Specification of a Network-on-Chip

David Lester and Dominic Richards
The University of Manchester
{drl,dar}@cs.man.ac.uk

***Abstract*—With the increasing complexity of Systems-on-Chip, the clear specification of Network-on-Chip (NoC) architectures is of growing importance. We present an approach which uses Concurrent Haskell to produce concise formal specifications. We present a detailed high-level specification of a real-world NoC, the SpiNNaker Communications NoC.**

I. Introduction

There have been many attempts to use formal methods to specify and verify hardware [Har03], [Rus00], [SB08], [Fox03], [BH91]; why do we wish to make another attempt? Our approach produces specifications which capture concurrent systems in a concrete way, and are executable. Whereas previous attempts to verify hardware have tended to focus on serial hardware units, which typically have 'functional' characteristics (limited side effects) [Har03], [Rus00], our approach uses a concurrent specification language to capture systems which do not readily lend themselves to direct specification in purely functional languages. For a brief introduction to Concurrent Haskell, see [JGF96]. Concurrent Haskell is a language which expresses concurrency at a high level of abstraction, allowing concise specifications of concurrent systems. It is an extension of the purely functional language Haskell [Jon03]. Pure Haskell and Concurrent Haskell have the dual identities of being programming languages and mathematical calculi; this means that programs are also formal specifications.

II. Introducing SpiNNaker

SpiNNaker [FTB06b], [FTB06a] is a bespoke chip multiprocessor (CMP) being designed at the University of Manchester, within an EPSRC-funded project in collaboration with the University of Southampton, ARM Limited and Silistix Limited. It is being produced specifically for the real-time simulation of large-scale spiking neural networks. The chip (along with its associated SDRAM chip) forms a node in a scalable parallel system, interconnected to the other nodes through self-timed links. Funding is currently in place for the construction of a 500 node system for the execution of models produced by the University's School of Psychological Sciences.

The on-chip processing power is provided by 20 ARM968 cores. Each ARM models up to 1,000 neurons in continuous 'real' time. Neurons communicate through atomic 'spike' events, and these are communicated as discrete packets via an on- and inter-chip communications fabric, formed by the concurrent operation of the chips' NoCs, together with self-timed inter-chip links. A packet contains only the address of the source neuron; this source-address is used as a routing key by the NoCs it passes through as it is forwarded to its destination neurons.

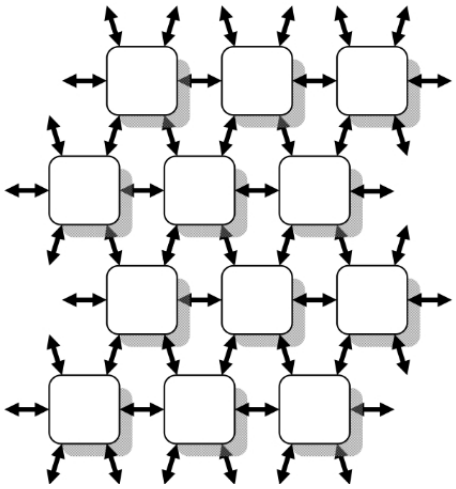

Fig. 1. Nodes (Chip+RAM) in a SpiNNaker System

A. The Communications Sub-System

The system is implemented as a 2D array of nodes interconnected through bi-directional links in a triangular formation as illustrated in Fig. 1.

Within the chip, a NoC connects the ARM cores and inter-chip links. The NoC is asynchronous, and serves to decouple the many different clock domains across the chip. Within the NoC there is an arbiter and a router. A packet entering the NoC from a processor or inter-chip link is passed through the arbiter to the router, which passes it to a subset of the processors and inter-chip links, as directed by a source-address lookup-table. In this way, the NoCs of each chip work together facilitate packet-routing across the entire system.

The SpiNNaker system places a heavy emphasis on fault tolerance. If an inter-chip link fails, a hardware mechanism exists to re-route packets via an adjacent node. If a processor fails, its workload can be re-distributed across other processors in the system. The details of the hardware re-routing mechanism will be introduced via the formal specification in the next section. The operation of workload-migration is not considered further in this paper.

III. Specifying the SpiNNaker NoC

We now specify the SpiNNaker multicast communication network in Concurrent Haskell. We model SpiNNaker chips as concurrent processes, communicating with each other via

Chans, which represent the inter-chip links. Each chip has a NoC model running as a concurrent process, and 20 ARM core models, again each running as concurrent process. Because we are only interested in the communications network, we use a very simple model for the ARM cores, which simply accepts and generates traffic; they are functions that wait to take packets and then emit new ones, with their own address as the source address. The type of their addresses is ProcID, which is a synonym of the type Int. Packets are represented by ProcID lists, together with a value of PacketType which is used for fault tolerance and will be explained below. When a packet is in the communications network, the head of the ProcID list is treated as the packet's source address. When a processor receives a packet, it adds its own address to the head of the list, and passes the new packet to its NoC; in this way, a packet 'trace' is formed as the list grows.

```
type ProcID     = Int

type Packet     = ([ProcID], PacketType)

data PacketType = N | E1 | N_E1 | E2
  deriving (Show, Eq)

proc :: ProcID -> Chan Packet -> Chan Packet
                                    -> IO()
proc id i o =
  loop (readChan i >>= \(ids,t) ->
                       writeChan o (id:ids,t))

loop :: IO() -> IO()
loop f = f >> (loop f)
```

loop achieves infinite looping by evaluating f, then discarding the result and calling itself with f. Its type definition specifies that it can only take an argument of the type IO() which is an entirely side-effecting action.

The behaviour of proc is to read its input Chan, which yields a value of type IO Packet, unwrap the Packet from this and write a new packet to its output Chan. The new Packet is identical to the Packet it received, except with the processor's ProcID added to the head of the ProcID list.

For the processor-NoC links, we take as a initial assumptions that:

- **Initial Assumption 1:** The NoC-processor links will never fail.
- **Initial Assumption 2:** The processors and NoC fabric will always process packets fast enough to prevent deadlock across the NoC-processor links.

The model for inter-chip links is slightly more complicated than the processor-NoC link model because we allow inter-chip links to break. To model potentially broken links, we introduce the data-type:

```
data Channel = Working (Chan Packet) | Broken
```

It turns out to be more convenient if we use this type for the processor-NoC links also, so we modify the definition of proc:

```
proc :: ProcID -> Channel -> Chan Packet -> IO()
proc id i (Working o) =
 loop (readChan i >>=
         \(ids,t) -> writeChan o (id:ids,t))
```

Trying to evaluate proc id i Broken would cause an execution error; this will never happen because Channels to and from procs will always be working.

We now introduce the NoC specification. The NoC consists of an arbiter and a router. The arbiter removes packets from all incoming inter-chip links and on-chip processor outputs, and writes them to a Chan which is read by the router.

```
type NocNode     = Int

arbiter::[Channel]->Chan(NocNode,Packet)->IO()
arbiter outputs routerIn =
 ioFold
  (\nodeID->launchNode (outputs!!nodeID) nodeID)
  [0.. length outputs - 1]
 where
  launchNode output id =
   case input of
    Working o->forkLoop (readChan o >>=
                  \p->writeChan (id,p) routerIn)
    Broken   ->return ()

ioFold :: (a -> IO()) -> [a] -> IO()
ioFold f ps =
 foldl (\io p -> io >> f p) (return ()) ps

forkLoop :: IO() -> IO()
forkLoop f = forkIO (loop f) >> return ()
```

The axillary function ioFold takes as arguments:

1) a function which takes a value of type a and returns a side-effecting action
2) a list of values of type a

It then applies the function to every element in the list. In this way it is similar the the foldl function introduced in Section 2, but for side-effecting functions.

We define the type NocNode as a synonym for Int; it is used to identify which NoC node a packet entered the NoC from. The function arbiter takes as arguments:

1) a list of Channels- these are the outputs from on-chip processors and inter-chip links
2) a Chan which takes NocNodes paired with Packets- this is the input channel for the router.

Its behaviour is to launch a concurrent process for each Working Chan it was passed, each carrying a unique ProcID, which is derived from the position the Channel occupied in the list that was passed to arbiter. These concurrent processes run an infinite loop, continuously executing the sub-function:

```
readChan o >>= \p -> writeChan (id,p) routerIn
```

This reads the NoC node output Chan it was passed, and places the Packet it receives, together the NocNode value, in the Chan that is read by the router. We introduce a new initial assumption:

- **Initial Assumption 3:** The NoC arbiter will be 'fair' in so much as all input requests will be serviced eventually.

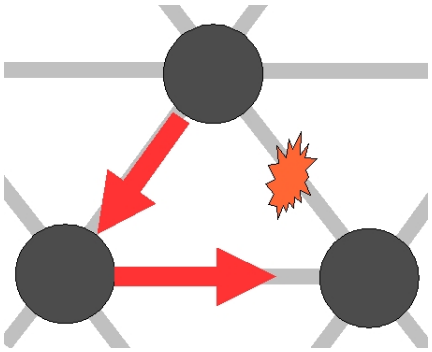

Fig. 2. Emergency Re-Routing

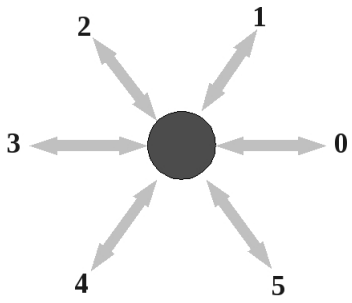

Fig. 3. Inter-Chip Links Numbering Convention

A. The NoC Router

The actual NoC Router is responsible for routing all packets that arrive at its input to one or more of the NoC outputs. Its primary function is to route multicast neural event packets, which it does through an associative multicast router sub-system, and this is the focus of our verification. It is also responsible for routing point-to-point packets (for which it uses a look-up table) and nearest-neighbour routing (which is a simple algorithmic process); these routing mechanisms are not verified in this paper, but we believe that their verification would be a simple extension of the current work.

Packets enter the NoC from on-chip processors and inter-chip links and are presented to the router one-at-a-time by the arbiter. The router then uses the packet's source-address to access a lookup table (in the hardware design, this is a content-addressed memory unit). The lookup table returns the set of NoC nodes to which the packet should be sent.

If an inter-chip link fails (temporarily, due to congestion, or permanently, due to component failure), the blocked link will be detected in hardware and subsequent packets rerouted via the other two sides of one of the routing triangles of which the suspect link was an edge (Fig. 2). The routing triangle selected is predetermined, as described in the specification below. If either of the two links on this new path are broken, the packet will be dropped. It is acceptable to drop a certain number of packets because of the robust nature of neural computation that SpiNNaker will simulate.

For our specification, we apply the numbering convention of Fig. 3 to a chips' inter-chip links. There are 26 NoC nodes; nodes 0-5 are inter-chip links and nodes 6-25 are on-chip processors. We also need a convention for referring to links in relation to other links on the same chip. For a given link i, we refer to:

- the clockwise neighbour $((i-1)\ mod\ 6)$
- the clockwise second neighbour $((i-2)\ mod\ 6)$
- the diametrically opposite link $((i \pm 3)\ mod\ 6)$
- the anti-clockwise second neighbour $((i+2)\ mod\ 6)$
- the anti-clockwise neighbour $((i+1)\ mod\ 6)$

The fault-tolerant re-routing mechanism, called emergency routing, proceeds as shown in Fig. 4. Imagine that chip A wants to send a packet down link AB to chip B, but link AB is unresponsive. The router on chip A's NoC attaches a label "E1" (emergency, stage 1) to the packet and tries to pass it to chip C. If link AC is unresponsive, the packet is dropped (Case 1 in Fig. 4). If link AC is functioning and chip C receives a packet with the label "E1" attached, it automatically changes the label to "E2" and tries to send it down the clockwise neighbour of the source link, which in this case is link CB. If link CB is unresponsive, it drops the packet (Case 2 in Fig. 4).

If the packet reaches chip B, its NoC router removes the "E2" label and treats it as a normal packet (Case 3 in Fig. 4).

The system also implements a 'default' routing mechanism, which allows packets to be routed through chips without an entry being required in the chip's look-up table. If a router receives a packet from an inter-chip link and the packet's source-address is not in its look-up table, is forwards the packet to the diametrically opposite inter-chip link. In this way, look-up table entries are required only when packets originate on the chip, are destined for on-chip processors, or change their 'direction of travel' at the chip.

If a packet is to be passed down an inter-chip link with the `E1` packet type, it is possible that the same packet should be passed down this link anyway with `N` packet type. If this is the case, the router sends a single packet with type `N_E1`. When a packet of this type is received, it is split into an `N` packet and an `E1` packet, which are then processed independently.

We model the router lookup table as a function of type `Lookup`, which maps from `Packet` to `Maybe [NoCNode]`. `Maybe` is a standard data-type:

```
type Lookup = Packet -> Maybe [NocNode]

data Maybe a = Nothing | Just a
```

We do not provide our concrete lookup function here as it is inconsequential to the specification.

Our Haskell router specification is given in Fig.5 in the appendix. A function `router` launches a concurrent process, which continually reads packets from the router input channel and passes them, along with a number of other items, to a supporting `route` function. The items it passed are:

- the packet
- the result from the router lookup table, which tells `route` which NoC nodes to forward the packet to.

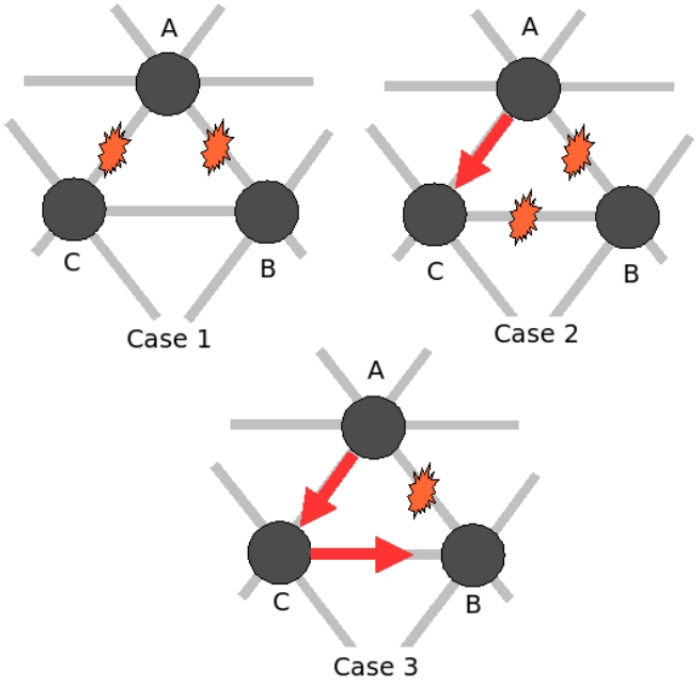

Fig. 4. Emergency Re-Routing Scenarios

- two lists of `Channel`s, which are the input channels for the on-chip processors and inter-chip links.

`route` is then responsible for copying the packet to the appropriate inter-chip links and on-chip processors, as well as initiating and handling emergency routing, and detecting erroneous packets. It achieves this as follows:

- If the packet originated from an on-chip processor and the packet type is not `N`, the expression `badLocalPacket` evaluates to `True`, triggering the expression to its right. This scenario should never occur because the only time a packet type is not `N` is during emergency routing, when it is being routed between chips and not from a processor to its NoC. If this situation occurs the function `badLocalPacketHandler` is called, the behaviour of which is not discussed here.
- If `badLocalPacket` evaluated to false, we next check whether the packet has type `N_E1`. If it is, the packet is split into two packets. Both are identical to the original except that one has packet type `N` and one `E1`. `route` is then called on both of these.
- If the lookup table returns `Nothing` and the packet came from an on-chip processor, `localMiss` evaluates to true true and we have detected another form of error. This situation should never happen because the only time the lookup table should return `Nothing` is for default routing, where a packet enters the chip through an inter-chip link and is forwarded to the diametrically opposite inter-chip link. In this case, the function `localMissHandler` is called.
- Whenever the state is not `N` or the lookup returns `Nothing`, `forwardPacket` is true and we have one of two scenarios:
  1) If the state is not `N`, then we know from the falsity of `badLocalPacket` that the packet originated from an inter-chip link. We have a packet which is being passed for emergency routing.
  2) If `dests == Nothing`, we know from the falsity of `localMiss` that the packet came from an inter-chip link. We have a valid case for default routing. Note that `forward` can be applied to an `E2` packet, if a packet should under normal conditions have been passed through the chip with default routing. If this is the case, the packet is passed to the anti-clockwise second neighbour of the source link, which is diametrically opposite to the broken link that the packet would have entered the chip from.

  The cases for default and emergency routing share in common the fact that they can be handled without referring to the lookup table. For this reason, they are both passed to a function `forward` which deals with this scenario.
- `otherwise`. If none of the guards evaluate to true, we know that we have a packet with type `N`, for which the lookup returned `Just ds`. This class of packets is handled by the multicasting routing function, `multicast`.

The function `forward` is called by `route` to deal with any situation where a packet is forwarded without referring to the lookup table entry. This is necessary for default and emergency routing:

- If `state == E1`, the procedure for "emergency routing, stage 1" is invoked. If the clockwise neighbour of the source link is working, the packet state is changed to `E2` and it is passed down the link. If the link is broken, the packet is dropped.
- If `state == E2`, the procedure for "emergency routing, stage 2" is invoked. The packet is passed to the function `writeToICLs`, together with a target inter-chip link, which is the anti-clockwise second neighbour of the source link. `writeToICLs` will then pass the packet to the appropriate inter-chip link. In this case, the effect will be to launch the `E2` packet along the same trajectory it was on before encountering the broken link.
- If `state == N`, we have straight-forward default routing. The packet is passed to the function `writeToICLs`, with a target inter-chip link which is diametrically opposite to the packet's source link.

The function `multicast` is invoked by `route` to write a packet to one or more NoC node inputs. `multicast` simply invokes two functions, `writeToProcs` and `writeToICLs`, which forward the packet to the appropriate on-chip processors and inter-chip links.

`writeToICLs`, which is called by both `forward` and `multicast`, has the task of passing packets to inter-chip links. In the case of a broken inter-chip link, this function also initiates the emergency re-routing procedure. The function cycles through all inter-chip links, looking for the following

conditions:

- If the link is working, but should not receive the packet, it may still have to carry an emergency re-routing packet. The anti-clockwise neighbour is inspected; if it is broken and should receive the packet, the emergency-routing procedure dictates that the packet should be sent down the present link. Hence, an `E1` packet is sent down the present link. This condition is recognised by the predicate `emergencyPacket`
- If the link is working and the packet should be passed down it, the anti-clockwise neighbour is inspected;
  1) If it is broken and should receive the packet, we have a case for emergency re-routing. An `N_E1` packet should be sent down the present link. This condition is recognised by the predicate `normAndEmerg`.
  2) If it is working, or not due to receive a packet, an `N` packet is sent down the present link. This condition is recognised by the predicate `normalPacket`.
- If the link is broken, the anti-clockwise neighbour is inspected; if it is broken and should receive the packet, the emergency-routing procedure dictates that the packet should be dropped. This condition is recognised by the predicate `dropped`.
- If none of the above conditions have occurred, no action should be taken for the present link.

B. Summary

This completes our description of the NoC and its formal specification. We hope this treatment conveyed the complexity of the design, and highlighted the need for clear and unambiguous specification in the production of such communications fabrics.

IV. Conclusions and Related work

We are pleased that the SpiNNaker NoC, evidently a complicated piece of hardware, can be specified so succinctly. The process of formulating the SpiNNaker specification had several practical benefits; as well as elucidating the design concept from the previous pseudo-code specification, it brought to light a shortcoming in the system, which would have allowed erroneous routing tables to create livelock.

For an alternative approach to NoC specification, see [SB08]. Note the more abstract encapsulation of concurrency in this approach. Also note the more restricted notion of system execution, which involves the one-time routing of a single packet list. [MGPM04] reports on the formal specification and verification of the fault tolerance aspects of a communications fabric, using the in the PVS Automated Theorem Prover.

References

[BH91] Bishop Brock and Warren A. Hunt. Report on the formal specification and partial verification of the VIPER microprocessor. In *Compass '91: 6th Annual Conference on Computer Assurance*, pages 91–98, Gaithersburg, Maryland, 1991. National Institute of Standards and Technology.

[Fox03] Anthony Fox. Formal specification and verification of arm6. In *18th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2005*, volume 2758, pages 25–40. Springer-Verlag, 2003.

[FTB06a] S. Furber, S. Temple, and A. Brown. On-chip and inter-chip networks for modeling large-scale neural systems. *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pages 4 pp.–, 21-24 May 2006.

[FTB06b] Steve Furber, Steve Temple, and Andrew Brown. High performance computing for systems of spiking neurons. In *Proceedings of Adaptation in Artificial and Biological Systems AISB 2006*, 2006.

[Har03] John Harrison. Formal verification at intel. In *LICS '03: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, page 45, Washington, DC, USA, 2003. IEEE Computer Society.

[JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.

[Jon03] Simon P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.

[MGPM04] Paul Miner, Alfons Geser, Lee Pike, and Jeffrey Maddalon. A unified fault-tolerance protocol. In *In Yassine Lakhnech and Sergio Yovine, editors, Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems (FORMATS-FTRTFT), volume 3253 of Lecture*, pages 167–182. Springer, 2004.

[Rus00] David M. Russinoff. A case study in fomal verification of register-transfer logic with acl2: The floating point adder of the amd athlontm processor. In *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 3–36, 2000.

[SB08] Julien Schmaltz and Dominique Borrione. A functional formalization of on chip communications. *Formal Asp. Comput.*, 20(3):241–258, 2008.

Appendix

Fig. 5 gives the main body of the specification of the SpiNNaker Communications NoC, less the code given in section 3.

```
router :: MCLookup-> Chan(NocNode,Packet)-> [Channel]-> [Channel]-> IO()
router r routerIn procIns iclIns =
  forkLoop (readChan routerIn >>= \(i,p)-> route (r p) procIns iclIns i p)

route :: Maybe [NocNode]-> [Channel]-> [Channel]-> NocNode-> Packet-> IO()
route dests procIns iclIns src (id,state)
  | badLocalPacket = badLocalPacketHandler
  | state == N_E1  =  route dests procIns iclIns src (id,N)
                   >> route dests procIns iclIns src (id,E1)
  | localMiss      = localMissHandler
  | forwardPacket  = forward iclIns src (id,state)
  | otherwise      = multicast dests procIns iclIns src (id,N)
  where
    badLocalPacket  = src > 5 && state != N
    localMiss       = dests == Nothing && src > 5
    forwardPacket   = state == E1 || dests == Nothing
    iclDests  dests = foldl (\ps p -> if p< 6 then  p   :ps else ps) [] dests
    procDests dests = foldl (\ps p -> if p>=6 then (p-6):ps else ps) [] dests

forward :: [Channel] -> Int -> Packet -> IO()
forward iclIns src (ids,state) =
  case state of
    E1 -> e1Forward
    E2 -> defaultForward 2 (ids,N)
    N  -> defaultForward 3 (ids,N)
  where
    e1Forward
      | not (broken (src-1)) = writeToChan (iclIns!!(mod(src-1)6)) (ids,E2)
      | otherwise            = drop (ids,state)
    defaultForward n packet  = writeToICLs iclIns [(mod (src+n) 6)] packet
    broken  node = broken' (iclIns!!(mod node 6))
    broken' chan = case chan of Broken->True;(Working _)->False

multicast :: Maybe [NocNode]-> [Channel]-> [Channel]-> NocNode-> Packet-> IO()
multicast dests procIns iclIns src (id,N) =
  let Just ds = dests in
    writeToProcs procIns     (procDests ds) (id, N) >>
    writeToICLs  iclIns  src (iclDests  ds) (id, N)

writeToProcs :: [Channel] -> [NocNode] -> Packet -> IO()
writeToProcs procIns dests (ids,state) = ioFold (writeToProc (ids,state)) dests
  where
    writeToProc (ids,t,N,c) dest = writeToChan (procIns!!dest) (ids,N)

writeToChan :: Channel -> Packet -> IO()
writeToChan (Working chan) packet = writeChan chan packet

writeToICLs :: [Channel] -> [NocNode] -> Packet -> IO()
writeToICLs iclIns dests pack = ioFold (writeToICL pack) [0..5]
 where
  writeToICL (ids,N) node
    | normalPacket    node = writeToChan (iclIns!!node) (ids,N   )
    | emergencyPacket node = writeToChan (iclIns!!node) (ids,E1  )
    | normAndEmerg    node = writeToChan (iclIns!!node) (ids,N_E1)
    | dropped         node = drop (ids,N)
    | otherwise            = return ()
    where
      normalPacket    node = (busy   node) && (not (jammed (node+1))
      emergencyPacket node = (quiet  node) && (     jammed (node+1))
      normAndEmerg    node = (busy   node) && (     jammed (node+1))
      dropped         node = (jammed node) && (     broken (node-1))
       where
        quiet         node = not (receiving node) && not (broken node)
        jammed        node =      receiving node  &&      broken node
        busy          node =      receiving node  && not (broken node)
         where
          broken      node = broken' (iclIns!!(mod node (length iclIns)))
          broken'     chan = case chan of Broken->True;(Working _)->False
          receiving   node = isIn (mod node (length iclIns)) dests
          isIn n ps        = foldl (\x y -> x || (y==n)) False ps
```

Fig. 5. A Specification of the SpiNNaker Communications NoC