Submission for ESEC/FSE '97

# Software Design, Specification, and Verification: Lessons Learned from the Rether Case Study*

Xiaoqun Du, Kevin McDonnell, Evangelos Nanos,
Y.S. Ramakrishna, Scott A. Smolka
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794–4400
USA

**Abstract**

RETHER is a software-based real-time ethernet protocol developed at SUNY Stony Brook. The purpose of this protocol is to provide guaranteed bandwidth and deterministic, periodic network access to multimedia applications over commodity ethernet hardware. It has been implemented in the FreeBSD 2.1.0 operating system, and is now being used to support the Stony Brook Video Server (SBVS), a low-cost, ethernet LAN-based server providing real-time delivery of videos to end users from the server's disk subsystem.

Using local model checking, as provided by the Concurrency Factory specification and verification environment, we showed (for a particular network configuration) that Rether indeed makes good on its bandwidth guarantees to real-time nodes without exposing non-real-time nodes to the possibility of starvation. In the course of specifying and verifying Rether, we identified an alternative design of the protocol that warranted further study due to potential efficiency gains. Again using model checking, we showed that this alternative design also possesses the properties of interest.

**Keywords:** Specification, verification, software design, model checking, multimedia, real-time protocols, concurrent systems

Contact Author's Address:

Scott Smolka
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, USA
+1 (516) 632-8470 (voice)
+1 (516) 632-8334 (fax)
sas@cs.sunysb.edu (email)

*Research supported in part by NSF grant CCR-9505562, and AFOSR grants F49620-95-1-0508 and F49620-96-1-0087.

# 1 Introduction

Rether is a software-based real-time ethernet protocol developed at SUNY Stony Brook by Tzicker Chiueh and Chitra Venkatramani [CV95]. The purpose of this protocol is to provide guaranteed bandwidth and deterministic, periodic network access to multimedia applications over commodity ethernet hardware. It has been implemented in the FreeBSD 2.1.0 operating system, and is now being used to support the Stony Brook Video Server (SBVS), a low-cost, ethernet LAN-based server providing real-time delivery of videos to end users from the server's disk subsystem.

The MAC layer of ethernet uses the CSMA/CD protocol, which provides nondeterministic channel access to the nodes on the network. This form of access is unsuitable for the real-time multimedia applications such as video conferencing. Rether is a datalink layer protocol that runs on top of CSMA/CD and requires no modification to the ethernet hardware. It accommodates real-time applications by using a token to regulate the transmission of packets in the network. At any time, only the node that holds the token is allowed to transmit. Therefore, nodes are provided deterministic and contention-free channel access.

In discussions with Rether's developers, it became clear that they would welcome some kind of formal reassurance that their protocol does indeed deliver guaranteed bandwidth to real-time nodes without running the risk of starving non-real-time nodes. To this end, we set out to formally verify these properties of Rether using the *Concurrency Factory* specification and verification environment [CLSS96]. The primary features of the Factory are its graphical user interface for specification and simulation of hierarchical networks of communicating state machines; its VPL textual specification language for describing protocols involving complex data structures and value passing; and its suite of analysis routines for verifying properties of specifications.

One of the main analysis routines supported by the Factory is *model checking* [CE81, QS82, CES86], a verification technique for determining whether a system specification possesses a property expressed as a temporal logic formula. Model checking has enjoyed wide success in verifying, or finding design errors in, real-life systems. An interesting account of a number of these success stories can be found in [CW96]. A distinguishing aspect of the Factory's model checker is its use of *local model checking*, a technique aimed at combating state explosion by performing state space search in a need-driven fashion.

To carry out the proposed verification of Rether, we encoded the protocol in the Factory's VPL specification language and submitted this specification to the local model checker. A number of temporal logic properties were model checked, including the two mentioned above, and in all cases the specification was shown to possess the property of interest.

In the course of using the Factory to verify Rether, it occurred to us that an alternative design of the protocol might also be worth investigating. Rether, as it is currently implemented in FreeBSD, services the real-time nodes first during any given token rotation cycle; the period of a cycle is determined by the frequency of the real-time data transmissions. After servicing the real-time nodes, non-real-time requests are serviced until the cycle bandwidth is exhausted. We refer to this method as the *real-time first* (RTF) node servicing strategy. In our alternative strategy, nodes are serviced in sequential order (SQO) by their node ids. One of the benefits of the SQO strategy is that, under certain reasonable assumptions, it incurs less token-passing overhead than RTF. This is because a node wishing to transmit real-time and non-real-time data within one cycle may do so in SQO during the same visit of the token. RTF, on the other hand, would require the token to make a second visit to the node in question in order for it to transmit its non-real-time data.

SQO also requires less information to be stored in the token: in RTF, the token must maintain a bit vector indicating which nodes are in real-time mode.

In order to verify that the Rether developers could safely use our new strategy, we submitted an SQO-based VPL specification of Rether to the Factory's local model checker. Like the original RTF-based protocol, the model checker revealed that it satisfied all properties of interest. Interestingly, our model checking experiments show that the size of the Rether state space goes down by a factor of four in moving from RTF to SQO.

In light of the above, the following lessons were learned from our Rether case study:

1. By requiring one to write an abstract and formally rigorous specification of a system, specification and verification can have a positive (and perhaps unexpected) influence on software design. In our case, this was the discovery of the SQO node servicing strategy. We believe this phenomenon is primarily one of being able to see the forest through the trees once an abstract specification is in hand. This view is consistent with that of Parnas in [Par96], who wrote on the topic of software jewels:

   > [These] Programs were not just written; they had been planned, often in some pseudocode or a language other than the actual programming language.

2. Alternative software designs can be shown "equivalent" through verification.

3. An alternative software design may lead to a smaller state space, thereby facilitating model checking and other forms of verification.

4. Interesting properties of real-time protocols can be verified without the use of real-time formalisms, such as timed automata [AD94] or real-time logic [Mok91]. For our Rether case study, we abstracted real time into "time slots" and, in this context, it sufficed to use an untimed value-passing language (VPL) along with a non-real-time temporal logic (the modal mu-calculus [Koz83]).

The structure of the rest of the paper is as follows. Sections 2 and 3 provide overviews of the Concurrency Factory and Rether protocol, respectively. Our alternative software design is the subject of Section 4. Section 5 describes how we specified the protocol in VPL, both for RTF and SQO. Our model checking results are presented in Section 6. Finally, Section 7 concludes.

## 2 The Concurrency Factory

The Concurrency Factory [CLSS96] is an integrated toolset for the specification, verification, and implementation of concurrent and distributed real-time systems such as communication protocols and process control systems. The main features of the Factory are: a *graphical user interface* that allows the non-expert to design and simulate concurrent systems using GCCS, a graphical process algebra; a *textual user interface* for VPL; a *suite of verification routines* that currently includes a local model checker for the modal mu-calculus [Sok96], a very expressive temporal logic, and a local model checker for a real-time extension of the modal mu-calculus; and a *graphical compiler* that transforms GCCS and VPL specifications into executable Facile [TLK96] code.

We briefly describe here the components of the Factory that are most germane to our verification of the Rether protocol: VPL and the local model checker for the modal mu-calculus. VPL is well suited for specifying concurrent systems with non-trivial control structure and data structures, and which involve value passing (as opposed to pure synchronization). VPL-supported data structures include integers of limited size and arrays and records composed of such integers. Structural equivalence of data types is employed, i.e. two integer types are considered the same if their sizes are equal and two records are equivalent when they have fields of the same type appearing in the same order.

A system specification in VPL is a tree-like hierarchy of subsystems. A subsystem is either a *network* or a *process*. A network consists of a collection of subsystems running in parallel and communicating with each other through typed channels. Processes are at the leaves of the hierarchy. Each subsystem, be it a process or a network, consists of a header, local declaration and body. The header specifies a unique name for the subsystem and a list of "formal channels" (by analogy with formal parameters of procedures in programming languages). The names of the formal channels of a subsystem can be used in the body of the subsystem and represent events visible to an external observer.

Declarations local to a network include specifications of the subsystems of the network and channels for communication between subsystems, hidden from the outside world. The body of a network is a parallel composition of its subsystems. A subsystem declared within a network can be used arbitrarily many times; each time a new copy of the subsystem is instantiated with actual channels substituted for the formal ones. Actual channels must match the formal ones introduced in the header and must be declared either as local channels or formal channels of the network immediately containing the subsystem.

Declarations local to a process consist of variable and procedure declarations. Procedure bodies, like process bodies, are sequences of statements. Simple statements of VPL are assignments of arithmetic or boolean expressions to variables, and input/output operations on channels. Complex statements include sequential composition, `if-then-else`, `while-do`, and nondeterministic choice in the form of the `select` statement.

VPL programs can be verified using LMC, the Concurrency Factory's local model checker for the so-called alternation-depth-2 fragment of the modal mu-calculus. LMC first produces a graph representation of the logical formula under consideration and then computes the product of this graph with the labeled transition system (guaranteed to be finite-state) underlying the VPL program. This "product graph" is constructed on-the-fly: each product graph node has an associated boolean value (representing the value of a subformula in an LTS state) and the value of a node is computed based on the values of its successors. Values of the terminal nodes in the product graph can be decided immediately based on the node itself. Nodes are never constructed unless their values are needed to determine the value of a previously constructed node.

To further reduce the size of the product graph, LMC avoids interleaving process transitions whenever the order of interleaving cannot affect the truth of a logical formula. Intuitively, this is achieved by enlarging the granularity of process steps so as to avoid redundant interleavings. LMC is also equipped with a diagnostic facility which allows the user to request that the contents of the depth-first search stack be displayed whenever a certain "significant event" occurs—for instance, when the search first encounters a state at which a logical variable is determined to be either true or false. This technique can be used to great effect, for instance, to discover execution paths leading to deadlocked states, or to livelock cycles.

# 3 The Rether Protocol

Rether is a contention-free token bus protocol for the data link level, designed to run on top of a CSMA/CD physical layer. A network running the protocol normally operates in CSMA/CD mode, transparently switching to Rether mode when one or more nodes generate requests for real-time (RT) connections. An initialization protocol is used to coordinate the switch to Rether mode. Once in Rether mode, a token is used to control access to the medium. That is, a node can transmit data on the network only when it has the token.

The token circulates the network in cycles, each of which is called a *token rotation cycle*. The period of a cycle, the *token cycle time*, is a configurable parameter determined by the frequency of real-time data transmissions. For instance, $TCT$ (as this parameter is called) can be set to 33.3 $ms$ for video applications, which typically transmit data at the rate of 30 frames per second. In each token cycle, all nodes that have successfully made a bandwidth reservation (the so-called "admitted nodes") can send out RT data in accordance with their reservations. In the time remaining in the cycle, nodes are permitted to send out non-real-time (NRT) data. To be fair to NRT transmission requests, part of each cycle, $T_{nrt}$, is set aside for NRT traffic.

An RT application is admitted into the system by the protocol only if there is sufficient bandwidth in the cycle. That is, suppose this application requests a reservation for an RT transmission of duration $THT_{RT_{new}}$ (where the $THT$ stands for "token holding time"). Then this request is granted only if the following inequality holds:

$$\left(\sum_{i \in RT\_set} THT_{RT_i}\right) + THT_{RT_{new}} + T_{nrt} \leq TCT$$

where $RT\_set$ is the set of admitted RT nodes, and $THT_{RT_i}$ is the token holding time of the $i$th RT node. It is precisely this policy that enables the protocol to guarantee cycle bandwidth to RT applications.

Admitted RT nodes are serviced first in a cycle, and we refer to this strategy as RTF. A newly admitted RT application adds its node id to $RT\_set$, which is stored in the token as a bit vector. A node releasing RT bandwidth removes its id removed from $RT\_set$ before forwarding the token to the next node. The network reverts back to CSMA/CD mode when the last RT node releases its bandwidth. The token also carries a "residual time" counter, which records the amount of time remaining in a cycle. At the start of a cycle this field is initialized to $TCT$. Following a data transmission by the token holding node, the residual time counter is decremented by that node's token holding time before the token is passed to the next node.

The admission policy ensures that at least $T_{nrt}$ time remains in each cycle for NRT traffic. NRT transmissions are scheduled in a round-robin fashion to prevent the possibility of starvation. For this purpose, the token carries a field containing the id of the last node that failed to transmit its NRT data in a given cycle because of insufficient residual time.

To illustrate the behavior of the protocol, and the RTF node servicing strategy, consider the example network depicted in Figure 1. $RT\_set$ contains 1, 3, and 5 and these nodes are serviced first during each token rotation cycle. Assuming that each cycle can accommodate three NRT transmissions (in addition to the three guaranteed RT transmissions), and that each node has NRT data to transmit, it takes two successive cycles for the NRT transmissions to be serviced in round robin fashion. In particular, the following sequences of node transmissions are possible in successive cycles:

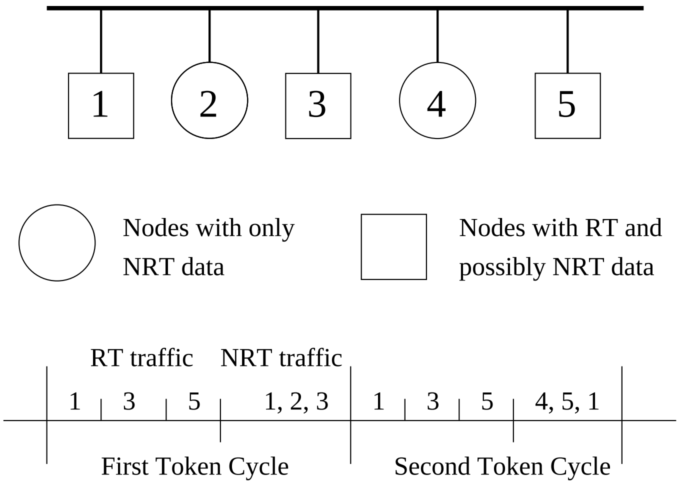


Figure 1: A Sample Network Configuration.

1 – 3 – 5 – 1 – 2 – 3
1 – 3 – 5 – 4 – 5 – 1

## 4 SQO: An Alternative Node Servicing Strategy

As mentioned in the Introduction, in the process of specifying and verifying Rether, we identified an alternative design of the protocol that warranted investigation because of potential efficiency gains. In this version, nodes are serviced in sequential order by their node ids, a strategy we refer to as SQO. In order to ensure periodic access to the token by RT nodes, the node with the smallest node id number is the first to receive the token in each cycle. Upon receiving the token, a node will transmit its RT data if it has previously been admitted by the protocol as an RT node. It will also, during the same token visit, transmit a fixed amount of NRT data if: (1) it has NRT data to transmit, and (2) the "next" pointer, which is stored in the token and used to implement round robin scheduling, indicates that sufficient NRT bandwidth remains in the current cycle to accommodate the node's transmission. The next-pointer is incremented when its value is $i$ and node $i$ has just completed an NRT transmission. Similarly, an NRT node (i.e. a node with only NRT data to transmit) transmits its data only if there is sufficient bandwidth. Otherwise, it simply passes the token to its neighbor. Note that, unlike RTF, there is no need in the SQO strategy to store $RT\_set$ in the token.

To further our understanding of the SQO strategy, consider once again the network of Figure 1. The following sequences of node transmissions are possible in successive cycles:

(1-1) – 2 – (3-3) – (4) – 5
(1-1) – (2) – 3 – 4 – (5-5)

where a sequence item of the form ($i$-$i$) means that node $i$ transmits both RT and NRT data during the same token visit. Also, an item of the form ($i$) means that $i$ is an NRT node that transmits no data but rather just passes the token along to its neighbor. At the beginning of the first sequence, the next-pointer is set to 1, which means that the NRT data transmissions of nodes 1, 2, and 3 can be accommodated during this cycle. The value of the next-pointer will be 4 at the beginning of the second sequence, allowing nodes 4, 5, and 1 to transmit NRT data. Comparing these sequences to the corresponding ones for RTF given above, we see that the token visits a total of five nodes per cycle in SQO, and six nodes per cycle in RTF.

In general, SQO will incur less token-passing overhead than RTF when the total number of per-cycle data transmissions (call this $m$) exceeds the number of nodes on the network (call this $n$). Moreover, the reduction in overhead increase linearly in $m - n$. If $m < n$, SQO could potentially require more token passing than RTF, depending on the relative ordering of RT and NRT nodes. In practice, though, the conditions favorable to SQO can be expected to occur frequently, and correspond to the situation where the network is not saturated by too many applications with RT demands.

# 5 VPL Specification of Rether

In this section, we describe how we specified the Rether protocol in VPL for both the original RTF version, and our alternative SQO version. The interested reader can find the VPL source code listings in Appendices A and B, respectively. The key to the specifications, and subsequent model checking, is our abstraction of time into "time slots." That is, each token cycle is divided into a fixed number of time slots, and each RT and NRT transmission consumes one time slot. Since the properties we wish to verify (admitted RT nodes receive their guaranteed cycle bandwidth and NRT transmissions are not starved) do not depend on the exact value of the token holding times, our time slot abstraction is sufficient for this purpose.

Both versions of the protocol are specified in VPL as a network of N+2 processes: `Node_0`, ..., `Node_N-1`, `Token`, and `Bandwidth`. The `Node_i` processes capture the behavior of the various nodes in the network, `Token` represents the Rether token, and `Bandwidth` is the process nodes interact with to reserve or release cycle bandwidth. A separate process is set aside for the token in order to localize the token-passing logic deployed by the protocol. Modeling the token in this fashion decreases the size of the specification's state space.

Figure 2 depicts the network topology of the specification. In particular, it indicates the names of the channels processes use to communicate with one another and with the outside world. To simplify the figure, only `Node_0` is depicted; the other nodes are connected to `Token`, `Bandwidth`, and the outside world in a similar fashion.

The connections to the outside world are for model-checking purposes. That is, actions over these channels will appear in the various modal mu-calculus formulas to be model checked. Each such action represents a pure synchronization and the significance of these actions is as follows:

| | |
|---|---|
| `start`: | indicates the beginning of a token cycle |
| `cycle`: | indicates the end of a token cycle |
| `reserve_i`: | `Node_i` successfully makes a bandwidth reservation |
| `rt_i`: | `Node_i` performs an RT transmission |
| `nrt_i`: | `Node_i` performs an NRT transmission |

Since the contents of the transmitted data is irrelevant to the correct operation of the protocol, it suffices to use the signals `rt_i` and `nrt_i` in their place.

Process `Bandwidth` is perhaps the simplest of the processes in the specification, and we start

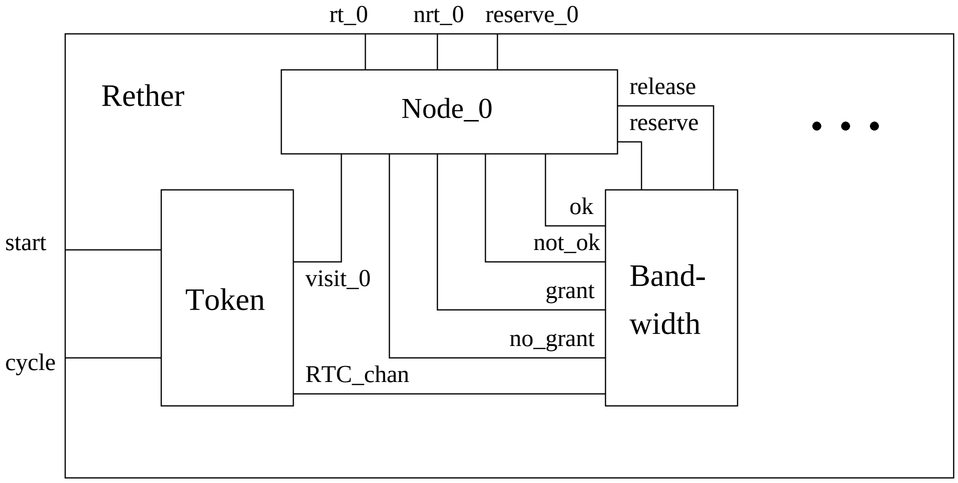


Figure 2: Network topology of the specification.

with it. It essentially behaves as a counting semaphore whose value should not exceed `RTSlots`, where `RTSlots` is the total number of time slots available in a token cycle for RT data transmissions. `RTSlots` is less then the total number of time slots (`Slots`) in a cycle since some slots are reserved for NRT traffic. An NRT node requests a bandwidth reservation via the signal `reserve`. The request is `granted` if a time slot is available; i.e., if `RT_count` < `RTSlots` where `RT_count` is a local variable that keeps track of the number of RT nodes in the system.

When an RT node is ready to release its reserved bandwidth, it signals `Bandwidth` with `release`. It is `ok` for the node to release its bandwidth and become an NRT node as long as the system contains at least one RT node at all times. This is to ensure that the protocol remains in Rether mode: the protocol reverts back to CSMA/CD mode when all RT connections terminate, and we do not model this mode of the protocol. `Bandwidth` also writes `RT_count` to channel `RTC_chan` when `Token` wants to read the number of RT nodes in the system.

Process `Bandwidth` is common to the RTF and SQO specifications. The difference in these versions of the protocol lies in the behavior of the other processes in the network. We consider first the behavior of these processes for the RTF version. The `Token` process determines the order in which the token visits the nodes in the network. It is also responsible for starting and ending the token rotation cycle and updating the information in the token at the beginning of each cycle. To perform these duties, it maintains a local data structure `tok_value` with the following fields: a bit-vector `node[]` used to encode $RT\_set$; an integer `NRT_count` indicating the number of time slots available for NRT data transmissions in the current cycle (`NRT_count` is decremented each time an NRT data transmission occurs); an integer `next`, which points to the next node with NRT data to transmit to be serviced in round robin fashion; and a boolean variable `serving_rt`, which indicates whether RT traffic or NRT traffic is currently being serviced.

`Token` starts a cycle by outputting the signal `start` to the outside world. It then passes the token to each of the nodes in $RT\_set$ in order by their node ids. `Token` passes the token to `Node_i` by writing the token data structure onto channel `visit[i]`; when `Node_i` is finished with the token it sends it back to `Token` over the same channel. After finishing with the RT nodes, `Token` circulates the token among the nodes in the system until all remaining time slots are consumed.

At this point, Token terminates the cycle by outputting the signal cycle and setting NRT_count to Slots − RT_count (recall that RT_count is maintained by the bandwidth process). It then begins a new cycle.

In the RTF version of Rether, process Node_i first receives the token from process Token into the local variable local_tok. It then checks local_tok.serving_rt to determine if it should call procedure RT_action or NRT_action. (In the RTF model, only an RT node can get the token with serving_rt true.) Procedure RT_action simulates an RT data transmission by emitting signal rt_i, and then nondeterministically chooses to release its bandwidth. If its request to release bandwidth is successful, the node is deleted from *RT_set* and local_tok.node[i] is updated accordingly.

Node_i executes procedure NRT_action when it receives the token and local_tok.serving_rt is false. The following conditions are guaranteed to hold at this point: Node_i is being pointed to by local_tok.next and the value of local_tok.NRT_count is greater than zero. Therefore, it simulates an NRT transmission by emitting signal nrt_i, decrementing local_tok.NRT_count, and incrementing local_tok.next. NRT_action then checks local_tok.node[i] to see if it is a RT node; if not, it nondeterministically chooses to issue a request for bandwidth reservation. If granted, local_tok.node[i] is set to true, making Node_i an RT node, and signal reserve_i is emitted to the outside world, indicating that Node_i has successfully made a bandwidth reservation. Upon completion of NRT_action, Node_i outputs local.tok to Token.

Process Token is somewhat simpler in the SQO-based specification of the protocol than in the RTF-based one. It has simpler token-passing logic—the token is passed from one node to another in sequential order by their node ids—and the token has less information in it since it no longer stores *RT_set*. In particular, tok_value has the following four fields: RT_count, NRT_count, next, and NRT_enabled. RT_count and NRT_count are the residual number of time slots for RT and NRT data transmissions, respectively, and next and NRT_enabled are used to implement round robin scheduling of NRT transmissions.

NRT_enabled is set to true by Token if the NRT data transmission of the next node to receive the token can be accommodated during the current cycle. This is the case if the following (somewhat complicated looking) condition is met:

```
NRT_count > 0 & (next = index | (index <next & total_NRT > N - next + index))
```

Here index is the id of the next node to receive the token, total_NRT is the total number of time slots available in the current cycle for NRT transmissions, and N is the total number of nodes in the network. The reader is referred to the discussion of the SQO strategy in Section 4 for the intuition underlying this logic.

Process Node_i in the SQO case has a local variable mode indicating whether or not the node is an RT node. Similar to the RTF case, once Node_i receives the token, it calls RT_action or NRT_action depending on the value of mode. In RT_action, it emits signal rt_i and decrements local_tok.RT_count to simulate an RT data transmission. If local_tok.NRT_enabled is true, it also emits signal nrt_i and decrements local_tok.NRT_count. Moreover, it increments local_tok.next if local_tok.next = i. Like in the RTF case, it can then nondeterministically choose to release its bandwidth reservation; if successful, it switches to NRT_mode.

The SQO version of NRT_action also checks local_tok.NRT_enabled to decide if it can perform an NRT data transmission. If it can, it behaves as does RT_action in this case, after which it may nondeterministically choose to request a bandwidth reservation. If this request is granted, Node_i

| Name | Formula |
|---|---|
| DLF | $\nu X.([-]X \wedge \langle - \rangle tt)$ |
| CC | $\nu X.([-]X \wedge [start]\mu Y.(\langle - \rangle tt \wedge [start]ff \wedge [-cycle]Y))$ |
| RT0 | $\nu X.([-]X \wedge [reserve_0]\nu Y.([-cycle]Y \wedge [rt_0]ff \wedge [cycle]\mu Z.(\langle - \rangle tt \wedge [-rt_0]Z \wedge [cycle]ff)))$ |
| RT1 | $\nu X.([-]X \wedge [reserve_1]\nu Y.([-cycle]Y \wedge [rt_1]ff \wedge [cycle]\mu Z.(\langle - \rangle tt \wedge [-rt_1]Z \wedge [cycle]ff)))$ |
| RT2 | $\nu X.([-]X \wedge [reserve_2]\nu Y.([-cycle]Y \wedge [rt_2]ff \wedge [cycle]\mu Z.(\langle - \rangle tt \wedge [-rt_2]Z \wedge [cycle]ff)))$ |
| RT3 | $\nu X.([-]X \wedge [reserve_3]\nu Y.([-cycle]Y \wedge [rt_3]ff \wedge [cycle]\mu Z.(\langle - \rangle tt \wedge [-rt_3]Z \wedge [cycle]ff)))$ |
| NS0 | $\nu X.([-]X \wedge [start]\nu Y.([-\{nrt_0, cycle\}]Y \wedge [cycle]\mu Z.([-nrt_0]Z \wedge \langle - \rangle tt)))$ |
| NS1 | $\nu X.([-]X \wedge [start]\nu Y.([-\{nrt_1, cycle\}]Y \wedge [cycle]\mu Z.([-nrt_1]Z \wedge \langle - \rangle tt)))$ |
| NS2 | $\nu X.([-]X \wedge [start]\nu Y.([-\{nrt_2, cycle\}]Y \wedge [cycle]\mu Z.([-nrt_2]Z \wedge \langle - \rangle tt)))$ |
| NS3 | $\nu X.([-]X \wedge [start]\nu Y.([-\{nrt_3, cycle\}]Y \wedge [cycle]\mu Z.([-nrt_3]Z \wedge \langle - \rangle tt)))$ |
| RTT | $\nu X.([-]X \wedge [start]\mu Y.([cycle]ff \wedge \langle - \rangle tt \wedge [S_{RT}]Y))$ |
| NRT | $\nu X.([-]X \wedge [start]\mu Y.([cycle]ff \wedge \langle - \rangle tt \wedge [S_{NRT}]Y))$ |
| T3T | $\nu X_0.([-]X_0 \wedge [start]\mu X_1.([-S]X_1 \wedge [cycle]ff \wedge [S]\mu X_2.([-S]X_2 \wedge [cycle]ff \wedge$ $[S]\mu X_3.([-S]X_3 \wedge [cycle]ff \wedge [S]\mu X_4.([-cycle]X_4 \wedge [S]ff)))))$ |
| RTF | $\nu X.([-]X \wedge [S_{NRT}]\mu Y.(\langle - \rangle tt \wedge [-cycle]Y \wedge [S_{RT}]ff))$ |

Table 1: Formulas we model checked. $S$ is the set $\{rt_i, nrt_i | 0 \leq i \leq 3\}$, $S_{RT}$ is the set $\{rt_i | 0 \leq i \leq 3\}$ and $S_{NRT}$ is the set $\{nrt_i | 0 \leq i \leq 3\}$.

switches to `RT_mode`.

# 6 Model Checking the Rether Protocol

In this section, we summarize the results we obtained applying the Concurrency Factory's local model checker to our VPL specifications of the Rether protocol. To ensure that we are model checking a finite-state system, we fixed the following parameters of the specification at compilation time: the number of nodes in the system is chosen to be 4, i.e. the system contains `Node_0` up to `Node_3`; the total number of time slots in each token rotation cycle (`Slots`) is 3; and one slot per cycle is reserved for NRT data transmission. Consequently, during each cycle there are either two RT transmissions plus one NRT transmission, or two NRT transmissions plus one RT transmission. Initially `Node_0` is designated as an RT node and is pointed to by `next`. The remaining nodes are initially NRT nodes.

We model checked both versions of the protocol (RTF and SQO) against 14 formulas of the modal mu-calculus, expressing different properties of the system. The syntax of the modal mu calculus makes use of propositional constants $tt$ and $ff$, logical variables $X$, $Y$ and $Z$, standard logical connectives $\vee$ and $\wedge$, dual modal operators $\langle - \rangle$ and $[-]$, and dual fixed point operators $\mu$ (least fixed point) and $\nu$ (greatest fixed point). A formula of the form $\langle - \rangle \phi$ is true of a state $s$ if it is possible for $s$ to perform an action and reach a state where $\phi$ holds. Dually, $[-]\phi$ is true of $s$ if necessarily after $s$ performs an action it reaches a state where $\phi$ holds. In place of '$-$' (any action) a specific action $a$ or an expression of the form '$-a$' (meaning any action except $a$) can appear inside the modal operators. Fixed point formulas are of the form $\mu X \,.\, \phi(X)$ or $\nu X \,.\, \phi(X)$. Intuitively, $\mu$ formulas capture eventuality properties and $\nu$ formulas capture invariant properties.

The modal mu-calculus formulas we model checked are listed in Table 1. Their significance can

| Formula | Result | States (K) | Transitions (K) | Memory (MB) | Time (min) |
|---|---|---|---|---|---|
| DLF | True | 49.2 | 53.4 | 11.0 | 80.6 |
| CC | True | 93.9 | 100.2 | 16.2 | 125.3 |
| RT0 | True | 40.5 | 42.9 | 7.9 | 42.6 |
| RT1 | True | 42.7 | 45.7 | 8.2 | 45.5 |
| RT2 | True | 43.9 | 47.0 | 8.4 | 47.4 |
| RT3 | True | 43.4 | 45.8 | 8.2 | 45.7 |
| NS0 | True | 100.2 | 107.0 | 17.2 | 129.8 |
| NS1 | True | 100.7 | 107.6 | 17.3 | 129.2 |
| NS2 | True | 100.8 | 107.7 | 17.3 | 128.9 |
| NS3 | True | 101.7 | 108.9 | 17.5 | 131.4 |
| RTT | True | 44.7 | 47.2 | 8.4 | 47.8 |
| NRT | True | 79.9 | 85.6 | 14.0 | 100.2 |
| T3T | True | 82.1 | 86.6 | 13.0 | 82.7 |
| RTF | True | 52.3 | 55.4 | 9.7 | 59.0 |

Table 2: Model cheking results for the RTF model.

be understood as follows:

- Formula 1 (DLF): Deadlock freedom.
- Formula 2 (CC): Cycle completion. This formula states that each token rotation cycle must be completed. It requires that whenever the beginning of a cycle is seen ($[start]$), then eventually the end of the cycle must also be seen by observing the signal *cycle*, and that there cannot be another *start* before the *cycle* is observed.
- Formulas 3 - 6 (RT0 - RT3): Bandwidth guarantee for RT nodes. These formulas state that whenever `Node_i` successfully makes a bandwidth reservation during a token rotation cycle ($[reserve_i]...[cycle]$), then during the immediate following cycle (i.e. before a second *cycle* signal is seen), `Node_i` will transmit RT data ($[rt_i]$). Also no RT data transmission for this node is allowed during the cycle in which it made the reservation (the node is not yet an RT node in that cycle).
- Formulas 7 - 10 (NS0 - NS3): No Starvation for NRT traffic. If during a particular cycle, a `Node_i` does not transmit any NRT data (i.e. no $nrt_i$ between *start* and *cycle*), then eventually an $nrt_i$ signal will be observed, meaning that `Node_i` eventually transmits some NRT data.
- Formula 11 (RTT): At least one RT data transmission in each cycle. Once a *start* is seen, at least one signal from the set $\{rt_i|0 \leq i \leq 3\}$ must be observed before the *cycle* signal.
- Formula 12 (NRT): At least one NRT data transmission in each cycle. Once a *start* is seen, at least one signal from the set $\{nrt_i|0 \leq i \leq 3\}$ must be observed before the *cycle* signal.
- Formula 13 (T3T): A total of three data transmissions in each cycle. Once a *start* is seen, there must be exactly three signals from the set $\{rt_i, nrt_i|0 \leq i \leq 3\}$ before the *cycle* signal.
- Formula 14 (RTF): RT-first property. This formula states that in each cycle, if an NRT data transmission is observed, no more RT data transmissions can be seen before the end of the

| Formula | Result | States (K) | Transitions (K) | Memory (MB) | Time (min) |
|---|---|---|---|---|---|
| DF | True | 12.0 | 13.7 | 3.0 | 19.9 |
| CC | True | 22.6 | 25.1 | 4.2 | 25.4 |
| RT0 | True | 10.9 | 12.0 | 2.3 | 10.7 |
| RT1 | True | 11.3 | 12.7 | 2.4 | 12.4 |
| RT2 | True | 11.5 | 12.9 | 2.4 | 12.1 |
| RT3 | True | 10.4 | 11.5 | 2.2 | 10.3 |
| NS0 | True | 19.4 | 21.4 | 3.6 | 22.8 |
| NS1 | True | 21.8 | 24.1 | 4.0 | 26.2 |
| NS2 | True | 21.9 | 24.3 | 4.0 | 25.7 |
| NS3 | True | 25.0 | 27.8 | 4.5 | 31.0 |
| RA | True | 13.7 | 15.2 | 2.8 | 15.1 |
| NRA | True | 14.7 | 16.5 | 3.0 | 17.3 |
| T3A | True | 19.8 | 21.7 | 3.4 | 18.8 |
| RF | False | 1.8 | 1.8 | 0.9 | 2.6 |

Table 3: Model checking results for the SQO model.

cycle. This implies that all RT data transmissions must precede any NRT data transmissions in each cycle.

Our model checking results for the RTF and SQO models are summarized in Tables 2 and 3, respectively. All data is obtained from a Sun4m Sparc machine with 65MB RAM and 520MB swap space. All the formulas turn out to be true for the RTF model, as intended. For the SQO model, all formulas except RTF are true, again as intended.

Within each model, for the formulas RT0 through RT3, there are some small differences in their state count. This is due to the fact that our system is not strictly symmetric, since `Node_0` starts out as an RT node, while the others start out as NRT nodes. Also, the token visits the nodes in a specific order in each cycle. These reasons also explain the minor variation among the data obtained for formulas NS0 through NS3.

Owing to its simpler token-passing logic, the SQO model has a smaller state space than the RTF model: there is roughly a factor of four difference between the number of the states explored by the Concurrency Factory's local model checker. Consequently, the number of transitions, the memory usage, and the execution time are all down by roughly a factor of four in the SQO model.

## 7 Conclusions

We used the Concurrency Factory's local model checker to verify a number of key properties of the Rether real-time ethernet protocol. In the course of specifying and verifying Rether, we identified an alternative design of the protocol that warranted further study because of potential efficiency gains. Model checking was used to show that this alternative design also possessed the properties of interest.

As for future work, it would be interesting to implement the SQO node servicing strategy in the FreeBSD kernel, so that its performance could be compared to the original protocol. It would also be interesting to observe its relative impact on the quality of service (e.g. jitter) in video applications.

Another research direction is to prove the Rether correct for an arbitrary number of nodes and cycle time slots. This will require induction-based techniques, e.g. [MK95], or more general theorem-proving techniques, e.g. [RSS95].

## References


[AD94] R. Alur and D. Dill. The theory of timed automata. *TCS*, 126(2), 1994.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

[CLSS96] R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The Concurrency Factory: A development environment for concurrent systems. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 398–401, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[CV95] T. Chiueh and C. Venkatramani. The design, implementation and evaluation of a software-based real-time ethernet protocol. In *Proceedings of ACM SIGCOMM '95*, pages 27–37, 1995.

[CW96] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4), December 1996.

[Koz83] D. Kozen. Results on the propositional $\mu$-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[MK95] K. L. McMillan and R. Kurshan. A structural induction theorem for processes. *Information and Computation*, 117:1–11, 1995.

[Mok91] A. K. Mok. Toward mechanization of real-time system design. pages 1–38. Kluwer Academic Publishers, 1991.

[Par96] D. L. Parnas. Why software jewels are rare. *IEEE Computer*, 29(2):57–61, February 1996.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, Berlin, 1982. Springer-Verlag.

[RSS95] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In P. Wolper, editor, *Computer Aided Verification (CAV '95)*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liége, Belgium, July 1995. Springer-Verlag.

[Sok96] O. Sokolsky. *Efficient Graph-Based Algorithms for Model Checking in the Modal Mu-Calculus.* PhD thesis, Department of Computer Science, SUNY at Stony Brook, April 1996.

[TLK96] B. Thomsen, L. Leth, and T.-M. Kuo. A Facile tutorial. In *Proceedings of the Seventh International Conference on Concurrency Theory (CONCUR '96),* Vol. 1119 of *Lecture Notes in Computer Science*, pages 278–298. Springer-Verlag, 1996.

## A VPL Listing of Rether – RTF Version

```
all addition and subtraction operations are modulo N (the number of nodes)
```

```
{========================beginning of vpl code=========================}

value N : 4 {a network of 4 nodes}
value Slots : 3 {total number of time slots}
value RTSlots : 2 {max number of RT time slots}

type count_type : N {NRT_count, RT_count and next are all <= N}

type tok_var : record
  node: array [N] of boolean; {node[i] is true if it is a RT node}
  NRT_count : count_type; {number of NRT slots left in the cycle}
  next : count_type;
  serving_rt: boolean;
end

{======================================================================}

network rether(start: synch, cycle: synch,
                reserve0: synch, reserve1: synch,
                reserve2: synch, reserve3: synch,
                rt0: synch, rt1: synch, rt2: synch, rt3: synch,
                nrt0: synch, nrt1: synch, nrt2: synch, nrt3: synch)

begin

  process bandwidth(reserve: synch, release:synch,
                        grant: synch, no_grant: synch,
                        ok: synch, not_ok: synch,
                        RTC_chan: count_type)
  begin

    var RT_count : count_type

    RT_count :=1;  {initially only node 0 is a RT node, so RT_count = 1}
    while true do
        select
           begin
             reserve?*;
             if (RT_count < RTSlots)  then
                begin  grant!*; RT_count := RT_count+1 end
```

```
            else   no_grant!*
          end
          %
          begin
            release?*;
            if (RT_count > 1)  then
               begin ok!*; RT_count := RT_count-1 end
            else  not_ok!*
          end
          %
          RTC_chan!RT_count
       end   {select}
  end  {while}
end;  {process bandwidth}

{---------------------------------------------------------------------}
process token( visit[]: tok_var,
                RTC_chan: count_type,
                tok_start: synch, tok_cycle: synch)
begin
  var tok_value: tok_var
  var RT_count: count_type
  var index: count_type

  tok_value.NRT_count := Slots-1;
  tok_value.next := 0;
  tok_value.node[0] := true;
  tok_value.node[1] := false;
  tok_value.node[2] := false;
  tok_value.node[3] := false;

  while true do
          tok_start!*;
          tok_value.serving_rt := true;
          index :=0;
          while (index < N) do
            if (tok_value.node[index] = true) then
               begin visit[index]!tok_value; visit[index]?tok_value end;
            index := index + 1
          end;

          {start visiting NRT nodes}
          tok_value.serving_rt := false;
          while ( ~(tok_value.NRT_count = 0) ) do
                visit[tok_value.next]!tok_value;
                visit[tok_value.next]?tok_value
          end;

          tok_cycle!*;
          RTC_chan?RT_count;
          tok_value.NRT_count := Slots-RT_count
  end  {outer while}
end;  {process token}
```

```
{----------------------------------------------------------------------}

  process node(var i: count_type, visit_i: tok_var,
                reserve: synch, release: synch, grant: synch,
                no_grant: synch,  ok: synch, not_ok: synch,
                reserve_i: synch, rt_i: synch, nrt_i: synch)
  begin
    var local_tok: tok_var

    procedure RT_action
    begin
        rt_i!*;
        select
                skip
                %
                begin
                  release!*;
                  select
                      begin
                        ok?*;
                        local_tok.node[i] := false
                      end
                      %
                      not_ok?*
                   end  {inner select}
                end {2nd choice of outer select}
        end {outer select}
    end; {procedure RT_action}

    procedure NRT_action
    begin
              nrt_i!*;
              local_tok.NRT_count := local_tok.NRT_count -1;
              local_tok.next := local_tok.next+1;
              if (local_tok.node[i] = false) then
              begin
                select
                  skip
                  %
                  begin
                    reserve!*;
                    select
                        begin
                          grant?*;
                          local_tok.node[i] := true;
                          reserve_i!*
                        end
                        %
                        no_grant?*
                    end  {inner select}
                  end {2nd choice of outer select}
                end {outer select}
```

```
              end {of if local_tok.node = false}
    end;  {procedure NRT_action}

   {-----------process node body-------------------}
    while true do
        visit_i?local_tok;
        if (local_tok.serving_rt = true)  then
                call RT_action
        else    call NRT_action;
        visit_i!local_tok
    end {while}
   end;  {process node}

 { -------------- body of  network rether --------}

   channel reserve: synch
   channel release: synch
   channel grant: synch
   channel no_grant: synch
   channel ok: synch
   channel not_ok: synch
   channel RTC_chan: count_type
   channel visit: array [N] of tok_var

   bandwidth(reserve, release, grant, no_grant, ok, not_ok, RTC_chan)
   | token(visit[], RTC_chan, start, cycle)
   | node(0, visit[0], reserve, release,
           grant, no_grant, ok, not_ok, reserve0, rt0, nrt0)
   | node(1, visit[1], reserve, release,
           grant, no_grant, ok, not_ok, reserve1, rt1, nrt1)
   | node(2, visit[2], reserve, release,
           grant, no_grant, ok, not_ok, reserve2, rt2, nrt2)
   | node(3, visit[3], reserve, release,
           grant, no_grant, ok, not_ok, reserve3, rt3, nrt3)

 end; {network rether}
```

# B VPL Listing of Rether – SQO Version

all addition and subtraction operations are modulo N (the number of nodes)

```
{=======================beginning of vpl code=====================}

value N : 4 {a network of 4 nodes}
value Slots : 3 {total number of time slots in each token rotation cycle}
value RTSlots : 2 {max number of RT time slots}

value RT_mode : 1
value NRT_mode : 0

type count_type : N {NRT_count, RT_count and next pointer are all <= N}
```

```
type tok_var : record
  NRT_count : count_type; {number of NRT slots remaining in the cycle}
  RT_count : count_type; {number of RT slots remaining in the cycle}
  next : count_type;
  NRT_enabled: boolean;
       {true if during a token visit this node will
        be allowed to transmit NRT data.}
end

{==================================================================}

network rether(start: synch, cycle: synch,
                reserve0: synch, reserve1: synch,
                reserve2: synch, reserve3: synch,
                rt0: synch, rt1: synch, rt2: synch, rt3: synch,
                nrt0: synch, nrt1: synch, nrt2: synch, nrt3: synch)

begin

  process bandwidth(reserve: synch, release:synch,
                        grant: synch, no_grant: synch,
                        ok: synch, not_ok: synch,
                        RTC_chan: count_type )
  begin

    var RT_count : count_type

    RT_count := 1;  {initially node 0 is a RT node, so RT_count = 1}
    while true do
        select
           begin
             reserve?*;
             if (RT_count < RTSlots)  then
                begin  grant!*; RT_count := RT_count+1 end
             else   no_grant!*
           end
           %
           begin
             release?*;
             if (RT_count > 1)  then
                begin ok!*; RT_count := RT_count-1 end
             else  not_ok!*
           end
           %
           RTC_chan!RT_count
        end   {select}
  end  {while}
end;  {process bandwidth}

{------------------------------------------------------------------}

process token(visit[]: tok_var,
```

```
                RTC_chan: count_type,
                tok_start: synch, tok_cycle: synch)
begin
  var tok_value: tok_var
  var RT_count: count_type
  var index: count_type
  var total_NRT: count_type

  tok_value.NRT_count := Slots-1;
  tok_value.RT_count := 1;
  tok_value.next := 0;
  tok_value.NRT_enabled := false;

  while true do
          index := 0;
          tok_start!*;
          while ((tok_value.NRT_count >0) | (tok_value.RT_count >0)) do
            tok_value.NRT_enabled := false;
            if (tok_value.NRT_count >0) then
              begin
                if  (tok_value.next = index)
                   then tok_value.NRT_enabled := true;
                if ( (index < tok_value.next) &
                       (total_NRT > N - tok_value.next + index) )
                then tok_value.NRT_enabled := true
              end;
            visit[index]!tok_value;
            visit[index]?tok_value;
            index := index +1
          end;

          tok_cycle!*;
          RTC_chan?RT_count;
          tok_value.NRT_count := Slots-RT_count;
          tok_value.RT_count := RT_count;
          total_NRT := tok_value.NRT_count
  end  {outer while}
end;  {process token}

{---------------------------------------------------------------------}

  process node(var i: count_type, visit_i: tok_var,
              reserve: synch, release: synch, grant: synch,
              no_grant: synch,  ok: synch, not_ok: synch,
              reserve_i: synch, rt_i: synch, nrt_i: synch)
  begin
    var local_tok: tok_var
    var mode: boolean {1=RT, 0=NRT}

    procedure RT_action
    begin
        if ~(local_tok.RT_count = 0) then
        begin
```

```
       rt_i!*;
       local_tok.RT_count := local_tok.RT_count-1
    end;
    if (local_tok.NRT_enabled = true)
    then begin
            nrt_i!*;
            local_tok.NRT_count := local_tok.NRT_count-1;
            if (local_tok.next = i)  then
               local_tok.next := local_tok.next+1
            {if next != ID, then do not move the next pointer}
    end;
    select
            skip
            %
            begin
              release!*;
              select
                  begin
                    ok?*;
                    mode := NRT_mode
                  end
                  %
                  begin
                    not_ok?*
                  end
               end  {inner select}
            end {2nd choice of outer select}
    end {outer select}
end; {procedure RT_action}

procedure NRT_action
begin
  if (local_tok.NRT_enabled = true)
    then  begin
          nrt_i!*;
          local_tok.NRT_count := local_tok.NRT_count-1;
          if (local_tok.next=i) then local_tok.next:=local_tok.next+1;
          select
            skip
            %
            begin
                reserve!*;
                select
                    begin
                      grant?*;
                      mode := RT_mode;
                      reserve_i!*
                    end
                    %
                    begin
                      no_grant?*
                    end
                end  {inner select}
```

```
                end {2nd choice of outer select}
              end {outer select}
        end {outer if}
    end;  {procedure NRT_action}

  {-----------process node body-------------------}
    if (i = 0) then mode := RT_mode  {0th  node initially RT}
    else mode := NRT_mode;
    while true do
        visit_i?local_tok;
        if (mode = RT_mode)  then
                call RT_action
        else    call NRT_action;
        visit_i!local_tok
    end {while}
  end;  {process node}

{--------------- body of  network rether --------}

  channel reserve: synch
  channel release: synch
  channel grant: synch
  channel no_grant: synch
  channel ok: synch
  channel not_ok: synch
  channel RTC_chan: count_type
  channel visit: array [N] of tok_var

  bandwidth(reserve, release, grant, no_grant, ok, not_ok, RTC_chan)
  | token(visit[], RTC_chan, start, cycle)
  | node(0, visit[0], reserve, release,
          grant, no_grant, ok, not_ok, reserve0, rt0, nrt0)
  | node(1, visit[1], reserve, release,
          grant, no_grant, ok, not_ok, reserve1, rt1, nrt1)
  | node(2, visit[2], reserve, release,
          grant, no_grant, ok, not_ok, reserve2, rt2, nrt2)
  | node(3, visit[3], reserve, release,
          grant, no_grant, ok, not_ok, reserve3, rt3, nrt3)
end;
```