

Verifying End-to-End Protocols Using Induction with CSP/FDR

S. J. Creese¹ and Joy Reed^{2*}

¹ Oxford University, Oxford UK

² Oxford Brookes University, Oxford UK

Sadie.Creese@comlab.ox.ac.uk, jnreed@brookes.ac.uk

Abstract. We investigate a technique, suitable for process algebraic, finite-state machine (model-checking) automated tools, for formally modelling arbitrary network topologies. We model aspects of a protocol for multiservice networks, and demonstrate how the technique can be used to verify end-to-end properties of protocols designed for arbitrary numbers of intermediate nodes. Our models are presented in a version of CSP allowing automatic verification with the FDR software tool. They encompass both inductive and non-inductive behaviours.

1 Introduction

Formal specification and verification of network protocols have focused in the main on link-to-link properties involving the interaction of a fixed number of components, e.g., providing reliable transmission between a sender and receiver using a fixed number of lower layer subcomponents communicating over an unreliable channel. However these techniques are not suitable for verifying end-to-end properties of modern multi-service networks, such as the Internet, which use complex protocols operating with arbitrary numbers of interacting components.

We illustrate a technique encompassing induction for proving correctness properties of network protocols operating with an arbitrary number of components. The technique is suitable for process algebraic, finite-state machine (model-checking) automated tools. Model-checking based techniques do not directly handle unbounded state problems; however this induction technique enables their use for certain classes of such applications, notably end-to-end protocols with arbitrary numbers of intermediate nodes. We illustrate the method using FDR [8, 23], 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. The tool is based on the mathematical theory of Communicating Sequential Processes (CSP), developed at Oxford University and subsequently applied successfully in a number of industrial applications.

* This work was supported by in part by the US Office of Naval Research. Technical staff at Formal Systems (Europe) Ltd provided valuable advice on the use of FDR.

Previous CSP/FDR applications do not address properties of arbitrary numbers of interacting components. The induction technique described here extends the basic method introduced in [22] by employing lazy abstraction and dealing with non-inductive behaviours. It provides a powerful way to establish complex properties of arbitrary network configurations. This technique should prove especially valuable for verifying livelock and deadlock freedom for complex end-to-end network protocols. We illustrate its applicability with an example patterned after the Resource reSerVation Protocol (RSVP) [3, 35], a protocol designed to support resource reservation for high-bandwidth multicasts over IP networks.

We present an overview of formal models of network protocols, CSP and FDR. We describe a reservation protocol, the induction method, and its application to the protocol. We conclude with remarks on applicability. Our FDR source scripts are available upon request.

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 of distributed services and protocols are LOTOS [14] (web bibliography [34]) and Estelle [9, 13].

Finite-state techniques are particularly suited for modelling *layered* protocols. 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” can be established by chaining together the service specifications for the fixed number of intermediate layers, ultimately arriving at the service guaranteed by the highest level. In a very natural way, the formal layered model reflects the specification structure of these protocols as adopted by the network and communications community, such as the seven-layer ISO OSI (Open Systems Interconnect) Reference Model. There are numerous examples of formalisations of layered protocols, including Ethernet - CSMA/CD (in non-automated TCSP [7]) (in non-automated algebraic-temporal logic [16]), TCP (in non-automated CSP [10]), DSS1 / ISDN SS7 gateway (in LOTOS [19]), ISDN Layer 3 (in LOTOS [20]), ISDN Link Access Protocol (in Estelle [11]), ATM signalling (in TLT, a temporal logic/UNITY formalism [1]). All of these examples deal with link rather than end system properties.

By way of contrast, we note that an unbounded network topology is modelled with action systems [2] and extended in [29]. Although such deductive-reasoning techniques are not possible for finite-state model checkers such as FDR, the advantage of finite-state methods is that they are fully automatable, indicating when properties are *not* satisfied as well as when they are.

3 CSP, CSP_M and FDR

CSP [12] 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, deadlock and livelock checking is provided by Formal Systems' FDR model checker.

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 S , written $S \sqsubseteq P$, if any possible behaviour of P is also a possible behaviour of S . 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 . The refinement relation is transitive. S can represent an idealised model of a system's behaviour, or an abstract property 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 user with an illustrative counter example. The definitive source book for CSP/FDR is found in [25].

CSP_M [8, 25, 26] combines the CSP process algebra with an expression language inspired by languages like Miranda-Orwell and Haskell-Gofer and modified to support the idioms of CSP. Hereafter we shall simply refer to CSP.

Unlike most packages of this type, FDR was specifically developed by Formal Systems for industrial applications, initially used to develop and verify communications hardware (in the Inmos 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 [17, 24, 31]), work has been carried out modelling discrete time with FDR [25, 28], including a class of embedded real-time scheduler implementations [15] and a traffic congestion algorithm [22].

4 A Reservation Protocol

We illustrate the induction technique on a protocol patterned after the RSVP reservation protocol intended for IP based networks. This protocol addresses those requirements associated with a new generation of applications, such as

remote video and multimedia conferencing, 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. One strategy for dealing with large demands for bandwidth is to “multicast” from sources to receivers with transmissions along a number of intermediate links shared by “downstream” nodes. The RSVP approach is to create and maintain resource reservations along each link of a previously determined multicast route.

A multicast route consists of multiple sources and receivers, and arbitrary numbers of intermediate nodes forming a path between sources and receivers. Messages carrying reservation requests originate at receivers and are passed upstream towards the sources. Along the way if any node rejects the reservation, a 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 source 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. A binary multicast tree is represented in Figure 3.

5 The Significance of Lazy Abstraction

Our technique reduces unbounded state space to a small finite one by hiding (and thus abstracting away) the unbounded components of our multicast network. We take the perspective of a downstream node in specifying the behaviour it sees on its upstream link.

Abstracting away the network along downstream channels, by hiding the channels using the CSP standard hiding operator \backslash , would have significant consequences for reasoning about behaviour on the visible downstream channel. Consider the process H below; if the event a were to be *eagerly* abstracted, i.e. hidden, then the process H would be in a state where it would be willing to do a b , as if the a had occurred.

$$\begin{aligned} H &= a \rightarrow b \rightarrow H \\ H \backslash a &= b \rightarrow H \end{aligned}$$

By similarly hiding an interface channel, we are not allowing the environment to be silent. Thus any proofs involving such abstraction would carry the “hidden” assumption that the environment is live on all offered events, and any properties thereby established could be significantly devalued.

We wish to establish that the behaviour seen on one downstream channel does not require liveness on other downstream channels. In order to claim that the behaviour over hidden interfaces does not compromise a successful refinement check, we simply show that the refinement holds even in the case of no communications on the hidden channels. By “gluing” *CHAOS* onto a *downstream* channel, we are effectively abstracting the rest of the downstream network, no matter what its behaviour. The *lazy* abstraction of a from H , $\mathcal{L}_a(H)$, is defined

below as the parallel composition of H with $CHAOS$ before hiding a , where $CHAOS$ is the process which may or may not do anything. In the example H above, $\mathcal{L}_a(H)$ is $b \rightarrow H \sqcap STOP$, which may fail to offer b (since $CHAOS$ can nondeterministically refuse a , thereby stopping b), in contrast to $H \setminus a$, which always offers b .

$$\begin{aligned}\mathcal{L}_a(H) &= (H \parallel a \parallel CHAOS(a)) \setminus a \\ CHAOS(A) &= \prod_{a:A} a \rightarrow CHAOS(A) \sqcap STOP\end{aligned}$$

6 The Method

We model a protocol exhibiting the traffic-reducing aspects of RSVP: the ability of an intermediate node to automatically respond to requests for resources for which previous requests had already established the response. If a reservation is already in place (for a particular source), then any request for the same amount or less can be automatically accepted without any messages forwarded upstream. Similarly, if a request for an amount of resource from a particular source has already been rejected, then any request for the same amount or more can be automatically rejected without forwarding upstream.

FDR is used to perform a *structural induction* to establish end-to-end properties of the protocol through a series of refinement checks. These checks correspond to the base case and the inductive step of proof by induction. In this example, the proposition to be proven for our protocol is:

An arbitrary number of intermediate traffic-reducing nodes may be placed transparently between sources and end receivers, i.e., a receiver sees the same behaviour from an upstream node connected by an arbitrary number of links to an end source, as it would see from an end source itself.

Let P represent this property. The induction proof obligations take the form:

$$\begin{aligned}P &\sqsubseteq SOURCE && (i) \\ P &\sqsubseteq \mathcal{L}_1^2(P [downstream \leftrightarrow upstream] NODE) && (ii) \\ P &\sqsubseteq \mathcal{L}_2^1(P [downstream \leftrightarrow upstream] NODE) && (iii)\end{aligned}$$

The refinement operator \sqsubseteq refers to refinement in the CSP *failures model*. $\mathcal{L}_i^j(Q)$ is the *lazy abstraction* [25] of the channel $downstream_i$ from the process Q , with $downstream_j$ renamed to $downstream$. By this mechanism, we abstract away unbounded parts of the network further along the $downstream_i$ channel. In the parallel composition $P[downstream \leftrightarrow upstream]NODE$ of (ii) and (iii), the communications between the P and $NODE$ are hidden and happen as soon as they are available according to CSP semantics for the hiding operator. We can safely eagerly (rather than lazily) abstract these communications by hiding the next-hop upstream channel, since this has no effect on our reasoning about communications along a given downstream channel.

Refinement (i) establishes the base case for the induction. Generally this case is simply an end source, although an example in Section 7 requires an intermediate node directly linked to a source.

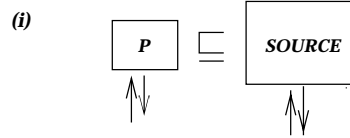


Fig. 1. Base Case: (i) A SOURCE must satisfy property P.

Refinements (ii) and (iii) establish the inductive step. Assuming that P is provided on its *upstream* channel, $NODE$ provides P to each of the *downstream* channels. The linked parallel operator used in the right hand sides of the induction proof obligations given above indicates that $NODE$ communicates on its *upstream* channel in a synchronised fashion with (a node satisfying) P on P 's *downstream* channel, with the linked communications hidden. Thus we are able to represent the perspective of the one downstream node, by hiding the other downstream channel and next-hop upstream communications. The refinement checks performed by FDR establish that no matter what communications take place over the *downstream1* channel the property P still holds on channel *downstream2*. Similarly, P holds on *downstream1* regardless of communications on *downstream2*.

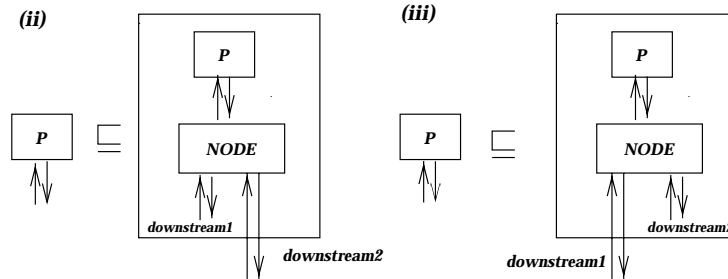


Fig. 2. Inductive Case: The property P connected on its downstream to $NODE$ on $NODE$'s upstream, with all communications hidden apart from the *downstream2* interface in (ii), and the *downstream1* interface in (iii).

By transitivity of refinement, relations (i) - (iii) allow us to claim that property P is provided along any of the downstream links for an arbitrary binary multicast tree, such as specified below and represented in Figure 3. These relations can be expressed as an **assertion** in FDR, which checks their validity. The techniques can be similarly applied to achieve arbitrary branching degree for sources and receivers.

```

((( SOURCE
  [downstream <-> upstream]
  NODE[[downstream1 <- A, downstream2 <- B]] )
  [A <-> upstream]
  NODE )
  [B <-> upstream]
  NODE[[downstream1 <- C ]]
  [C <-> upstream]
  NODE

```

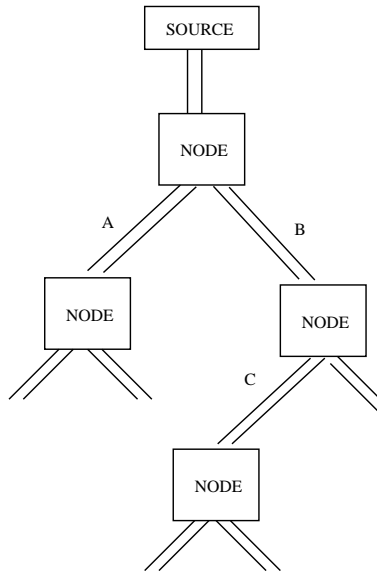


Fig. 3. An example binary multicast. Relations (i) - (iii) ensure that P is provided on each downstream link.

7 The Model

Our protocol operates with a given multicast tree with the source as the root, receivers as leaves and with an arbitrary number of intermediate nodes. A receiver initiates a request for a certain amount of resource by presenting a request to the node immediately upstream. In order to minimise network traffic, reservation requests are merged at intermediate nodes, which automatically respond to any downstream request whenever the appropriate reply is determined by responses previously relayed from above; otherwise the request is relayed upstream and

the subsequent response relayed downstream. A receiver is ultimately given an accept (successful) response if all upstream nodes, including the source, have sufficient local resources to satisfy the request, and a reject (failure) response otherwise – due to some node along the path having insufficient local resources.

We build a general model of a network node, and inductively establish appropriate properties. The general communication convention is that an intermediate node has access to bi-directional channels: one upstream towards the source and two downstream towards receivers. We model resources as small integers, and define a single type to distinguish successful reservations from rejections.

```

MAX_R = 3 // system parameter, set to 3
RESOURCE = {0 . . MAX_R}
datatype RESULT = accept | reject
datatype MESSAGE = request.RESOURCE
                | reply.RESULT.RESOURCE

channel upstream, downstream, downstream1, downstream2,
        local, local1, local2, A : MESSAGE

```

A source S with a fixed amount q of local resources is always live on downstream requests, accepting those for which it has adequate resources and rejecting otherwise. Replies are merged, i.e., consecutive identical requests may generate only a single reply. A general $SOURCE$ behaves as a source with an arbitrary amount of local resources q .

```

S(q) = let
  S'(q, {v}) = downstream.request?v -> S'(q, {v})
  S'(q, seen) =
    (downstream.request?vv -> S'(q, union(seen, {vv})))
  []
  (|~| v: seen @ if v <= q
    then downstream.reply!accept.v -> S'(q, diff(seen, {v}))
    else downstream.reply!reject.v -> S'(q, diff(seen, {v}))))
within S'(q, {})

SOURCE = |~| q:RESOURCE @ S(q)

```

An intermediate node linking source and receivers has a downstream interface to handle requests from other nodes immediately downstream, which it may or may not forward upstream. The $DOWN$ process is live on downstream requests and upstream replies. It remembers upstream replies in order to automatically issues replies when appropriate. Whenever it cannot already know the response for downstream request it must forward the request upstream. Rather than using a single state variable for recording pending requests, we model a node as a series of “slices”, processes handling requests and replies for exactly one resource. This device for avoiding state-space explosion is analogous to using n binary variables, b_1, \dots, b_n for representing a set containing values drawn from $1 \dots n$.

In practice this would be implemented by fewer processes sharing state. The slices interleave on downstream requests, downstream replies, and upstream requests. They synchronise on upstream replies, since an accept for an amount ma implies that requests for all $v \leq ma$ should be accepted, and a reject for lr implies that all $v \geq lr$ should be rejected. Inconsistent upstream replies are ignored.

```

Down_slice(v, know, r, repdown, requp) =
  downstream.request.v ->
    (if know
     then Down_slice(v, true, r, true, false) // merge reply down
     else Down_slice(v, false, r, repdown, true)) // merge req up
[]
upstream.reply?rr?vv ->
  (if know or (vv>v and rr==reject) or (vv<v and rr==accept)
   then Down_slice(v, know, r, repdown, requp) //ignore reply
   else Down_slice(v, true, rr, (repdown or requp), false))
[]
(repdown and know & // reply down enabled
 downstream.reply!r!v -> Down_slice(v, true, r, false, false))
[]
(requp and know==false & // request up enabled
 upstream.request!v -> Down_slice(v, false, r, true, false))

DOWNV(v) = Down_slice(v, false, accept, false, false) // initialised

DOWN = || v: RESOURCE @ // parallel combination of slices
  [{| downstream.request.v, downstream.reply.reject.v,
    downstream.reply.accept.v, upstream.request.v,
    upstream.reply, |}]
  DOWNV(v)

```

We model an intermediate node having an upstream channel and two downstream channels as a node with two downstream interfaces and a single local controller behaving as a source. The local controller examines requests relayed from the downstream interfaces, and is prepared to reject automatically or to merge the request upstream. Each downstream interface takes in downstream requests, and is immediately prepared to respond or to merge them to the local controller. Upstream replies are merged downstream. The downstream interfaces interleave downstream requests, downstream replies, and upstream requests, and synchronise on upstream replies.

```

NODE = (((SOURCE [[ downstream.request <- local1.request,
                  downstream.request <- local2.request,
                  downstream.reply.reject <- local.reply.reject,
                  downstream.reply.accept <- upstream.request ]])
        [|{| local1, local |} |])
  DOWN [[ upstream.request <- local1.request,
          upstream.reply <- local.reply,
          downstream <- downstream1,
          upstream.reply <- upstream.reply ]])
        \{| local1 |})
  [|{| local2, local, upstream.reply |} |])
  DOWN [[ upstream.request <- local2.request,
          upstream.reply <- local.reply,
          downstream <- downstream2,
          upstream.reply <- upstream.reply ]])
        \{| local2, local |})

```

We inductively establish that any collection of intermediate nodes behave as a source on each of its downstream channels. Thus this is an end-to-end property guaranteed by a source to an end receiver. The *base case* is trivial in that we model an end sender as a source, which is certainly refined by itself.

The *inductive case* for each downstream channel, corresponding to 5 (ii) and (iii) is checked with FDR using the `assert` statement :

```

assert SOURCE [F= (((SOURCE [downstream <-> upstream] NODE)
                  [| {| downstream1 |} |]) CHAOS(|{|downstream1|}))
                \ {|downstream1|})
              [[downstream2 <- downstream]]

assert SOURCE [F= (((SOURCE [downstream <-> upstream] NODE)
                  [| {| downstream2 |} |]) CHAOS(|{|downstream2|}))
                \ {|downstream2|})
              [[downstream1 <- downstream]]

```

Also of interest is that the behaviour of an intermediate node along each of its downstream channels (given below as `DownBviour`) can be established inductively, provided that it is ultimately linked to a sender source. The base case is nontrivial in that a simple source does not satisfy the property `DownBviour` since the behaviour of `SOURCE` is less deterministic than `DownBviour`. We require two base cases corresponding to 5(i), one for each downstream channel:

```

DownBviour = (SOURCE [[ downstream <- A ]])
              [| {| A |} |])
              DOWN [[ upstream <- A ]]) \ {| A |}

Basecase = SOURCE [downstream <-> upstream] NODE

```

```

assert DownBviour [F= ((Basecase
  [|{| downstream2 |}|] CHAOS({|downstream2|}))
  \ {| downstream2|})
  [[downstream1 <- downstream]]
assert DownBviour [F= ((Basecase
  [|{| downstream2 |}|] CHAOS({|downstream2|}))
  \ {| downstream2|})
  [[downstream1 <- downstream]]

```

And the inductive case (corresponding to 5 (ii) and (iii) :

```

assert DownBviour [F= (((DownBviour [downstream <-> upstream] NODE)
  [|{| downstream1 |} |] CHAOS({|downstream1|}))
  \ {|downstream1|})
  [[downstream2 <- downstream]]

assert DownBviour [F= (((DownBviour [downstream <-> upstream] NODE)
  [|{| downstream2 |} |] CHAOS({|downstream2|}))
  \ {|downstream2|})
  [[downstream1 <- downstream]]

```

Our nodes additionally satisfy another property – minimising traffic on the network – by guaranteeing that after some reaction delay after an up reply, they do not issue requests for which they can already know the response. We specify that such a receiver process may repeatedly issue requests, but only for ones not determined by past responses. It remembers the maximum accepted and least rejected amounts, and adjusts these according to new responses:

```

adjust(accept,qq,ma,lr) = if qq < lr then max(qq,ma) else ma
adjust(reject,qq,ma,lr) = if qq > ma then min(qq,lr) else lr
channel tock

```

R specifies that a process must be live on upstream responses, and may or may not issue upstream requests for which it cannot determine the upstream response by past responses. It is allowed some time (tocks) before it must adjust according to new upstream responses.

```

R(ma,lr,t,k,ma',lr') =
  (upstream.reply.accept?vv ->
    RR(ma,lr,0,2,adjust(accept,vv,ma,lr),lr))
  []
  (upstream.reply.reject?vv ->
    RR(ma,lr,0,2,ma,adjust(reject,vv,ma,lr)))
  []
  ((|~| v: RESOURCE @ ma < v and v < lr and t <= k &
    upstream.request!v -> RR(ma,lr,t,k,ma',lr')) |~| STOP )
  [] ( t <= k & tock -> if t = k
    then RR(ma',lr',t+1,k,ma',lr') // reset
    else RR(ma,lr,t+1,k,ma',lr') ) // advance clock

```

A receiver `REC` initially has maximum accepted `ma = 0` and least rejected `lr = MAX_R`, and must adjust to replies within 2 internal `tock` cycles:

```
REC = R(0,MAX_R,0,2,0,MAX_R) \ {| tock |}
```

An intermediate node satisfies the traffic reducing property on its upstream channel. That is, from the perspective of its upstream node, an intermediate node behaves as a receiver – no matter what the behaviour along the next-hop downstream links. This property is established without using induction simply by lazy abstracting all downstream channels, and asserting that the result refines the `REC` property:

```
CHAOSDOWN = CHAOS(|downstream1,downstream2|)
```

```
assert REC [F= (NODE[{| downstream1, downstream2 |}] CHAOSDOWN)
              \ {| downstream1,downstream2|}
```

8 Conclusions and Comparison with Other Work

There are a variety of different techniques based on induction for reducing a system of unbounded processes to a finite-state problem. Not surprisingly all of these, as well as ours, rely on proof obligations corresponding to a base case and an inductive case. The methods differ in the particular mechanisms, derived from their underlying process theory, for abstracting away the unbounded parts of the system. Techniques appearing in the literature illustrated on simple protocols include formulations in process algebra applied to a bounded buffer and token ring [33], network grammars and abstraction applied to a token ring algorithm and binary tree parity checker [4], general induction theorem applied to a distributed replication algorithm and dining philosophers [18], PVS model checking applied to a mutual exclusion protocol [27]. Creese and Roscoe [6] are developing a form of induction for CSP/FDR using data independence to handle unbounded collections of processes retaining their separate identities, rather than abstracted away as in the technique presented here. In [5, 22], simpler properties of the RSVP protocol are modelled with CSP/FDR.

Wolper and Lovinfosse [33] establish that network invariants do not always exist, that is, there are true properties for which there are no corresponding network invariants (although they do not give an example of such). The novelty of our work is that it is applied to a complex multiservice network protocol, incorporating both inductive and non-inductive behaviours which very much depend on the data values along visible interfaces. This demonstrates that our technique can be a practical strategy for reducing a large protocol so that it can be handled comfortably by a model checker. The key is to isolate invariant properties amenable to inductive proof, and abstract away non-inductive behaviours. Hence, we isolate interesting properties of each pair of downstream channels alone, and check the induction proof obligations for each pair.

For this protocol there are important properties which are not inductive in the sense defined by refinement relations 5 (i) - (iii). For example, the traffic reduction property REC does not satisfy these relations. The specified behaviour for this property by necessity allows a fixed amount of reaction delay before new upstream replies must be assimilated for subsequent automatic responses to downstream requests. Such data-dependent behaviour appearing on the right-hand side of the refinement relations increases the reaction delay over that of the left-hand side – in effect, by composing arbitrary nodes satisfying this property this reaction delay becomes unbounded. The most we can show by model checking is that such properties are link-to-link, and hence we rely on a different use of abstraction to establish that

If a node exhibits desirable behaviour along a link within t for some time t , then an arbitrary composition of nodes exhibits desirable behaviour along the link within t' for some t' .

It is interesting to note that although certain behaviours are inductive in the above sense, it could be significant that they fail to be inductive in a model-checking sense. For some protocols, link reaction delays could accumulate to an unacceptable end-delay.

Our induction technique is naturally suited for verifying certain end-to-end properties of protocols designed to operate with arbitrary numbers of interacting nodes. This is not surprising, since these protocols are inductive by design. The strength of our technique is that it can handle complex protocols exhibiting a mixture of model-checking inductive and non-inductive behaviours.

Finally, we note that one important property which our technique establishes is that our multicast protocol is deadlock free. Our method can be used to model complex end-to-end protocols in order to establish deadlock and divergence freedom, or to reveal unexpected deadlock or livelock for these protocols.

Acknowledgements The authors would like to thank Michael Goldsmith, Bryan Scattergood, Bill Roscoe and Carroll Morgan for valuable advice and discussions. Also appreciation to the reviewers for their careful reading.

References

1. 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.
2. R Butler. A CSP Approach to Action Systems, DPhil Thesis, Oxford U., 1992.
3. R Braden, L Zhang, S. Berson, S. Herzog and S. Jamin. Resource reSerVation Protocol (RSVP), Ver. 1, Functional Spec. Internet Draft, IETF 1996.
4. E Clarke, O Grumberg and S Jha, Verifying parameterized networks using abstraction and regular languages, *Proc. CONCUR'95*, LNCS 962, Springer 1995.

5. S Creese, An inductive technique for modelling arbitrarily configured networks, MSc Thesis, Oxford U., 1997.
6. SJ Creese and AW Roscoe, Verifying an infinite family of inductions simultaneously using data independence and FDR, (Submitted).
7. J Davies, Specification and Proof in Real-time Systems, D.Phil Thesis, Oxford U., 1991.
8. Formal Systems (Europe) Ltd. Failures Divergence Refinement. *User Manual and Tutorial*, version 2.11.
9. Estelle Specifications, <ftp://louie.udel.edu/pub/grope/estelle-specs>
10. J Guttman and D Johnson, Three Applications of Formal Methods at MITRE, *Formal Methods Europe*, LNCS873, Naftolin, Denfir, Barcelona '94.
11. 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>
12. CAR Hoare. *Communicating Sequential Processes*. Prentice-Hall 1985.
13. ISO Rec. 9074, The Extended State Transition Language (Estelle), 1989.
14. ISO: Information Processing System - Open System Interconnection - LOTOS - A Formal Description Technique based on Temporal Ordering of Observational Behavior, IS8807, 1988.
15. DM Jackson. Experiences in Embedded Scheduling. *Formal Methods Europe*, Oxford, 1996.
16. M Jmail, An Algebraic-temporal Specification of CSMA/CD Protocol, *Proc. IFIP WG6.1 Inter. Sym. on Protocol Spec., Testing and Verification XV*, Dembrinski and Sredniawa, eds, Warsaw Poland, June '95, Chapman Hall.
17. A Kay and JN Reed. A Rely and Guarantee Method for TCSP, A Specification and Design of a Telephone Exchange. *IEEE TSE*. 19,6 1993, pp 625-629.
18. RP Kurshan and M McMillan, A structural induction theorem for processes, *Proc. 8th Symposium on Principles of Distributed Computing*, 1989.
19. G Leon, J Yelmo, C Sanchez, F Carrasco and J Gil, An Industrial Experience on LOTOS-based Prototyping for Switching Systems Design, *Formal Methods Europe*, LNCS 670, Woodcock and Larsen, eds., Odense Denmark, '93.
20. J Navarro and P 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.
21. K Paliwoda and JW Sanders. An Incremental Specification of the Sliding-window Protocol. *Distributed Computing*. May 1991, pp 83-94.
22. J Reed, D Jackson, B Deianov and G Reed, Automated Formal Analysis of Networks: FDR Models of Arbitrary Topologies and Flow-Control Mechanisms, ETAPS-FASE98 Fund. Approaches to Soft. Eng., Lisbon, LNCS 1382 Mar '98.
23. AW Roscoe, PHB Gardiner, MH Goldsmith, JR Hulance, DM Jackson, JB Scattergood. H hierarchical compression for model-checking CSP or How to check 10^{20} dining philosophers for deadlock, Springer LNCS 1019.
24. 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.
25. AW Roscoe, **Theory and Practice of Concurrency**, Prentice Hall, 1998.
26. B Scattergood, **Tools for CSP and Timed CSP**, D.Phil Thesis, Oxford U., 1998.
27. N Shankar, Machine-Assisted Verification Usin Automated Theorem Proving and Model Checking, *Math. Prog. Methodology*, ed M Broy.

28. K Sidle, Pi Bus, *Formal Methods Europe*, Barcelona, 1993.
29. J Sinclair, Action Systems, Determinism, and the Development of Secure Systems, PhD Thesis, Open University, 1997.
30. AS Tanenbaum. *Computer Networks*. 3rd edition. Prentice-Hall 1996.
31. J Davies, D Jackson, G Reed, J Reed, A Roscoe, and S Schneider, Timed CSP: Theory and practice. *Proc. REX Workshop, Nijmegen*, LNCS 600, Springer, '92.
32. JS Turner. New Directions in Communications (or Which Way to the Information Age). *IEEE Commun. Magazine*. vol 24, pp 8 -15, Oct 1986.
33. P Wolper and V Lovinfosse, Verifying properties of large sets of processes with network invariants, *Proc. International Workshop on Automatic Verification Methods for Finite-State Machines*, LNCS 407, Springer-Verlaag, 1989.
34. LOTOS Bibliography, <http://www.cs.stir.ac.uk/~kjt/research/well/bib.html>
35. 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 synchronise on it. The CSP processes that we use are constructed from the following:

STOP is the simplest CSP process; it never engages in any action, nor terminates.

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

$a \rightarrow P$ 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 ($c!v \rightarrow P$) and inputs ($c?x \rightarrow P(x)$) of values along named channels.

$P \mid \sim \mid Q$ represents internal choice. It behaves as P or Q *nondeterministically*.

$P \llbracket \mid Q$ represents external or *deterministic* choice. It will offer the initial actions of both P and Q to its environment at first; its subsequent behaviour 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 $\mid \sim \mid$). **STOP** $\llbracket P$ behaves as P .

$P \llbracket [A] \mid Q$ represents parallel composition. P and Q evolve concurrently, except that events in A occur only when P and Q agree (i.e. *synchronise*) to perform them.

$P \parallel \mid Q$ 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 .

$P \setminus A$ 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.

$P \llbracket [a \leftarrow b] \mid$ represents the process P with a renamed to b .

$P \llbracket [a \leftrightarrow b] \mid Q$ is the linked parallel operator. P and Q synchronise on channels a and b , which have been renamed to the same and hidden.

There are also straightforward generalisations of the choice operators over non-empty sets, written $\mid \sim \mid x:X @ P(x)$ and $\llbracket [x:X @ P(x)$.