Automated Formal Analysis of Networks: FDR Models of Arbitrary Topologies and Flow-Control Mechanisms *

JN Reed[1], DM Jackson[2], B Deianov[3], and GM Reed[4]

[1] Oxford Brooks University, Oxford, UK
[2] Praxis Critical Systems, Ltd, Bath, UK
[3] Cornell University, Ithaca, NY, USA
[4] Oxford University, Oxford, UK

Abstract. We present new techniques for formally modeling arbitrary network topologies and control-flow schemes, applicable to high-speed networks. A novel induction technique suitable for process algebraic, finite-state machine techniques is described which can be used to verify end-to-end properties of certain arbitrarily configured networks. We also present a formal model of an algorithm for regulating burstiness of network traffic, which incorporates discrete timing constraints. Our models are presented in CSP with automatic verification by FDR.

1 Introduction

The dynamic nature and arbitrary configuration of advanced network environments and network protocols make the problems of their design, control and analysis inherently complex. This is particularly the case where timeliness as well as correctness of service delivery is a priority.

This paper presents elements of formal models of networks which capture various properties of resource-management and control-flow schemes, of special relevance for high-speed, multiservice networks. These models are analysed with FDR [FDR94,RGG95], a software package offered by Formal Systems (Europe) Ltd, which allows automatic checking of many properties of finite state systems and the interactive investigation of processes which fail these checks. It is based on the mathematical theory of Communicating Sequential Processes, developed at Oxford University and subsequently applied successfully in a number of industrial applications.

Previous CSP/FDR network applications primarily centre on protocols. These applications do not specifically address arbitrary network topologies nor rate-based, flow-control mechanisms for network traffic. In this paper we describe a novel induction technique which used in conjunction with hiding and renaming

* This work was supported by the US Office of Naval Research and a research grant from Oxford Brookes University. Technical staff at Formal Systems (Europe) Ltd provided valuable advice on the use of FDR.

can be used to establish properties of arbitrary network configurations. This technique would prove extremely valuable for verifying livelock and deadlock freedom for complex protocols exercised by arbitrary numbers of network nodes. We illustrate its applicability with an example patterned after the Resource reSerVation Protocol (RSVP) [ZDE93,BZB96], a protocol designed to support resource reservation for high-bandwidth multicast transmissions over IP networks.

We also formalise the *leaky bucket* algorithm, a scheme for regulating *burstiness* (variance of delay) of transmitted traffic at a network node. A key component in this model is a ticking clock capturing aspects of a discrete time model.

2 Formal Models of Network Protocols

CSP/FDR belong to the class of formalisms which combine programming languages, and finite state machines. Two similar approaches standardised by ISO for specification and verification/validation of distributed services and protocols are LOTOS [ISOL] [wwwl] and Estelle [ISOE] [ftpe] These techniques are particularly suited for modeling layered protocols, which has come to be a conventional approach for formalising computer networks.

These layered protocols are structured as a fixed number of layers, each with fixed service interfaces. Correctness properties for a given layer typically take the form of an assumption of correct service from the immediate lower level in order to guarantee correct service to the immediate higher level. Properties of the entire "protocol stack" are established by chaining together the service specifications for the fixed number of intermediate layers, ultimately arriving at the service guaranteed by the highest level. The formal layered model naturally reflects the specification and implementation structure of these protocols as adopted by the network and communications community, such as the seven-layer OSI Reference Model developed by the International Standards Organisation. There are numerous examples of formalisations of layered protocols, including Ethernet: CSMA/CD (in non-automated TCSP [Dav91]) (in non-automated algebraic-temporal logic [Jma95]), TCP (in non-automated CSP [GJ94]), DSS1 / ISDN SS7 gateway (in LOTOS [LY93]), ISDN Layer 3 (in LOTOS [NM90]), ISDN Link Access Protocol (in Estelle [GPB91]), ATM signalling (in TLT, a temporal logic/UNITY formalism [BC95]).

An essential feature of these approaches is that system correctness properties are specified in terms of a high-level black box, with a predetermined set of intermediate subcomponents. None of these examples incorporate an unspecified (nor even arbitrary but fixed) set of intermediate nodes. For example, a useful approach for verifying correctness of communication protocols suitable for mechanical support is to prove that an implementation satisfies a variation of what is sometimes known as the *COPY* property, whereby a message is passed by a "black box" process from a specific *sender* to a specific *receiver*. Examples include the alternating bit, sliding window, and multiplexed switches [PS91,FDR94]. In all of these examples the black box connecting the sender to the receiver is refined

by an implementation with a fixed number of subcomponents, each with a fixed interface (set of communication channels).

An *arbitrary* network topology is modelled with action systems [But92] and extended in [Sin97]. The system consists of an (arbitrary but) fixed set of node pairs denoting pairwise channels, together with a complete, noncyclic set of routes. A correctness property analogous to *Copy* is straightforwardly established for the store and forward network. Such deductive-reasoning techniques are not possible for model-checkers such as FDR.

3 CSP and FDR

CSP [Hoa85] models a system as a *process* which interacts with its environment by means of atomic *events*. Communication is synchronous; that is, an event takes place precisely when both the process and environment agree on its occurrence. CSP comprises a process-algebraic programming language (see appendix), together with a related series of semantic models capturing different aspects of behaviour. A powerful notion of refinement intuitively captures the idea that one system implements another. Mechanical support for refinement checking is provided by Formal Systems' FDR refinement checker, which also checks for system properties such as deadlock or livelock.

The simplest semantic model identifies a process as the sequences of events, or *traces* it can perform. We refer to such sequences as *behaviours*. More sophisticated models introduce additional information to behaviours which can be used to determine liveness properties of processes.

We say that a process P is a refinement of process Q, written $Q \sqsubseteq P$, if any possible behaviour of P is also a possible behaviour of Q. Intuitively, suppose S (for "specification") is a process for which all behaviours are in some sense acceptable. If P refines S, then the same acceptability must apply to all behaviours of P. S can represent an idealised model of a system's behaviour, or an abstract property corresponding to a correctness constraint, such as deadlock freedom.

The theory of refinement in CSP allows a wide range of correctness conditions to be encoded as refinement checks between processes. FDR performs a check by invoking a normalisation procedure for the specification process, which represents the specification in a form where the implementation can be checked against it by simple model-checking techniques. When a refinement check fails, FDR provides the means to explore the way the error arose. The system provides the user with a description of the state of the implementation (and its subprocesses) at the point where the error was detected, as well as the sequence of events that lead to the error. The definitive sourcebook for CSP/FDR can now be found in [Ros97].

Unlike most packages of this type, FDR was specifically developed by Formal Systems for industrial applications, in the first instance at Inmos where it is used to develop and verify communications hardware (in the T9000 Transputer and the C104 routing chip). Existing applications include VLSI design, protocol

development and implementation, control, signalling, fault-tolerant systems and security. Although the underlying semantic models for FDR do not specifically address time (in contrast to Timed CSP formalism [RR86,TCSP92,KR93]), work has been carried out modeling discrete time with FDR [Sid93,Ros97]. A class of embedded real-time scheduler implementations [Jac96] is analysed with FDR by extracting numerical information from refinement checks to show not only that a timing requirement is satisfied, but also to determine the margin by which it is met.

4 Properties of Arbitrarily Configured Networks

Certain desirable network properties may not be expressible in terms of predetermined numbers of nodes and interfaces. For example, we might wish to establish deadlock-livelock freedom for an end-to-end protocol which operates with an arbitrary number of intermediate nodes. We would therefore want to express models and properties in a topology dependent manner. To achieve this, we base our specification on single network nodes plus immediate neighbours, and inductively establish the property for arbitrary chains of such nodes. Further discussion of our inductive technique is given in [CR,Cre].

Suppose for a single node we can characterise the interface which a sender or routing node presents to the next node downstream by a property P. If we can demonstrate that under the assumption that all incoming interfaces satisfy P then so do all outgoing onees, we have established an inductive step which allows arbitrary acyclic graphs to be built up, always presenting an interface satisfying P to the nodes downstream. The essential base condition, of course, is that an individual data source meets P. The symmetric case starting with a property of a receiving node and building back towards a source is equally sound. The power of this proof strategy depends on the properties which can be proven of particular nodes, and the ability to structure a collection of nodes inductively with these nodes. Shankar [Shan] uses an induction scheme for PVS model checking for a shared memory algorithm for mutual exclusion, but to our knowledge there has been no published work addressing network protocols.

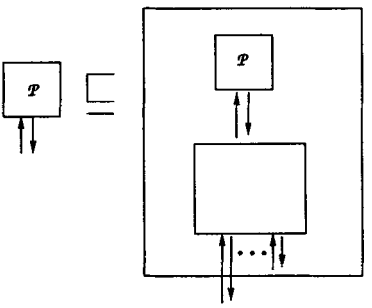

Fig. 1. Simple Induction Scheme

5 RSVP and CSP Models of Reservation Protocols

RSVP is a protocol for multicast resource reservation intended for IP based networks. The protocol addresses those requirements associated with a new generation of applications such as remote video, multimedia conferencing, and virtual reality, which are sensitive to the quality of service provided by the network. These applications depend on certain levels of resource (bandwidth, buffer space, etc.) allocation in order to operate acceptably. The RSVP approach is to create and maintain resource reservations along each link of a previously determined multicast route, with receivers initiating resource requests. This is analogous to a signalling phase prior to packet/cell transmission (such as found in ATM networks).

The technical specification for RSVP as given by its developers appears as a working document of the Internet Engineering Task Force [BZB96]. The protocol assumes a multicast route, which may consist of multiple senders and receivers. RSVP messages carrying reservation requests originate at receivers and are passed upstream towards the senders. Along the way if any node rejects the reservation, a RSVP reject message is sent back to the receiver and the reservation message discarded; otherwise the reservation message is propagated as far as the closest point along the way to the sender where a reservation level greater than or equal to it has been made. Thus reservations become "merged" as they travel upstream; a node forwards upstream only the "maximum" request.

Receivers can request confirmation messages to indicate that the request was (probably) successful. A successful reservation propagates upstream until it reaches a node where there is a (pending) smaller or equal request; the arriving request is then merged with the request in place and a confirmation sent back to the receiver. The receipt of this confirmation is thus a high-probability indication rather than a guarantee of a successful reservation. There is no easy way for a receiver to determine if the reservation is ultimately successful although enhancements involving control packets travelling downstream contain pertinent information to predict the result.

Several interesting aspects emerge from the intuitive description of the RSVP protocol. The protocol is defined for arbitrary routing graphs consisting of several senders and receivers. Confirmations sent by intermediate nodes to receivers are ultimately valid only for the receiver making the largest request; i.e., a requester may receive a confirmation although subsequently the end-to-end reservations fails because of further upstream denial. Global views involving intermediate nodes, (e.g., successful reservations propagate upstream until there are pending smaller or equal requests) present problems for building models consisting of predetermined sets of components. Clearly we are dealing with end-to-end properties inherently defined for arbitrary configurations of intermediate nodes.

We note some interesting design decisions distinctive to RSVP but which are not explicit in [BZB96]. Acknowledgements returned to receivers are only a reflection of a full path back to the specified source for the receiver which has made the (globally) largest request – other receivers may receive acknowledgements when reservations are in place along part of the path. Acknowledgements

from different sources are considered independently: a receiver requesting an acknowledgement which is greater than any existing one will receive an acknowledgement from each data source. Receivers making smaller reservations may receive acknowledgements from intermediate nodes or from sources, depending on the partial ordering among requests.

Extending RSVP to provide more exact information for sender/receiver pairs would involve algorithmic changes, including maintaining more state at intermediate nodes. An interesting technical consideration arises in the context of mechanical verification, where we might identify a hierarchy of approaches: if we maintain state for each reservation, then the system will be potentially infinite, as duplicate reservations must be counted; if we maintain confirmation state for only a single request for each interface, we lose the ability to provide exact acknowledgements. As a compromise, we maintain a record of the confirmed status of each *unique* request, and ignore duplicates.

5.1 CSP Models for Reservation Protocols

We build a general model of a network node, and inductively establish appropriate properties desirable from a receiver's perspective. We illustrate here a very simple model (immediate acknowledgements for previously accepted requests). Similar properties amenable to inductive argument but requiring more complex models include automatic rejection of requests exceeding those previously rejected upstream and filtering requests according to selected sources.

The general communications convention used is that a node has access to two channels, one upstream to toward the source, and another downstream towards the sender. We model resources as small integers and define a single type to distinguish acknowledgements from errors, and define internal channels to relay messages and implement a voting protocol.

```
MAX_RESOUCE = 3
RESOURCE = {0 . . MAX_RESOURCE}
datatype RESULT = accept | reject
datatype MESSAGE = request . RESOURCE
                 | reply . RESOURCE . RESULT

channel upstream, downstream: MESSAGE

datatype INTERNAL = msg . MESSAGE |
     sync . RESOURCE | vote . RESOURCE . RESULT
channel internal : INTERNAL
```

– PROTOCOL NODE: To avoid state explosion, a node is structured as a series of "slices" each maintaining one value of the resource. In practice this would be implemented by fewer processes sharing state. The "down" section of a node manages downstream communications with receivers. Each slice has a parameter v indicating which resource value it is concerned with, and maintains a current reservation state, and an idle flag which indicates if a request is pending.

```
DownSlice(curr, v, idle) =
  idle & downstream . request . v ->
    (if curr >= v
      then DownSlice(curr, v, false)
      else (internal . msg . request ! v -> DownSlice(curr, v, false)
            |~|
            downstream . reply ! v ! reject -> DownSlice(curr, v, true)))
  []
  not idle and curr >= v & downstream . reply ! v ! accept ->
    DownSlice(curr, v, true)
  []
  internal . msg . reply ? vv ! reject ->
    (if vv == v and not idle
      then downstream . reply ! v ! reject -> DownSlice(curr, v, true)
      else DownSlice(curr, v, idle))
  []
  internal . msg . reply ? vv ! accept -> DownSlice(max(curr,vv), v, idle)
```

– Requests which cannot be trivially satisfied are forwarded on the internal message channel. The whole receiver interface is a combination of such slices:

```
Down = || v: RESOURCE @ [{|internal . msg . reply ,
                           internal. msg.request.v,
                           downstream.request.v,
                           downstream.reply.v|}] DownSlice(0, v, true)
```

– The interface to a sender is similarly structured: In the RSVP wild-card model, it simply relays requests upstream and then votes on the returned status value. If a reservation succeeds, it will allow both synchronisation's to happen without intervening events. Otherwise, it will insist that the rejection is recorded. Different upstream interfaces all synchronise on `sync` but interleave on `vote`.

```
UpSlice(v) =
  internal . msg . request . v -> upstream . request ! v ->
    upstream . reply . v ? result -> internal . sync ! v  ->
    if result == accept
      then internal . sync ! v  -> UpSlice(v)
      else internal . vote ! v ! reject ->
             internal . sync ! v  -> UpSlice(v)
```

– Instantiation:

```
Up = || v: RESOURCE @ [{|internal . msg . request.v,
                         internal.vote.v,
                         internal. sync.v,
                         upstream.request.v,
                         upstream.reply.v|}] UpSlice(v)
```

– The final part of the node is a central co-ordinator which is responsible for monitoring the results of propagated requests and passing the resulting vote down to the receiver components. All upstream interfaces should register their vote on `sync`, with only rejecting also choosing to vote `reject`.

– This rather complex voting structure accommodates multiple upstream interfaces; a downstream request is ultimately accepted only if all upstream interfaces accept.

```
CoordinatorSlice(v) =
let
  Accept = internal . sync . v ->
             internal . msg . reply ! v ! accept ->
             CoordinatorSlice(v)
           []
           internal . vote . v . reject -> Reject
  Reject = internal . sync . v ->
             internal . msg . reply ! v ! reject ->
             CoordinatorSlice(v)
           []
           internal . vote . v . reject -> Reject
  within internal . sync . v -> Accept
```

– In this case there is no communication between co-ordinator slices.

```
Coordinator = ||| v : RESOURCE @ CoordinatorSlice(v)
```

– The simplest possible node has a single upstream and single downstream interface, and a coordinator (see Fig. 2):

```
SimpleNode =  ((Up
                 [|{|internal.vote,internal.sync|}|]
                 Coordinator \ {|internal.vote,internal.sync|})
                 [|{|internal|}|]
                 Down \ {|internal|})
```

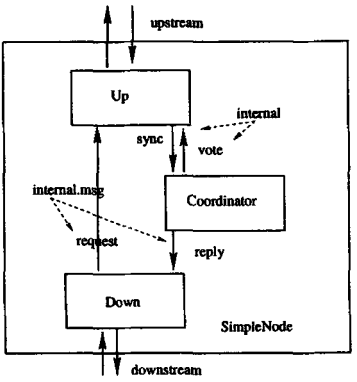

Fig. 2. Simple Node

– The following specification for a receiver access point has two principle properties: acknowledgements are issued only for previously observed request values, and requests for values which have not yet been seen are always accepted.

```
RA0 =
let
SPEC({}) = downstream.request?v -> SPEC({v})
SPEC(seen) =
            (downstream.request?v -> SPEC(union(seen,{v})))
             |~|
     downstream.request?v:diff(RESOURCE,seen) ->
SPEC(union(seen,{v}))
     |~|
(|~| h : seen,  v: RESULT @
                      downstream . reply ! h ! v -> SPEC(seen))
within SPEC({})
```

– The simplest specification that a sender must satisfy is that it must present an interface satisfies this condition; we achieve this by using RA0. We can then connect such an abstract source to a simple protocol node as follows:

```
SimpleSystem = (RA0[[ downstream <- upstream ]]
[| {|upstream|} |]
SimpleNode) \ {| upstream |}
```

– This should preserve the RA0 property (see Figure 3):

```
assert RA0 [FD= SimpleSystem
```

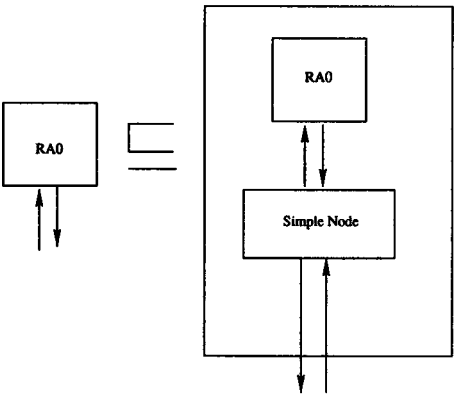

Fig. 3. Receiver's perspective

– To add a second receiver interface, we introduce a second downstream channel, and add an appropriately renamed interface process.

```
channel down' : MESSAGE
channel localreq : RESOURCE
```

– A new local request channel is introduced to allow requests from the two downstream channels to be interleaved. The `intrnal.msg.request` channel seen by the first down node (`down'`) is renamed to `localreq` before it becomes hidden (and thus unavailable to the second down node). Then this `localreq` is renamed back to `internal.msg.request` in order to open it up to the second down node. This clever technique enables the synchronised parallel operator (`[| |]`) with both synchronous (`internal.msg.reply`) and asynchronous channels (`internal.msg.request`).

```
DualNode = ((( (normal(((Up
          [|{|internal.vote,internal.sync|}|]
          Coordinator \ {|internal.vote,internal.sync|})
                    [[ internal.msg.request  <- localreq,
 internal.msg.request <- internal.msg.request ]]
          [|{|internal|}|]
          Down[[downstream <- down']]) \ {|internal.msg.request|})
      [[ localreq <- internal.msg.request ]])
   [|{|internal.msg.reply|}|]
          Down)) \ {|internal|})
```

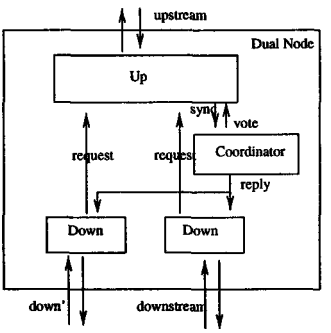

Fig. 4. Node with two downstream channels

– Again we can add a pseudo-sender (see Figures 4 and 5). The system should present the same interface on both `downstream` and `down'`: :

```
DualSystem = (RAO[[ downstream <- upstream ]]
[| {|upstream|} |]
DualNode) \ {| upstream |}

assert RAO  [F= DualSystem \ {|down'|}
assert RAO[[downstream <- down']] [F= DualSystem \ {|downstream|}
```

In the refinement above we employ the CSP hiding operator \, but a stronger criteria is obtained if we use lazy evaluation [Ros97]. The assertions illustrated in Fig 5 inductively establishing properties for, in this instance, any network made up of nodes with downstream branching degree of two. Thus we can prove correctness for finite but unbounded topologies.

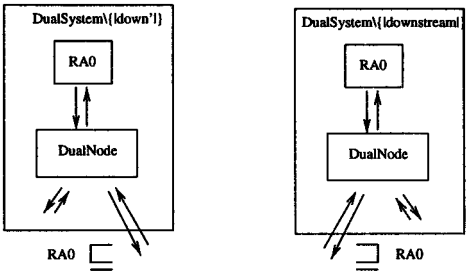

Fig. 5. Each downstream channel satisfies RA0

6 Leaky Bucket

Traffic congestion caused by burstiness of data transmission presents problems for today's multimedia networks, particularly for video and audio which do not tolerate variable rates of flow well. One approach to congestion management in ATM networks is called traffic shaping, which attempts to regulate the average rate and burstiness of network traffic. If senders agree to certain transmission patterns, then the network can agree to provide a certain quality of service. Monitoring a traffic flow for conformance of transmission pattern is called traffic policing.

The leaky bucket algorithm [Tur86] attempts to regulate traffic burstiness at a network node. It does not preclude buffer overflow nor guarantee an upper bound on packet delay; but with proper choice of bucket parameters, overflow and delay can be reduced.

Our model utilises an idiom introduced in [Ros97] which was applied very effectively to the verification of timing requirements of a real-time embedded scheduler [Jac96]. Passing of time is marked by a clock process synchronising with other system components. However because of CSP's particular treatment of internal events, care must be taken to prevent the system from diverging (livelocking).

Overview of Algorithm

This description is adapted from Tanenbaum [Tan96]. Imagine a bucket with capacity L which holds packets and leaks them at a given rate I when not empty and rate 0 when empty. Arriving packets join the bucket if the bucket is not full and are marked *conforming*. If the bucket is full, arriving packets do not join and are marked as *nonconforming*. This mechanism can be used to smooth out burstiness to a more even flow.

For example, assume that we want to achieve a transmission rate of 2MB/sec, and assume that data is coming in at a rate of 25MB/sec for the first 40 msec of a 1 second period. If the bucket has capacity 1MB, and leaks packets at a rate of 2 MB/sec, then the described burst of 1MB/sec is conforming. However an additional 25MB/sec over the next 10 msec would be marked as nonconforming.

The first example below models a policing function using a leaky buffer. The second example contains an additional space controller which attempts to smooth burstiness by buffering incoming cells.

We outline a model indicated in Fig 7 of the Leaky-Bucket algorithm, with and without a Space Controller. We omit the code for most of the components, but explicitly give that for the crucial modules Space Controller and the Timer.

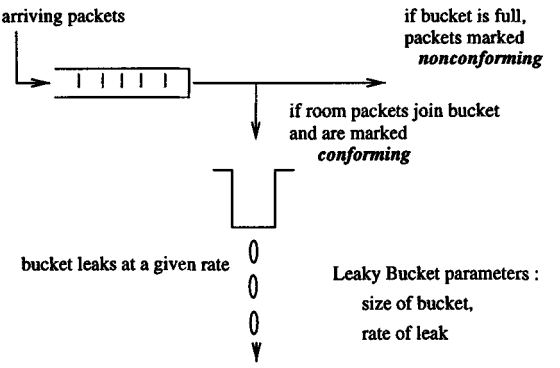

Fig. 6. Leaky Bucket

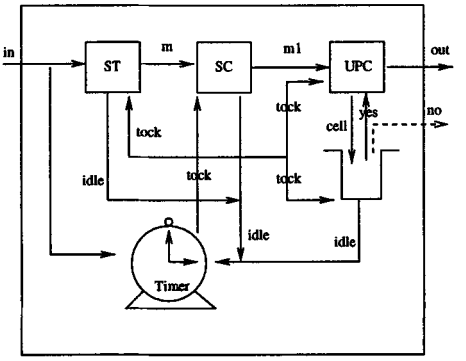

Fig. 7. Leaky Bucket with Space Controller

```
DATA = {0,1}
channel in,out,m,m1 : DATA
channel cell,yes,no,error,tock,idle
```

A tock event indicates that one unit of time has passed. A process performing the idle event will not change state until some non-tock event takes place. BUFF(B) is a B place buffer.

LB(I,L) models a leaky bucket with parameters I (agreed rate) and L (tolerance/bucket size); it remembers the current value of the bucket. With each tock, when the bucket is not empty, leak one otherwise indicate that the bucket is idle. A cell event is non-conforming (output no) if the bucket is too full and conforming (yes) otherwise; put I more in the bucket. The significance of a no event is that a non-conforming cell will cause the system to fail the specification. The bucket is initially empty.

The UPC process simply relays out those cells which are nonconforming. It must always listen to tock events.

A space controller with parameters `I` (release rate) and `B` (buffer size); `s` is the current contents of the buffer; `t` is the time since the last cell was released. If enough time has passed release the next cell – otherwise record the passage of time. The space

controller is idle only if its buffer has been empty for I units of time. If there is space in the buffer, store cells when they arrive –if not, release a cell even if it is too early, to prevent the buffer from overflowing.

```
SC(I,B) = let
 SC1(s,t) = if t == 0 and not null(s)
              then m1!head(s) -> SC1(tail(s),I)
              else (tock -> if t > 0
                            then SC1(s,t-1)
                            else if null(s)
                                   then idle -> SC1(<>,0)
                                   else SC1(s,0))
                   []
                   (m?c -> if #s < B
                             then SC1(s^<c>,t)
                             else m1!head(s) -> SC1(tail(s)^<c>,I))
within SC1(<>,0)
```

A transmitter process ST(I,B) inputs cells from the environment and puts them into the system, but no more than B every I*B units of time (overall rate 1 every I tocks); it maintains a value bits as a binary record of the last I*B+B events: a bit is 1 if the event was a cell put into the system, 0 if it was a tock; count is the number of 1s in bits. If no cells have been transmitted recently, the transmitter is idle – otherwise change the state with the passage of time. If no more than B cells have been transmitted in the last B*I tocks transmit the next cell and record that.

The purpose of TIMER is to stop time when the system is idle - this stops the system from diverging as it can't perform the tock event an infinite number of times without engaging in external communication. N is the number of components in the system; M is the number of components that are still not idle, when 0 time stops. An input from the timer wakes up the clock. Only allow the tock event if the system is not idle.

```
TIMER(N) = let
TIMER1(M) = M != 0 & (tock -> TIMER1(N)
                         []
                         idle -> TIMER1(M-1))
              []
              in?c -> TIMER1(N)
within TIMER1(N)
```

A complete system (without space controller) with parameters I1 (transmission rate), B1 (burstiness), I2 (agreed rate), L2 (bucket size/tolerance).

```
SYS(I1,B1,I2,L2) = (ST(I1,B1) [|{|m,tock|}|] LB(I2,L2) \ {|m|})
                   [|{|in,tock,idle|}|]
                   TIMER(2) \ {|tock,idle|}
```

A complete system (with space controller). Additional parameter: B2 (space controller buffer size).

```
SYSC(I1,B1,I2,L2,B2) = ((ST(I1,B1) [|{|m,tock|}|]
                          SC(I2,B2) \ {|m|}) [|{|m1,tock|}|]
                        LB(I2,L2) [[m <- m1]] \ {|m1|})
                       [|{|in,tock,idle|}|]
                       TIMER(3) \ {|tock,idle|}


assert BUFF(1) [FD= SYS(5,1,5,3)          -- Checks !
```

– Transmission is at agreed rate with equal delay between cells; all cells are conforming.

```
assert BUFF(2) [FD= SYS(5,3,5,8) -- Fails !
```

– Transmission at the agreed rate, but the traffic is bursty; the bucket is too small and the third cell is non-conforming.

```
assert BUFF(3) [FD= SYSC(5,3,5,8,2) -- Checks !
```

– Same as above, but with a space controller; all cells are now conforming.

```
assert BUFF(6) [FD= SYSC(4,1,5,8,5) -- Fails !
```

– Transmission is too fast and eventually both the buffer of the space controller and the leaky bucket overflow.

7 Conclusions

We described an induction technique for proving properties of arbitrary configurations of nodes. This technique was illustrated with RSVP, a resource reservation protocol which is intended for and most naturally described using arbitrary network topologies. Whilst unspecified topologies are straightforwardly handled by state-based formal methods such as action systems or Z, corresponding methods for automated model-checking approaches such as FDR have not been identified. Our contribution is to identify induction schemes which require no extension to the underlying theory, but which have not been used in previous applications and rely on various "coding tricks" which have not been illustrated in previously published works. Such techniques would prove especially valuable for proving deadlock/livelock freedom for complex protocols among arbitrary numbers of nodes, provided that we can model the protocol using an inductive structure.

We have also presented an FDR model incorporating discrete time which is applied to the leaky bucket algorithm for traffic policing. FDR is not immediately associated with applications dealing with time, but the treatment of discrete time proves very effective in this case.

References

[BC95] D Barnard and Simon Crosby, The Specification and Verification of an Experimental ATM Signalling Protocol, *Proc. IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, Dembrinski and Sredniawa, eds,Warsaw, Poland, June 1995, Chapman Hall.

[But92] R Butler. A CSP Approach to Action Sysems, DPhil Thesis, University of Oxford, 1992.

[BZB96] R Braden, L Zhang, S. Berson, S. Herzog and S. Jamin. Resource reSerVation Protocol (RSVP) – Version 1, Functional Specification. Internet Draft, Internet Engineering Task Force. 1996.

[Cre] S Creese, An inductive technique for modelling arbitrarily configured networks, MSc Thesis, University of Oxford, 1997.

[CR] S Creese and J Reed, Inductive Properties and Automatic Proof for Computer Networks, (to appear).

[Dav91] J Davies, Specification and Proof in Real-time Systems, D.Phil Thesis, Univ. of Oxford, 1991.

[FDR94] Formal Systems (Europe) Ltd. Failures Divergence Refinement. *User Manual and Tutorial*, version 1.4 1994.

[ftpe] Estelle Specifications, ftp://louie.udel.edu/pub/grope/estelle-specs

[GJ94] JD Guttman and DM Johnson, Three Applications of Formal Methods at MITRE, *Formal Methods Europe*, LNCS 873, M Naftolin, T Denfir, eds, Barcelona 1994.

[GPB91] R Groz, M Phalippou, M Brossard, Specification of the ISDN Linc Access Protocol for D-channel (LAPD), CCITT Recommendation Q.921, ftp://louie.udel.edu/pub/grope/estelle-specs/lapd.e

[Hoa85] CAR Hoare. *Communicating Sequential Processes.* Prentice-Hall 1985.

[ISOE] ISO Recommendation 9074, The Extended State Transition Language (Estelle), 1989.

[ISOL] ISO: Information Processing System - Open System Interconnection - LOTOS - A Formal Description Technique based on Temporal Ordering of Observational Behavior, IS8807, 1988.

[Jac96] DM Jackson. Experiences in Embedded Scheduling. *Formal Methods Europe*, Oxford, 1996.

[Jma95] M Jmail, An Algebraic-temporal Specification of a CSMA/CD Protocol, *Proc. IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, Dembrinski and Sredniawa, eds,Warsaw, Poland, June 1995, Chapman Hall.

[KR93] A Kay and JN Reed. A Rely and Guarantee Method for TCSP, A Specification and Design of a Telephone Exchange. *IEEE Trans. Soft. Eng.*. 19,6 June 1993, pp 625-629.

[LY93] G Leon, JC Yelmo, C Sanchez, FJ Carrasco and JJ Gil, An Industrial Experience on LOTOS-based Prototyping for Switching Systems Design, *Formal Methods Europe*, LNCS 670, JCP Woodcock and DG Larsen, eds., Odense Denmark, 1993.

[NM90] J Navarro and P s Martin, Experience in the Development of an ISDN Layer 3 Service in LOTOS, *Proc. Formal Description Techniques III*, J Quemada, JA Manas, E Vazquez, eds, North-Holland, 1990.

[PS91] K Paliwoda and JW Sanders. An Incremental Specification of the Sliding-window Protocol. *Distributed Computing.* May 1991, pp 83-94.

[RGG95] AW Roscoe, PHB Gardiner, MH Goldsmith, JR Hulance, DM Jackson, JB Scattergood. H ierarchical compression for model-checking CSP or How to check 10^{20} dining philosphers for deadlock, Springer LNCS 1019.

[Ros97] AW Roscoe. *The CSP Handbook*, Prentice-Hall International, 1997.

[RR86] GM Reed and AW Roscoe, A timed model for communicating sequential processes, Proceedings of ICALP'86, Springer LNCS 226 (1986), 314-323; *Theoretical Computer Science* 58, 249-261.

[Shan] N Shankar, Machine-Assisted Verification Using Automated Theorm Proving and Model Checking, *Math. Prog. Meothodology*, ed. M Broy.

[Sid93] K Sidle, Pi Bus, *Formal Methods Europe*, Barcelona, 1993.

[Sin97] J Sinclair, Action Systems, Determinism, and the Development of Secure Systems, PHd Thesis, Open University, 1997.

[Tan96] AS Tanenbaum. *Computer Networks.* 3rd edition. Prentice-Hall 1996.

[TCSP92] J Davies, DM Jackson, GM Reed, JN Reed, AW Roscoe, and SA Schneider, Timed CSP: Theory and practice. *Proceedings of REX Workshop, Nijmegen*, LNCS 600, Springer-Verlag, 1992.

[Tur86] JS Turner. New Directions in Communications (or Which Way to the Information Age). *IEEE Commun. Magazine.* vol 24, pp 8 -15, Oct 1986.

[wwwl] LOTOS Bibliography, http://www.cs.stir.ac.uk/ kjt/research/well/bib.html

[ZDE93] L Zhang, S Deering, D Estrin, S Shenker and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, September 1993.

Appendix A. The CSP Language

The CSP language is a means of describing components of systems, *processes* whose external actions are the communication or refusal of instantaneous atomic *events*. All the participants in an event must agree on its performance. The CSP processes that we use are constructed from the following (overview from [Jac96]):

`STOP` is the simplest CSP process; it never engages in any action, and never terminates.

`SKIP` similarly never performs any action, but instead terminates successfully, passing control to the next process in sequence (see ; below).

`a -> P` is the most basic program constructor. It waits to perform the event `a` and after this has occurred subsequently behaves as process `P`. The same notation is used for outputs (`c!v -> P`) and inputs (`c?x -> P(x)`) of values along named channels.

`P |~| Q` represents *nondeterministic* or internal choice. It may behave as `P` or `Q` arbitrarily.

`P [] Q` represents external or *deterministic* choice. It will offer the initial actions of both `P` and `Q` to its encironment at first; its subsequent behaviousr is like *P* if the initial action chosen was possible only for `P`, and like `Q` if the action selected `Q`. If both `P` and `Q` have common initial actions, its subsequent behaviour is nondeterministic (like `|~|`). A deterministic choice between `STOP` and another process, `STOP [] P` is identical to `P`.

`P |[ A ]| Q` represents parallel (concurrent) composition. `P` and `Q` evolve separately, except that events in `A` occur only when `P` and `Q` agree (i.e. *synchronise*) to perform them.

`P ||| Q` represents the interleaved parallel composition. `P` and `Q` evolve separately, and do not synchronize on their events.

`P ; Q` is a sequential, rather than parallel, composition. It behaves as `P` until and unless `P` terminates successfully: its subsequent behaviour is that of `Q`.

`P \ A` is the CSP abstraction or hiding operator. This process behaves as `P` except that events in set `A` are hidden from the environment and are solely determined by `P`; the environment can neither observe nor influence them.

`P [[ a <- b ]]` represents the process `P` with `a` renamed to `b`.

There are also straighforward generalisations of the choice operators over non-empty sets, written `|~| x:X @ P(x)` and `[] x:X @ P(x)`.