP(n:PreNode):bool = ni(n) OR router(n) OR rcu(n)
```

```
NODE: TYPE = (IP)

end node
```

A.4 Slot table

Slot table definition with router output ports and time slot as an input and router input port as output.

```
slot_table: THEORY
begin

importing node

%The output port is from the router
SL_OUTPORT: TYPE = {p:OUTPORT | router(node(p)) AND r_port(p)}

%The input port is from the router or dmy
% dmy (dummy input port) means the outport is not reserved
SL_INPORT: TYPE =  {p:INPORT  | dmy(p)
                                OR ( router(node(p)) AND r_port(p))}

SLOT_TABLE: TYPE = [SL_OUTPORT,nat->SL_INPORT]

reserved_by(stb:SLOT_TABLE,o1:SL_OUTPORT,st:nat,i1:SL_INPORT):bool =
        notdmy(i1) AND i1=stb(o1,st)

unreserved(stb:SLOT_TABLE,o1:SL_OUTPORT,st:nat):bool =
        dmy(stb(o1,st))

already_reserved(stb:SLOT_TABLE,o1:SL_OUTPORT,st:nat):bool =
        notdmy(stb(o1,st))

end slot_table
```

A.5 Buffer Address

Definition of buffer as a Cartesian product of input and output port of a node and Buffer classification into system and acknowledgment classes.

```
buffer_address: Theory
Begin

IMPORTING node

BUFFER: TYPE =
        [# inport:(notdmy[INPORT]),
            outport:{o1:(notdmy[OUTPORT]) | node(o1) = node(inport)} #]

b: VAR BUFFER

node(b):NODE = node(inport(b))

CAPACITY:  TYPE = nat
capacity:[BUFFER->CAPACITY]

% [sp,sp] buffer - for RCU
rcu_buffer(b):bool =  sp(inport(b)) AND  sp(outport(b))

% [sp,normal] buffer - for ROUTER
router_from_rcu_buffer(b):bool =  sp(inport(b)) AND  normal(outport(b))

% [normal,sp] buffer - for ROUTER
router_sys_buffer(b):bool =  normal(inport(b)) AND  sp(outport(b))

% [normal,normal] buffer - for ROUTER
router_ack_buffer(b):bool =  normal(inport(b)) AND  normal(outport(b))
```

```
% [sp,ap] buffer - for NI
ni_ack_buffer(b):bool =  sp(inport(b)) AND  app(outport(b))

% [app,sp] buffer - for NI
ni_sys_buffer(b):bool =  app(inport(b)) AND  sp(outport(b))

%anip and pnip system/acknowledgment buffer
anip_sys_buffer(b):bool = ni_sys_buffer(b) AND anip(node(b))
anip_ack_buffer(b):bool = ni_sys_buffer(b) AND anip(node(b))
anip_buffer(b):bool = anip_ack_buffer(b)OR anip_sys_buffer(b)

pnip_sys_buffer(b):bool = ni_sys_buffer(b) AND pnip(node(b))
pnip_ack_buffer(b):bool = ni_sys_buffer(b) AND pnip(node(b))
pnip_buffer(b):bool = pnip_ack_buffer(b)OR pnip_sys_buffer(b)

% ANIP and PNIP buffers
ni_buffers(b): bool = ni_ack_buffer(b) OR ni_sys_buffer(b)

%router buffers
router_buffer(b):bool = router_sys_buffer(b)  OR
                        router_from_rcu_buffer(b) OR
                        router_ack_buffer(b)

%system buffer
sys_buffer(b):bool = router_sys_buffer(b)  OR
                     router_from_rcu_buffer(b) OR
                     ni_sys_buffer(b) OR rcu_buffer(b)

%ack_buffer
ack_buffer(b):bool =         router_ack_buffer(b) OR
                        router_from_rcu_buffer(b) OR
                        ni_ack_buffer(b)

%non final = not app(outport(b))
non_final_buffer(b):bool =  router_ack_buffer(b) OR sys_buffer(b)

END buffer_address
```

A.6 Packet

Definition of packet as a record and operations on packet

```
packet[NI:TYPE,anip:[NI->bool],pnip:[NI->bool]]: THEORY

BEGIN

PACKET_TYPE: TYPE = {SETUP,TDOWN,ACK,NACK,EMPTY}
SLOT: TYPE = nat
HUB: TYPE = nat

PACKET: TYPE = [# ptype: PACKET_TYPE,        % type of the packet
                  psrc: NI,                 % source (ANIP or PNIP)
                  pdes: NI,                 % destination
                  pslot: SLOT,                 % slot number to reserve
                  phub: HUB                 % hub counter
               #]

MAX_SLOT: nat
MINIMUM_SLOT_OVERHEAD: nat

p,p2: VAR PACKET
%n1: VAR NODE
st1,st2: VAR SLOT

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

empty(p):PACKET = p WITH [ ptype:= EMPTY]

empty(p):bool = ptype(p) = EMPTY
```

```
%system packet
sys_packet(p):bool = (ptype(p)=SETUP OR ptype(p)=TDOWN ) AND
                        pnip(pdes(p)) AND anip(psrc(p))

%acknowledgment packet
ack_packet(p):bool =  (ptype(p)=NACK OR ptype(p)=ACK ) AND
                        anip(pdes(p)) AND pnip(psrc(p))

promote_SETUP(p):PACKET =
        p WITH  [ phub := phub(p) + 1, pslot:=pslot(p)+1]

promote_TDOWN(p):PACKET =
                p WITH  [ phub := phub(p) - 1, %  > 0 from "state.pvs"
                          pslot:=pslot(p)+1]

new_packet(p:PACKET,a:(anip),cur_slot:SLOT):bool =
        ptype(p) = SETUP         % type is SETUP
        AND phub(p) = 0         % hub counter is zero
        %its slot number should refer to future time (well ahead)
        AND pslot(p) > cur_slot + MINIMUM_SLOT_OVERHEAD
        % the source is the generating ANIP
        AND psrc(p) = a
        % destination is a PNIP
        AND pnip(pdes(p))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% a router can bounce the packet back to the source
% if the slot required is not availble
fail(p):PACKET =
        (# ptype:= NACK,
           psrc := pdes(p),
           pdes := psrc(p),
           pslot:= pslot(p),
           phub := phub(p)   #)

%TDOWN packets are generated in ANIP due to the arrival of NACK packet
%the src and destination of the TDWN packet is the destination and src
%of the NACK packet. The slot number is the same as the NACK
NEW_TDOWN_packet(p):PACKET =
        (# ptype := TDOWN,
           psrc  := pdes(p),
           pdes  := psrc(p),
           pslot := pslot(p)-phub(p), % >= 0, from state.pvs
           phub  := phub(p)        #)

%PNIP response
reply:[PACKET->PACKET]

reply_ax: AXIOM reply(p) = p2 IMPLIES
        (ptype(p2) = ACK OR ptype(p2) = NACK) AND
        psrc(p2) = pdes(p) AND
        pdes(p2) = psrc(p) AND
        pslot(p2) = pslot(p) AND
        phub(p2) = phub(p)

%After a TDOWN packet is generated from the NACK, then
% Mark the NACK to avoid multple TDOWN generation
% ... later the application will consume it
MARK_packet(p):PACKET = p WITH [phub:=0]

NEW_NACK(p):bool =
        ptype(p)=NACK and not p = MARK_packet(p)

END packet
```

A.7 Buffer Data

Definition of buffer content as a queue of packets and buffer capacity.

```
buffer_data: Theory
Begin

IMPORTING buffer_address
IMPORTING packet [NI,(anip),(pnip)]
IMPORTING Queue

BUFFERD: TYPE = Queue[PACKET]

end buffer_data
```

A.8 Local credit

Definition of local credit and operations on local credit.

```
credit[BUFFER_A:TYPE]:THEORY
BEGIN

LOCAL_CREDIT: TYPE = [ BUFFER_A -> nat]

lcredit: VAR LOCAL_CREDIT

credit_up_lcredit(lcredit:nat,sending:bool):nat =

             if(sending) then        lcredit+1
        else                        lcredit
        endif

credit_down_lcredit(lcredit:nat,receiving:bool):nat =

             if(receiving) then        lcredit-1
        else                        lcredit
        endif

up_eecredit(eecredit:nat,old_length:nat,new_length:nat):nat =
        eecredit + (old_length - new_length)

end credit
```

A.9 Network Data

The complete list of network content and operations that affect the content of the network as a whole. These operations are: sending, receiving, generating, consuming, PNIP's replay, updating flags and flow control credit counters and reservation.

```
network_data: Theory

begin

IMPORTING route
IMPORTING credit[BUFFER]
IMPORTING slot_table

DATA: TYPE = [#
        buffer:                [BUFFER -> BUFFERD],
        seli:                 [(nport[OUTPORT]) -> INPORT],
        flag:                 [BUFFER->bool],
        link:                 [(nport[INPORT]) -> PACKET],
        lcredit:        LOCAL_CREDIT,
        eecredit:         [(anip_sys_buffer) -> CAPACITY],
        slot_table:         SLOT_TABLE
        #]

b,b2: VAR BUFFER
p: VAR PACKET
intb:VAR BUFFERD
i: VAR nat
```

```
initial_data(d:DATA):bool =
FORALL b: empty?(buffer(d)(b)) AND
          dmy(seli(d)(outport(b))) AND
          flag(d)(b) AND empty(link(d)(inport(b))) AND
          lcredit(d)(b) = 0 AND
          ( anip_sys_buffer(b) IMPLIES
                ( FORALL b2: anip_ack_buffer(b2) AND node(b2) = node(b) AND
                        eecredit(d)(b) <= capacity(b) AND
                        eecredit(d)(b) <= capacity(b2))) AND
          ( router_buffer(b) IMPLIES
                ( FORALL i: dmy(slot_table(d)(outport(b),i))))



up_lcredit(old,new:DATA):bool =
FORALL b: lcredit(new)(b) =
        credit_up_lcredit(lcredit(old)(b),flag(old)(b))

down_lcredit(old,new:DATA):bool =
FORALL b: lcredit(new)(b) =
        credit_down_lcredit( lcredit(old)(b),
                             (inport(b)=seli(old)(outport(b))))

update_flag(old,new:DATA):bool =
FORALL b:  flag(new)(b) = (inport(b) = seli(old)(outport(b)))

read_from_link(old,new:DATA):bool =
FORALL b:
        LET        oldd:BUFFERD =  buffer(old)(b),
                  newd:BUFFERD =  buffer(new)(b),
                  linkp:PACKET =  link(old)(inport(b)) IN

        IF(NOT empty(linkp) AND outport(b) = route(inport(b),linkp)) THEN
                  newd = enqueue(linkp,oldd)
        ELSE
                  newd = oldd
        ENDIF

write_to_link(old,new:DATA):bool =
FORALL b:
        IF(inport(b)=seli(old)(outport(b))) THEN
                buffer(new)(b) = dequeue(buffer(old)(b)) AND
                link(new)(peer_oi(outport(b))) = first( buffer(old)(b))
        ELSE
                buffer(new)(b) = buffer(old)(b)
        ENDIF

%ANIP generate SETUP
generate_SETUP(old,new:DATA,cur_slot:SLOT):bool =
FORALL b: anip_sys_buffer(b)  AND

LET         %oldab = old value of the acknowledgment buffer
        oldab:BUFFERD = buffer(old)(b),
        newab:BUFFERD = buffer(new)(b)   IN

         ( buffer(new)(b) = buffer(old)(b)
    OR
        (eecredit(old)(b) > 0 AND
           ( EXISTS p: new_packet(p,node(b),cur_slot) AND
               buffer(new)(b) = enqueue(p,buffer(old)(b)))
        )
    ) AND
        eecredit(new)(b) =  up_eecredit(eecredit(old)(b),
                                length(buffer(old)(b)),
                                length(buffer(new)(b)))

%ANIP generate TDOWN
generate_TDOWN(old,new:DATA): bool =
FORALL b: anip_ack_buffer(b) AND
EXISTS b2: anip_sys_buffer(b2) AND node(b2) = node(b) AND

LET         %oldab = old value of the acknowledgment buffer
```

```
        oldab:BUFFERD = buffer(old)(b),
        newab:BUFFERD = buffer(new)(b),
        oldsb:BUFFERD = buffer(old)(b2),
        newsb:BUFFERD = buffer(new)(b2)   IN

IF ( not empty?(oldab)) THEN

        LET  p:PACKET = first(oldab) IN

        IF(NEW_NACK(p)) THEN
                        EXISTS intb: oldab = enqueue(p,intb) AND
                        newab = enqueue(MARK_packet(p),intb) AND
                        newsb = enqueue(NEW_TDOWN_packet(p),oldsb)
         ELSE
                        newab = oldab AND newsb=oldsb
        ENDIF
ELSE
        newab = oldab AND newsb=oldsb
ENDIF

%ANIP consume
consume(old,new:DATA): bool =
FORALL b: ni_ack_buffer(b)  AND anip(node(b)) AND
        (  buffer(new)(b) = buffer(old)(b) OR
           buffer(new)(b) = dequeue(buffer(old)(b)) )
        AND eecredit(new)(b) =
                up_eecredit( eecredit(old)(b),
                length(buffer(old)(b)),
                length(buffer(new)(b))        )

%PNIP operation
pnip_reply(old,new:DATA): bool =
FORALL b: pnip_ack_buffer(b) AND

        EXISTS b2: ni_sys_buffer(b2) AND node(b2) = node(b) AND

        LET         %oldab = old value of the acknowledgment buffer
                oldab:BUFFERD = buffer(old)(b),
                newab:BUFFERD = buffer(new)(b),
                oldsb:BUFFERD = buffer(old)(b2),
                newsb:BUFFERD = buffer(new)(b2)   IN

        (newab = oldab AND newsb = oldsb)
        OR
        (
         length(oldsb)>0 and length(oldab)<capacity(b) AND
         newab = enqueue(reply(first(oldsb)),oldab) AND
                newsb = dequeue(oldsb)
        )

%RCU operation - Reserving and unreserving
reserve(old,new:DATA):bool =
FORALL b: rcu_buffer(b) AND
        LET
                p:PACKET = last(buffer(old)(b)),
                o2:OUTPORT = peer_io(inport(b))               IN

        (( empty(p) IMPLIES old = new) OR (

        LET        o1:OUTPORT = route(peer_oi(outport(b)),p),
                i1:INPORT = seli(old)(o2)                  IN

CASES ptype(p) OF

SETUP:          IF already_reserved(slot_table(old),o1,pslot(p)) THEN
          buffer(new)(b) = enqueue(fail(p),dequeue(buffer(old)(b)))
        ELSE
          buffer(new)(b) = enqueue(promote_SETUP(p),dequeue(buffer(old)(b)))
          AND reserved_by(slot_table(new),o1,pslot(p),i1)
        ENDIF,
TDOWN: unreserved(slot_table(new),o1,pslot(p)) AND
        IF(phub(p)>0) THEN
          buffer(new)(b) = enqueue(promote_TDOWN(p),dequeue(buffer(old)(b)))
```

```
        ELSE
             buffer(new)(b)= dequeue(buffer(old)(b)) %delete the TDOWN packet
             ENDIF
ENDCASES )
)


End network_data
```

A.10 Route

Route function, axioms and dependency graph.

```
route: THEORY

begin IMPORTING buffer_data

route:[        i1:(notdmy[INPORT]),p:PACKET
                -> {o1:(notdmy[OUTPORT])| node(i1) = node(o1)}
      ]

i1: VAR (notdmy[INPORT])
o1: VAR (notdmy[OUTPORT])
p: VAR PACKET

route_ax: AXIOM
        route(i1,p) = o1 IMPLIES
                LET n1:NODE = node(i1),
                    b:BUFFER = (#inport:=i1,outport:=o1#) IN

                ( sys_packet(p) AND sys_buffer(b) ) OR
                ( ack_packet(p) AND ack_buffer(b) ) OR
                ( pnip(n1) AND ni_ack_buffer(b) )

dep_graph(b1,b2:BUFFER):bool =
EXISTS p: route(peer_oi(outport(b1)),p) = outport(b2)

%The properties of buffer routing holds for dep_graph
IMPORTING broute[BUFFER,(sys_buffer), (dep_graph)]

end route
```

A.10.1 Dependency Graph

Dependency graph, paths and cycles in dependency graph. Lemma on absence of cycle in dependency graph.

```
broute[T: TYPE, ack?:[T->bool],buffer_class_routing:[T,T->bool]]: THEORY
BEGIN

ASSUMING

t,t1,t2: VAR T
lt: VAR list[T]
i,j: VAR nat

buffer_class_routing_as: ASSUMPTION
        buffer_class_routing(t1,t2) = ack?(t1) OR NOT ack?(t2)

ENDASSUMING

sys?(t):bool = NOT ack?(t)

path?(lt):bool =
length(lt) > 1 AND
       FORALL i: i > 0 AND i < length(lt) IMPLIES
         buffer_class_routing(nth(lt,i-1),nth(lt,i))
```

```
ackpath?(lt):bool =
        FORALL i: i >= 0 AND i < length(lt) IMPLIES ack?(nth(lt,i))

syspath?(lt):bool =
        FORALL i: i >= 0 AND i < length(lt) IMPLIES sys?(nth(lt,i))

prop2: LEMMA FORALL i: i >= 0 AND i < length(lt) AND path?(lt) IMPLIES
                ( sys?(nth(lt,i)) OR ack?(nth(lt,i)) )

prop2a: LEMMA FORALL i: i > 0 AND i < length(lt) AND path?(lt)
                AND sys?(nth(lt,i-1)) IMPLIES
                        sys?(nth(lt,i))

prop2b: LEMMA FORALL i: i > 0 AND i < length(lt) AND path?(lt)
                AND ack?(nth(lt,i)) IMPLIES
                        ack?(nth(lt,i-1))

prop3c: LEMMA  FORALL i: i >= 0 AND
                i < length(lt) AND
                path?(lt) AND
                sys?(nth(lt,i)) IMPLIES (
                FORALL j: i+j<length(lt) IMPLIES sys?(nth(lt,i+j)))

prop3d: LEMMA  FORALL i: i >= 0 AND
                i < length(lt) AND
                path?(lt) AND
                ack?(nth(lt,i)) IMPLIES (
                FORALL j: i-j>=0 IMPLIES ack?(nth(lt,i-j)))

prop3ca: LEMMA
        path?(lt) AND sys?(nth(lt,0)) IMPLIES syspath?(lt)


prop3da: LEMMA
        path?(lt) AND ack?(nth(lt,length(lt)-1)) IMPLIES ack?(nth(lt,0))


prop3de: LEMMA
        path?(lt) AND ack?(nth(lt,length(lt)-1)) IMPLIES ackpath?(lt)


cycle?(lt):bool = length(lt)>1 AND nth(lt,0) = nth(lt,length(lt)-1)

smcy: AXIOM ackpath?(lt) IMPLIES not cycle?(lt)

bgcy: AXIOM syspath?(lt) IMPLIES not cycle?(lt)

nocy:  LEMMA path?(lt) IMPLIES not cycle?(lt)

END broute
```

A.11 Arbiter

The arbiter function and its property

```
arbiter:THEORY

begin

IMPORTING network_data

i1,i2: VAR INPORT
o1,o2: VAR (nport[OUTPORT])
b1,b2:  VAR BUFFER
st:  VAR SLOT
data1: VAR DATA

arbiter:[DATA,(nport[OUTPORT]),SLOT->INPORT]

schedule(old,new:DATA,cur_slot:SLOT):bool =
FORALL o1:  seli(new)(o1) = arbiter(old, o1,cur_slot)
```

```
%i1 is dummy iff one of the three
%        1. o1 is already reserved
%       2. All destination buffers of the non empty source buffers are full
%           this means either
%                2.1 source buffers are empty or,
%                2.2 destination buffers are full

arbiter_ax1: AXIOM arbiter(data1,o1,st) = i1 AND dmy(i1) IFF
 (
        already_reserved(slot_table(data1),o1,st) OR

        (FORALL b1: outport(b1) = o1 AND nonempty?(buffer(data1)(b1)) IMPLIES
                LET i2 = peer_oi(o1),
                    p1 = first(buffer(data1)(b1)),
                       o2 = route(i2,p1),
                        b2 = (#inport := i2, outport:=o2#) IN
                 lcredit(data1)(b2)=0
        )
)

arbiter_ax2: AXIOM arbiter(data1,o1,st) = i1 AND NOT dmy(i1) IMPLIES

        LET b1 = (#inport := i1, outport:=o1#) IN

        NOT empty?(buffer(data1)(b1)) AND

        LET i2 = peer_oi(o1),
            p1 = first(buffer(data1)(b1)),
            o2 = route(i2,p1),
            b2 = (#inport := i2, outport:=o2#) IN
                lcredit(data1)(b2) >=1

%Two buffers of the same input port are never served similtanously
arbiter_ax3: AXIOM
 NOT o1 = o2 AND arbiter(data1,o1,st) = arbiter(data1,o2,st)
        IMPLIES dmy(arbiter(data1,o1,st))


end arbiter
```

A.12 State Transition

Network activity as a state transition. State variables and the three phases of operation.

```
state: THEORY

BEGIN

%IMPORTING route
IMPORTING arbiter

%The network operates in three phases
PHASE_TYPE: DATATYPE
   BEGIN    READ: READ?
            EXECUTE: EXECUTE?
            WRITE: WRITE?
   END PHASE_TYPE

%State variables
State: TYPE=[# data:DATA,
                phase: PHASE_TYPE,                  % transition phase
                cur_slot: SLOT                          % current time slot
        #]


read:[State,State->bool]
execute:[State,State->bool]
write:[State,State->bool]

s,t: VAR State
b: VAR BUFFER
```

```
read_ax: AXIOM read(s,t) IFF

        phase(s) = READ AND phase(t) = EXECUTE AND

        % in read phase
        % 1. new packets generated in anips and eecredit is updated
        % 2. local credit is updated
        % 3. buffers get data from the link(wire)
        read_from_link(data(s),data(t)) AND
        up_lcredit(data(s),data(t))  AND
        generate_SETUP(data(s),data(t),cur_slot(s)) AND

        % the following state variables will remain
        % unchanged in READ phase
        slot_table(data(t)) = slot_table(data(s)) AND
        seli(data(t)) = seli(data(s)) AND
        link(data(t)) = link(data(s)) AND
        cur_slot(t) = cur_slot(s) AND
        flag(data(t)) = flag(data(s))

execute_ax: AXIOM execute(s,t) IFF

        phase(s) = EXECUTE AND        phase(t) = WRITE AND

        schedule(data(s),data(t),cur_slot(s)) AND
        reserve(data(s),data(t)) AND
        generate_TDOWN(data(s),data(t)) AND

        % the following state variables will remain
        % unchanged in  this phase
        link(data(t)) = link(data(s)) AND
        cur_slot(t) = cur_slot(s) AND
        flag(data(t)) = flag(data(s)) AND
        eecredit(data(t)) = eecredit(data(s)) AND
        lcredit(data(t)) = lcredit(data(s)) AND
        (FORALL b: NOT (anip_buffer(b) or rcu_buffer(b)) AND
        buffer(data(t))(b) = buffer(data(t))(b))


write_ax: AXIOM write(s,t) IFF

        phase(s) = WRITE AND        phase(t) = READ AND
        cur_slot(t) = cur_slot(s)+1 AND

        write_to_link(data(t),data(s)) AND
        update_flag(data(t),data(s)) AND
        consume(data(t),data(s)) AND
        pnip_reply(data(t),data(s)) AND
        down_lcredit(data(s),data(t))  AND
        % the following state variables will remain
        % unchanged in  this phase
        slot_table(data(t)) = slot_table(data(s)) AND
        seli(data(t)) = seli(data(s)) AND
        eecredit(data(t)) = eecredit(data(s))


start_state(s):bool =
        initial_data(data(s)) AND
        phase(s) = READ AND
        cur_slot(s) = 0

end state
```

A.13 Automata and Reachability

Definition automata and reachability concept in PVS. Imported in theory NOC automaton.

```
automaton[Action: TYPE, State: TYPE]: THEORY
BEGIN
```

```
Automaton : TYPE =
        [# starts : setof[State],
           steps: [State,Action,State->bool] #]

A: VAR Automaton
s,t: VAR State
a: VAR Action

reachable_n(A,s,(n:nat)) : RECURSIVE bool =
   IF n = 0 THEN starts(A)(s) ELSE
     (EXISTS (t:State),(a:Action) :
                reachable_n(A,t,n-1) AND steps(A)(t,a,s))
   ENDIF
   MEASURE n

reachable(A,s): bool = EXISTS (n:nat): reachable_n(A,s,n)

isinv_n: LEMMA FORALL (A:Automaton, I:[State->bool]):
  (FORALL s: starts(A)(s) IMPLIES I(s)) AND
  (FORALL a,s,t: I(s) AND reachable(A,s) AND steps(A)(s,a,t) IMPLIES I(t))
 IMPLIES
  (FORALL (n:nat),s: reachable_n(A,s,n) IMPLIES I(s))

isinv: LEMMA FORALL (A:Automaton, I:[State->bool]):
  (FORALL s: starts(A)(s) IMPLIES I(s)) AND
  (FORALL a,s,t: I(s) AND reachable(A,s) AND steps(A)(s,a,t) IMPLIES I(t))
  IMPLIES
  (FORALL s: reachable(A,s) IMPLIES I(s))

isinv?(A: Automaton)(I:[State->bool]):bool = FORALL s: reachable(A,s) IMPLIES I(s)

END automaton
```

A.14 Deadlock

Deadlock definition as a state property.

```
deadlock: THEORY

BEGIN

IMPORTING state

b1,b2: VAR BUFFER
lb: VAR list[BUFFER]
i: VAR nat

deadlock?(s:State):bool =
EXISTS lb: path?(lb) AND cycle?(lb) AND
      FORALL i:        i>0 AND i<length(lb) AND
        LET        b1 = nth(lb,i-1),
                   b2 = nth(lb,i)  IN
        length(buffer(data(s))(b1)) = capacity(b1)

end deadlock
```

A.15 Simplified ANIP

Abstracted ANIP and its invariant

```
anip: THEORY

BEGIN

%capacity
C:[nat->nat]

state: TYPE=[#         ee: nat, l1: nat, l2: nat,
                       n: nat, % nack in buffer2, l2-n = ack buffers in buffer2
```

```
                g: nat, % setup packets sent
                m: nat  % setup packets not yet sent
                        % l1-m = tdown packets in buffer1 not yet sent
        #]

s,s1,s2: VAR state
i: VAR nat

ax1: AXIOM FORALL s1,s2: C(l1(s1)) = C(l1(s2))
ax2: AXIOM FORALL s1,s2: C(l2(s1)) = C(l2(s2))

generate_setup(s1,s2):bool =
if(ee(s1)>0 and l1(s1)<C(l1(s1)))

        THEN (s2 = s1 WITH [ee := ee(s1)-1, m:=m(s1)+1, l1:=l1(s1) + 1]) OR
                (s2 = s1)
        ELSE        (s2 = s1)
ENDIF

generate_tdown(s1,s2):bool =
if(n(s1)>0 and l1(s1)<C(l1(s1)))

THEN (s2 = s1 WITH [ n:= n(s1)-1, l1:=l1(s1)+1])
        OR        (s2 = s1)
        ELSE        (s2 = s1)
ENDIF

send_setup(s1,s2):bool =
if(l1(s1)>0 and m(s1)>0)

THEN (s2 = s1 WITH [ g:= g(s1)+1, l1:=l1(s1)-1, m:=m(s1)-1])
        OR        (s2 = s1)
        ELSE        (s2 = s1)
ENDIF

send_tdown(s1,s2):bool =
if(l1(s1)>0 and l1(s1)>m(s1))

THEN (s2 = s1 WITH [l1:=l1(s1)-1])
        OR        (s2 = s1)
        ELSE        (s2 = s1)
ENDIF

consume(s1,s2):bool =
if(l2(s1)>n(s1))

THEN (s2 = s1 WITH [ l2:= l2(s1)-1, ee:=ee(s1)+1])
        OR        (s2 = s1)
        ELSE        (s2 = s1)
ENDIF

yconsume(s1,s2):bool =
if(l2(s1)>n(s1))

THEN ( l2(s2) = l2(s1)-1 AND g(s2)=g(s1) AND m(s2)=m(s1) AND
        ee(s2)=ee(s1)+1 AND n(s2)=n(s1) AND l1(s2)=l1(s1))

        OR        (s2 = s1)
        ELSE        (s2 = s1)
ENDIF

receive_nack(s1,s2):bool =
if(g(s1)>n(s1))

THEN (s2 = s1 WITH [ l2:= l2(s1)+1, g:=g(s1)-1, n:= n(s1)+1])
        OR        (s2 = s1)
        ELSE        (s2 = s1)
ENDIF

receive_ack(s1,s2):bool =
if(g(s1)>n(s1))

THEN (s2 = s1 WITH [ l2:= l2(s1)+1, g:=g(s1)-1])
```

```
        OR        (s2 = s1)
        ELSE        (s2 = s1)
ENDIF

steps(s1,s2):bool =
generate_setup(s1,s2) OR
generate_tdown(s1,s2) OR
send_setup(s1,s2) OR
send_tdown(s1,s2) OR
consume(s1,s2) OR
receive_nack(s1,s2) OR
receive_ack(s1,s2)


start_state(s1):bool =
l1(s1)= 0 AND l2(s1)= 0     AND
g(s1)= 0  AND n(s1)= 0      AND
m(s1)= 0  AND
ee(s1) <= C(l1(s1)) AND
ee(s1) <= C(l2(s1))

sl: VAR list[state]

reachable_path(sl):bool = length(sl) > 0 AND start_state(nth(sl,0)) AND
( FORALL i: i < length(sl)-1 IMPLIES steps(nth(sl,i),nth(sl,i+1)))

I1(s1):bool = ee(s1) + g(s1) + m(s1) + l2(s1) <= C(l2(s1))
I2(s1):bool = l2(s1) = C(l2(s1)) IMPLIES g(s1) = 0

p1: LEMMA  reachable_path(sl) IMPLIES
        (FORALL i: i< length(sl) IMPLIES I1(nth(sl,i)))

p2: LEMMA reachable_path(sl) IMPLIES
        (FORALL i: i< length(sl) IMPLIES I2(nth(sl,i)))

END anip
```

A.16 NoC automaton

The top level PVS specification of the NOC. NOC invariants.

```
noc_automaton: THEORY

BEGIN

IMPORTING deadlock
IMPORTING automaton
IMPORTING anip

step(s:State,ph: PHASE_TYPE, t:State):bool =
CASES ph OF
        READ:          read(s,t),
        EXECUTE:  execute(s,t),
        WRITE:          write(s,t)
ENDCASES

noc_automaton: Automaton[PHASE_TYPE,State] =
         (# starts:= start_state,
            steps:= step
         #)

s: VAR State
I1: LEMMA NOT deadlock?(s)

end noc_automaton
```