

A Coloured Petri Net Approach to Protocol Verification*

Jonathan Billington, Guy Edward Gallasch, and Bing Han

Computer Systems Engineering Centre
University of South Australia
Mawson Lakes Campus, SA 5095, Australia
`jonathan.billington@unisa.edu.au`,
`{guy.gallasch,bing.han}@postgrads.unisa.edu.au`

Abstract. The correct operation of communication and co-operation protocols, including signalling systems in various networks, is essential for the reliability of the many distributed systems that facilitate our global economy. This paper presents a methodology for the formal specification, analysis and verification of protocols based on the use of Coloured Petri nets and automata theory. The methodology is illustrated using two case studies. The first belongs to the category of data transfer protocols, called Stop-and-Wait Protocols, while the second investigates the connection management part of the Internet's Transmission Control Protocol (TCP). Stop-and-Wait protocols (SWP) incorporate retransmission strategies to recover from data transmission errors that occur on noisy transmission media. Although relatively simple, their basic mechanisms are important for practical protocols such as the data transfer procedures of TCP. The SWP case study is quite detailed. It considers a class of protocols characterized by two parameters: the maximum sequence number (MaxSeqNo) and the maximum number of retransmissions (MaxRetrans). We investigate the operation of the protocol over (lossy) in-sequence (FIFO) channels, and then over (lossy) re-ordering media, such as that provided by the Internet Protocol. Four properties are considered: the bound on the number of messages that can be in the communication channels; whether or not the protocol provides the expected service of alternating sends and receives; (unknowing) loss of messages (i.e. data sent but not received, and not detected as lost by the protocol); and the acceptance of duplicates as new messages. The model is analysed using a combination of hand proofs and automatic techniques. A new result for the bound of the channels ($2\text{MaxRetrans}+1$) is proved for FIFO channels. It is further shown that for re-ordering channels, the channels are unbounded, loss and duplication can occur, and that the SWP does not provide the expected service. We discuss the relevance of these results to the Transmission Control Protocol and indicate the limitations of our approach and the need for further work. The second case study (TCP) illustrates the use of hierarchies to provide a compact and readable CPN model for a complex protocol. We advocate an incremental approach to

* Partially supported by an Australian Research Council (ARC) Discovery Grant (DP0210524).

both modelling and analysis. The importance of stating the assumptions involved is emphasised and we illustrate how they affect the abstractions that can be made to simplify the model. The incremental approach to analysis allows us to validate the model against the TCP definition and to show how errors in the connection establishment procedures can be found. Finally we provide some observations and tips on the how the methodology can be used based on many years of experience. The emphasis of the paper is on providing a tutorial style introduction to the methodology, examining case studies in depth, rather than breadth, and giving some insight into the process while noting its limitations.

1 Introduction

The global economy is becoming more and more dependent on distributed systems. An important example is the Internet which connects millions of computers all over the world via the interconnection of thousands of networks. It provides the infrastructure for the world wide web and email and the development of new information services such as electronic commerce, GRID computing, web services and mobile data services. At the heart of distributed systems are the communication and co-operation protocols that ensure that the required services are provided to their users. It is thus vitally important that these protocols operate correctly.

A protocol needs to satisfy a set of properties defined for the communication service it is meant to provide (e.g. data is neither lost nor duplicated and arrives in sequence, and there are no deadlocks). Proving that a protocol satisfies its required properties is known as protocol verification. Protocol verification [40,70] is a difficult problem due to inherent complexity [77].

This paper summarises a protocol verification methodology set in the context of the broader field of protocol engineering [9]. The paper does not attempt to compare the merits of this approach with other approaches. For a comparison of the main techniques for protocol specification and analysis, including the Petri net approach, the reader is referred to a recent survey by Babich and Deotto [4].

Coloured Petri Nets [48–50,53] have been used successfully for the modelling and analysis of a wide range of concurrent and distributed systems [50] including communication systems and protocols [11,14,23,24,31,32,51,83]. Thus the methodology uses Coloured Petri Nets for the specification of protocol behaviour.

In order to prove that the *protocol specification* satisfies the requirements of its users, a higher level specification, known as the *service specification* is also modelled with Coloured Petri Nets (CPNs). We then wish to prove that the protocol specification is a refinement of the service specification, in that it complies with the sequencing constraints on user observable events (known as service primitives) that are embodied in the service specification. These constraints can be expressed as the *service language*: the set of sequences of service primitives at each of the user interfaces. In principle, the service language can be derived from the CPN service specification by generating its occurrence graph (reachability graph, state space), converting it to an automaton by nominating halt states and

labelling events that are not observable (by users) as empty, and then reducing this automaton to its minimised deterministic form using standard automata reduction techniques.

We can follow the same approach with the protocol specification, masking out internal events such as retransmissions, to obtain the *protocol language*: the set of sequences of service primitives generated by the protocol. We then compare the protocol and service languages. If they are the same, then we can say that the protocol specification satisfies the service sequence constraints. If the protocol language is a subset of the service language, then the protocol satisfies the constraints, but may not provide all the desired features of the service. Finally if the protocol language is not a subset of the service language, then it contains sequences that are erroneous (if the service language has been defined correctly).

We are also interested in other behaviour of the protocol, such as whether or not it will deadlock or livelock in various circumstances. In general, we need to define (a priori) a set of properties that the protocol needs to fulfill, such as correct termination or transparent delivery of data. These properties can be expressed in some language (often a temporal logic) and model checking techniques used over the occurrence graph (OG) to prove their existence or otherwise. Thus our methodology comprises two parts: the first checks sequencing constraints are satisfied at the user interface; the second proves general properties (such as boundedness or absence of deadlock) and specific protocol properties by hand proofs on the CPN model or by investigating the occurrence graph.

After presenting the methodology in some detail in Section 2, this paper formalises the methodology in Section 3 and illustrates it with two case studies. The first of these is concerned with data transfer procedures and comprises the class of Stop-and-Wait protocols [73] (see Sections 4 to 8). This is motivated by their basic mechanisms being important for practical protocols such as the Internet's Transmission Control Protocol (TCP). The second case study (Sections 9 to 12) investigates the connection management part of protocols, using the rather complex 3-way handshake of TCP. The paper also attempts to provide some guidelines for modelling and analysing protocols using high-level Petri net techniques based on twenty-five years experience of the first author and the work of his colleagues and students. The paper does not aim to be complete, it does however aim to give some insight and detailed illustrations of a Coloured Petri Net approach to protocol verification.

2 Protocol Verification Methodology

Our first attempts to develop a protocol verification methodology were published in [19]. Since then we have used it with some success to verify that industrial scale protocols do or do not meet their service specifications [34–37, 39, 55, 58, 59, 74, 78, 80].

The main steps of the methodology are:

1. Service specification: specify the service to be provided to the users of the protocol under investigation;
2. Protocol specification: specify a protocol entity for each party involved in communication;
3. Underlying service specification: specify the characteristics of the communication medium by which the different protocol entities communicate, by defining the communication service provided by the underlying service in the protocol architecture;
4. Composite specification: combine the protocol specification with the specification of the medium to obtain a composite specification of the protocol entities communicating over the underlying service;
5. Analysis: analyse the composite specification using reachability analysis and/or theorem proving to investigate desired properties of the system; and
6. Comparison: compare the service specification with the composite specification to see if the composite specification is a faithful refinement of the service.

2.1 Service Specification

This first step of the methodology has led to the development of formal service specifications using high-level nets for a number of protocols. The first of these was for the ISO Open Systems Interconnection (OSI) Transport Service [7]. The OSI standardisation effort had strongly supported the notion of service specification and promulgated guidelines for their development, known as the OSI service conventions [46]. This has greatly assisted the development of formal service specifications. In the case of OSI and other protocol development forums, such as ITU-T [41] and the Wireless Application Protocol Forum [81], this has led to the inclusion of service definitions as an integral part of developing protocol specifications. In contrast, this has not been the case in the development of Internet drafts and Request for Comments (RFCs) used in the Internet community.

Integral to the development of service specifications is the notion of a *service primitive*. A service primitive represents an interaction between the user of the service (often another protocol entity in a higher layer) and the provider of the service (i.e. the protocol operating over its underlying service). It corresponds to some feature of the service, such as a request by a user to establish or release a connection or to transfer data, or an indication by the provider that a connection has been requested by a remote user. Primitives are meant to be implementation independent, allowing them to be implemented in various ways such as message passing or procedure calls. They are also considered to be atomic events in service specifications, and are readily modelled by labelling transitions in a Coloured Petri net with the name of the primitive.

In an attempt to verify industrial protocols we have recently developed a number of service specifications. These have included the Wireless Application

Protocol (WAP) transaction layer [35] and the capability exchange signalling (CES) protocol of ITU-T recommendation H.245 [56], where the service definitions exist in the standards documents. We have also attempted to create service specifications for Internet Request for Comments such as the Resource Reservation Protocol (RSVP) [78,79], the Internet Open Trading Protocol (IOTP) [58,60] and the Internet's Transmission Control Protocol (TCP) [15,16]. We have found that the specification of services has ranged from relatively straightforward (WAP) to requiring significant ingenuity.

Although the CES protocol and service are very simple, ensuring that the service specification properly reflected the sequences of service primitives required complex synchronisation mechanisms [56]. The work for RSVP and IOTP was much more complicated, firstly due to the complex nature of the protocols and secondly that no service definitions had been written as part of the standardisation process. With IOTP there was the added complexity of catering for 4 interfaces, due to there being 4 user roles (Consumer, Merchant, Payment Handler and Delivery Handler). This complexity has led to the development of local automata which express the sequencing constraints at each one of the interfaces separately, before trying to define the global sequences by converting the automata into Coloured Petri nets (CPNs) and synchronising them via queues. We considered that attempting to directly build the correct CPN (covering all interfaces at once) would be too error prone, and that a divide and conquer approach of specifying local interface sequences first, as is done in the service conventions, would be an easier task. This has led to defining a validation step when specifying services this way. The validation step comprises proving that the CPN service specification does conform to the local sequences as defined by the local automata. This is done using reachability analysis and automata reduction techniques [5]. Interested readers can consult [60] for the details. For complex service specifications (such as IOTP), this validation step has proved to be of significant value, as now an iterative approach is used to remove errors from the CPN specification of the service.

Our experience with both the CES protocol and TCP has shown that the reachability graph for many service specifications is not finite. This is due to the service provider (e.g. the Internet) having an unknown storage capacity (number of buffers) and that the service allows an arbitrary number of service data units to be accepted by the service provider, before they are delivered to a peer user. This has important ramifications for our comparison step, which we shall return to below in section 2.3.

2.2 Protocol, Underlying Service and Composite Specifications

Currently we tend to consider steps 2 to 4 together, whereas step 1 is quite independent, and could be performed by a separate member of the protocol verification team, concurrently with steps 2 to 4. The reasons for considering these 3 steps together are that:

- it is important that the level of abstraction used for modelling the underlying service and the protocol entities is the same;

- for verification, the precise architectural location where the underlying service is considered, may not correspond to a strict layer boundary; and
- separate specifications for the protocol and underlying service tend to generate a larger state space when combined, due to there being ways of optimising the specification at the boundaries when the composite specification is considered.

Regarding the first point, normally we consider protocol data units (packets) as being transferred through the underlying medium, rather than service data units.

We illustrate the second point by considering how to model the error detection mechanism in protocols that need to recover from transmission errors. To detect transmission errors, packets contain redundant bits known as a checksum. The operation of checksums is very well known and either does not need to be verified or can be verified separately. We thus can assume that the checksum works correctly and then use a non-deterministic approach to model the *effect* of the checksum: a packet is accepted as correct (passes the checksum) or is discarded (fails the checksum). The effect is that one possible action is to discard (or lose) the packet. The processing of a checksum is an operation that occurs on the contents of the whole packet, and thus it is done before the details of the main protocol mechanisms are considered. Thus checksum processing, although part of the protocol, can be considered as a preliminary layer that can be combined with the characteristics of the medium (underlying service). This loss of a packet (due to the checksum) can then be combined with a lossy medium - such as that provided by the Internet Protocol (IP) [61] - where packets can be dropped at routers due to congestion. Hence we only need to model the loss of a packet once. It may be due to a bit error, or due to congestion, but from the point of view of the major protocol mechanisms, it does not matter. Thus when modelling a transport protocol such as TCP, we only model *above* the checksum procedures of TCP, and hence the boundary for verification is not strictly at the TCP and Internet Protocol boundary.

When using hierarchical CPNs it is natural to model the medium (underlying service) between protocol entities as a hierarchical substitution transition as in [53]. This makes a lot of sense from a specification point of view, as the details of the operation of the medium can be hidden and specified separately. Further, the medium can be changed separately as we change our view of its characteristics. However, this can have a penalty when considering verification and state space explosion. The use of a substitution transition requires that there is both an input place (perhaps representing a sending entity buffer) and an output place (representing a receiving entity buffer). This can lead to a combinatorial explosion of states of the buffers in both directions of information flow in the medium. To avoid this component of combinatorial state space explosion, we can often provide a coarser model, where the storage of the sending buffer, the medium and the receiving buffer are combined and modelled by one place. Thus we do not model any of the details of transferring packets from one buffer to another, which is not relevant to most protocol mechanisms, and hence

avoid these different buffer states and the consequential state space explosion that results. However, this does mean that we can not hide the medium in the specification using hierarchical CPNs, as no hierarchical place is provided. Thus it is important to keep in mind sources of state space explosion when creating a model that is to be verified using model checking approaches.

2.3 Analysis and Comparison

The methodology in [19] proposed to concentrate on comparing sequences of service primitive events at each of the interfaces, between the service specification and the composite specification. It was assumed that other properties, such as absence of *deadlock* or *livelock*, or boundedness properties could be decided by querying the reachability graph.

Deadlocks can be determined by examining the dead markings of the reachability graph of the composite specification, to see if they are desired or not. (Dead markings correspond to the leaf nodes of the reachability graph.) Desired dead markings correspond to required terminal states, such as in a connection establishment protocol, where both protocol entities perform an initialisation sequence (for example to synchronise sequence numbers or to set flow control window sizes) before data is transferred. The connection establishment protocol needs to place each protocol entity in the *data transfer* state, and have no packets left in the underlying medium. This would be a dead marking corresponding to the desired terminal state of correct establishment. Dead markings that are not desired, we then call deadlocks. These may correspond to *unspecified receptions*, where a packet is left in the medium because the protocol does not define a procedure for processing the packet in some circumstances, or to when the different protocol entities are not synchronised (e.g. one is established, while the other is closed).

Livelocks occur when the protocol entities are involved in the exchange of control information (such as a reset) but no progress is made with respect to the aim of the protocol, which is to transfer data (for example), and that there is no way out of this cyclic behaviour. Livelocks can be detected by calculating the strongly connected components (SCC) of the reachability graph. Each terminal SCC can then be examined. It may be a dead marking or it may be a component that involves cycles. Each of these cycles needs to be checked to see if it is desired or not. In a data transfer protocol, the main loop sending and receiving data is an expected component. However, other components may not be desired (such as each end constantly sending each other resets) and they would be considered to be livelocks. Non-terminal SCCs that contain cycles are not livelocks, but may correspond to *tempo blocking* behaviour, such as repeated loss and retransmission. The difference is that there is always the possibility of escaping from this cyclic behaviour.

Also of interest are bounds on the maximum number of messages in the communication channels, as this may affect buffer and network dimensioning or the need for congestion control procedures. These properties are generic for protocols. There may also be many other properties that are specific to a particular

protocol that we wish to prove, and this can be done using a model checking approach on the reachability graph.

Given that we can determine *divergent* behaviour (deadlocks and livelocks) from the reachability graph of the composite specification, it is not important if this information is lost when comparing the service and composite specifications. Once we have determined the presence or absence of deadlocks or livelocks in our protocol we are interested in whether or not the sequences of primitives that occur in the composite specification are compatible with the service specification. Thus we can use the notion of *language equivalence*¹ or *language inclusion* to check this compatibility. We refer to the set of sequences of service primitives that occur in the service specification as the *service language* and similarly, those that occur in the composite specification as the *protocol language*.

If the protocol language is the same as the service language, then the protocol is a faithful refinement of the service specification. If the protocol language is included in the service language, then we may be able to define conditions under which the protocol is also a faithful refinement, for example, if there are some options which the protocol does not include, or some concurrent behaviour that is acceptable but not essential, see [58,59]. It may also be the case that the protocol does not implement an essential part of the service, in which case the protocol needs to be revised to include it (for example, it is unlikely that the empty set would be an acceptable subset of service primitive sequences!). However, if the protocol language is not a subset of the service language, then there is at least one sequence in the protocol that does not exist in the service specification. This means that there is an error, either in the service specification or in the protocol specification. If the error is in the service specification, then it is normally readily fixed by inspecting the sequences and understanding how they occur. If it is in the protocol specification (which is more usually the case) then the sequence of service primitives needs to be traced back to protocol behaviour (in the reachability graph) to see how the sequence was generated, normally a more difficult task, as there are usually many more epsilon transitions in the protocol specification.

Obtaining the Service Language. The service language is obtained from the service specification by generating its occurrence graph and using automata reduction techniques [5]. Normally, transitions in the service specification are labelled by service primitive names, except for some transitions that may only relate to synchronisation transitions or garbage collection. Once the OG is generated, it contains all sequences of transitions that can occur in the CPN for the initial marking. To obtain the service language, any non-primitive transition is mapped to an internal transition (epsilon transition), and acceptance (halt)

¹ There are many other equivalence notions defined in the literature (155 reported in 1993 by van Glabbeek) [77], including observational, failures, testing and Valmari's Chaos-Free Failures Divergences (CFFD). The problem with language equivalence is that progress properties, such as the absence of deadlock and livelock are not preserved. However, since these properties can be obtained directly from the protocol's state space [49] language equivalence is sufficient.

states are designated. Designation of halt states may not be trivial and requires some experience of protocols on the part of the verifier. In connection management and transaction protocol services halt states will often correspond to dead markings, while in data transfer services there may only be one halt state corresponding to the initial marking. The OG can then be considered as a Finite State Automaton (FSA), that encodes a language. This FSA is then transformed into an equivalent deterministic and minimum FSA that preserves the sequences of primitives while removing the epsilon transitions. This uses 5 algorithms and is implemented in tools such as FSM from AT&T [33]. This minimum deterministic FSA is the service language.

Obtaining the Protocol Language. We use the same procedure to obtain the protocol language from the composite specification. An important difference is that the service specification has been created with service primitives in mind. In the composite specification, there may be few guidelines as to which transitions correspond to service primitives, because the protocol may have been developed without a service specification, as is the case for Internet protocols. In this case, significant judgement is required to label transitions correctly. It is worse than that because if the protocol is modelled first (a natural way to proceed to get a good understanding of the protocol before trying to retrofit its service), it may be that decisions have been made in the protocol model which mean that for a particular transition, a service primitive will only correspond to some of its modes and not others. Thus the creation of the FSA corresponding to the OG needs to map transition modes (rather than transitions) to primitives or epsilons.

Comparing Languages. Languages can be compared if they are represented by deterministic automata. We use the FSM tool to obtain the difference between the service language and the protocol language and vice versa. For details of how this is done see [34, 58, 78].

Infinite State Systems. We have assumed in this section that the systems we are dealing with are finite state and for physical systems, this seems to be a reasonable assumption. Unfortunately, there are times when we do not know the range of a parameter, even though we may be sure it is finite. An example of this is the storage capacity of the Internet. In this case, we would like our results to be general, that is, to apply to any arbitrary value of the storage capacity. Our approach to this problem [17, 56] has been to introduce a parameter representing the storage of the Internet (as the length of a queue) into the model. We can then obtain results for small values of this capacity using standard reachability analysis. In the case of the CES protocol service, we find that the OG grows linearly with the length of the queue [56]. We can thus derive recursive formulae for the OG for any value of the length of the queue. Further, we have been able to show that the corresponding deterministic automaton (DFSA) also grows linearly in the size of the queue and have derived a recursive formula for it [57]. The hope is that we shall be able to derive a similar recursive formula for the

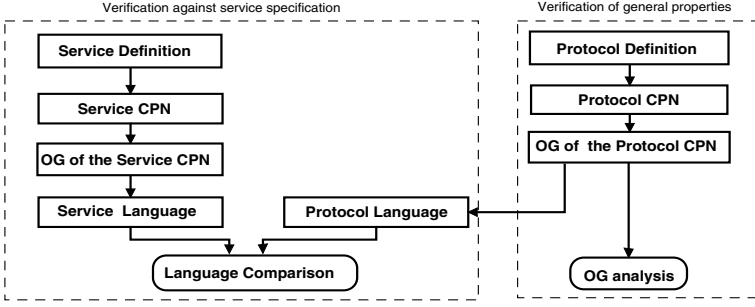


Fig. 1. Protocol verification methodology.

protocol language DFSA and then be able to extend language comparison to recursive FSAs. In the case of TCP [17], we have been able to show that the OG of the service grows exponentially with the size of the queue and have been able to derive expressions for the nodes and arcs of the OG directly, without the need for recursive formulae.

2.4 Summary

The verification methodology is summarised in Fig. 1. The dashed box on the right shows the procedures for verifying properties of a protocol. We start with the protocol definition (often provided by an international standard) and model it with CPNs. From the CPN model, we use a software package for CPNs called Design/CPN [29] to generate its OG. By analysing the OG we can obtain information about the dynamic behaviour and properties of the protocol. This may be proving correct termination (e.g. absence of deadlock and livelock), investigating boundedness properties and message sequences, or more specific properties that could be written in a temporal logic or other technique suitable for querying OGs.

The dashed box on the left of Fig. 1 illustrates the steps required to verify a protocol against its service language [19]. We do this by comparing the sequences of *service primitives* that occur as a result of the protocol's operation (the *protocol language*), with the sequences specified in the service specification (the *service language*). We firstly create a CPN model of the service specification, in which service primitives are associated with CPN transitions. The OG of the CPN model is calculated. The OG includes all the possible occurrence sequences of CPN transitions. The CPN model may include transitions that do not model service primitives, but rather internal events of the service provider required to ensure correct operation. We need to eliminate these internal transitions while preserving service primitive sequences. To do this, we treat the OG as a FSA and use standard FSA reduction techniques [5]. This minimised and deterministic FSA embodies the *service language*. In a similar way, we generate the *protocol language*. These are compared to see if all the sequences of service primitives in the protocol language are sequences specified in the service language, to discover

if there are any inconsistencies between the protocol definition and the service definition. The automata reductions and comparison algorithms are automated in tools such as FSM [33].

3 Definitions

3.1 High-Level Petri Nets

We use Coloured Petri nets and Design/CPN for the specification of protocols and services, and refer readers to [48–50, 53] for an introduction to CPNs and their definitions. However, for some of the proofs in this paper, it is useful to use the definition of the High-level Petri Net (HLPNs) semantic model presented in clause 5 of international standard ISO/IEC 15909-1 [44]. It is drawn from similar work published in [8, 10] and earlier work in [47].

In order to be self-contained, we begin by presenting definitions concerning multisets and vectors, followed by the semantic model from [44]. We then define occurrence sequences for high-level nets and provide some propositions that we require in the analysis of the Stop-and-Wait protocol.

Multisets

Definition 1. *A multiset, $B : A \rightarrow \mathbb{N}$, is a function that associates a natural number, known as the multiplicity, with each of the elements of a non-empty basis set, A .*

The multiplicity of $a \in A$ in B , is given by $B(a)$. The set of all multisets over A is denoted by μA .

Multiset Operations

Definition 2. Equality:

Two multisets, $B1, B2 \in \mu A$, are equal, $B1 = B2$, iff $\forall a \in A, B1(a) = B2(a)$.

Definition 3. Comparison:

$B1$ is less than or equal to $B2$, $B1 \leq B2$, iff $\forall a \in A, B1(a) \leq B2(a)$ and $B1$ is greater than or equal to $B2$, $B1 \geq B2$, iff $\forall a \in A, B1(a) \geq B2(a)$.

We define addition and subtraction on multisets $B1, B2 \in \mu A$ as follows.

Definition 4. Addition and Subtraction:

$B = B1 + B2$ iff $B(a) = B1(a) + B2(a)$

$B = B1 - B2$ iff $B(a) = B1(a) - B2(a)$ if $B1(a) \geq B2(a)$

There are times when we wish to subtract one multiset from another when the above restriction on multiset subtraction does not apply. We then need to consider multisets as vectors.

Vectors

Definition 5. A vector V over a basis set A is a function $V : A \rightarrow \mathbb{Z}$ where \mathbb{Z} is the set of integers.

The set of all vectors over A is denoted by νA . Subtraction is a closed operation for vectors, defined componentwise as follows.

Definition 6. Vector Subtraction:

For $V1, V2 \in \nu A$, $V = V1 - V2$ iff $\forall a \in A, V(a) = V1(a) - V2(a)$.

High-Level Petri Net

We now define a High-level Petri net (HLPN) [8, 44].

Definition 7. $HLPN = (P, T, D; Type, Pre, Post, M_0)$ where

- P is a finite set of Places.
- T is a finite set of Transitions, disjoint from P ($P \cap T = \emptyset$).
- D is a non-empty finite set of non-empty domains where each element of D is called a type.
- $Type : P \cup T \rightarrow D$ is a function used to assign types to places and to determine transition modes.
- $Pre, Post : TM \rightarrow \mu PLACE$ are the pre and post mappings with
 - $TM = \{(t, m) | t \in T, m \in Type(t)\}$, the set of transition modes; and
 - $PLACE = \{(p, g) | p \in P, g \in Type(p)\}$, the set of elementary places.
- $M_0 \in \mu PLACE$ is a multiset called the initial marking of the net.

Marking of a HLPN

Definition 8. A Marking of the HLPN is a multiset, $M \in \mu PLACE$.

Enabling of Transition Modes

Definition 9. A single transition mode, $tm \in TM$, is enabled at a marking M iff $Pre(tm) \leq M$.

We can also define the concurrent enabling of a finite multiset of transition modes (see [10, 44]) but this is not required for this paper.

Transition Rule

Definition 10. The transition rule for a single transition mode $tm \in TM$ enabled at a marking M is given by

$$M' = M - Pre(tm) + Post(tm)$$

where an occurrence of tm results in the new marking M' .

The occurrence of a single transition mode $tm \in TM$ in marking M is denoted by $M[tm]M'$ or $M \xrightarrow{tm} M'$.

Occurrence Sequences and Reachability

In addition to that defined in [44], we define the following in relation to occurrence sequences.

Definition 11. Let M be a marking of the HLPN. A finite sequence of transition modes, $tm_1, tm_2, \dots, tm_k \in TM, k \in \mathbb{N}^+$, is called a finite occurrence sequence able to occur from M , if there are markings M_1, M_2, \dots, M_k such that

$$M \xrightarrow{tm_1} M_1 \xrightarrow{tm_2} M_2 \dots \xrightarrow{tm_k} M_k$$

We denote a finite occurrence sequence by $\sigma_k = tm_1 tm_2 \dots tm_k$ and write $M \xrightarrow{\sigma_k} M_k$.

Definition 12. A marking M' is reachable from a marking M if there is a finite occurrence sequence σ_k leading from M to M' , i.e. $M \xrightarrow{\sigma_k} M'$.

Definition 13. An infinite sequence of transition modes, $\sigma = tm_1 tm_2 tm_3 \dots$ is called an infinite occurrence sequence, able to occur from a marking M , if there are markings M_1, M_2, \dots such that

$$M \xrightarrow{tm_1} M_1 \xrightarrow{tm_2} M_2 \xrightarrow{tm_3} \dots$$

Following directly from these definitions are Propositions 1 and 2. They are HLPN extensions of the propositions given in [28] and are used in the proof of Theorem 4 in Section 6.

Proposition 1. An infinite occurrence sequence σ of transition modes can occur from a marking M if and only if every finite prefix of σ can also occur from M .

Proposition 2. If M and L are markings of a HLPN and for a finite occurrence sequence σ_k , $M \xrightarrow{\sigma_k} M'$ and $L \xrightarrow{\sigma_k} L'$ then (using vector subtraction) $M' - M = L' - L$.

The following proposition is useful for proving that a finite occurrence sequence of transition modes can be repeated indefinitely.

Proposition 3. If M and L are markings satisfying $M \geq L$ then every occurrence sequence that can occur from L can also occur from M .

Proof. Consider the infinite occurrence sequence $\sigma = tm_1 tm_2 tm_3 \dots$ that can occur from L . Now all finite prefixes of σ (i.e. $\sigma_k = tm_1 tm_2 \dots tm_k, k \in \mathbb{N}^+$) can occur from L according to Proposition 1, i.e.

$$L \xrightarrow{tm_1} L_1 \xrightarrow{tm_2} \dots \xrightarrow{tm_k} L_k$$

Using the enabling condition (and transition rule) as defined above, this means that $L \geq Pre(tm_1)$. Occurrence of tm_1 at marking L leads to a marking L_1 in which the next transition mode in the sequence, tm_2 , is enabled, and from which the rest of the sequence $tm_2 tm_3 \dots tm_k$ can occur.

Now consider the marking M . We know that $M \geq L$. If $L \geq \text{Pre}(tm_1)$ then this means that $M \geq \text{Pre}(tm_1)$ also. Thus transition mode tm_1 is enabled in M .

Occurrence of tm_1 from M will lead to a new marking M_1 , i.e. $M \xrightarrow{tm_1} M_1$, and by Proposition 2 we know that $M_1 - M = L_1 - L$ (again using vector subtraction). Knowing $M \geq L$ and substituting L for M we obtain

$$M_1 - L \geq L_1 - L$$

$$\Rightarrow M_1 \geq L_1$$

We know that tm_2 is enabled in L_1 , and by the above arguments, tm_2 is also enabled in M_1 . By repeated application of the above arguments, we obtain that for every intermediate marking $L_n (1 \leq n \leq k)$ during execution of the occurrence sequence σ_k there is a corresponding marking M_n such that $M_n \geq L_n$. So all finite prefixes σ_k of the infinite occurrence sequence σ can occur from M , and by Proposition 1, σ can occur from M . Thus any occurrence sequence that can occur from L can also occur from M . \square

3.2 Definitions of Occurrence Graphs and Associated Automata

Part of the methodology requires the generation of a CPN's occurrence graph (OG) and its transformation to an appropriate finite state automaton (FSA). This section provides the definitions that are useful for this purpose.

Occurrence Graphs

We consider that an OG can be defined as a labelled directed graph, where the nodes of the graph represent markings of the CPN, and the directed arcs represent the transition modes that can occur in all executions of the net from the initial marking. The arcs are thus labelled by the transition mode. We thus start by defining a labelled directed graph.

Definition 14. *A labelled directed graph is a triple $G = (V, L, E)$ where*

- V is the set of vertices or nodes;
- L is a set of labels; and
- $E \subseteq V \times L \times V$ is a set of labelled directed edges.

Definition 15. *An occurrence graph of a HLPN with an initial marking M_0 , is a labelled directed graph $OG = (V, TM, A)$ where*

- $V = [M_0]$ is the set of markings reachable from M_0 (the reachability set);
- TM is the set of transition modes of the HLPN; and
- $A = \{(M, tm, M') \in V \times TM \times V \mid M[tm\rangle M'\}$ is the set of arcs (directed edges) labelled by transition modes.

Abstract OGs

When we only consider sequences of primitives, there are two abstractions of the OG which are useful. The first removes the modes (variable bindings) from the transition modes to give transitions only. The second removes the details of the markings, so that they are just represented by integers. Both abstractions may be used together. We formalise these abstractions in the following definitions.

For an occurrence graph where we are only interested in the transition names, rather than transition modes, e.g. in the case of service primitives where we are not concerned with service primitive parameter values, but just the primitive name, then an abstract OG with respect to transitions, AOG^T , is defined as follows.

Definition 16. *An abstract OG, with respect to transitions, of a HLPN with an initial marking M_0 and a set of transition modes, TM , is a labelled directed graph $AOG^T = (V, T, A)$ where*

- $V = [M_0]$ is the set of markings reachable from M_0 ;
- T is the set of transitions of the HLPN; and
- $A = \{(M, t, M') \in V \times T \times V \mid (t, m) \in TM \text{ and } M[(t, m)]M'\}$ is a set of arcs labelled with transition names.

In the case where we are only interested in the identification of the markings for the nodes, and not the details of the markings, we introduce an injection, I , mapping the set of reachable markings into the set of positive integers:

$$I : [M_0] \longrightarrow \mathbb{N}^+$$

where $I(M_0) = 1$ represents the initial marking.

Definition 17. *An abstract OG, with respect to markings, of a HLPN with an initial marking M_0 and a set of transition modes TM , is a labelled directed graph $AOG^M = (V, TM, A)$ where*

- $V = \{I(M) \mid M \in [M_0]\}$ is the set of nodes;
- TM is the set of transition modes of the HLPN; and
- $A = \{(I(M), tm, I(M')) \in V \times TM \times V \mid M[tm]M'\}$ is the set of arcs labelled with transition modes.

This definition is useful for the analysis of the Stop-and-Wait protocol (see section 6).

Finally we combine the two abstractions to obtain an abstract OG with respect to markings and transitions, AOG^{MT} .

Definition 18. *An abstract OG, with respect to markings and transitions, of a HLPN with an initial marking, M_0 , and a set of transition modes, TM , and given an injection, I , mapping markings to positive integers, is a labelled directed graph $AOG^{MT} = (V, T, A)$ where*

- $V = \{I(M) | M \in [M_0]\}$ is the set of nodes;
- T is the set of transitions of the HLPN; and
- $A = \{(I(M), t, I(M')) \in V \times T \times V | (t, m) \in TM \text{ and } M[(t, m))M']\}$ is the set of arcs labelled with transition names.

This last definition is useful when we are only interested in sequences of transitions, e.g., sequences of service primitive names.

FSA Associated with the OG

The next step is then defining the mapping from an abstract OG to its FSA. In our methodology, the FSAs are only used to determine language equivalence (or inclusion) and thus we are not concerned with the details of the markings, and hence we can use an abstract OG that does not include the marking details. Hence we just represent the nodes by positive integers.

When we construct the CPN model of the protocol, we normally do so with primitive events in mind, and thus label transitions in the CPN with service primitive names. However, in some cases (such as for the stop-and-wait protocol) it is convenient to have a more general mapping from transition modes to service primitives. (In the following, we restrict our attention to a mapping to service primitive names, as that is our current focus, but in general the mapping could be to service primitives in general, i.e. where service primitive parameters are included as well as the name.) We thus need a function that maps each transition mode in the abstract OG to either a service primitive name, or to an epsilon. Lets call this function *Prim* as it returns a primitive name (or epsilon). Thus we have

$$Prim : TM' \longrightarrow SP \cup \{\epsilon\}$$

where

- $TM' \subseteq TM$ is the set of transition modes used to label arcs in the abstract OG; and
- SP is the set of service primitive names in the system we are describing.

Given an abstract OG with respect to markings $AOG^M = (V, TM, A)$ (see Definition 17) we can formulate its corresponding FSA as

Definition 19. $FSA_{AOG^M} = (V, SP, A_{SP}, v_0, F)$ where

- V is the set of nodes of the abstract OG (the states of the FSA);
- SP is the set of service primitive names of the system of interest (the alphabet of the FSA);
- $A_{SP} = \{(v, Prim(tm), v') | (v, tm, v') \in A\}$ is the set of transitions labelled by service primitives or epsilons for internal events (the transition relation of the FSA);
- $v_0 = 1$ corresponds to the abstract initial marking (initial state of the FSA); and
- $F \subseteq V$ is the set of final states.

4 Stop-and-Wait Protocols

This part of the paper illustrates our approach to the verification of data transfer protocols by investigating the class of Stop-and-Wait protocols (SWP) [66, 73]. The work presented here is a major elaboration and extension of that recently published by the first two authors [12] and is based on [13].

We choose the SWP class because the protocol mechanisms are readily understood and because they are the simplest representative class of data transfer protocols since they include sequence numbers and retransmission counters. The class of SWP protocols is characterised by two parameters: the maximum sequence number and the maximum number of retransmissions.

Stop-and-Wait is an elementary form of flow control [66, 73] between a sender and a receiver. The sender stops after transmitting a message and waits until it receives an acknowledgement indicating that the receiver is ready to receive the next message. Stop-and-Wait Protocols often operate over noisy channels and combine flow control with error recovery using a timeout and retransmission scheme, known as Automatic Repeat ReQuest (ARQ) [73]. In this case, a checksum [73] is included to detect transmission errors. Messages that pass the checksum are acknowledged as received correctly. A message that fails the checksum is discarded by the receiver. In this case, the sender of the message will not receive an acknowledgement within its specified timeout period, and thus retransmits the message. This works well if the cause of not receiving the acknowledgement is due to the message being discarded (due to errors). However, the acknowledgement is also error protected by a checksum and it could have been discarded. In this case the retransmitted message is an unnecessary duplicate of the original message that has already been received correctly. To prevent duplicate messages being accepted as new messages a sequence number is appended to each message.

The class of SWPs are important because many practical data transfer protocols use *sliding window* mechanisms that have their foundations based on Stop-and-Wait principles. Sliding Window protocols [66, 73] improve the efficiency of SWPs by allowing many messages (rather than one) to be sent before requiring an acknowledgement. The number of messages that can be sent before the sender must stop and wait to receive an acknowledgement is known as the *window*. Cumulative acknowledgements and more sophisticated error retransmission schemes (such as Selective Reject) [66] can also improve efficiency. These schemes are used in many practical protocols such as TCP [62]. The underlying principles of ARQ used in sliding window protocols are the same as those used in SWPs, so that a window size of 1 corresponds to a Stop-and-Wait protocol. Thus it is essential that the stop-and-wait mechanisms work correctly if the more advanced protocols are also to be correct.

It is well known [73] that for sliding window protocols to work properly in detecting and discarding duplicates, the sequence number space needs to be one greater than the number of unacknowledged messages (the window). In the case of Stop-and-Wait protocols which have just one outstanding unacknowledged message, the sequence number space can be just two numbers, usually $\{0,1\}$.

When a SWP uses the sequence numbers $\{0,1\}$ it is called an Alternating Bit protocol (ABP) [6], because the sequence number can be implemented using just one bit in the header of the message, and the sequence number value alternates between 0 and 1. Acknowledgement messages in this case serve a dual purpose: that of flow control (indicating that the receiver is ready to receive another message) and transmission error recovery (informing the sender not to retransmit as the data has been received correctly). This is the simplest class of SWPs where the maximum sequence number is instantiated to 1. We consider Stop-and-Wait protocols with an arbitrary maximum sequence number as this takes us a step closer to sliding window protocols, where the window size is arbitrary, and hence the sequence number space (which must be at least one greater than the window size) is also arbitrary. It may also be the case that SWPs with larger sequence number spaces can work correctly over media with a limited amount of re-ordering (see [52]), but we do not consider this situation in this paper.

A number of papers, articles and books [1, 3, 6, 18, 30, 63, 68, 71–73, 76] have been written about the ABP. Many demonstrate that the ABP will work perfectly over an underlying medium that behaves in a FIFO (First-In First-Out) manner and that may also include loss. The ABP is often used as a case study when developing a new modelling language or a derivation from an existing modelling language, to demonstrate the use or effectiveness of the new language. This is the case in [68] where the ABP is used as an example to illustrate a new Timed Rewriting Logic (TRL) for capturing the static and dynamic aspects of SDL (Specification and Description Language) [45]. Another example of this is in [71] and [72] where the ABP is formally modelled and analysed using Temporal Petri nets (derivations of Petri nets with restrictions on the firing of transitions based on formulae containing temporal operators.) The ABP is used to illustrate modelling and analysis of protocols using Petri Nets in [30]. The Abracadabra Service and Protocol Example [76] describes a protocol using Alternating Bit sequence numbers, Retransmissions on timeout, Acknowledgements, Connection And Disconnection (ABRACAD), and is one of a graded set of examples used to provide guidelines for the application of three standardised formal description techniques, namely Estelle [21], LOTOS (Language Of Temporal Ordering Specifications) [20] and SDL [45]. Billington et al [18] use a variant of the ABP [22] to demonstrate a software tool. Reisig [63] develops the ABP in a series of steps as part of a case study on acknowledged messages, developed incrementally using simple Petri net models to illustrate the principles and operation of the ABP over FIFO (First-In First-Out) communication channels.

Some of the above papers demonstrate that the ABP will work as expected over FIFO channels that may also include loss. It appears, however, that the situation in which messages may be re-ordered by the medium has not been considered. The ABP was originally designed to provide a reliable data link service over an unreliable point-to-point physical link. In this situation overtaking of messages does not occur. However, the same ARQ mechanisms are used in transport level protocols, such as TCP [62], that operate over a medium that does not guarantee in-sequence delivery and may also lose messages [73]. It is therefore useful to investigate this situation for SWPs.

In Section 5 we present and explain our Coloured Petri Net (CPN) [48, 53] model of the SWP and discuss some of the modelling decisions made during its construction. The model is then analysed in Section 6 using a combination of hand proofs and language analysis. A discussion of the impact of the analysis results on the Transmission Control Protocol is given in Section 7, as well as the identification and discussion of a limitation of our approach. Finally some concluding remarks are presented in Section 8.

5 The Stop-and-Wait Protocol Model

We model a Stop-and-Wait protocol that includes error recovery using retransmissions operating over a lossy re-ordering medium. The CPN model of our SWP is given in Figs. 2 and 3. Figure 2 presents the graphical representation of the system, while Fig. 3 defines all the constants, sets and functions required and declares the types of the variables used in the annotations associated with the graphical representation. The software tool, Design/CPN [29], was used for the construction of the model. Design/CPN has four main facilities: an editor, a simulator, a state space tool and a performance tool. The simulation engine and state space tool are built using CPN ML [27], a variant of the functional programming language Standard ML of New Jersey (SML/NJ) [67]. CPN ML is used for the net annotations in Fig. 2 and the declarations in Fig. 3.

In Fig. 3 colour sets (types) are defined using the keyword `color` and enumerated types (`Sender`, `Seq`, `Retranscounter`) are created with the set constructor `with`. Variables are declared using the keyword `var` and are typed by a colour set, e.g. the variables `sn` and `rn` of type `Seq` (sequence number). Functions are defined using the keyword `fun` and values (e.g. constants) are defined using the keyword `val`.

We now describe the CPN model of the SWP in detail. The model comprises three main parts: the *Sender* (on the left), the *Receiver* (on the right) and an underlying bidirectional communication medium, *Network*, in the middle.

5.1 Sender

The sender consists of four places, four transitions and their interconnecting arcs. The places, `sender_ready` and `wait_ack`, represent the two states of the sender (either ready to send a new data message or awaiting an acknowledgement) and are typed by the colour set `Sender`, representing a single sender. The place, `sender_ready`, has an initial marking of one `s` token, indicating that the Sender is initially in the ready state. The `seq_no` place stores the sender sequence number, which is either the number of the message just sent (an unacknowledged message) or if acknowledged, the number of the next message to be sent. It is typed by the colour set `Seq` (sequence number) and has an initial marking of a single `0` token, indicating that the first message to be sent will have sequence number 0. The current number of retransmissions is recorded in place `retrans_counter`, typed by the colour set `RetransCounter` and is initially 0.

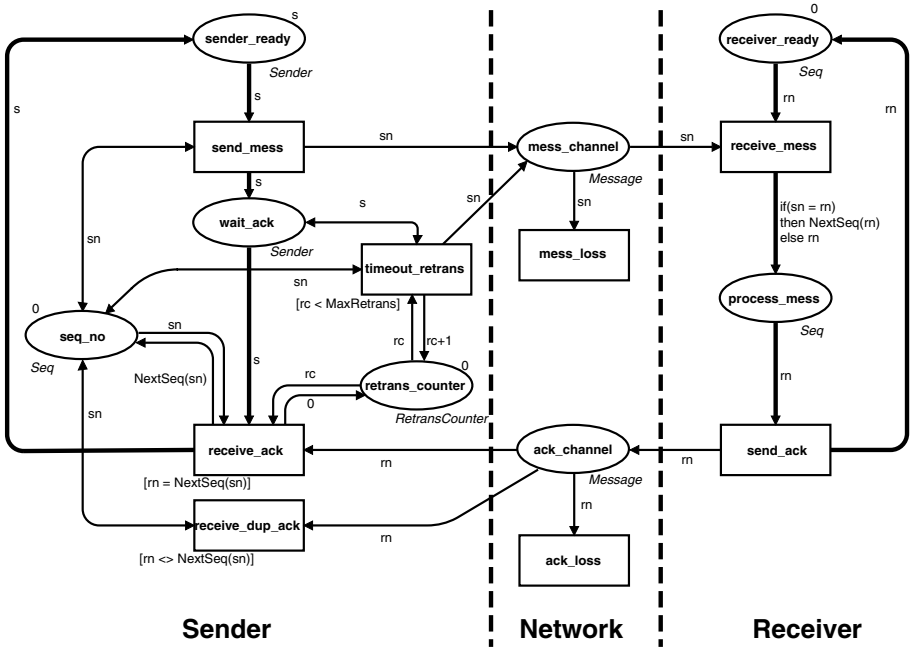


Fig. 2. The CPN of the Stop-and-Wait Protocol operating over a lossy re-ordering channel.

```

val MaxRetrans = 1;
val MaxSeqNo = 1;

color Sender = with s;
color Seq = int with 0..MaxSeqNo;
color RetransCounter = int with 0..MaxRetrans;
color Message = Seq;

var sn, rn : Seq;
var rc : RetransCounter;

fun NextSeq(n) = if (n = MaxSeqNo) then 0 else n+1;

```

Fig. 3. Global Declarations for the Stop-and-Wait Protocol CPN.

Transition `send_mess` models the sending of a message to the receiver. Message content is not represented as no protocol operations involve it (the protocol behaves the same way irrespective of content). The same is true for the addresses of sender and receiver, as we only have one of each in this model. Consequently, a message (or an acknowledgement) can be modelled by just its sequence number.

When the sender is ready `send_mess` may occur. It writes its sequence number (as the message) to the message channel and changes state to waiting for an acknowledgement.

The `timeout_retrans` transition models the expiry of the retransmission timer and the retransmission of the currently unacknowledged message. This transition can only occur if the sender is waiting for an acknowledgement and there have been less than `MaxRetrans` retransmissions of this message (see the guard). When `timeout_retrans` occurs, the retransmission counter is incremented by 1 and the retransmitted message is placed into the message channel.

Transition `receive_ack` models the receipt of expected acknowledgements from the receiver, i.e. those that acknowledge the currently outstanding message. Duplicate acknowledgements are received and discarded by transition `receive_dup_ack`. These may result from acknowledged retransmissions, where delay rather than loss was the cause of the retransmission. The complementary guards on these transitions identify the acknowledgement as being expected or a duplicate. An expected acknowledgement will have a sequence number one greater than the sender sequence number. The function `NextSeq` is used to increment the sequence number modulo (`MaxSeqNo` + 1), as shown in Fig. 3. An occurrence of `receive_ack` will remove the acknowledgement from the channel, return the sender to the ready state, reset the retransmission counter to 0 and increment the sequence number stored in `seq_no` using modulo arithmetic. The transition `receive_dup_ack` discards duplicate acknowledgements irrespective of the state of the sender.

5.2 Receiver

The receiver consists of two places and two transitions. The places `receiver_ready` and `process_mess` model the states of the receiver and are typed by the colour set `Seq`. A sequence number token present on one of these places indicates that the receiver is in that state (either ready to receive a message, or processing a message.) The `receiver_ready` place has an initial marking of one 0 token, indicating that initially the receiver is in the ready state and expecting a message with sequence number 0.

Transition `receive_mess` models the receipt of a message from the sender. The annotation on the arc from `receive_mess` to `process_mess` compares the sequence number of the message (`sn`) with the sequence number expected by the receiver (`rn`). If they match, then the message is the one expected (and is passed onto the user, a process that is not modelled) and the sequence number is incremented modulo (`MaxSeqNo` + 1) by the `NextSeq` function and placed in `process_mess`. If they don't match, a duplicate is detected (and discarded) and the receive sequence number is placed in `process_mess` unchanged. Transition `send_ack` occurs when the receiver has finished processing the message, indicating that enough buffer space is available to receive another message. This transition sends an acknowledgement containing the next sequence number expected by the receiver and returns the receiver to the ready state. Sending an acknowledgement when a duplicate message is received is necessary because if an acknowledgement of a

(new) message is lost and subsequent retransmissions of the same message are not acknowledged, the system will fail to progress as no acknowledgement will ever be delivered to the sender.

5.3 Underlying Medium

The underlying communication medium is modelled as a bidirectional channel consisting of one place and one transition for each direction of communication. The `mess_channel` place models the message channel while the `ack_channel` place models the acknowledgement channel. Both channel places are typed by the colour set `Message` (a sequence number, see Fig. 3) and are both initially empty. This models the overtaking behaviour of the communication medium. The two transitions `mess_loss` and `ack_loss` model the loss of messages and acknowledgements respectively. This corresponds to either loss in the network (due to congestion and buffer overflow in a router), or to discarding messages (and acknowledgements) due to checksum failures.

5.4 Discussion of Modelling Decisions

A straightforward way to begin modelling a system such as this is to use one place for each state of an entity, one place for each data item, and one transition for each action. This is evident in the Sender, where we have one place for each state (i.e. ready, waiting for an acknowledgement), one place for each item of data (i.e. sequence number, retransmission counter) and one transition for each action. A representation such as this gives a clearer visual indication of the control flow within the Sender than if the sender states were folded and represented by a single place typed by the set of states. However, this is normally only possible for protocols with very few states. As the number of states increases, so do the number of arcs, which leads to a visually complex diagram with many arc crossings, distracting from the major flows. This is alleviated to some extent by the use of thicker lines for arcs to emphasise major flows, as is illustrated for control flow in Fig. 2.

The receiver has been modelled using a different style, where there has been a folding of the receiver sequence number and receiver state. The transition `receive_mess` represents both the receipt of an expected message and the discarding of duplicate messages, requiring a complex arc annotation. This provides a more compact representation of the receiver clearly highlighting the control flow loop. This demonstrates the versatility of CPNs in being able to illustrate visually control flow and data flow. To do this well requires significant experience, especially for complex protocols that require a hierarchical approach. We used different styles for the sender and receiver to illustrate the different approaches. When modelling a complex protocol this would rarely be done, and a consistent style throughout the whole model is advocated.

We may want the model structure to reflect the structure of the real life system, given a certain amount of abstraction. For example, we have illustrated this by modelling the sequence number at the sender in the net structure as a

separate place. This is to reflect the fact that in an implementation, the sequence number as an entity of data may exist separately from, and regardless of, the state of the sender. It also simplifies modelling of the sender, because duplicate acknowledgement messages can be received and discarded regardless of the state of the sender. Conversely, the meaning of the value of the sequence number (either the next to be sent when in state `sender_ready`, or the message to be acknowledged when in state `wait_ack`) is dependent on the state, and hence this would favour folding the sequence number into the state places, as in the receiver. Modelling the loss of messages and acknowledgements by separate transitions (instead folding them into the receive transitions) allows for more flexibility in analysis, as to analyse a system without a lossy channel requires nothing more than adding a `[false]` guard to each loss transition. Thus various trade-offs present themselves to the modeller, even in models as simple as this.

6 SWP CPN Model Analysis

6.1 Properties of Interest

As described previously, with the SWP it is usual to place an upper bound (`MaxRetrans`) on the number of retransmissions that are allowed per message. When this limit is reached, the communication medium is considered to be down. In practice, an indication is given to a management entity that invokes a procedure to deal with the fault. This interaction (and procedure) is not modelled as it is not part of the SWP. In our model, the protocol will just terminate in a state where the retransmission counter has reached its maximum value (`MaxRetrans`). This is an expected terminal state, indicating that the network is down, and that the last message sent may have been lost.

Thus we are not particularly concerned with terminal states. Instead we focus on properties that are quintessential for correct operation of the SWP. The first concerns bounds on the channels, the second that duplicates are not accepted as new messages, the third that messages are not lost unknowingly and the fourth that the protocol conforms to the Stop-and-Wait service of alternating sends and (correct) receives, ensuring that messages are received in the same order as they were sent.

6.2 FIFO Channels

Our first step is to consider the SWP operating over communication channels that preserve sequence. This corresponds to the SWP operating over a physical link (as is the case for data link protocols) and is thus important in its own right. It is also important from the point of view of incremental analysis of the SWP operating over re-ordering channels. This is because most networks will provide a FIFO channel most of the time. Thus it is important to ensure that the SWP will operate correctly over FIFO channels, before we investigate the re-ordering case.

The CPN in Fig. 4 and associated declarations in Fig. 5 show our Stop-and-Wait protocol operating over a lossy FIFO channel. Places `mess_channel` and `ack_channel` are modified to operate as FIFO queues by altering their colour set from `Message` to `MessList` (a list of messages), giving them an initial marking of the empty list and modifying appropriate arc expressions on incoming and outgoing arcs to manipulate the list as a FIFO queue. All arcs placing a message into one of the channels do so by appending it to the *end* of the message list using the infix append operator (`^^`). All arcs removing a message from one of the channels do so by removing a message from the *beginning* of the list using the infix ‘cons’ operator (`::`). Loss in the medium is modelled by transitions `mess_loss` or `ack_loss`. This loss behaviour includes the discarding of corrupted messages (due to failing the checksum) and loss due to routers dropping packets (if applicable).

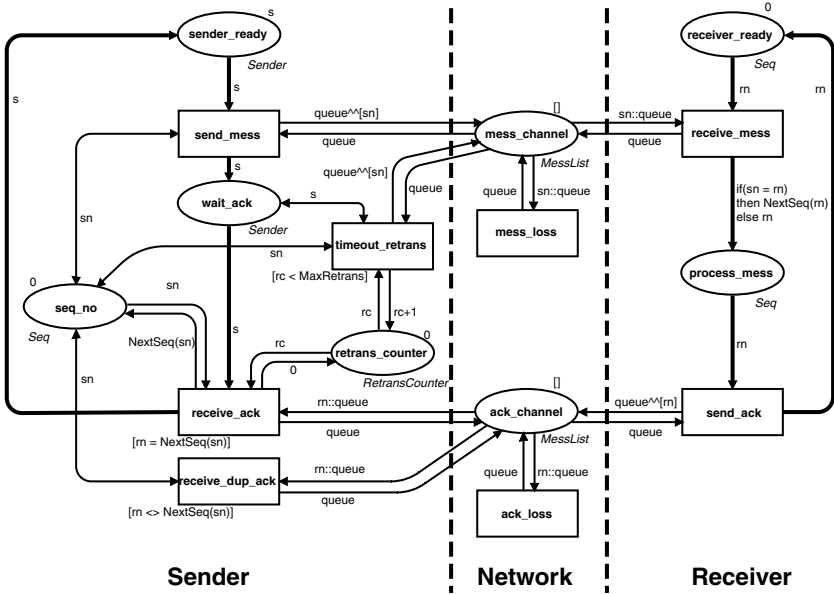


Fig. 4. A CPN of the Stop-and-Wait Protocol operating over an in-order medium.

We now consider the first property of interest, that of the bounds on the channels, and then investigate the other three properties (loss, duplication and SWP service).

Channel Bounds. In this section we state and prove two theorems regarding the maximum length of each of the message queues in places `mess_channel` and `ack_channel`.


```

val MaxRetrans = 1;
val MaxSeqNo = 1;

color Sender = with s;
color Seq = int with 0..MaxSeqNo;
color RetransCounter = int with 0..MaxRetrans;
color Message = Seq;
color MessList = list Message;

var sn, rn : Seq;
var rc : RetransCounter;
var queue : MessList;

fun NextSeq(n) = if(n = MaxSeqNo) then 0 else n+1;

```

Fig. 5. Declarations of the CPN shown in Fig. 4.

Theorem 1. *For the Stop-and-Wait CPN of Figs. 4 and 5 with $\text{MaxRetrans} \geq 0$ and $\text{MaxSeqNo} \geq 1$, the message queue length in place `mess_channel` is bounded by $(2\text{MaxRetrans} + 1)$, i.e. $\forall M \in [M_0], |queue| \leq (2\text{MaxRetrans} + 1)$ where $M(\text{mess_channel}) = 1'queue$, $queue \in \{0, \dots, \text{MaxSeqNo}\}^*$ and $|queue|$ is the length of the list 'queue'.*

Proof. We use the notation \oplus for addition modulo $(\text{MaxSeqNo} + 1)$ and 'n-message' as shorthand for 'a message with sequence number n'.

We firstly examine the case where $\text{MaxRetrans} = 0$. From the initial marking of the CPN as shown in Fig. 4, the only transition that can occur is `send_mess`, which inserts a 0-message into the message queue, so $|queue| = 1$. Transition `timeout_retrans` will never occur because of its guard. The only enabled transitions are `mess_loss` and `receive_mess`, both of which remove the 0-message, resulting in $|queue| = 0$. An occurrence of `mess_loss` leads to a dead marking and hence the theorem holds in this case. An occurrence of `receive_mess` with the variable bindings $sn = rn = 0$ indicates this is the message expected by the receiver. The receiver sequence number is incremented by the `NextSeq` function on the arc from `receive_mess` to `process_mess`. Now the only enabled transition is `send_ack`, inserting an acknowledgement ($rn = 1$) into the acknowledgement queue on place `ack_channel`. The only possible action now is to remove this acknowledgement from the queue, either through the occurrence of `ack_loss` (leading to a dead marking and hence the theorem holds) or through `receive_ack` as $rn = \text{NextSeq}(sn) = 1$. An occurrence of `receive_ack` will return the model to a state identical to the initial state except that all sequence numbers are now 1 instead of 0. The behaviour described above is the *only* behaviour of the system. This behaviour repeats, each time leading to the 'same' state except for the sequence numbers that have been incremented modulo MaxSeqNo . When the sequence number wraps back to zero, the CPN returns to the initial state. This demonstrates that for any $\text{MaxSeqNo} \geq 1$, $|queue| \leq 1$ for all markings when $\text{MaxRetrans} = 0$. This proves the theorem for $\text{MaxRetrans} = 0$.

For the more general case of $\text{MaxRetrans} \geq 1$, the situation is complicated by the possibility of duplicate messages and duplicate acknowledgements. Consider that we start with empty queues. The maximum number of messages with a given sequence number n that can be inserted into the message queue is the original (`send_mess` occurs) plus MaxRetrans duplicates (by MaxRetrans occurrences of `timeout_retrans`) giving $|queue| = (\text{MaxRetrans} + 1)$ (given the queue was empty). At this point the sender must stop and wait until it receives an acknowledgement (`receive_ack`) for the n -message. The minimum number of n -messages that need to be received and acknowledged (i.e. when no loss occurs) is one, leaving MaxRetrans n -messages in the message queue. When this acknowledgement is received, the retransmission counter is reset to zero and $(\text{MaxRetrans} + 1)$ $(n \oplus 1)$ -messages can be sent, giving a queue with MaxRetrans n -messages followed by $(\text{MaxRetrans} + 1)$ $(n \oplus 1)$ -messages and $|queue| = (2\text{MaxRetrans} + 1)$. Because of the FIFO property of the communication channels, the remaining MaxRetrans n -messages must be removed (by loss or receipt) before the first $(n \oplus 1)$ -message can be received and acknowledged allowing messages with sequence number $n \oplus 2$ to be placed in the message channel. Thus before any new message can be sent, the length of the queue can be no more than MaxRetrans . As already discussed, only $(\text{MaxRetrans} + 1)$ new messages can be added to the queue, giving a maximum queue length of $2\text{MaxRetrans} + 1$. \square

A similar theorem to that stated in Theorem 1 holds for the acknowledgement channel.

Theorem 2. *For the Stop-and-Wait CPN of Figs. 4 and 5 with $\text{MaxRetrans} \geq 0$ and $\text{MaxSeqNo} \geq 1$, the acknowledgement queue in place `ack_channel` is bounded by $(2\text{MaxRetrans} + 1)$, i.e. $\forall M \in [M_0], |queue| \leq (2\text{MaxRetrans} + 1)$ where $M(\text{ack_channel}) = 1'queue, queue \in \{0, \dots, \text{MaxSeqNo}\}^*$ and $|queue|$ is the length of the list 'queue'.*

Proof. From Theorem 1 at most $(2\text{MaxRetrans} + 1)$ messages can be in the message queue. Also, from the proof of Theorem 1 when there are $(2\text{MaxRetrans} + 1)$ messages in the message queue the acknowledgement queue is empty. We know that exactly one acknowledgement is generated for each message accepted by the receiver, by the occurrence of `receive_mess` followed by `send_ack`, which implies that a message is removed from `mess_channel` for every acknowledgement generated by the receiver. Further, from the proof of Theorem 1, the sum of messages and acknowledgements in the channel places can be no more than MaxRetrans before a new message can be sent. Thus although the removal of one acknowledgement can result in the addition of $(\text{MaxRetrans} + 1)$ messages, this can only happen when the sum of messages and acknowledgements in the channels is MaxRetrans . Thus (taking loss into account) the sum of the messages and acknowledgements in any marking must be $\leq (2\text{MaxRetrans} + 1)$. Therefore the maximum number of acknowledgements that can be in `ack_channel` is $(2\text{MaxRetrans} + 1)$ (when all the messages in `mess_channel` have been received and acknowledgements deposited in `ack_channel`). \square

Loss, Duplication and Stop-and-Wait Property

We firstly prove that the SWP satisfies the SW property for a range of parameter values and then consider whether or not loss and duplication can occur. We would like to prove the following theorem.

Theorem 3. *The Stop-and-Wait CPN of Figs. 4 and 5 where $\text{MaxRetrans} \geq 1$ and $\text{MaxSeqNo} \geq 1$, satisfies the Stop-and-Wait property.*

We use language analysis to prove this theorem for a significant range of values of MaxRetrans and MaxSeqNo .

For the SWP service, we define two primitives: a *send* at the sender entity interface; and a *receive* at the receiver entity interface. We can then define the *service language* as 0 or more repetitions of the sequence (send, receive). This can be represented by the regular expression $(\text{send receive})^*$ or by the Finite State Automaton (FSA) shown in Fig. 6.

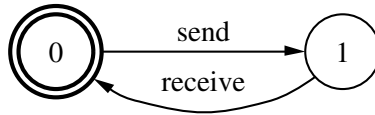


Fig. 6. FSA for the SWP service.

The next step is to generate the *protocol language* from the *protocol specification*. The protocol language just contains service primitive events. Our protocol specification is the CPN model of Figs. 4 and 5. In this CPN we can consider that the send primitive occurs when the `send_mess` transition occurs and that the receive primitive occurs when `receive_mess` occurs when the bindings of `sn` and `rn` are the same ($\text{sn} = \text{rn}$). (Otherwise the occurrence of `receive_mess` represents the discarding of duplicates, which does not correspond to a receive primitive.)

Following the verification methodology, the protocol language is obtained from the CPN's reachability graph by treating it as a FSA. All non-service primitive transitions (i.e. those associated with sending and receiving acknowledgements, with loss, retransmission or discarding duplicates) are replaced by empty (ϵ) transitions and the resulting FSA minimised [5] to produce the minimum deterministic FSA. This FSA represents all possible sequences of service primitives, generated from the protocol, and is thus the protocol language. We use the suite of tools available in the FSM package [33] for FSA minimisation and comparison.

Reachability graphs of the CPN defined in Fig. 4 and Fig. 5 were generated using Design/CPN [29] for both lossy and lossless media. (Loss is disabled by adding a guard of `[false]` to each of the two loss transitions.) We generated the OGs for a range of values of the parameters MaxRetrans and MaxSeqNo . Table 1 gives the statistics for some selected values of the parameters for lossy FIFO channels. Each OG is generated on a 2.4 Ghz PC with 1 GByte of memory.

Table 1. OG Results for SWP operating over lossy FIFO channels.

SeqNo. Bits	MaxSeqNo	MaxRetrans	Nodes	Arcs	Dead	Channel Bound	Time (hh:mm:ss)
1	1	0	12	12	4	1	00:00:00
1	1	1	80	194	4	3	00:00:00
1	1	2	264	834	4	5	00:00:00
1	1	3	640	2278	4	7	00:00:00
1	1	4	1300	4956	4	9	00:00:00
2	3	0	24	24	8	1	00:00:00
2	3	1	160	388	8	3	00:00:00
2	3	2	528	1668	8	5	00:00:00
2	3	3	1280	4556	8	7	00:00:00
2	3	4	2600	9912	8	9	00:00:00
9	511	0	3072	3072	1024	1	00:00:01
9	511	1	20480	49664	1024	3	00:00:29
9	511	2	67584	213504	1024	5	00:03:22
9	511	3	163840	583168	1024	7	00:16:34
9	511	4	332800	1268736	1024	9	00:55:32
10	1023	0	6144	6144	2048	1	00:00:04
10	1023	1	40960	99328	2048	3	00:01:32
10	1023	2	135168	427008	2048	5	00:11:49
10	1023	3	327680	1166336	2048	7	00:57:07
10	1023	4	665600	2537472	2048	9	04:02:14

The first two columns give the value of the number of bits required to encode the sequence number and the corresponding maximum sequence number. The next column records **MaxRetrans**. The next three columns list the numbers of nodes (markings), arcs, and dead markings in each OG, respectively. The second last column indicates the maximum number of messages in the message queue and the maximum number of acknowledgements in the acknowledgement queue, confirming Theorems 1 and 2. The last column records the time it took to generate the OG in hours, minutes and seconds.

The results indicate that the state space is linear in the size of the sequence number space ($\text{MaxSeqNo} + 1$) and the number of dead markings is given by $2(\text{MaxSeqNo} + 1)$. We expect the number of dead markings to be independent of the number of retransmissions, and for there to be a dead marking for each sequence number for two cases: firstly, when all messages are lost; and secondly, when all acknowledgements are lost.

Generating the reachability graph and answering our analysis questions is readily achieved with Design/CPN for small values of the **MaxSeqNo** and **MaxRetrans** parameters. We can obtain results for some practical values of sequence numbers. For example, the X.25 protocol allows the use of 3 bit, 7 bit and 15 bit sequence numbers. For **MaxSeqNo** = 127 (7 bit sequence numbers) and **MaxRetrans** = 3, we find the reachability graph contains 40960 states and takes 94 seconds to generate. Increasing **MaxSeqNo** to 1023 (10 bit sequence numbers)

and for `MaxRetrans` = 4 we find that there are 665600 reachable states, taking over 4 hours to generate. At around 1.5 million states the generation time becomes too slow to be feasible. (1.5 million states takes Design/CPN days to generate and will exhaust the memory on a PC with 1Gb of RAM.) Thus obtaining a result for 15 bit sequence numbers is problematic. Further, TCP, which uses 32 bit sequence numbers (`MaxSeqNo` = 4294967295), would require the generation of a reachability graph containing over 10^{12} states (for `MaxRetrans` = 3). Clearly this is not feasible with Design/CPN.

We generated a similar set of statistics for the case without loss. In this case there are no dead markings and the size of the state space is smaller but still is linear in the size of the sequence number space. For example, without loss, the reachability graph for `MaxRetrans`=1 and `MaxSeqNo`=1 comprises 48 nodes and 86 arcs, as opposed to 80 nodes and 194 arcs in the lossy case. This reachability graph is shown in Fig. 7.

We have abbreviated the names of transitions (S for send, R for receive and T for timeout), included the sequence number and indicated when a duplicate is received. We can see that the behaviour is quite complex even for this simple case. The usual behaviour when there are no retransmissions is given by the cycle of nodes 1,2,3,5,7,9,12,17,1. The rest of the graph depicts the behaviour when retransmissions occur, leading to the need to receive duplicate acknowledgements. It is worth noting that this graph is strongly connected (all markings are mutually reachable from each other).

The suite of tools available in the FSM package [33] was used for FSA generation and manipulation. A mapping was provided for the reachability graph to distinguish between the transition occurrences of interest and internal events (ϵ transitions). Final (halt) states were chosen to be those states in which the sender and receiver are both in their ready states with the same sequence numbers, as the protocol can terminate after sending an arbitrary number of messages, not necessarily a multiple of the modulo value. All the reachability graphs that were generated in both the lossy and lossless cases were then converted into a format understandable by the FSM tools. For the lossless FIFO medium, all the minimum deterministic FSA produced were identical to that shown in Fig. 6. Thus for the range of parameters tested (`MaxSeqNo` up to 1023 and `MaxRetrans` up to 4), the SWP operating over a lossless in-order medium is language equivalent to its service. For the lossy case, the FSA in Fig. 8 was produced for each combination of parameters tested. It shows that there can be sequences of alternating sends and receives that may end after either a send or a receive. This is expected, as the sequence may end after a send if a message and all its retransmissions are lost, ending in a dead marking, where the medium is declared down. (In this case the sender cannot tell if the last message was lost or successfully received, as it could have been that the acknowledgements were lost instead.) Given that this is inevitable for a finite number of retransmissions, we consider that the SWP satisfies the stop-and-wait property in this case.

We have thus shown that the SWP satisfies the stop-and-wait property for the values of the parameters tested. We also conjecture that this result implies

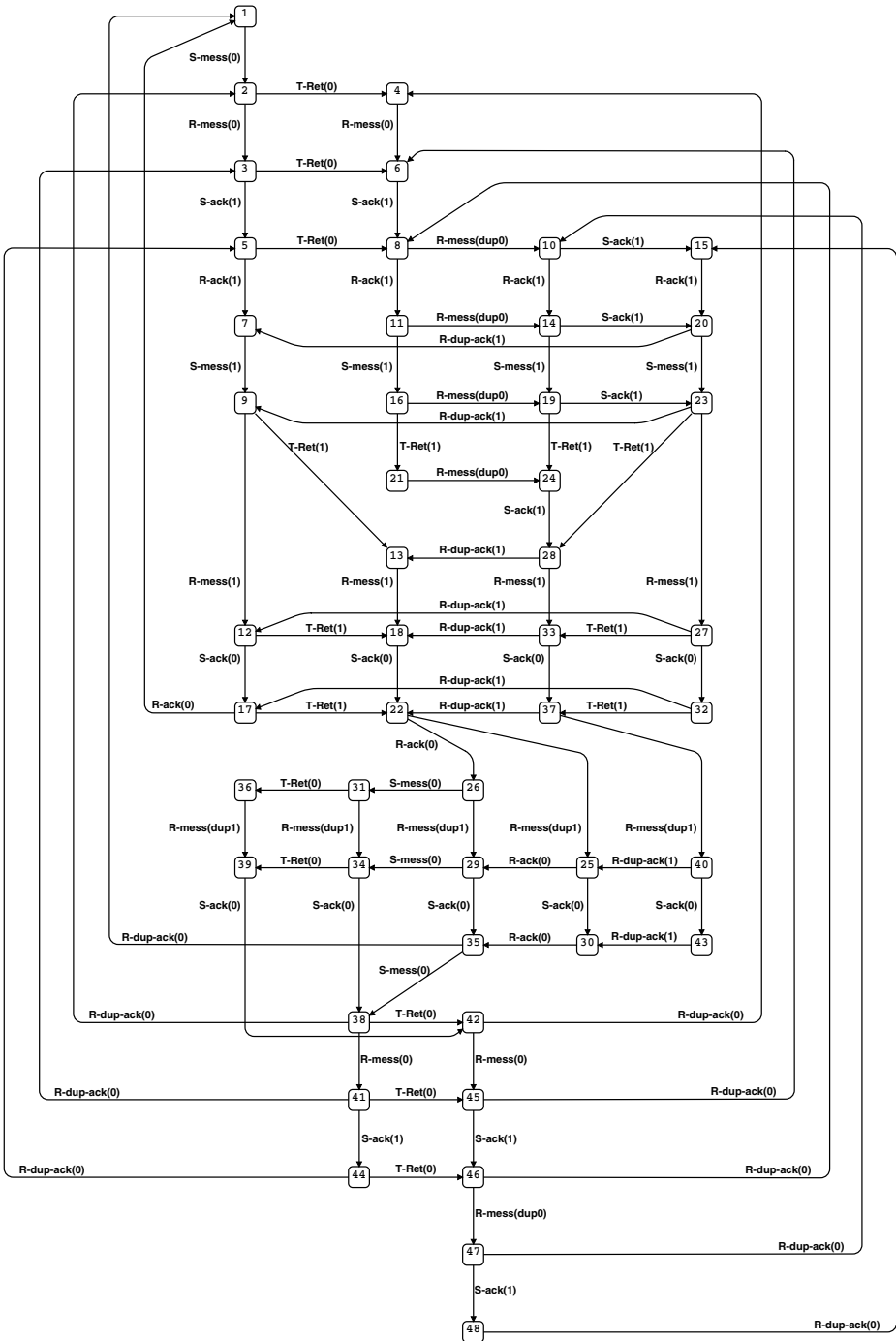


Fig. 7. OG of the Stop-and-Wait protocol, with `MaxRetrans=1`, `MaxSeqNo=1`, operating over a FIFO channel.

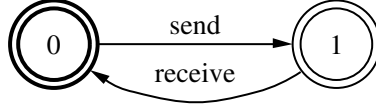


Fig. 8. A FSA showing sequences of send and receive primitives when the Stop-and-Wait protocol operates over a lossy FIFO channel.

that no duplicates are accepted as new messages by the receiver, and no messages are lost, except for possibly the last message in the case of a lossy medium as discussed above. We have further confidence that these conjectures are true because the Alternating Bit Protocol is widely accepted as being correct over lossy FIFO channels [2].

We thus conclude that the SWP operates as expected over lossy FIFO channels. If we allow messages in the channel to be re-ordered, will these properties still be satisfied? The next section shows they are not.

6.3 Re-ordering Channels

Channel Bounds. We wish to prove that the number of messages in the communication channel has the potential to grow without bound. This is formally captured in the following theorem.

Theorem 4. *For the Stop-and-Wait CPN of Figs. 2 and 3 where $\text{MaxRetrans} \geq 1$ and $\text{MaxSeqNo} \geq 1$, the message channel (place `mess_channel`) is unbounded.*

Proof. To prove this theorem, we show that a cycle of transition occurrences exists where the total effect of each cycle is to increase the number of messages in the message channel by one, and that this cycle can be repeated indefinitely. The following lemma is used in our proof.

Lemma 1. *Let σ_k be a finite occurrence sequence that can occur from a marking L , i.e. $L \xrightarrow{\sigma_k} L'$. If $L' \geq L$, then the occurrence sequence σ_k can be repeated indefinitely from marking L .*

Proof. Given $L \xrightarrow{\sigma_k} L'$ and $L' \geq L$, from Proposition 3 we know that σ_k can also occur from L' , i.e. $L' \xrightarrow{\sigma_k} L''$. From Proposition 2, we know that $L'' - L' = L' - L$ and because $L' \geq L$ we know that $L'' \geq L$. Thus from Proposition 3 we know that σ_k can occur from L'' . Thus by repeated application of Propositions 2 and 3, σ_k can repeat indefinitely from marking L . \square

All that is left to do to complete the proof of Theorem 4 is to identify a finite occurrence sequence σ_k of transitions in our CPN model, such that σ_k can be repeated indefinitely (i.e. $L \xrightarrow{\sigma_k} L'$ with $L' \geq L$), and that the total effect of an occurrence of the sequence σ_k is to increase the number of messages in the communication channel.

Consider our CPN model from Fig. 2 with declarations as shown in Fig. 3 but with $\text{MaxRetrans} \geq 1$ and $\text{MaxSeqNo} \geq 1$. From the initial marking, M_0 , only the

`send_mess` transition is enabled (with the variable `sn` bound to 0) indicating that the sender is ready to begin a transmission. The occurrence of this transition leads to a new marking M_1 in which there is a message (a '0' token) in the `mess_channel` place.

Transitions `timeout_retrans`, `mess_loss` and `receive_mess` are all enabled at marking M_1 . We note that the medium is lossy, but that this does not mean that messages *must* be lost. An occurrence of `receive_mess`, with variables `sn` and `rn` bound to 0, models the receipt of this message and leads to a marking M_2 . The arc inscription of the output arc from transition `receive_mess` to place `process_mess` determines that this message is not a duplicate (`sn = rn = 0`) and indicates this by placing a '1' token (through evaluation of `NextSend(0)`) into this place.

Transition `send_ack` is enabled in marking M_2 and when fired returns the receiver to the ready state and places an acknowledgement message into the acknowledgement channel (place `ack_channel`). This results in marking M_3 .

The `timeout_retrans` transition now occurs, with `rc` and `sn` bound to 0. The guard on `timeout_retrans` evaluates to true, because `MaxRetrans` ≥ 1 . This leads to a marking M_4 in which the retransmission counter has been incremented (a '1' token on `retrans_counter`) and a duplicate message '0' is in the message channel (place `mess_channel`).

Transition `receive_ack` rather than `receive_dup_ack` is enabled in M_4 , due to the complimentary guards, with a binding of `rn = 1`, `sn = 0` and `rc = 1`. Occurrence of `receive_ack` removes the acknowledgement message from the acknowledgement channel, returns the sender to the ready state, increments the sequence number (`NextSeq(0) = 1`), and resets the retransmission counter to 0. The resulting marking is M_5 , with

$$\begin{array}{ll} M_5(\text{sender_ready})=1's & M_5(\text{retrans_counter})=1'0 \\ M_5(\text{seq_no})=1'1 & M_5(\text{receiver_ready})=1'1 \\ M_5(\text{ack_channel})=\emptyset & M_5(\text{wait_ack})=\emptyset \\ M_5(\text{process_mess})=\emptyset & M_5(\text{mess_channel})=1'0 \end{array}$$

M_0 and M_5 are similar in many respects, but $M_5 \not\sim M_0$ as the sequence numbers stored at the sender and at the receiver have been incremented by one. Note that there is an additional message ('0') left in the message channel. Let us refer to the above sequence of transition occurrences as σ_0 where $\sigma_0 = \text{send_mess} \langle \text{sn}=0 \rangle, \text{receive_mess} \langle \text{rn}=0, \text{sn}=0 \rangle, \text{send_ack} \langle \text{rn}=1 \rangle, \text{timeout_retrans} \langle \text{rc}=0, \text{sn}=0 \rangle, \text{receive_ack} \langle \text{rc}=1, \text{rn}=1, \text{sn}=0 \rangle$ and $M_0 \xrightarrow{\sigma_0} M_5$. The binding of variables for each transition occurrence is written inside angular brackets.

For illustration purposes, we firstly consider alternating bit sequence numbers (`MaxSeqNo = 1`). Consider the sequence of transition modes σ_1 where $\sigma_1 = \text{send_mess} \langle \text{sn}=1 \rangle, \text{receive_mess} \langle \text{rn}=1, \text{sn}=1 \rangle, \text{send_ack} \langle \text{rn}=0 \rangle, \text{timeout_retrans} \langle \text{rc}=0, \text{sn}=1 \rangle, \text{receive_ack} \langle \text{rc}=1, \text{rn}=0, \text{sn}=1 \rangle$

σ_1 is very similar to σ_0 , with the exception of the bindings of `sn` and `rn`. In all instances, the values to which `sn` and `rn` are bound have been incremented, modulo (`MaxSeqNo+1`) (modulo 2 in this case). σ_1 can occur from M_5 , resulting in a marking M_{10} , with

$$\begin{array}{ll}
M_{10}(\text{sender_ready})=1's & M_{10}(\text{retrans_counter})=1'0 \\
M_{10}(\text{seq_no})=1'0 & M_{10}(\text{receiver_ready})=1'0 \\
M_{10}(\text{ack_channel})=\emptyset & M_{10}(\text{wait_ack})=\emptyset \\
M_{10}(\text{process_mess})=\emptyset & M_{10}(\text{mess_channel})=1'0 + 1'1
\end{array}$$

We note that $M_{10} = M_0 + \{((\text{mess_channel}, 0), 1), ((\text{mess_channel}, 1), 1)\}$. Thus M_{10} is a covering marking of M_0 , i.e. $M_{10} \geq M_0$. The sequence numbers at sender and receiver have wrapped back to their original value of 0 and M_{10} is identical to M_0 with the addition of the two extra messages in the message channel. According to the transition rule for HLPNs, additional tokens on a place will not disable any transitions that were previously enabled. From Lemma 1, the occurrence sequence $\sigma_0\sigma_1$ can repeat indefinitely, increasing the number of tokens in `mess_channel` by two each cycle. Thus `mess_channel` is unbounded and we have proved Theorem 4 for `MaxSeqNo` = 1.

Generalising for `MaxSeqNo` ≥ 1 , we have `MaxSeqNo`+1 sequence numbers and thus require $\sigma_0, \sigma_1, \dots, \sigma_{\text{MaxSeqNo}}$, defined in the same way as σ_0 and σ_1 above. For $0 \leq j \leq \text{MaxSeqNo}$, $\sigma_j = \text{send_mess } \langle \text{sn}=j \rangle, \text{receive_mess } \langle \text{rn}=\text{sn}=j \rangle, \text{send_ack } \langle \text{rn}=(j \oplus 1) \rangle, \text{timeout_retrans } \langle \text{rc}=0, \text{sn}=j \rangle, \text{receive_ack } \langle \text{rc}=1, \text{rn}=(j \oplus 1), \text{sn}=j \rangle$.

The occurrence of $\sigma_0\sigma_1 \dots \sigma_{\text{MaxSeqNo}}$ in marking M_0 leads to a marking M_m , where $m = 5\text{MaxSeqNo}$ and

$$\begin{array}{ll}
M_m(\text{sender_ready})=1's & M_m(\text{retrans_counter})=1'0 \\
M_m(\text{seq_no})=1'0 & M_m(\text{receiver_ready})=1'0 \\
M_m(\text{ack_channel})=\emptyset & M_m(\text{wait_ack})=\emptyset \\
M_m(\text{process_mess})=\emptyset & \\
M_m(\text{mess_channel})=1'0 + 1'1 + \dots + 1'\text{MaxSeqNo}
\end{array}$$

M_m covers marking M_0 so that $\sigma_0\sigma_1 \dots \sigma_{\text{MaxSeqNo}}$ can repeat indefinitely from marking M_0 , resulting in `MaxSeqNo` additional messages in the message channel for each repetition. Thus the message channel is unbounded. \square

A similar theorem holds for the acknowledgement channel.

Theorem 5. *For the Stop-and-Wait CPN of Figs. 2 and 3 where `MaxRetrans` ≥ 1 and `MaxSeqNo` ≥ 1 , the acknowledgement channel (place `ack_channel`) is unbounded.*

Proof. The proof is similar to that of Theorem 4, hence we just provide a sketch. Consider the transition sequence `send_mess`, `receive_mess` $\langle \text{rn}=\text{sn} \rangle$, `send_ack`, `timeout_retrans`, `receive_ack`, `receive_mess` $\langle \text{sn} \neq \text{rn} \rangle$ and `send_ack`. (Binding elements have been omitted where they are not important.) Transition occurrence sequences $\sigma_0, \sigma_1, \dots, \sigma_{\text{MaxSeqNo}}$ are defined in a similar way. The occurrence sequence $\sigma_0, \sigma_1, \dots, \sigma_{\text{MaxSeqNo}}$ can be repeated indefinitely from M_0 , resulting in `MaxSeqNo` additional acknowledgements in the acknowledgement channel for each repetition. \square

Loss, Duplication and Stop-and-Wait Property. As previously discussed, when the Stop-and-Wait protocol operates as required, one message will be received correctly at the receiver for every original message sent by the sender. It turns out that this is not always the case for the Stop-and-Wait protocol operating over a medium that reorders messages. This demonstrates that the protocol does not satisfy the Stop-and-Wait service. Further we can show that sequences of sends and receives exist where there are more receives than sends, indicating that duplicates are accepted. Finally we can also demonstrate that there are sequences in which there are more sends than receives, indicating that messages can be lost.

We summarise these results in the following theorems.

Theorem 6. *The Stop-and-Wait CPN of Figs. 2 and 3 where $\text{MaxRetrans} \geq 1$ and $\text{MaxSeqNo} \geq 1$, does not satisfy the Stop-and-Wait service.*

Theorem 7. *For the Stop-and-Wait CPN of Figs. 2 and 3 where $\text{MaxRetrans} \geq 1$ and $\text{MaxSeqNo} \geq 1$, the receiver may incorrectly accept duplicate messages as new messages.*

Theorem 8. *For the Stop-and-Wait CPN of Figs. 2 and 3 where $\text{MaxRetrans} \geq 1$ and $\text{MaxSeqNo} \geq 1$, messages can be lost without the sender or receiver being aware of it.*

Proof. We use language analysis to prove the above theorems. Due to the unbounded communication channels in our original model shown in Fig. 2, the resulting reachability graph is infinite. However, to prove our theorems, we only need to demonstrate that it is possible for the system to malfunction. We therefore limit the capacity of the communication channels to two. The rationale behind this is that if the protocol operates incorrectly with a channel capacity of two messages, the same incorrect behaviour will also be present in a channel with capacity greater than two. Capacities of 0 and 1 are not appropriate, as a capacity of 0 results in no communication and a capacity of 1 prohibits overtaking. Thus a capacity of two is the minimum needed to show interesting behaviour.

To obtain the smallest reachability graph of interest, we also set $\text{MaxRetrans} = 1$ and $\text{MaxSeqNo} = 1$. We argue that if incorrect behaviour is evident when $\text{MaxRetrans} = 1$ then the same behaviour can occur for $\text{MaxRetrans} \geq 1$ (as it includes $\text{MaxRetrans} = 1$) and similarly for MaxSeqNo (as sequence numbers always wrap, but the sequences illustrating the incorrect behaviour will be longer).

Channel capacity has been implemented as shown in Fig. 9 with declarations shown in Fig. 10. The initial marking of the `mess_channel` and `ack_channel` places has been modified so that each place contains a certain number of `empty` tokens, in this case two each, representing empty buffers. Each time a message is placed in the channel, an `empty` token must be removed, and whenever a message is removed, an `empty` token must be put back. This is shown on the arc expressions connecting the `mess_channel` and `ack_channel` places to the surrounding transitions.

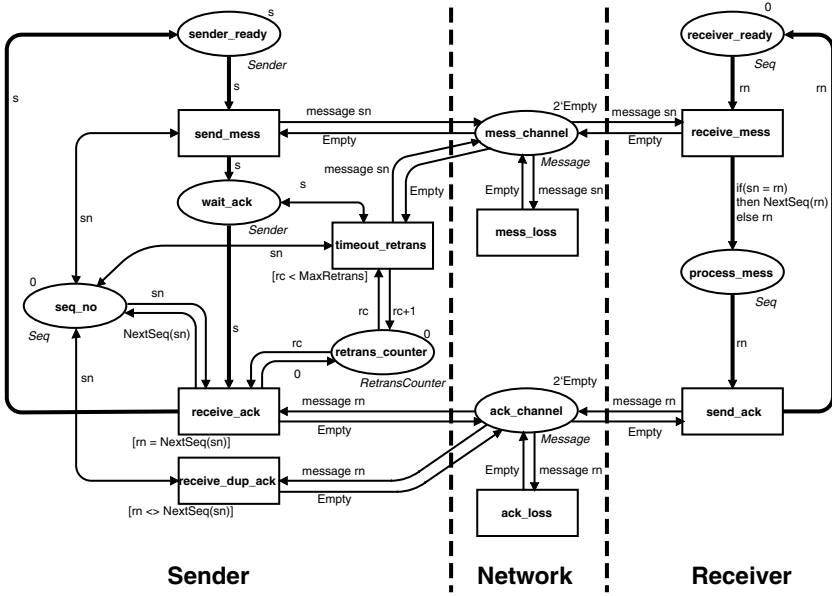


Fig. 9. A CPN of the Stop-and-Wait Protocol operating over a reordering medium with finite capacity.

```

val MaxRetrans = 1;
val MaxSeqNo = 1;

color Sender = with s;
color Seq = int with 0..MaxSeqNo;
color RetransCounter = int with 0..MaxRetrans;
color Message = union message : Seq + Empty;

var sn, rn : Seq;
var rc : RetransCounter;

fun NextSeq(n) = if(n = MaxSeqNo) then 0 else n+1;

```

Fig. 10. Declarations of the CPN shown in Fig. 9.

Design/CPN [29] was used to generate the reachability graph of this CPN (Fig. 9) for the configuration shown in Fig. 10, without loss in the channel. The reachability graph contains 410 nodes and 848 arcs. After interpreting this as a FSA, the FSM package was used to obtain the equivalent minimum deterministic FSA as shown in Fig. 11. We have replaced `send` with `s` and `receive` with `r` in the figure due to size constraints.

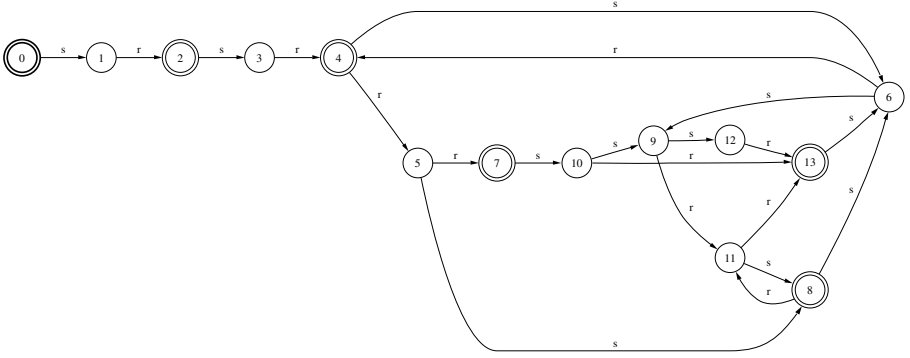


Fig. 11. FSA showing erroneous sequences of send and receive primitives for the Stop-and-Wait protocol operating over a lossless reordering medium.

This FSA shows that the SWP operating over a medium that reorders messages does not satisfy its service. For example, we see that the sequence $4 \xrightarrow{r} 5 \xrightarrow{s} 8 \xrightarrow{s} 6 \xrightarrow{r} 4$ violates the Stop-and-Wait service of alternating send and receive events. There are other more interesting sequences also. There are incorrect sequences of send and receive primitives, indicating that the receiver can mistakenly accept duplicate messages as new messages. For example, the cycle $7 \xrightarrow{s} 10 \xrightarrow{r} 13 \xrightarrow{s} 6 \xrightarrow{r} 4 \xrightarrow{r} 5 \xrightarrow{r} 7$ shows that it is possible for the system to enter a loop where the receiver accepts four messages as legitimate messages for every two sent by the sender. Another such loop is $5 \xrightarrow{s} 8 \xrightarrow{r} 11 \xrightarrow{r} 13 \xrightarrow{s} 6 \xrightarrow{r} 4 \xrightarrow{r} 5$. To illustrate how duplication can happen in the protocol, we have included a protocol trace corresponding to the initial sequence $0 \xrightarrow{s} 1 \xrightarrow{r} 2 \xrightarrow{s} 3 \xrightarrow{r} 4 \xrightarrow{r} 5$ as shown in Fig. 12.

In Fig. 12, event 1 corresponds to sending a message (send primitive) with sequence number 0 (mess(0)), which is received (event 2: receive) and acknowledged (event 3) by sending ack(1). The timer then expires at the sender, and mess(0) is retransmitted (event 4) giving rise to a duplicate (mess(0)[dup]) which is delayed in the medium. The sender then receives ack(1) (event 5) and sends out its next message, mess(1), at event 6 (send). At this stage there are two messages in the channel. Because the retransmitted mess(0) is delayed, it is overtaken by mess(1) which is expected and received normally by the receiver (event 7: receive) who acknowledges it with ack(0) at event 8. At this point, the primitive events have been as expected: send, receive, send, receive. Next the sender retransmits mess(1) (mess(1)[dup]) at event 9 (not relevant to this discussion). Then at event 10 (receive), the receiver is expecting a mess(0) and receives it. However, it is a duplicate of the first message, and not a new message. The receiver wrongly interprets it as a new message and a receive primitive occurs, giving the sequence send, receive, send, receive, receive.

We now consider a third cycle in Fig. 11, given by $13 \xrightarrow{s} 6 \xrightarrow{s} 9 \xrightarrow{s} 12 \xrightarrow{r} 13$. This cycle shows that for every 3 messages sent, only one is received, demon-

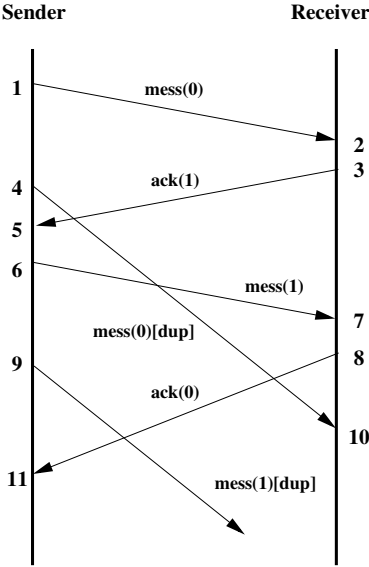


Fig. 12. Time Sequence diagram showing the acceptance of a duplicate for the Stop-and-Wait protocol operating over a lossless reordering medium.

strating message loss even though there is no loss in the medium! We illustrate this behaviour with the protocol trace shown in Fig. 13.

The sequence starts as expected with the first two messages (`mess(0)` and `mess(1)`) sent (events 1 and 8) and received (2 and 9) correctly. Retransmissions occur for both `mess(0)` (event 4) and `mess(1)` (event 11), but these are correctly discarded as duplicates (events 6 and 13). However, they give rise to duplicate acknowledgements (events 7 and 14) which the sender incorrectly interprets as acknowledgements (events 16 and 18) for the new messages sent (events 15 and 17). For example, this is due to `ack(0)` overtaking `ack(1)` and `mess(0)` being sent before `ack(1)` arrives. Now `mess(1)` (the fourth message sent) overtakes `mess(0)` (the third message sent) and is misinterpreted by the receiver as a duplicate and discarded (event 19) because the receiver is expecting `mess(0)`. An acknowledgement (`ack(0)`) is thus sent (event 20) indicating that the receiver is expecting a message with a sequence number 0. Meanwhile, the sender has received duplicate `ack(0)` (event 18) which it interprets as a good acknowledgement for the fourth message (which is discarded by the receiver, as already discussed) and transmits (event 21) the fifth message (`mess(0)`). This gives us the primitive sequence: send, receive, send, receive, send, send, send. The receiver now receives the third message (`mess(0)`) correctly (event 22) and acknowledges it (event 23). The receiver is thus expecting to receive a message with sequence number 1, but receives the new fifth message (event 25) and discards it as a duplicate. However, the sender now receives `ack(1)` (event 24) and believes that the fifth message has been received correctly. This sequence demonstrates how two mes-

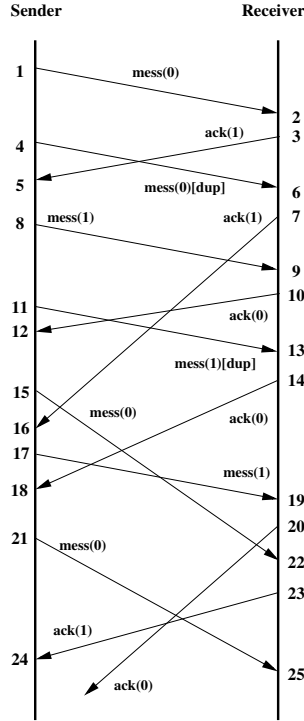


Fig. 13. Time Sequence diagram showing how loss can occur for the Stop-and-Wait protocol operating over a lossless reordering medium.

sages (the fourth and fifth) can be lost due to incorrect protocol mechanisms, while both the sender and receiver believe that there is no problem. Note that this sequence only requires two retransmissions to occur.

It is interesting to note that problems with acceptance of incorrect messages do not occur until the sequence numbers wrap, i.e. at node 4 in Fig. 11.

We also considered the case when the channel was lossy (in addition to re-ordering). The same parameter settings were used. The reachability graph contained 624 nodes and 2484 arcs. The reduced FSA showing the protocol language for this configuration contains 29 nodes and 47 arcs, and is shown in Fig. 14. There are many incorrect sequences in this language also. \square

7 Discussion

7.1 Practical Relevance

In order to understand the relevance of these results to practical protocols, let us consider the error recovery and flow control strategies implemented in TCP [62]. TCP uses retransmission on timeout to recover from packet loss and a sliding

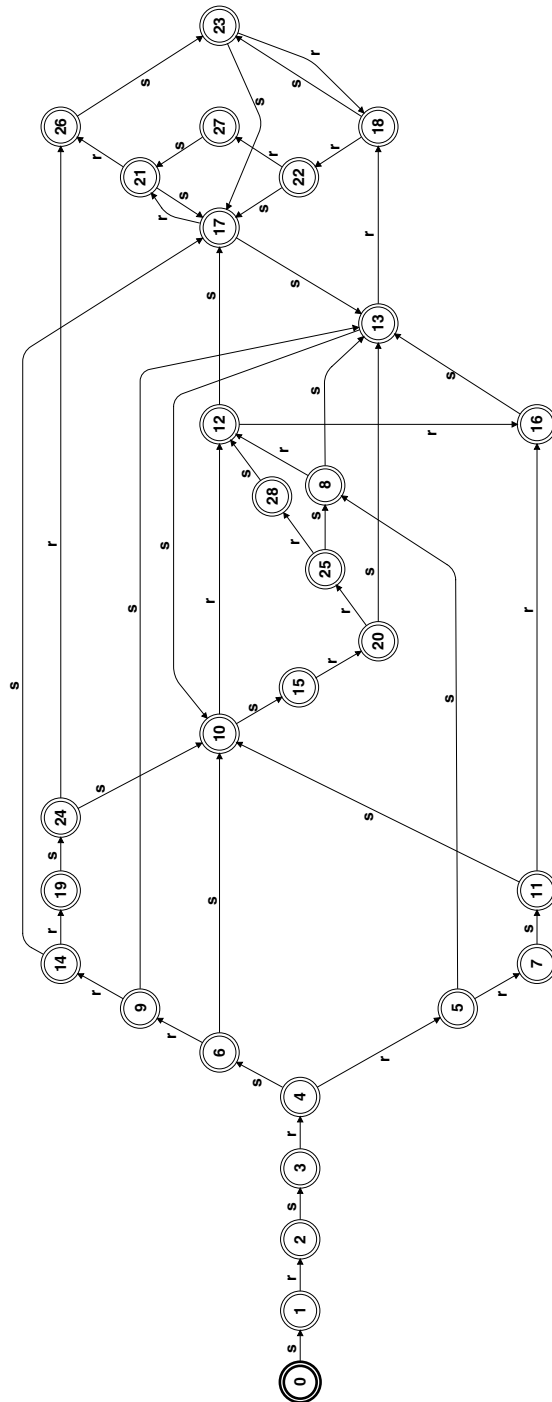


Fig. 14. FSA showing erroneous sequences of send and receive primitives for the Stop-and-Wait protocol operating over a lossy reordering medium.

window mechanism for flow control, which includes dynamic window changes. TCP operates over IP (the Internet Protocol), which allows packets (known as segments) to be dropped or reordered. The correctness of TCP's data transfer procedures can thus be related to the correctness of the Stop-and-Wait protocol operating over a medium that allows reordering.

It is necessary to distinguish between 'old' duplicates, those left from a previous connection, and duplicates caused by retransmissions within a connection. TCP uses a 32 bit sequence number, giving $2^{32} = 4294967296$ sequence numbers. Each sequence number is associated with 1 byte of data. Apart from unboundedness, the problems associated with the Stop-and-Wait protocol only arise after sequence numbers wrap, so that delayed duplicates can disrupt the acknowledgement mechanism, resulting in loss of messages or mistaken acceptance of duplicates as new messages. This will only happen in TCP after 4Gbytes of data have been transmitted and duplicates still remain in the network.

The threat posed by old duplicates was recognised by the designers of the Internet. They introduced the concept of a life-time for a packet in the IP layer, known as *time-to-live*. (This is implemented as a 'hop count' in practice.) The idea is that duplicate packets left floating around a network will be discarded once their time-to-live expires. TCP also implements a 3-way handshake for connection establishment, such that the starting sequence number for a connection can be chosen carefully for each new connection. In this way, (time-to-live combined with the 3-way handshake) TCP tried to avoid the problem caused by old duplicate packets being accepted due to wrapping sequence numbers.

RFC (Request for Comment) 793 [62], the protocol specification for TCP maintained by the Internet Engineering Task Force (IETF) [42], states that the maximum lifetime for a segment of data (MSL) is two minutes. Thus there will not be a problem with duplicate packets if they are destroyed before sequence numbers can wrap. Every byte is given a sequence number, thus for a transmission rate of 1 megabit per second (125 kBytes/sec) and ideal conditions for data transfer, the sequence number space will be exhausted in approximately $2^{32}/(1.25 * 10^5) \approx 9.5$ hours. Clearly this is not a problem. For a transmission rate of 100 megabits/sec (12.5 megabytes/sec) we see that the sequence numbers will wrap after $2^{32}/(1.25 * 10^7) \approx 5$ minutes and 45 seconds. This is getting close to the maximum packet lifetime, but should not pose a problem unless the hop count mechanism takes longer than 2 minutes to quash packets. With the introduction of Gigabit networks [73] the sequence numbers of TCP could wrap after only 34 seconds of data transfer at 1 gigabit/second. Although the maximum throughput of a network rarely approaches the theoretical maximum, it would not be unreasonable to assume that with a very large window size and very large data transfers, wrapping of sequence numbers would occur after about one minute, allowing for the possibility of duplicates being in the channel at the same time as new packets with the same sequence numbers.

This is the condition necessary for packet loss and the acceptance of duplicates (as a new packet) to occur. However, to get duplicates, there must be retransmissions caused by additional delay due to network congestion or lack

of responsiveness in the receiver (e.g. an overloaded web server) which will reduce throughput. This delay, however, does not need to be very great to cause retransmissions, and hence the effect on throughput may not be significant. Another factor limiting throughput is TCP's window size and the round trip delay (RTD). In standard TCP implementations, the maximum window size is 2^{16} bytes, limiting the throughput to $(2^{16}/\text{RTD})$ bytes/sec. The speed of light propagation delay contribution to RTD will then provide a limit irrespective of the transmission speed. However, to allow users to take advantage of high-speed networks, RFC 1323 [43] proposes to increase the maximum window size to 2^{30} bytes or 1Gbyte, in which case the speed of light delays are no longer a limiting factor.

It is unlikely that duplicates are a problem for TCP with the current speed of networks, however these problems may become more probable if network speed were to increase by another order of magnitude, i.e. 10 gigabit/second. There are additional ramifications to be considered if incorrect acceptance of duplicates or loss of data becomes a problem. For safety critical applications operating over the Internet the consequences could be catastrophic.

There are a number of suggested ways in which this problem could be solved, or at least alleviated. RFC 1323 [43] specifies a number of TCP extensions for high performance. The extension for a larger window size has already been mentioned. Another extension is Protect Against Wrapped Sequence Numbers (PAWS) which proposes a solution to wrapping sequence numbers within a connection, by including a 32 bit time-stamp in every segment. Another solution involves extending the sequence number space, to 2^{64} , i.e. 64 bit sequence numbers. Even at 10 gigabit/second, a 64 bit sequence number field would take 470 years to wrap. The procedure of sequence numbering may also be reviewed, as currently every byte is given a sequence number. Providing a sequence number for every packet would extend the usefulness of the existing 32-bit sequence numbers.

How likely is it that unbounded growth of messages in the communication channels will actually occur? The unbounded growth is caused by retransmissions due to delayed acknowledgements. Given the variability of the round trip delay (due to the unpredictability of network congestion or overloaded servers) it is not uncommon for these delays to occur. This is countered to some extent by TCP measuring round trip delay and setting its retransmission timeout period accordingly. However, due to transients, unnecessary retransmissions will always occur. The unbounded growth, however, only occurs because the duplicates are not received by the receiver. This is highly unlikely. Also those that are delayed in the network will be expunged after their time-to-live limit has expired. Thus TCP already has mechanisms in place to prevent unbounded growth. TCP has also developed sophisticated techniques to cope with network congestion [73], so we don't see that our unboundedness result for re-ordering media will cause major difficulties with protocols such as TCP. Nonetheless, as network speeds increase the problem will get worse, particularly if the time to live value is maintained at 2 minutes. In general we can say that the contribution to congestion

over FIFO channels is well contained (determined by the maximum number of re-transmissions allowed) whereas over channels that re-order messages, it requires other mechanisms to contain congestion.

One avenue of further study is to generalise these results for other flow control mechanisms, such as the Sliding Window mechanism used in TCP and other protocols.

7.2 Shortcomings of Our Approach

The language analysis performed in Section 6.3 uncovered a number of errors, but was not able to detect all errors. We discovered many scenarios in which the Stop-and-Wait protocol, operating over a reordering channel (bounded or unbounded), can generate sequences of alternating send and receive events in which duplicate data is accepted and messages can be lost.

The following sequence of events is just one of many in which the above two problems are evident. This particular sequence was chosen because it is one of the shortest. Starting from the initial state of the CPN as depicted in Figs. 9 and 10, it illustrates the possibility of acceptance of duplicate data and message loss. This scenario represents a loop in the reachability graph of the model and exists independently of the boundedness and lossy properties of the channel. In the following event sequence, we use ‘message n’ as shorthand for ‘the message with sequence number n’. The corresponding transition and binding of variables is written after each action.

1. Send message 0 (`send_mess <sn=0>`)
2. Receive message 0 (`receive_mess <rn=0, sn=0>`)
3. Send acknowledgement for message 0 (`send_ack <rn=1>`)
4. Timeout and retransmit message 0 (`timeout_retrans <rc=0, sn=0>`)
- A duplicate message 0 remains in the channel.
5. Receive the ack of message 0 (`receive_ack <rc=1, rn=1, sn=0>`)
6. Send message 1 (`send_mess <sn=1>`)
7. Receive message 1 (`receive_mess <rn=1, sn=1>`)
- Message 1 has overtaken message 0.
8. Send acknowledgement for message 1 (`send_ack <rn=0>`)
9. Receive the ack of message 1 (`receive_ack <rc=0, rn=0, sn=1>`)
10. Send message 0 (`send_mess <sn=0>`)
- Now there are two message 0’s in the channel, a new message 0 and the duplicate of the previous message 0 from line 4.
11. Receive message 0 (`receive_mess <rn=0, sn=0>`)
- Duplicate data accepted (unless the new message 0 overtakes the old message 0.) The receiver believes this to be the correct (new) message 0.
12. Send acknowledgement for message 0 (`send_ack <rn=1>`)
13. Receive and discard a duplicate message 0 (`receive_mess <rn=1, sn=0>`)
- Loss of the data in the new message 0 (unless overtaking occurred in step 11, in which case we are discarding the duplicate message 0 from step 4.)
14. Send a duplicate acknowledgement of message 0 (`send_ack <rn=1>`)

15. Receive the ack of message 0 (`receive_ack <rc=0, rn=1, sn=0>`)
16. Receive the duplicate ack of message 0 (`receive_dup_ack <rn=1, sn=1>`)
17. Send message 1 (`send_mess <sn=1>`)
18. Receive message 1 (`receive_mess <rn=1, sn=1>`)
19. Send acknowledgement of message 1 (`send_ack <rn=0>`)
20. Receive ack of message 1 (`receive_ack <rc=0, rn=0, sn=1>`)
21. Repeat from the beginning.

The problem arises in steps 11 and 13 of the above sequence. Because we do not include message data in our model, we can not determine which message is being received at step 11 (the original message from step 10 or the duplicate from step 4) and which is being discarded at step 13. Note that the global sequence of alternating send and receive events, as defined in our service, still holds.

Language equivalence is sufficient to prove or disprove that the correct *sequences* of events were occurring in the protocol, as defined in our service specification. From it we were able to detect numerous errors relating to the sequences of events, and to infer from those incorrect sequences that loss or duplication was occurring. However, given our existing service and protocol specifications, we were unable to detect the incorrect data acceptance and loss problems identified above, when the event sequences were as expected. The fault lies not with language analysis itself but with what we are applying language analysis to. The service specification is incomplete. Indeed, the service only specifies that there must be alternating send and receive events, i.e. that every send event (for a new message) is followed by a receive event (for what the receiver believes is the same new message). It abstracts from the data that is sent and received. What appears to be a reasonable abstraction (that is, the data to be sent is not required as it does not affect the operation of the protocol) is adequate for proving or disproving properties such as deadlocks and livelocks, and for determining the sequences of events, but says nothing about the data delivered on the occurrence of such events.

The assumption we made in defining the service is related to the notion of *data independence* [54, 64, 65, 77, 82]. Conceptually, a system is said to be data independent if the operation of the system is independent of the specific data it is operating on. Sabnani [65] uses data independence principles to define and verify properties about the Alternating Bit Protocol operating over a link-layer channel with a capacity of one. For example, to prove that data is passed up to the receiving user in the same order in which it is supplied by the sending user, both Wolper [82] and Sabnani [65] tell us that we need a minimum of three distinct data values. Incorporating these ideas into our service and protocol specifications may provide a solution.

Another avenue for investigation is the work presented in [52]. Knuth essentially derives design rules for appropriate bounds on sequence numbers for data transfer protocols operating over FIFO channels. He then provides a generalisation of these rules for channels that are basically FIFO in nature but may exhibit limited (bounded) reordering. Investigating the derivation of these rules may provide insight into our use of sequence numbers in the analysis of the protocol specification.

8 Concluding Remarks on the Stop-and-Wait Protocol

The class of Stop-and-Wait protocols (SWPs) include message acknowledgements and retransmission on time-out procedures to recover from transmission errors or from dropped packets in a communication medium such as the Internet. They form the basis of the data transfer procedures for both data link layer protocols and transport level protocols. The retransmission procedure can result in duplicate messages due to acknowledgements being lost or delayed. To detect duplicates a SWP inserts sequence numbers into messages and keeps track of the sequence numbers at both ends. However, sequence numbers need to be from a finite sequence number space and the number of times that a message can be retransmitted is also limited. We thus characterise SWPs using two parameters: the maximum sequence number (MaxSeqNo) and the maximum number of retransmissions (MaxRetrans). We demonstrate how a simple CPN model can be built that is parameterised by MaxSeqNo and MaxRetrans and discuss some of the modelling decisions. The first model operates over a lossy re-ordering medium. We also show how the CPN model can be modified to operate over (lossy) FIFO channels (applicable to data link protocols) and that this is an important starting point for analysing the re-ordering case. We consider four properties that we believe are important for stop-and-wait protocols: the bound on the channels; (unknowing) loss of messages; acceptance of duplicates as new messages; and the stop-and-wait property of alternating sends and receives. In the case of lossy FIFO channels, we manually prove that the communication channels are bounded by one more than twice the maximum number of retransmissions ($2\text{MaxRetrans} + 1$). We believe this is a new result. This illustrates an approach to proving properties of protocols for arbitrary parameter values using manual proofs. Using the protocol verification methodology, i.e., by generating the occurrence graph of the protocol and using automata reduction, we also show that the stop-and-wait property holds for small values of the parameters and conjecture that this implies that no loss or duplication occurs.

Protocols (such as TCP) operating over the Internet Protocol have to contend not only with loss due to transmission errors and packets dropped at routers, but also with the possibility that the order of packets is not maintained. Since TCP can behave as a Stop-and-Wait protocol under certain conditions it is interesting to investigate the behaviour of SWPs over a reordering medium. We analysed our CPN model with re-ordering channels and proved that: the communication channels are unbounded; messages can be lost, although the sender believes they have been confirmed by the receiver; duplicates can be accepted as new messages by the receiver; and that the SWP does not satisfy its service of alternating sends and receives. We provided a manual proof that the channels were unbounded so long as both parameters were positive giving a general result. The last 3 properties were obtained using our automatic verification method for the case when the channel capacity was 2 and MaxRetrans and MaxSeqNo were both 1. We then argued that this would also imply that these error conditions would occur for any channel capacity greater than one, and any positive values of the parameters. We also noted that the first result is independent of sequence number

wrap, while the last three results depend on sequence numbers wrapping before the problems occur.

We discuss the practical relevance of these results to TCP. We conclude that sequence number wrap is possible in Gigabit networks, particularly if the extended window size option is used. RFC 1323 discusses this problem and suggests a mechanism (PAWS) using 32 bit time stamps to reject old duplicates, which hopefully will eliminate the problems associated with sequence number wrap. The problem with unbounded channels is not serious, but could add to congestion problems as the speed of networks increases. Our discussion is at a high-level and does not investigate in detail TCP's procedures for data transfer, nor the suggested PAWS scheme.

Our language analysis results allow us to detect incorrect sequences of events in our protocol specification and to deduce that loss and duplication are occurring. However, we illustrate that our data abstraction assumption (that data is not required as it does not affect the operation of the protocol) prevents us from detecting data loss and duplication in situations where the sequences of events are correct (i.e. correspond to the SWP service). Thus, with the data abstraction used, language analysis does not provide a method for verifying correct operation of the protocol in terms of absence of loss and duplication. To solve this problem we plan to apply data independence principles and techniques in order to define service and protocol specifications that capture the required information.

9 Transmission Control Protocol

The purpose of this part of the paper is to provide an example of an application of the methodology to an important complex protocol of the Internet, the Transmission Control Protocol (TCP). Our concern here will be to illustrate the first steps in modelling and analysing a complex protocol in the hope that this experience will help others to tackle other complex protocols. We concentrate on the connection management aspects of the protocol (especially establishment), rather than data transfer, which has already been discussed in detail for the Stop-and-Wait protocol.

TCP is specified in Internet Request For Comments (RFC) number 793 [62]. Its goal is to establish, maintain and close point-to-point connections between host computers attached to the Internet. Its main purpose is the reliable transfer of data between host computers. It also provides facilities for many connections to be running simultaneously to support multiple Internet application sessions such as those related to the World Wide Web and Email.

9.1 TCP Messages

In order to establish and release connections and to transfer data, RFC 793 defines a set of messages that are exchanged between the two computers. A TCP message, known as a segment, comprises a header field and a data field that carries application data. The TCP header field provides control information

for handling multiple connections, their management (the opening and closing of connections) and reliable data transfer including end-to-end flow control. The format of a TCP segment is given in Fig. 15.

Source Port							Destination Port						
Sequence Number													
Acknowledgment Number													
Data Offset	Reserved	U	A	P	R	S	F	Window					
		R	C	S	S	Y	I						
		G	K	H	T	N	N						
Checksum							Urgent Pointer						
Options										Padding			
Data													

Fig. 15. TCP Segment Format.

The 16 bit source and destination port fields are used to identify the application that is going to use the connection and allow multiple connections to be running simultaneously. On a particular connection, a sequence number is associated with every octet of data that is to be sent from one host computer to another. When transmitting data, the 32 bit sequence number field specifies the sequence number of the first data octet in the segment. The 32 bit acknowledgment number field indicates the successful receipt of data octets and contains the next sequence number of the data octet that the sender of the acknowledgement segment is expecting to receive. The four bit data offset field contains the header length (in 32 bit words). The next 6 bits are reserved, then there are a set of 6 control bits that are vitally important for TCP connection management. We describe them in detail in the next paragraph. A 16 bit window field is used for flow control. It signals to the receiver of the segment the number of data octets that the sender of the segment is prepared to receive, beginning with the acknowledgement number in the segment.

As already mentioned the header contains six 1-bit control flags: URG (urgent), ACK (acknowledgement), PSH (push), RST (reset), SYN (synchronisation) and FIN (finish). The URG flag if set, is used in conjunction with an urgent pointer field, to indicate to the receiver the position of data in the octet stream that needs priority when being delivered to the user. A segment with the ACK flag set indicates that the acknowledgement number field is valid. A set PSH flag indicates to the receiving TCP process that all queued data, including that just received, must be immediately delivered to the user. When set, the RST flag informs the receiver of the segment to reset the connection. This normally

results in the connection being aborted. A TCP connection is initiated by setting the SYN bit in a segment. The SYN carries an initial sequence number which indicates that the first octet of data to be sent will carry the next sequence number (initial sequence number plus one). The initial sequence number is chosen according to the value of a clock that the host runs, instead of always being set to zero. This is to reduce the probability of delayed old duplicate SYNs interfering with the connection. Finally, a segment with the FIN bit set indicates that the sender of the segment has no more data to send. It is used to gracefully close the connection (i.e. without data loss). The sequence number of the FIN is that of the last data octet sent plus one. A TCP segment is usually named after the control bits that are set. For example, a SYNACK segment refers to the segment which has both the SYN and ACK bits set.

The remaining header fields comprise a 16 bit checksum used to detect transmission errors, a 16 bit urgent pointer required for urgent data (already discussed above) and an options field (allowing, for example, a maximum segment size to be indicated in a SYN segment).

9.2 TCP Connection Management Procedures

A TCP state diagram [26, 62, 69] is included in the RFC to illustrate TCP connection management procedures. It defines TCP's 11 states and a core set of state changes related to processing user calls and connection management segments. It is incomplete in that it does not include TCP's state variables nor does it incorporate reset processing and error handling. A much more comprehensive pseudo-code like description of TCP's procedures is given in Section 3.9 of RFC 793. In this section we just illustrate the procedures using message sequence diagrams.

Figure 16 (a) is a message sequence diagram for normal connection setup and tear down. On the left is the client (the initiator of the connection) and on the right is the server. Time progresses down the page. The client's states (e.g. CLOSED, SYN_SENT, ESTABLISHED) are written to the left of the vertical line representing the client. A similar convention is adopted for the server side. User commands (i.e. *active open*, *passive open*, and *close*) are written in parentheses on top of some states, indicating when they occur. TCP uses a "three-way handshake" [75] to establish a connection, i.e., three segments are used by the two communicating hosts to open the connection. In Fig. 16, the sequence number and the acknowledgement number (when relevant) are included with the segment name. The procedure is initiated by the TCP entity on one host (client), and responded to by the TCP entity on the other (server). In Fig. 16 (a), after receiving an active open command from its user, the TCP client sends out a SYN segment with a sequence number ISS1, its initial sequence number for the connection. The client also changes state from CLOSED to SYN_SENT. The TCP server enters LISTEN after receiving a passive open command from its user. To respond to the SYN from the client, it sends out a SYNACK segment with an acknowledgement number ISS1+1 as well as a sequence number ISS2, the server's initial sequence number for the connection. After sending the SYNACK segment, the TCP server

changes state from LISTEN to SYN_RCVD. To respond to the SYNACK, the TCP client sends an ACK with sequence number $ISS1+1$ and acknowledgement number $ISS2+1$ and changes state from SYN_SENT to ESTABLISHED. After receiving the ACK, the server goes into ESTABLISHED from SYN_RCVD. The connection is now set up between the client and the server.

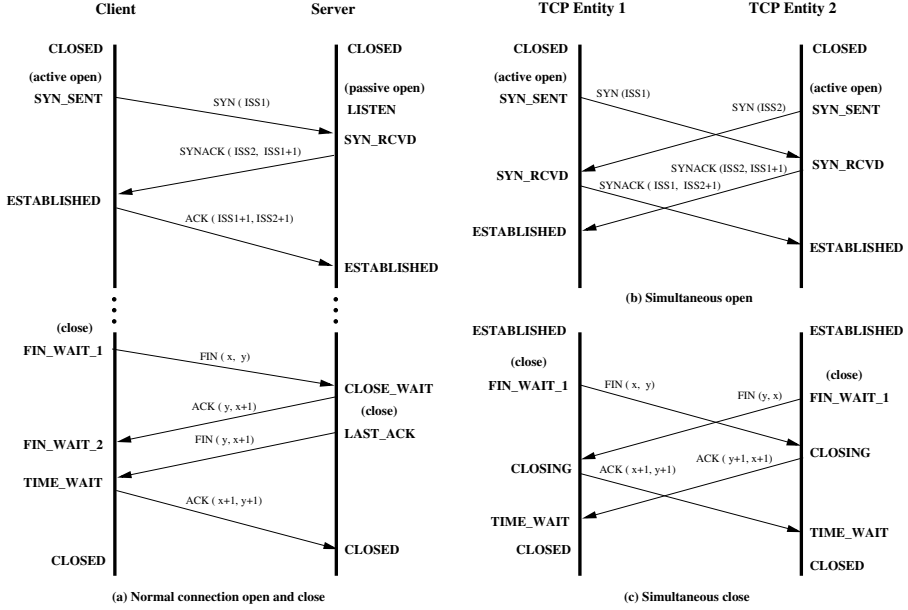


Fig. 16. Message sequences for TCP connection management.

Graceful connection termination is illustrated at the bottom of Figure 16 (a). Firstly, the TCP client user indicates that it has no further data to send by issuing a close command. The TCP client sends a FIN segment to the server and enters the FIN_WAIT_1 state. On receipt of the FIN, the server acknowledges it and informs its user that the client is closing the connection. When the server user has no more data to send, it issues a close command to the server TCP entity, which sends a FIN to the client, closing the connection from the server to the client. Finally the FIN is acknowledged by the client. The sequence number (x) and acknowledgement number (y) used in this message sequence diagram assume that the server had no more data to send.

The TCP connection management protocol also incorporates procedures for simultaneously opening and simultaneously closing connections by both TCP entities. Figs. 16 (b) and (c) show the sequences respectively. For more information about them, see [62, 69].

As well as the major states defined in the state diagram, TCP state variables are used to maintain the state of a TCP connection and are stored in a

record called the *Transmission Control Block* (TCB). Important TCP variables for connection management are: `SND_UNA`, `SND_NXT`, `ISS` and `RCV_NXT`. `SND_UNA` records the oldest sequence number of a segment that has been sent but has yet to be acknowledged. `SND_NXT` stores the sequence number of the next segment to be sent. `ISS` represents the initial send sequence number of the initiating TCP entity, while `RCV_NXT` stores the sequence number of the next expected incoming segment.

10 CPN Model of TCP Connection Management

In this section we explore building a CPN model of TCP's connection management procedures. We start by discussing the assumptions and abstractions used.

10.1 Modelling Assumptions and Abstractions

We make five assumptions when modelling TCP connection management.

1. The communication channel does not lose, corrupt or duplicate packets, but may delay and re-order packets.

The reason for starting from a non-lossy channel is that a lossy channel may hide possible deadlocks in the protocol, such as unspecified receptions which the channel can conveniently lose, but mostly will not! Thus this anomaly would be missed when inspecting the leaf nodes of the reachability graph if arbitrary channel loss is included, but nonetheless it would be a problem. Excluding loss initially allows these imperfections to be detected by inspecting dead markings.

2. There is no retransmission.

We would like to ensure that the procedures work without the complication of retransmissions, which we know will cause state space explosion. Thus, when no loss is involved, retransmissions are not necessary for the procedures to operate correctly. Later, we shall need to investigate the effect of loss and the use of retransmissions, once the basic behaviour is confirmed to be correct. This assumption allows us to ignore the retransmission procedures for SYN and FIN segments.

3. We consider a single instance of a TCP connection.

Because all instances of a connection operate the same way, we can just consider one instance. It is only when we wish to consider contention for resources (such as buffer space in a host) between a number of running connections, that we need to consider multiple connections. This is ruled out of the scope of the initial analysis of TCP. This assumption allows us to ignore the source and destination port numbers.

4. The receive buffer is big enough to store all the incoming segments.

Since we are only considering connection management, and not data transfer, it is reasonable to assume that the receiver will be able to store the connection management segments. This allows us ignore the flow control window and the calculations and comparisons based on it. This simplifies the model and thus reduces the size of the state space.

5. The user issues four commands to the TCP entity: active open, passive open, send and close.

The TCP user interface also allows for three other calls: abort, receive and status. We do not model the abort call at this stage, as we wish to investigate TCP's basic behaviour of establishing and gracefully releasing connections, before including arbitrary aborts. Once we know the core behaviour is satisfactory, then we can start to investigate the rarer and more complex behaviour that includes user aborts. The receive and status calls do not affect the operation of the protocol and can be considered as local interface matters. They are also not modelled.

The TCP RFC says little about feedback to the user (such as receipt of data, or indications that the connection has been requested or is established) and thus at this stage of the investigation, we do not consider it. It is however of vital importance with respect to whether or not TCP satisfies its service, where we must be explicit about such interactions. Since RFC 793 does not define a service, we firstly investigate the operation of the protocol without it. Once more experience is gained, then we are in a much better position to define the service [15–17].

We model TCP segments at the level of detail needed for analysing the connection management procedures, given the above assumptions. A TCP segment is thus modelled by including: the sequence number, the acknowledgement number and four control flags: SYN, ACK, RST and FIN. Other fields in the TCP header can be omitted because they do not affect the operation of the connection management procedure. For example, we do not need to model the checksum as discussed in Section 2.2, and the flags PSH and URG, the urgent pointer field and the window are only concerned with the data transfer procedures. We also do not consider options.

10.2 Architecture

Our CPN model comprises 6 places and 87 transitions. It is organised into three hierarchical levels, as shown in Fig. 17.

The first level has one page called TCP_Overview. The second level also has one page, named TCP_Entity. Since TCP is symmetrical (both ends implement the same procedures) we only need to define the procedures once, and then instantiate them for each end, using *page instances* [53]. The third level has eleven pages, one for each TCP state. This is standard practice in many protocol definitions and has been used in the modelling of other communication protocols,

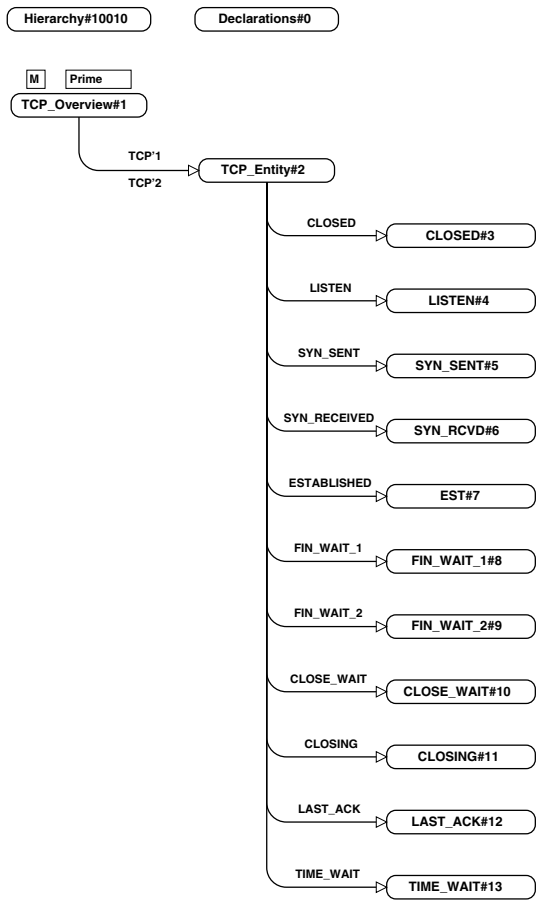


Fig. 17. Hierarchy page for the CPN model.

for example, [34]. This approach is also consistent with the way TCP is specified in Section 3.9 of RFC 793. (It turns out that other structures are possible which take advantage of common procedures in different states ([37]). However, the use of a state-based approach provides a clean and readily followed structure when first starting to model protocols, especially if they include state diagrams or state tables in their definitions. Once this experience has been obtained, further important optimisations of the structure can be undertaken.)

10.3 Declarations for the TCP Model

The declarations shown in Fig. 18 define the colour sets and any associated variables for User Commands, TCP segments and the TCB and include values for the initial sequence numbers.

```

1 (* User Commands *)
2 color COMMAND = with A_Open | P_Open | Send | Close;
3
4 (* TCP Segments *)
5 color Int = int;
6 color CTLbit = with SYN | RST | ACK | FIN;
7 color ACKflag = with on | off;
8 color SEG_CTL = product CTLbit*ACKflag;
9 color SEG = record
10     SEQ: Int *
11     ACK: Int *
12     CTL: SEG_CTL;
13 var seg: SEG;
14
15 (* Transmission Control Block *)
16 color STATE = with CLOSED | LISTEN | SYN_SENT | SYN_RCVD | EST |
17     CLOSE_WAIT | LAST_ACK | FIN_W1 | FIN_W2 | CLOSING | TIME_WAIT;
18 color SV = record
19     RCV_NXT: Int *
20     SND_NXT: Int *
21     SND_UNA: Int *
22     ISS: Int;
23 var v: SV;
24 color LISTENstat = with lis | cls;
25 var i: LISTENstat;
26 color TCB = product STATE*SV*LISTENstat;
27
28 (* ISS *)
29 val ISS_tcp1 = 10;
30 val ISS_tcp2 = 20;

```

Fig. 18. Declarations for TCP user commands, segments and TCB.

The colour set `COMMAND` (line 2) defines four commands: `A_Open` (active open); `P_Open` (passive open); `Send`; and `Close`, that are issued by users. The colour set `SEG` (lines 9–12) defines a TCP segment as a record with three entries: `SEQ`, `ACK` and `CTL`. `SEQ` is used to model the sequence number and `ACK`, the acknowledgement number. Both numbers are defined as integers. Note that the actual sequence number and acknowledgement number space is finite, ranging from 0 to $2^{32} - 1$. Because we are only dealing with connection management and not data transfer, only a very small portion of the sequence number space is used for establishing and releasing a connection. Thus we can also assume that the sequence numbers do not wrap (i.e. cycle back to zero), so that modulo arithmetic is not required. (Note that including modulo arithmetic is essential when modelling the data transfer procedures.)

`CTL` (line 12), typed by the colour set `SEG_CTL`, is used to model TCP's control flags. `SEG_CTL` (line 8) is a product of `CTLbit` (line 6) and `ACKflag`

(line 7) that model the four segment types and the ACK flag respectively. The ACKflag indicates the status (on or off) of the Acknowledgement Number field. If the ACK flag is on it indicates that the Acknowledgement field is valid. Variable seg (line 13), the variable for TCP segments, is declared as type SEG. An example of a SYNACK using ML syntax is $\{SEQ = 20, ACK = 11, CTL = (SYN, on)\}$.

The Transmission Control Block (lines 15–30) defines at line 26 a colour set, TCB, as a product of STATE (line 16), SV (lines 18–22) and LISTENstat (line 24). STATE comprises the 11 TCP states. SV defines the four TCP variables explained in Section 9.2: RCV_NXT, SND_NXT, SND_UNA and ISS. We use integers for state variable values (instead of 32 bit integers) for the same reason as given above for sequence numbers. LISTENstat is a Boolean used to keep track of whether or not the TCP entity has previously been in the LISTEN state. If it has, variable i, typed by LISTENstat (line 25), will take the value lis, otherwise it will take the value cls, indicating that the entity has not previously been in LISTEN but CLOSED. This is used to determine the next state that TCP enters from states SYN_SENT and SYN_RCVD on receipt of a RESET segment (see RFC 793, Section 3.4). Variable v (line 23), the variable for the record of TCP state variables, can take any value belonging to SV, such as $\{RCV_NXT = 21, SND_NXT = 11, SND_UNA = 11, ISS = 10\}$. ISS_tcp1 and ISS_tcp2 (lines 29–30) define the initial send sequence numbers for each TCP entity.

We also define six functions that create TCP segments, depending on the value of state variables or incoming segments, as shown in Fig. 19. All the functions are similar, so we just illustrate in detail the creation of a segment by using the function SYNseg (lines 32–35) that creates a SYN segment, as an example. The initial sequence number, stored in TCB as the fourth component of the state variable record and specified in ML as $\#ISS(v)$, is assigned to the sequence number field of the SYN segment. By convention, the acknowledgement number field is assigned the value 0 (for null), because the SYN segment is used to initiate a connection and therefore cannot carry a valid acknowledgement number. In the SYN segment, the control bit SYN is on and ACK is off (indicating that the acknowledgement number is not valid), so entry CTL of the segment record is assigned (SYN,off). Other TCP segments are modelled in a similar way. Note that functions RSTackon and RSTackoff take incoming segments as their argument rather than TCB's state variables. They are used to model the RST segment with ACK on and off respectively.

10.4 The Top and Second Level Pages

The top level page is shown in Fig. 20, which provides an abstract view of the protocol.

There are 6 places in Fig. 20. Places User_1 and User_2, typed by COMMAND, model TCP user commands. Changing the initial marking of a user place will change the command issued to TCP, resulting in modelling different cases. Places TCB_1 and TCB_2, typed by TCB, model TCP state information. A token in either place represents a local TCP (compound) state. Places H1_H2

```

31 (* Functions for TCP Segments *)
32 fun SYNseg(v: SV): SEG =
33   {SEQ = #ISS(v),
34    ACK = 0,
35    CTL = (SYN,off)};
36
37 fun SYNACKseg(v: SV): SEG =
38   {SEQ = #ISS(v),
39    ACK = #RCV_NXT(v),
40    CTL = (SYN,on)};
41
42 fun ACKseg(v: SV): SEG =
43   {SEQ = #SND_NXT(v),
44    ACK = #RCV_NXT(v),
45    CTL = (ACK,on)};
46
47 fun FINseg(v: SV): SEG =
48   {SEQ = #SND_NXT(v),
49    ACK = #RCV_NXT(v),
50    CTL = (FIN,on)};
51
52 fun RSTackon(seg: SEG): SEG =
53   {SEQ = 0,
54    ACK = #SEQ(seg)+1,
55    CTL = (RST,on)};
56
57 fun RSTackoff(seg: SEG): SEG =
58   {SEQ = #ACK(seg),
59    ACK = 0,
60    CTL = (RST,off)};

```

Fig. 19. Functions for creating TCP Segments.

and H2_H1, typed by SEG, each model a unidirectional communication channel. H1_H2 indicates the transmission direction is from host 1 to host 2, whereas H2_H1 indicates the opposite direction. A token in the communication channel place represents that a segment is in transit from one TCP entity to its peer TCP entity, and may be anywhere in the network or in a buffer of either local entity.

Also in Fig. 20, are two *substitution transitions* named TCP'1 and TCP'2. Each represents a TCP connection management process that implements both the establishment and termination procedures. A substitution transition may be viewed as a macro that is linked with another CPN page (known as a *subpage*) that is called when the CPN executes. TCP'1 and TCP'2 are both linked with the second level page (Fig. 21), which serves as a page instance.

The places associated with a substitution transition need to be assigned to places on the subpage. For example, places User_1 and User_2 in Fig. 20 are both

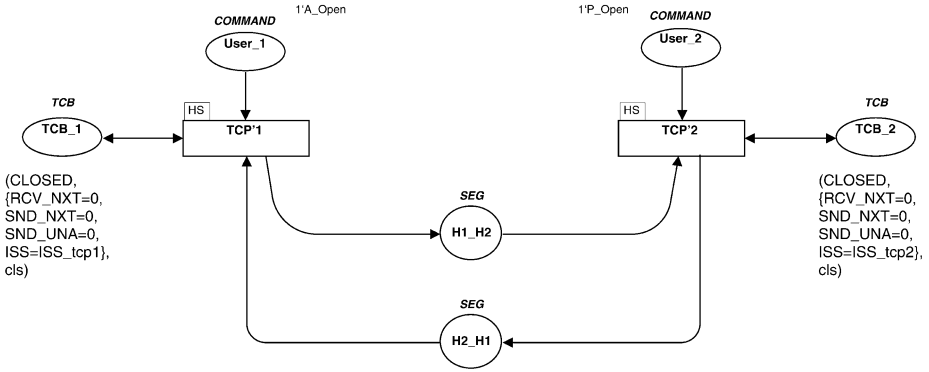


Fig. 20. Top level CPN page: TCP_Overview.

assigned to place User in Fig. 21. In contrast, Place H1_H2 in Fig. 20 is assigned to place Out for TCP'1 and In for TCP'2 (see Fig. 21) and vice versa for H2_H1.

The second level page structures the TCP connection management process into eleven substitution transitions. Each transition is named by a TCP state and is linked with a page at the third level, which models TCP's behaviour for that state.

10.5 Third Level Pages

We illustrate the TCP model at the executable level by considering four pages at the third level: CLOSED, LISTEN, SYN_SENT and SYN_RCVD.

The CLOSED Page. The CLOSED page (Fig. 22) models TCP's behaviour for the CLOSED state. It has 4 transitions and 4 places that have already been described.

When a server wishes to receive connection requests, it issues a passive open to its TCP entity. This is modelled by transition Passive.Open. TCP enters the LISTEN state (with state variables unchanged) after receiving a passive open (P_Open) command. When transition Passive.Open occurs, the value of the LISTENstat flag of the token in TCB is changed from *cls* to *lis*, indicating that TCP has been in LISTEN. Transition Active.Open models the expected behaviour of TCP sending out a SYN after receiving an active open (A_Open) command from its user. TCP enters SYN_SENT and updates its state variables as shown on the output arc to place TCB.

When TCP is CLOSED, it is not expecting any incoming segments. If it receives one, then it needs to inform the sender that it is closed, by sending a reset. However, if the incoming segment is a reset, then it is discarded, as the receiver does not need to inform its peer to close. As indicated by its guard, transition Rcv_noRST models TCP's response to any incoming segment that is not a reset (i.e. the RST bit is not set). TCP sends a RST segment to its peer

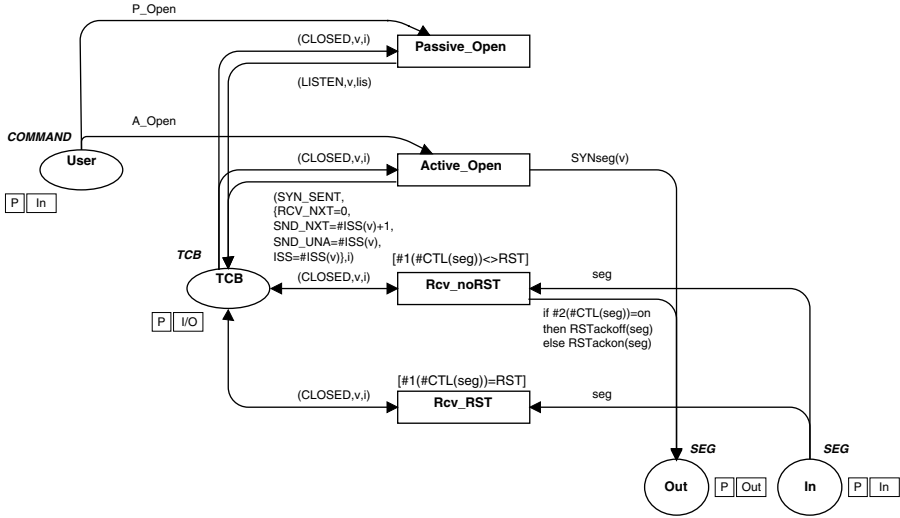


Fig. 22. The CLOSED page.

Transition `Rcv_ACK` and `Rcv_RST` model unexpected behaviour. Transition `Rcv_ACK` specifies TCP's response to any incoming segment with its ACK on that is not a reset. It replies with a RST segment and remains in `LISTEN`. Transition `Rcv_RST` models TCP remaining in `LISTEN` after receiving and discarding a RST segment.

When in `LISTEN`, TCP expects to receive a connection request. Transition `Rcv_SYN` models this situation by receiving a SYN segment. TCP returns a SYNACK and enters `SYN_RCVD` with its state variables updated.

The SYN_SENT Page. The `SYN_SENT` page (Fig. 24) has 9 transitions that are created in accordance with the TCP specification for the `SYN_SENT` state.

Transition `Close` models TCP entering `CLOSED` from `SYN_SENT` in response to a Close command. Transitions `Rcv_uACK` and `Rcv_uACK_RST` model TCP's response to an unacceptable ACK segment with the RST bit off and on respectively. Transitions `RST_LISTEN` and `RST_CLOSED` are used to respond to a RST with an acceptable ACK number based on TCP's previous state information. That is, if it has previously been in `LISTEN`, indicated by the `LISTENstat` flag having the value *lis*, TCP changes state to `LISTEN` from `SYN_SENT` (modelled by transition `RST_LISTEN`). Otherwise, it goes into `CLOSED` (modelled by transition `RST_CLOSED`). Transition `Rcv_RSTnoACK` is used to respond to a RST without an ACK. Transitions `Rcv_SYNACK` and `Rcv_SYN` model TCP's response to an incoming segment with the SYN bit on under different conditions. If the incoming segment acknowledges the SYN, TCP goes into `ESTABLISHED` from `SYN_SENT` and sends out an ACK (modelled by `Rcv_SYNACK`). Otherwise, it enters `SYN_RCVD` and sends out a SYNACK (modelled by `Rcv_SYN`). If the incoming segment has neither SYN nor RST set (normally an acceptable

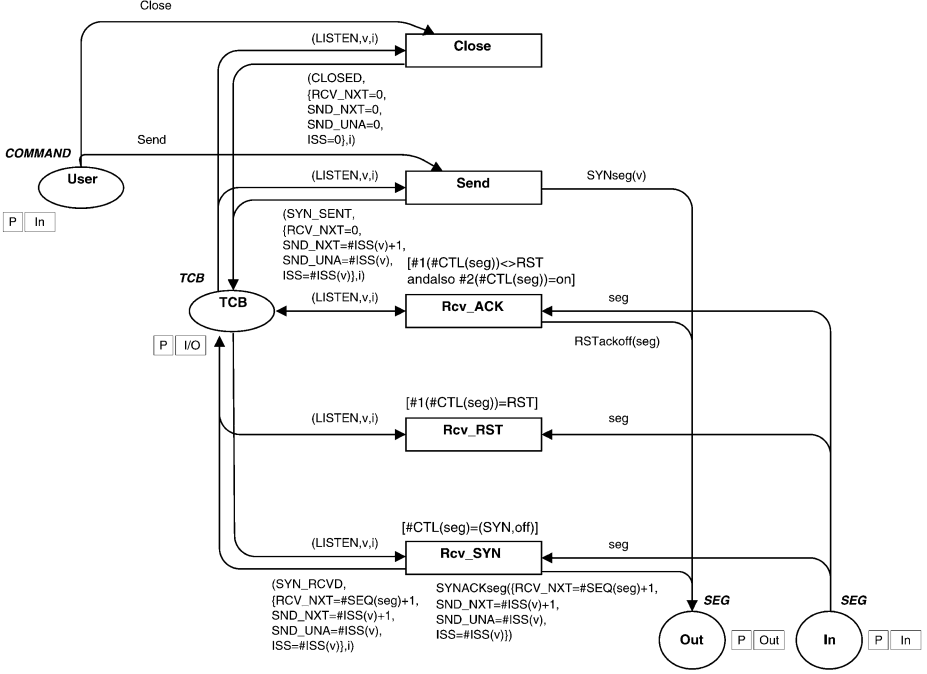


Fig. 23. The LISTEN page.

ACK), TCP drops the segment. This is modelled by transition `noSYN_noRST`. Note that if the ACK in an incoming segment is not acceptable, TCP sends out a RST, as specified by transition `Rcv_uACK`.

The SYN_RCVD Page depicted in Fig. 25 has 11 transitions. Transition `Close` models TCP sending a FIN segment and entering `FIN_WAIT_1` after receiving a *Close* command from its user. `Rcv_uSeq` and `Rcv_uSeq_RST` are used to respond to an incoming segment with an unacceptable sequence number (first item of the guard) with RST off or on respectively.

Other transitions in Fig. 25 require that the sequence number of the incoming segment is acceptable (first item of the guard). Transitions `RST_LISTEN` and `RST_CLOSED` model TCP's response to a RST with an acceptable sequence number based on TCP's previous state. That is, if it has previously been in `LISTEN`, TCP enters `LISTEN` from `SYN_RCVD`, otherwise, it changes state to `CLOSED`. Also depending on TCP's previous state, transitions `SYN_inw_LISTEN` and `SYN_inw_CLOSED` model TCP sending out a RST after receiving an acceptable SYN (i.e. its sequence number is in the receive window). If the SYN's sequence number is not in the window, an ACK is sent as a result of the sequence number check (see transition `Rcv_uSeq`). Transition `Rcv_ACKOff` models TCP dropping the incoming segment and remaining in `SYN_RCVD` in response to a segment with ACK off. Transitions `Rcv_ACKOnA` and `Rcv_ACKOnU` model TCP's behaviour on receipt of an ACK under different conditions. If the

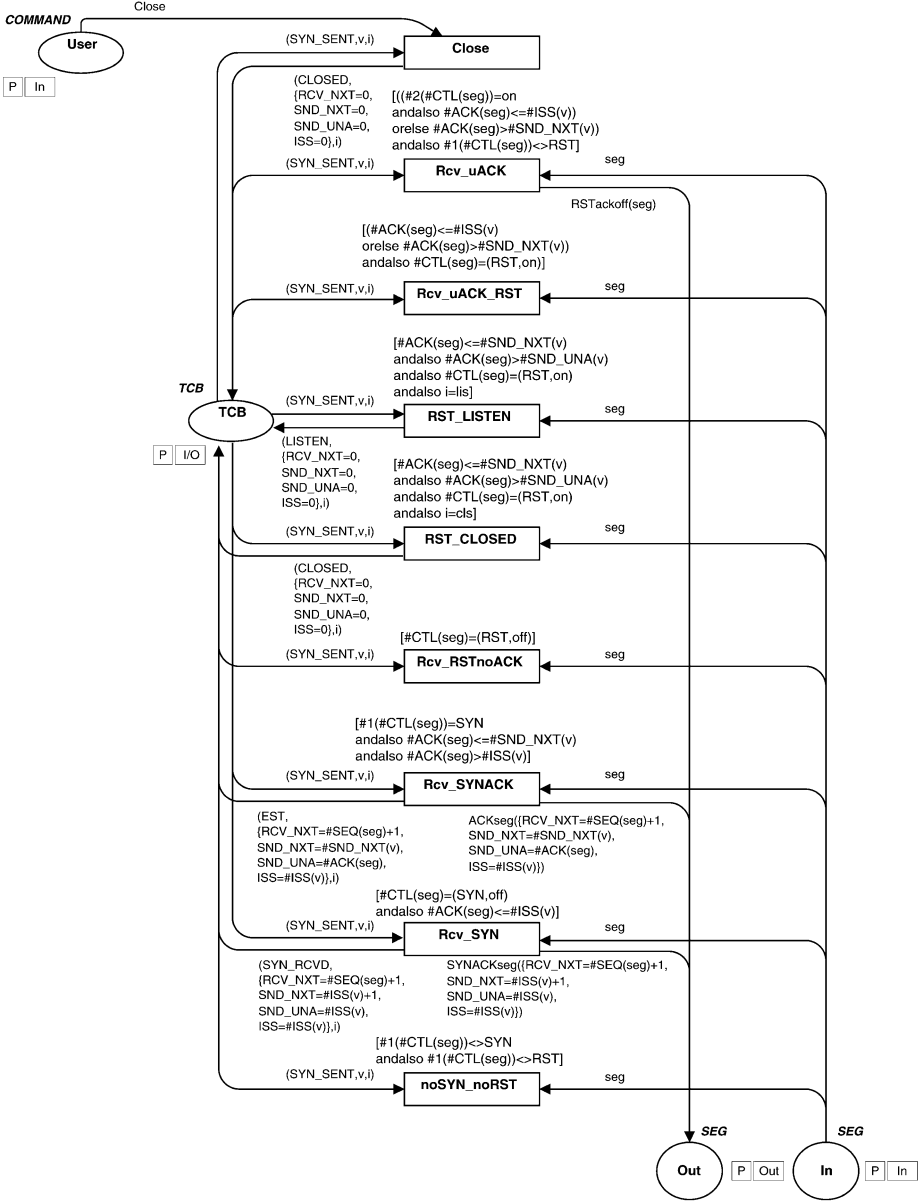


Fig. 24. The SYN_SENT page.

ACK is acceptable, TCP enters ESTABLISHED (modelled by Rcv_ACKOnA), otherwise, TCP sends out a RST (modelled by Rcv_ACKOnU). Finally, transition Rcv_FIN models TCP entering CLOSE_WAIT and sending an ACK on receipt of a FIN segment.

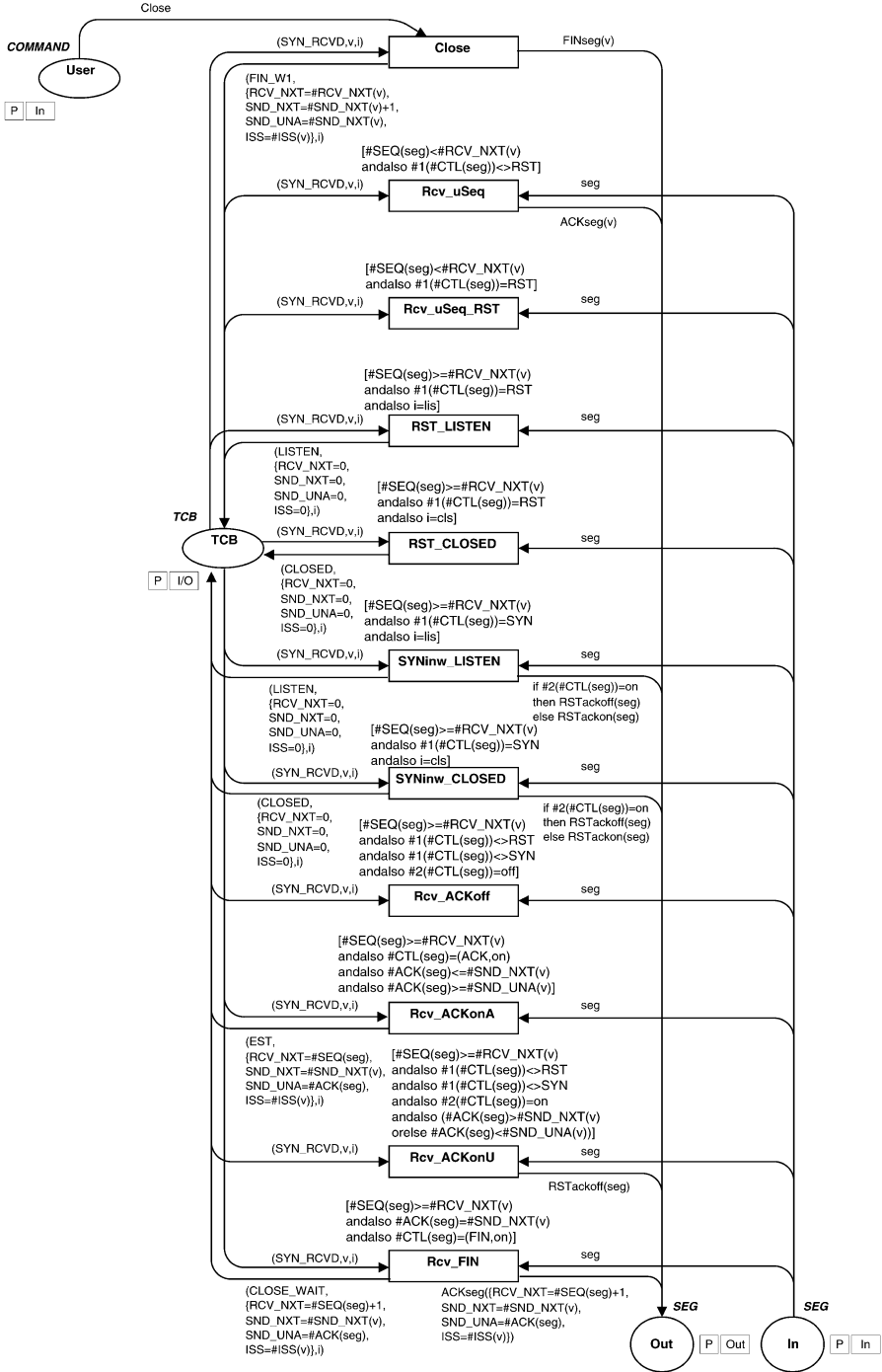


Fig. 25. The SYN.RCVD page.

11 Functional Analysis of TCP Connection Management

The TCP connection management protocol is analysed by generating occurrence graphs (OGs) using Design/CPN 4.0.5 running under Linux on a 800MHZ Intel Celeron laptop computer with 128 MB RAM.

11.1 Initial Configurations

We can analyse many different connection management scenarios by choosing a number of different initial markings of places User_1 and User_2 in Fig. 20. This allows us to start simply by just considering the connection establishment protocol, before analysing the effect of releasing connections. This incremental approach has two advantages. Firstly it allows us to gain confidence in the model by providing the simplest analysis results which can be checked against the specification (i.e. RFC 793). This is an important step in model validation. Secondly, we may discover errors in the protocol, which we can more easily debug in a simpler reachability graph. Once we gain confidence that the model is correct, we can then look at more complex scenarios which exercise all parts of the specification, by choosing the initial marking of places User_1 and User_2 accordingly. For example, we could include an active open and a close command in one user place, and a passive open, send and close in the other.

In this paper we shall just illustrate the approach by considering two cases that just relate to connection establishment. In Case 1, place User_1 has an A.Open (active open) command as the initial marking, while place User_2 has a P.Open (passive open) command as its initial marking. This represents the usual client-server opening scenario that would occur, for example, when requesting a web page. In Case 2, both user places have an A.Open as the initial marking, which allows for the simultaneous opening of a connection, that may occur in peer to peer applications. The initial markings of the remaining places are the same for the different cases. The channel places are empty, while the TCB places are as shown in Fig. 20. Since the initial send sequence (ISS) number for each TCB can be chosen arbitrarily within the 32 bit integer range, we arbitrarily select ISS=10 for TCB_1 and ISS=20 for TCB_2.

11.2 Analysis Results

When we analysed our CPN model for the above two cases, we found that there was a problem with the simultaneous open procedures (Case 2). We modified the original model (Model A) twice to remove the problems. We refer to the first modified model as Model B and to its modification as Model C.

Analysis of Model A

OGs for Model A are obtained for both cases. The results are summarised in Table 2 and the OGs for Cases 1 and 2 are given in Figs. 26 and 27.

The table shows the number of nodes, arcs and dead markings in the OG. The final column indicates whether or not the dead markings are considered to

Table 2. Results for Model A.

Model A	Nodes	Arcs	Dead Markings	Deadlocks
Case 1	11	12	2(0)	0
Case 2	42	60	2(1)	0

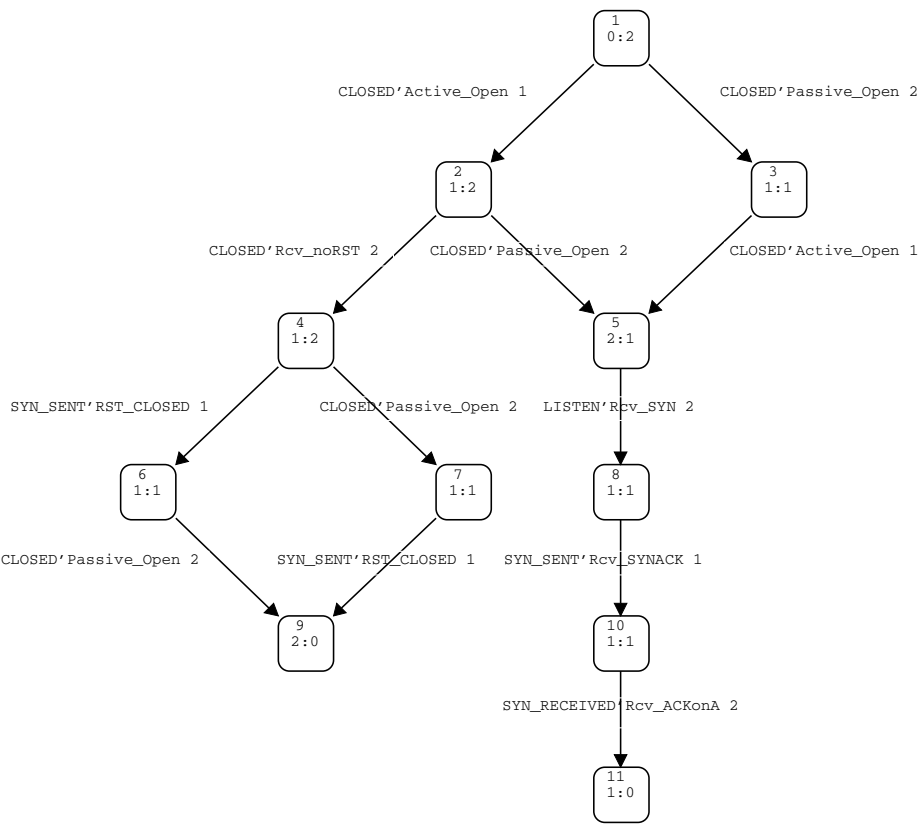


Fig. 26. OG of Model A for Case 1.

be deadlocks (i.e. undesired terminal markings). The integer in parenthesis in column Dead Markings shows the number of dead markings reached through undesired transition sequences. Both OGs are small and are generated in less than 1 second.

Case 1 has two dead markings, nodes 9 and 11 in Fig. 27. The details of the dead markings are given in Fig. 28. Node 9 has one TCP entity in CLOSED, the other in LISTEN, and nothing in the channel. This is an expected terminal state because when the TCP server is in CLOSED (the passive open command has yet to be issued) and receives a SYN, it sends out a RST that changes the state of the TCP client from SYN_SENT to CLOSED. The server TCP entity then

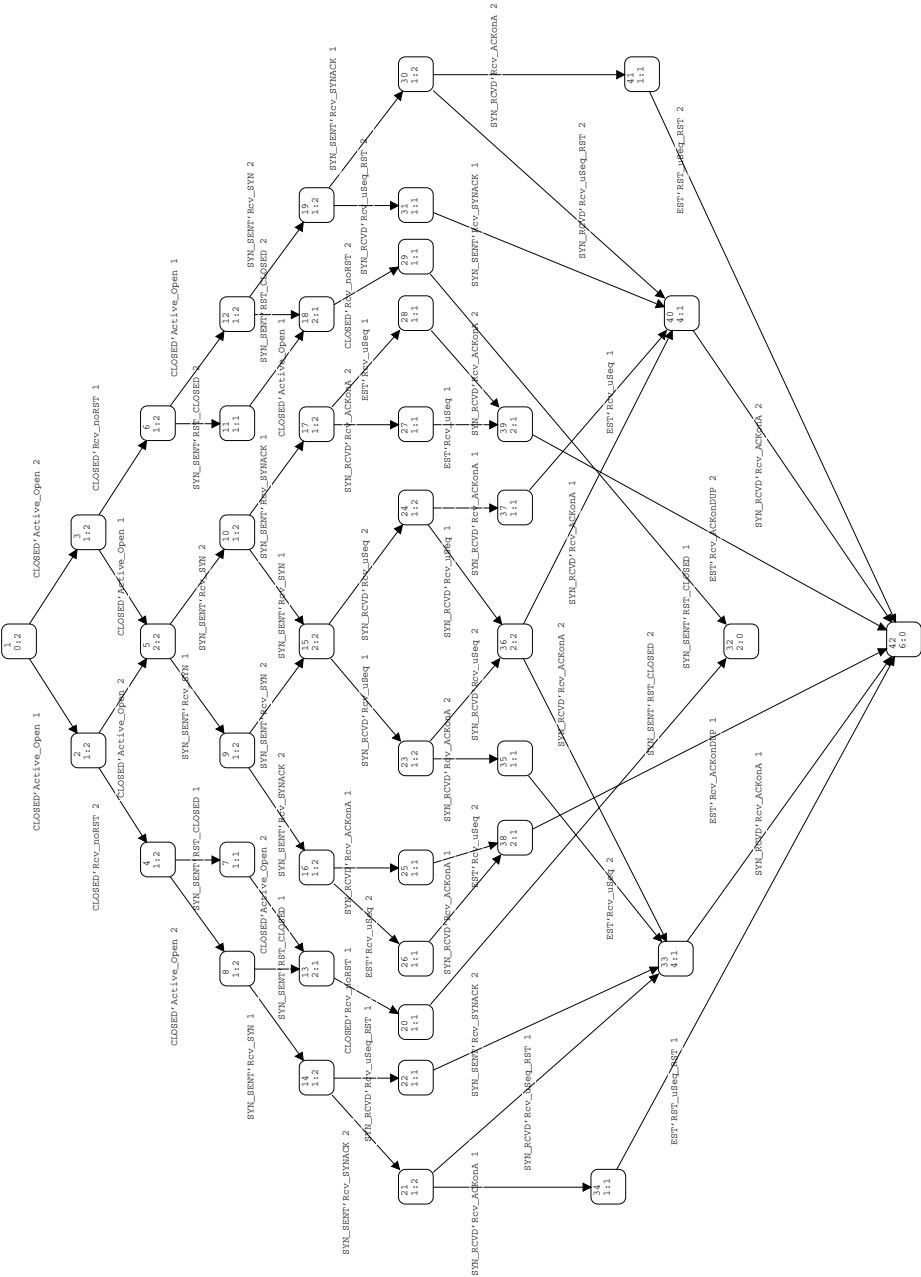


Fig. 27. OG of Model A for Case 2.

goes into LISTEN after receiving the passive open command. The other dead marking (Node 11) has both TCP entities in the ESTABLISHED state and

nothing in the channel. Also, the sequence numbers are synchronised at both ends, that is, the send next (SND_NXT) and the send oldest unacknowledged (SND_UNA) numbers are equal on one side and also equal the receive next number (RCV_NXT) at the other end of the connection. This is thus the desired terminal state in which the connection is properly established at both ends. Examining the sequences leading to the two dead markings (see Fig. 26) indicates that they are reached through desired transition sequences.

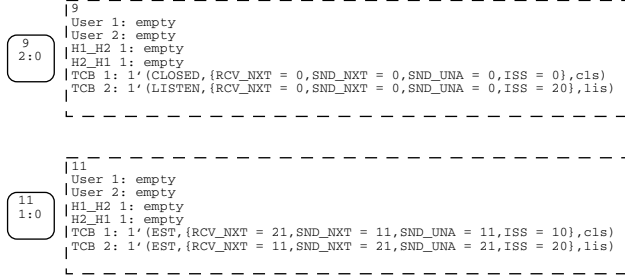


Fig. 28. The dead markings for Case 1 (Model A).

In Case 2 there are two dead markings, nodes 32 and 42 as shown in Fig. 27. Both are desired terminal states as can be seen from the marking details given in Fig. 29. Node 42 has both TCP entities in the ESTABLISHED state with synchronised state variables and nothing in the channel. The other dead marking (node 32) has both TCPs in the CLOSED state and nothing in the channel.

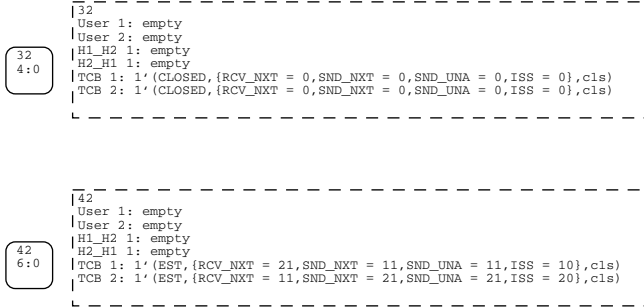


Fig. 29. The dead markings for Case 2 (Model A).

There are 38 sequences of transitions from node 1 to node 42, of length 7 or 8. Thus we do not find the sequence corresponding to the scenario shown in Fig. 16(b), which is of length 6. Instead, we find sequences similar to the trace obtained from the OG and shown in Fig. 30. This scenario is drawn as a message



Fig. 30. A sequence of the OG for Case 2 (Model A).

sequence diagram in Fig. 31(a), which shows that an ACK is sent in response to each SYNACK before each TCP entity enters the ESTABLISHED state. This behaviour is not desired because it adds delay when there is a simultaneous open and also is not according to TCP's intent as described by the message sequence diagrams in [62].

By examining the trace in Fig. 30, we see that it is transition Rcv_uSeq in Fig. 25 that sends out the redundant ACK, on receipt of segment SYNACK (see the transition from node 24 to node 36 in Fig. 30). Transition Rcv_uSeq

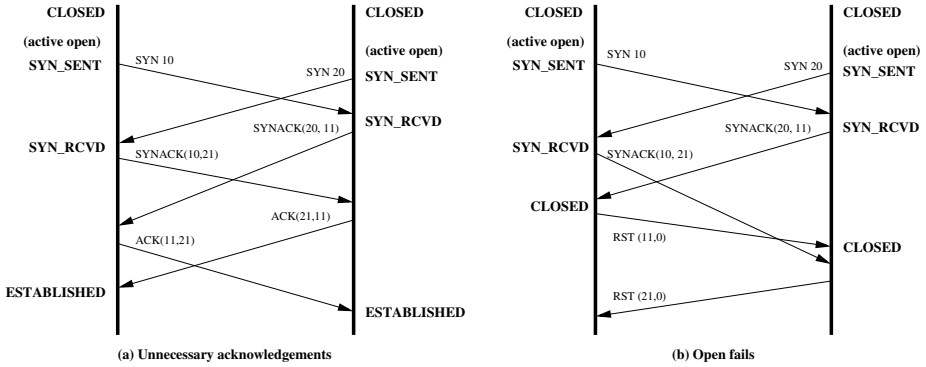


Fig. 31. Problematic simultaneous open scenarios.

sends out an ACK in response to any incoming segment that satisfies the two inequalities on its transition guard, according to the specification on page 69 of RFC 793.

“If an incoming segment is not acceptable, an acknowledgment should be sent in reply (unless the RST bit is set, if so drop the segment and return) ...”

This means that the sequence number of the incoming SYNACK is less than the value of RCV_NXT. In addition, the second inequality is satisfied because the SYNACK is not a RST segment. As the inequalities are applicable not only to the expected SYNACK, but also to other segments (including unexpected SYNACKs), we do not revise them to remove the problematic ACK for this case. Instead, we question the value of RCV_NXT on whether or not it reflects the sequence number of the next segment that TCP is expecting after receiving the SYN segment. The next segment that TCP is expecting is a SYNACK which has the same sequence number as that of the SYN (see Fig. 8, page 32 of RFC 793 and Fig. 16(b)). Examining the marking immediately prior to the firing of transition Rcv_uSeq, we find that the value of RCV_NXT is updated to $SEQ(seg)+1$ by transition Rcv_SYN on the SYN_SENT page (Fig. 24), according to processing a SYN in SYN_SENT on page 68 of RFC 793. To remove this inconsistency, we propose not to update RCV_NXT in the case of simultaneous open, keeping its value as $SEQ(seg)$. Therefore, $\#SEQ(seg) < \#RCV_NXT(v)$ will be false, and hence transition Rcv_uSeq will not be enabled.

To test our assertion, we make the corresponding modification, assigning $\#SEQ(seg)$ to RCV_NXT, on the output arc of transition Rcv_SYN to place TCB (see Fig. 24). This gives us Model B.

Analysis of Model B

The analysis results of Model B are again generated in less than 1 second and shown in Table 3. The results for Case 1 are the same as those of Case 1 for Model A.

Table 3. Results for Model B.

Model B	Nodes	Arcs	Dead Markings	Deadlocks
Case 1	11	12	2(0)	0
Case 2	44	62	2(1)	0

Examining the sequences leading to the dead markings of Case 2 shows that this undesired sequence is removed. However, it reveals another problematic sequence, which involves the generation of RSTs by each TCP entity in response to a SYNACK, as shown in Fig. 32. The corresponding scenario is drawn in Fig. 31(b), where the connection is unnecessarily terminated.

Examining the trace in Fig. 32, we see that it is transition SYNinw_CLOSED in Fig. 25 that sends out the RST. SYNinw_CLOSED rejects the SYN in the window by sending out a RST, as specified on page 71 of RFC 793. However, the SYNACK is mistakenly treated as an old duplicate, because RFC 793 specifies that for state SYN_RCVD, all segments with the SYN bit on (hence for a SYNACK) are rejected by sending out a RST. It is worth mentioning that our proposed solution for the first problem helps to reveal this. To remove the second problem, we must check the ACK bit while checking the SYN bit for state SYN_RCVD in the case of simultaneous open.

We make the corresponding modifications to the guards of transitions SYNinw_CLOSED and Rcv_ACKonA in Fig. 25. Firstly, we replace $\#1(\#CTL(seg)) = SYN$ with $\#CTL(seg) = (SYN, off)$ for transition SYNinw_CLOSED to exclude the SYNACK. Then we replace $\#CTL(seg) = (ACK, on)$ with $\#1(\#CTL(seg)) <> RST$ and $\#1(\#CTL(seg)) <> FIN$ for transition Rcv_ACKonA to accept the SYNACK. We also replace the annotation on the arc from Rcv_ACKonA to TCB by function. It is the same as the previous annotation except that it updates the value of RCV_NXT with $\#SEQ(seg)+1$ if a SYNACK is received and with $\#SEQ(seg)$ if an ACK is received. We thus get Model C.

Analysis of Model C

The analysis results of Model C are shown in Table 4.

Table 4. Results for Model C.

Model C	Nodes	Arcs	Dead Markings	Deadlocks
Case 1	11	12	2(0)	0
Case 2	39	54	2(0)	0

The results of Case 1 are the same as those of Case 1 for Models A and B. Analysis of Case 2 shows that there are two dead markings that are desired and reached through desired sequences. One dead marking (Node 35 in Fig. 33) has TCP entities in ESTABLISHED, state variables synchronised and nothing in the channel. The other (Node 32) has both TCPs in CLOSED and nothing in the channel.

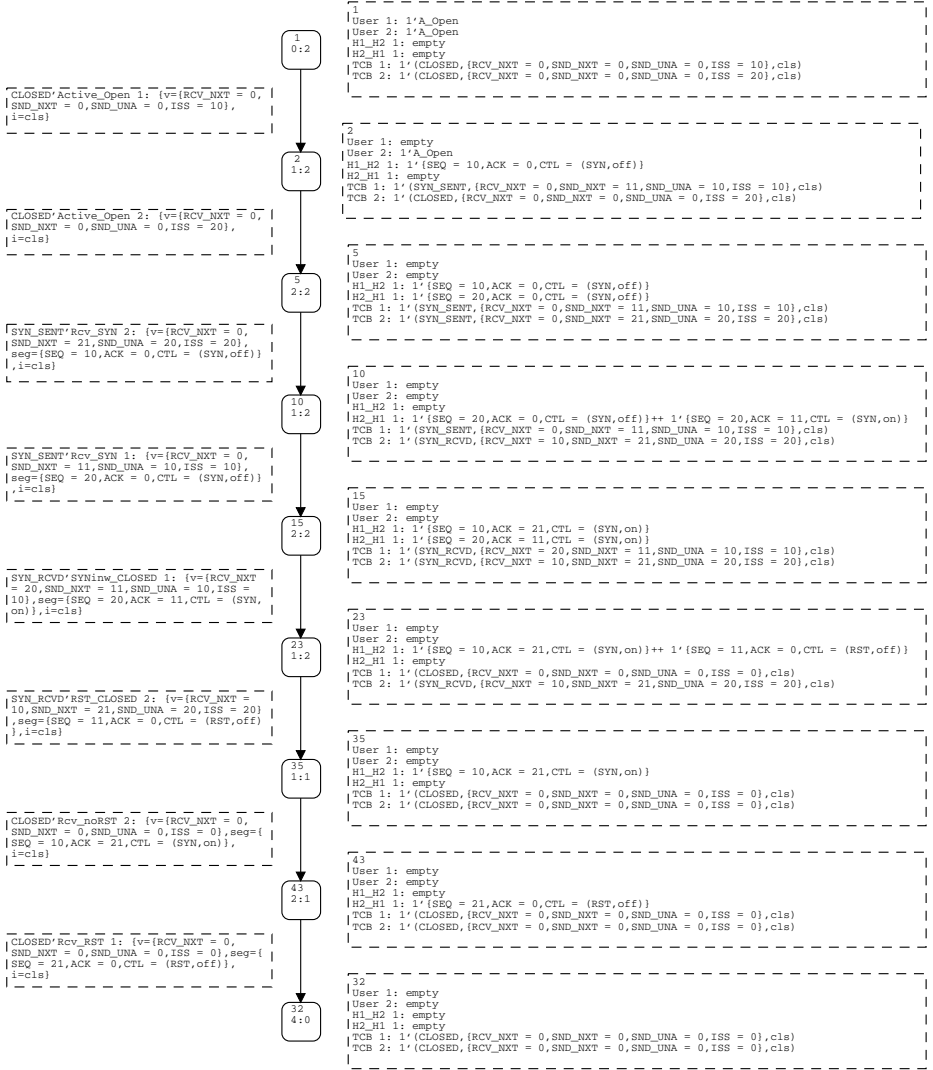


Fig. 32. A sequence of the OG for Case 2 (Model B).

Based on the sequences from the reachability analysis, three typical simultaneous open scenarios that are expected are obtained. They are presented in Fig. 16(b), and Fig. 34 (a) and (b). The scenarios shown in Fig. 34 are not discussed in RFC 793, nor in the text books (e.g., [26,69]).

12 Concluding Remarks on TCP

In this part of the paper we have illustrated two parts of the verification methodology for a complex practical protocol. Firstly we have provided some guidelines

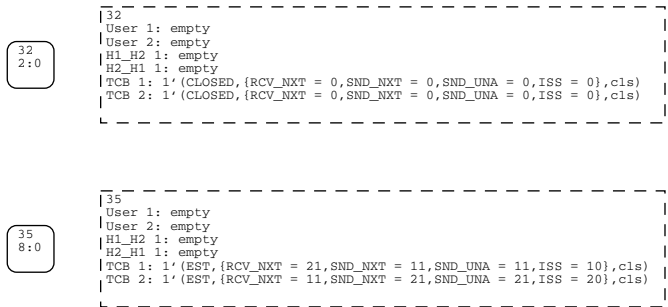


Fig. 33. The dead markings for Case 2 (Model C).

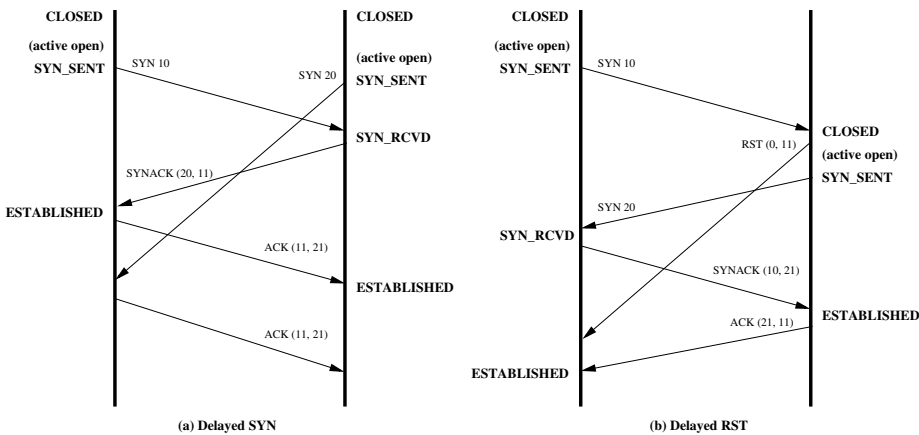


Fig. 34. Two expected simultaneous open scenarios.

for taking the first steps when modelling a complex protocol. The protocol is divided into its connection management part and its data transfer part. We firstly model the connection management part and importantly list the assumptions made. We then illustrate the use of hierarchical CPNs to structure the specification, taking advantage of symmetry, to just specify the connection management procedures once, but call them for each TCP entity by using page instances. This reduces the complexity of the model and eases maintenance. The detailed part of the model is structured into the processing that occurs per state which is a standard way of specifying protocols and a useful starting point for organising specifications. Secondly, we have illustrated the process of analysing a connection management protocol using reachability analysis with the help of Design/CPN.

The analysis has revealed some problems with the procedures for simultaneously opening a connection as specified in Section 3.9 of RFC 793. One problem causes delay and additional traffic due to an unnecessary exchange of acknowledgements when establishing the connection. A solution to this was incorporated and analysed revealing a more serious problem where the procedure can fail to

establish the connection. This is due to another error in the functional specification of RFC 793. A further correction to the model is made which removes the problem. Note that the second error also occurs in the finite state machine (FSM) diagram of RFC 793. So the FSM and narrative descriptions of Section 3.9 of RFC 793 are consistently in error. No deadlocks or other subtle errors were found. These results were reported in [38, 39].

This part has only touched the surface regarding the analysis of TCP connection management, where we have presented results just for the case of opening a connection. Further work addresses closing of the connection, relaxes some of the modelling assumptions, for example, allowing segments to be lost in the channel and recovered by retransmissions, develops a service specification [15–17] and compares the connection management protocol language with its service language. This work is consolidated in [37].

13 Some Observations Concerning Specification and Verification

13.1 Modelling Assumptions

After reading the definition of the protocol or service in a standards document (or other primary reference) it is vitally important to write down the assumptions that are made when modelling the protocol (or service). This is to ensure that the analysis results obtained are with respect to the set of assumptions made and that this is firmly in the mind of both the verifier and the reader of the results. The assumptions can be with respect to scope or restrictions within the scope, or abstractions that are made. Illustrations of these for complex protocols are given in [34, 37, 58, 78].

13.2 Specification Structure

There are several ways to structure a specification using hierarchical nets such as Coloured Petri Nets. One of the most popular ways is to structure specifications according to the (major) states of the protocol entities. For each major state, each action in that state is modelled by a transition, for example, the processing of an incoming message or a command from a user. Many international standards are structured this way by using state tables, or Specification and Description Language (SDL) [45] processes where there is one SDL diagram for each major state. If the CPN structure matches the structure in the international standard, then this aids in validating the CPN model against the standard, to ensure that the CPN accurately reflects the standard.

However, there are some specifications where this approach has significant drawbacks and can lead to specifications that have a lot of redundancy. This is the case when the processing of various input messages are treated the same way in a *set of states*. For example in TCP, the state machine comprises 11 states, and processing of some actions such as reset is essentially the same for

8 of these states. Thus the specifier needs to be aware of commonalities that exist for various states and to structure the specification according to processing actions (such as opening connections, resetting, closing connections and dealing with timeouts and retransmissions) rather than slavishly following a state-based approach. This leads to a more elegant specification that has fewer transitions, where actions that are common are clearly seen to be the same, and where maintenance is facilitated. This is following the usual rules of good programming and good writing, which is at the core of the object-based approach. The reduction in the specification is achieved by the folding of transitions that have the same actions for different states, and using a variable that runs over states, with the appropriate restrictions placed in a guard (see [37]).

13.3 Specification Validation

The validation of a CPN specification against the definition of a protocol provided in a standards document or in a proprietary definition is a very important step in making sure that the analysis results do apply to the system of interest. Validation may involve several steps. Firstly developing specifications incrementally and checking that each transition does accurately reflect the intent of the system is essential. This may include stepping through the model using interactive simulation on a tool such as Design/CPN. The next step is to incrementally analyse the model using the OG. Errors discovered may be in the model or in the protocol definition. This is detected by carefully checking if the error is in fact in the system or introduced into the model due to some misinterpretation or inaccurate assumption. Errors in the model need to be removed and then the model re-analysed, iteratively removing inaccuracies. In circumstances where the protocol definition can be discussed with its inventors, this is a vital step in resolving assumptions made in the model.

13.4 Specification versus Verification

Various concerns are more important when developing specifications for implementation rather than for verification. A specification for implementation needs to be readable to ease understanding, and complete so that it contains all essential details to ensure that implementations will interwork correctly. Due to complexity, the verifier will want to modify the specification to just concentrate on essential aspects that need verification. Otherwise the verification task becomes impossible. The main approach concerns making the right abstractions and dividing the specification into manageable and separable components that can be verified independently.

When considering the development aspects of protocol engineering in full, the protocol engineering team will need to develop the protocol architecture and complete service and protocol specifications. The art of the protocol verifier is then to have sufficient insight into how the protocol specification can be divided into manageable parts for verification. This may then lead to proof obligations

that the independence assumptions are valid for the properties that require verification.

Let us briefly look at some of the techniques that are used.

Independence of Protocol Mechanisms. Transaction protocols, such as the Internet Open Trading Protocol, define a set of transactions that may be carried out by the different parties. In this case, it is possible to treat each transaction separately. This greatly reduces the complexity of verification. It is only when transactions interact in some way (such as when a refund is dependent on a previous purchase), that we need to consider transactions together. Hence a lot of insight can be obtained and errors can be detected by considering each transaction independently (see [58]).

For connection-oriented protocols, we can divide the operation of the protocol into phases: connection establishment; data transfer; and connection release, and verify each phase separately, before considering their interactions. The next step will normally be to consider the release or abortion of connections at any time during the life of the connection, considering connection establishment and termination together.

Data Abstraction. Protocol messages may include fields that are not used or affected by the protocol mechanisms under investigation. For example, address and multiplexing fields (such as port numbers in TCP) allow multiple connections to be in progress at the same time between a large number of end systems. However, the operation of each connection is the same, and as long as we are not concerned about resources that are shared between connections (such as buffer space), we can just consider one connection as a representative and analyse its behaviour. This means that we can greatly reduce the address space and multiplexing fields. For example, in the case of TCP, we do not need to include these fields at all, and can just consider two protocol entities interacting over a medium representing the operation of IP, where every message that is sent is destined for the peer entity. This is easily achieved by having two places for the medium, one for each direction of information flow. Further, if our objective is to verify TCP connection management procedures, then we can safely ignore fields that are concerned with data transfer, such as windows for flow control, and urgent pointers and flags that are concerned with urgent data transfer. Moreover, we can omit the checksum (used to detect transmission errors) and model its effect using non-determinism as discussed in Section 2.2. We also do not need to model the header length or options (such as the maximum packet size), nor the data that may be transferred. This then reduces the message to a small tuple of values. In the case of TCP a message can be reduced to a triple where we model the message type (such as a connect request (SYN) or reset (RST)), its sequence number and an acknowledgement number. This also reduces the amount of state information that needs to be modelled in the TCP entities, such as the window size. An illustration of the approach is given in [39] and explained earlier in this paper.

With data transfer protocols it is normally sufficient to consider the sender and receiver parts of a protocol entity separately. Thus we just consider one way flow of data across the medium between a sender and receiver, with a return path for control information such as acknowledgements. However, ignoring the data field in data transfer protocols can lead to situations where although service primitive sequences are satisfied, duplicate data and data loss may be occurring, as discussed in Section 7.2.

13.5 Modelling the Underlying Medium

The behaviour of the protocol depends on the medium over which it operates. It is important to start with the simplest medium that makes sense. This is often a FIFO queue. For example, although the Internet Protocol allows for re-ordering, loss, delay and duplication of messages, most of the time it behaves like a FIFO queue. Thus as a first step, it is important to verify that the protocol will operate correctly over this perfect medium. The reason for this is that media that can lose messages or reset or disconnect connections, can hide undesirable behaviour such as the occurrence of deadlocks. However, that the medium may occasionally lose a message or reset a connection is cold comfort to the users of the protocol who may have to wait a long time for such an event to occur to remove the deadlock. Once the protocol is shown to operate correctly over this friendly medium, then medium imperfections can be introduced and the protocol re-analysed.

13.6 Incremental Approach

To gain confidence in the CPN model and with its verification it is important to use an incremental approach in modelling and analysing the system under investigation. This is illustrated in [78] where RSVP mechanisms are examined one at a time. A similar approach is taken for IOTP [58] where error free and successful transactions are examined first and then arbitrary cancellation and error handling procedures are added.

13.7 Parameters

Complex protocols include several parameters of significance such as the size of the sequence number space, flow control and congestion control window sizes, the maximum number of retransmissions, and the data that is to be sent. Another parameter may also be the capacity of the medium over which the protocol operates. For example, in the Stop-and-Wait protocol we have two parameters of interest, the maximum sequence number and the maximum number of retransmissions.

Reachability analysis requires that parameters of the system be instantiated with particular values before the reachability graph can be generated. Thus we have to generate the OG and its equivalent deterministic automaton for

every parameter value. The first step is to start with the smallest values that make sense (such as a medium capacity of one or two, no retransmissions, and maximum sequence number of one) to obtain results to give a feel for how the system operates.

As the values of these parameters increase, reachability analysis suffers from the state explosion problem [77] and becomes intractable. It also means that we can only obtain results for a (small) subset of values, rather than a general result for any value. It may be that results can be obtained for the values of interest of the parameters involved, such as the maximum values of the retransmission counters in the transaction service of WAP [34]. However, in general we would like a result that is valid for any value of the parameters involved. In this case we resort to theorem proving, quite often based on the results obtained from reachability analysis for small parameter values.

In some cases we can invoke the notion of *data independence* [82] to reduce what could be an infinite set (e.g. data items) to a finite and possibly very small number (such as 3), when protocol operations do not affect the data. This can be the case for the data that is sent, when only read/write or assignment operations are involved [65].

Recently, we have managed to obtain symbolic expressions (possibly recursive) for the reachability graphs and their deterministic automata in terms of the medium capacity for the Capability Exchange Signalling service and TCP's data transfer service as discussed under infinite state systems in Section 2.3.

14 Conclusions

This paper summarises the steps of a protocol verification methodology and discusses them in some detail based on several years of attempting to use the methodology for the verification of industrial scale protocols. The methodology uses Coloured Petri Nets (CPNs) to specify both the service provided by the protocol to its users, and the composite specification of the protocol entities interacting over a communication medium or channel. The composite specification is analysed using tool supported reachability analysis to discover behavioural properties, such as undesirable terminal states, livelocks, channel bounds or sequences of events which are inefficient, without recourse to the service specification. Some of this behaviour (inefficient sequences or bounds) are not visible to the users and are thus not captured in service specifications. However, because they have impact on the use of network resources they are worthy of investigation.

Most protocols are characterised by a set of parameters (such as window sizes, sequence number range and maximum number of retransmissions) which need to be instantiated for automatic reachability analysis. For parameters with a small range, such as the maximum number of retransmissions for the Wireless Application Protocol transaction layer (where no more than 8 is required) repeated automated runs may be sufficient to obtain the desired results [34]. The range of results obtained may be extended by using advanced state space techniques, such as the sweep-line method [25, 36], partial orders or equivalence

classes or some combination [14]. However, for many industrial protocols, the number of parameters and their significant ranges, preclude obtaining results for all but very small values of the parameters using automated analysis. When this is the case, the automated approach may be able to be supplemented by hand proofs to obtain general results for all values of the parameters. Quite often the intuition behind these hand proofs arises from the results obtained from using automated reachability analysis for small values of parameters.

It is also valuable to compare the behaviour of the composite specification with that of the service specification. For some protocols the service has also been defined as part of the process of defining the protocol. This greatly facilitates the creation of a CPN model of the service. However, the service definitions invariably do not completely specify the global sequences of user observable events (known as service primitives) concentrating instead on defining the local interface sequences. The CPN modeller must then use their intuition to obtain a consistent set of global sequences. Sometimes this involves complex structures in the CPN to ensure that the correct sequences occur, while in other cases it is quite straightforward. However, as far as we are aware, none of the Internet protocols have included service definitions, and thus a service specification must be created, based on the protocol definition, interface definitions if they exist and the experience of the modeller. In this circumstance, it is recommended that the protocol be modelled and analysed first, and then the service developed, based on the experience gained. Once a service model is obtained, the sequences of service primitives embedded in the service model (the service language) can be compared with the service primitive sequences that occur in the composite specification (the protocol language). This can be done automatically for finite state systems using well-known reachability analysis and automata reduction and comparison algorithms. This approach also suffers from the state explosion problem. For infinite state systems, or where parameter values are unknown, we briefly discuss an approach using recursive techniques to provide symbolic representations of the occurrence graphs and associated automata. The intuition behind these symbolic representations has been obtained from using reachability analysis for small values of parameters. In some cases it is possible to derive the symbolic representations directly without the need for recursive formulations.

The methodology is illustrated by two case studies. One considers the class of Stop-and-Wait protocols (SWP) as a representative of the class of data transfer protocols. This involves the inclusion of two parameters: the maximum sequence number and the maximum number of retransmissions. We define 4 properties of interest (queue bounds, data loss, duplication and the Stop-and-Wait property) and prove that the SWP operates correctly over FIFO channels. The channel bound depends the maximum number of retransmissions (MaxRetrans) and is given by $2\text{MaxRetrans} + 1$ for both channels. We believe this to be a new result. This is obtained using hand proofs, which we develop in detail to illustrate the process. To prove that there is no loss or duplication and that the Stop-and-Wait property holds (i.e. that the protocol satisfies its service of alternating sends and receives) we use automatic reachability and language analysis. We do this for a

significant range of parameter values e.g. up to 10 bit sequence numbers for up to 4 retransmissions, which gives confidence that the results are general. However, we have no general proof. We also prove that the properties do not hold when the SWP operates over lossy (or lossless) re-ordering channels. Again hand proofs are used to prove that the channels are unbounded, giving a general result when both the maximum number of retransmissions and the maximum sequence number are any positive integer. We use language analysis for the other proofs, and argue that they are generally applicable for medium capacities of two or greater, and for the other two parameters (`MaxRetrans` and `MaxSeqNo` being positive integers). The results for loss and duplication are illustrated with time sequence diagrams. We discuss the practical relevance of these results by considering their impact on the Transmission Control Protocol. We conclude that the problems can occur once transmission rates are at about 10 Gbit/s. We also discuss a limitation of our approach. We have shown that loss and duplication can occur in the SWP by considering sequences where there are more sends than receives (loss) or more receives than sends (duplication). We also illustrate that both loss and duplication can occur even when the Stop-and-Wait property of alternating sends and receives holds. This is due to our data abstraction assumption, which is too strong. We note that using the notion of data independence may overcome this limitation.

The second case study examines the connection management procedures of the Transmission Control Protocol, as a representative example of connection management procedures as opposed to the data transfer procedures of the SWP. It also illustrates the application of the methodology to a practical protocol of major significance. We exemplify the process of writing down assumptions, regarding the creation of the model, that simplify the analysis task. We also stress the importance of this step. We build a model of significant complexity and analyse the connection establishment protocol. This allows us to discover two problems with the simultaneous open procedures, and provide a solution which we then verify to be correct. This is easily handled by brute force reachability analysis. However, we have only illustrated the use of the methodology for the analysis of the very simplest of procedures. Further work is in hand to provide a comprehensive verification of the connection management procedures, including release, abort and the use of retransmissions [37].

Finally we end the paper with some observations and recommendations regarding the use of the methodology. Better tool support is required for the methodology to allow the seamless integration of reachability analysis and language analysis and to allow language equivalence or inclusion to be done on-the-fly using advanced techniques such as the sweep-line method. Promising areas of future work include the incorporation of the notion of data independence into the methodology and the use of recursive techniques to obtain general results for parameteric verification.

Acknowledgements

We gratefully acknowledge the contributions of many people to this work. First and foremost Jonathan greatly appreciates the guidance and encouragement of Professor Fred Symons who introduced the work to Telecom Australia Research Laboratories (TRL) in the late 1970s. His former colleagues in TRL, especially Geoffrey Wheeler, Michael Wilbur-Ham, Tim Batten, John Gilmore and Greg Findlow have provided important contributions and insights. Past and present CSEC postgraduate students have contributed significantly to the area, in particular: Lin Liu for her work on methodology, including recursive formulae for occurrence graphs and automata; Chun Ouyang for work on methodology, complex protocol specification and defining acceptable subsets of service specifications using IOTP; Dr. Maria Villapol for her work on incremental specification and verification with RSVP; Dr. Steven Gordon for applying the methodology, including the sweep-line technique, to WAP; and Mat Farrington who pointed out the relevance of RFC 1323 to our investigations. We would also like to thank Dr. Lars Kristensen, a post doctoral fellow with CSEC for two years from 2000 to 2002, for his many valuable suggestions regarding the methodology, the use of Design/CPN and his work on the development of the sweep-line technique.

References

1. P. Aziz Abdulla and B. Jonsson. Verifying Programs with Unreliable Channels. *Information and Computation*, 127(2):91–101, June 1996.
2. Y. Afek, H. Attiya, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, D. Wang, and L. Zuck. Reliable Communication Over Unreliable Channels. *Journal of the ACM*, 41(6):1267–1297, Nov. 1994.
3. Y. Afek and G.M. Brown. Self-Stabilization of the Alternating Bit Protocol. In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pages 80–83. IEEE Comput. Soc. Press, 1989.
4. F. Babich and L. Deotto. Formal Methods for the Specification and Analysis of Communication Protocols. *IEEE Communications Surveys*, 4(1):2–20, Third Quarter 2002.
5. W.A. Barrett and J.D. Couch. *Compiler Construction: Theory and Practice*. Science Research Associates, 1979.
6. K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A Note on Reliable Full-Duplex Transmission over Half-Duplex Links. *Communications of the ACM*, 12(5):260–261, May 1969.
7. J. Billington. *Abstract Specification of the ISO Transport Service Definition using Labelled Numerical Petri Nets*, volume III of *Protocol Specification, Testing, and Verification*, pages 173–185. Elsevier Science Publishers, 1983.
8. J. Billington. Extensions to Coloured Petri Nets. In *Petri Nets and Performance Models, The Proceedings of the Third International Workshop, PNPM '89, Kyoto, Japan, December 11-13*, pages 61–70. IEEE Computer Society, 1989.
9. J. Billington. Formal specification of protocols: Protocol Engineering. In *Encyclopedia of Microcomputers*, volume 7, pages 299–314. Marcel Dekker, New York, 1991.

10. J. Billington. *Protocol Specification using P-Graphs, a Technique based on Coloured Petri Nets*, volume 1492 of *Lecture Notes in Computer Science, Lectures on Petri Nets II: Applications*, pages 293–330. Springer-Verlag, 1998.
11. J. Billington, M. Diaz, and G. Rozenberg, editors. *Application of Petri Nets to Communication Networks*, volume 1605 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
12. J. Billington and G. E. Gallasch. How Stop and Wait Protocols Can Fail Over The Internet. In *Proceedings of FORTE'03*, volume 2767 of *Lecture Notes in Computer Science*, pages 209–223. Springer-Verlag, 2003.
13. J. Billington and G. E. Gallasch. An Investigation of the Properties of Stop-and-Wait Protocols over Channels which can Re-order messages. Technical Report 15, Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, Australia., 2004.
14. J. Billington, G.E. Gallasch, L.M. Kristensen, and T. Mailund. Exploiting equivalence reduction and the sweep-line method for detecting terminal states. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 34(1):23–37, January 2004.
15. J. Billington and B. Han. Formalising the TCP Symmetrical Connection Management Service. In *Proceedings of the Design, Analysis and Simulation of Distributed Systems Conference, Orlando, Florida, USA*, pages 178–184, March 2003.
16. J. Billington and B. Han. On Defining the Service Provided by TCP. In *Proceedings of the 26th Australasian Computer Science Conference, Adelaide, Australia*, volume 16 of *Conferences in Research and Practice in Information Technology*, pages 129–138, February 2003.
17. J. Billington and B. Han. Closed Form Expressions for the State Space of TCP's Data Transfer Service Operating over Unbounded Channels. In *Proceedings of the 27th Australasian Computer Science Conference, Dunedin, New Zealand*, volume 26 of *Conferences in Research and Practice in Information Technology*, pages 31–39, January 2004.
18. J. Billington, G.R. Wheeler, and M.C. Wilbur-Ham. PROTEAN: A High-level Petri Net Tool for the Specification and Verification of Communication Protocols. *IEEE Transactions on Software Engineering*, 14(3):301–316, March 1988.
19. J. Billington, M.C. Wilbur-Ham, and M.Y. Bearman. Automated Protocol Verification. In *Protocol Specification, Testing and Verification, V*, pages 59–70. North Holland, Amsterdam, 1986.
20. T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Comput. Networks and ISDN Sys.*, 14(1):25–59, 1987.
21. S. Budkowski and P. Dembinski. An Introduction to Estelle: A Specification Language for Distributed Systems. *Comput. Networks and ISDN Sys.*, 14(1):3–23, 1987.
22. CCITT. ISDN user-network interface data link layer specification. Technical report, Draft Recommendation Q.921, Working Party XI/6, Issue 7, Jan. 1984.
23. S. Christensen and L. O. Jepsen. Modelling and Simulation of a Network Management System Using Hierarchical Coloured Petri Nets. In *Proceedings of the 1991 European Simulation Multiconference*, pages 47–52. Society for Computer Simulation, 1991.
24. S. Christensen and J.B. Jørgensen. Analysis of Bang and Olufsen's BeoLink Audio/Video System Using Coloured Petri Nets. In *Proceedings of ICATPN'97*, volume 1248 of *Lecture Notes in Computer Science*, pages 387–406. Springer-Verlag, 1997.

25. S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proceedings of TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 450–464. Springer-Verlag, 2001.
26. D.E. Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture*, volume 1. Prentice Hall, Upper Saddle River, NJ, 2000.
27. CPN ML: An Extension of Standard ML.
<http://www.daimi.au.dk/designCPN/sml/cpnml.html>.
28. J. Desel and W. Reisig. *Place/Transition Petri Nets*, volume 1491 of *Lecture Notes in Computer Science, Lectures on Petri Nets I: Basic Models*, pages 122–173. Springer-Verlag, 1998.
29. Design/CPN Online. <http://www.daimi.au.dk/designCPN/>.
30. M. Diaz. Modelling and Analysis of Communication and Co-operation Protocols Using Petri Net Based Models. In *Protocol Specification, Testing and Verification*, pages 465–510. North-Holland, 1982.
31. G. A. Findlow, G. S. Gerrand, J. Billington, and R. J. Fone. Modelling ISDN Supplementary Services Using Coloured Petri Nets. In *Proceedings of Communications '92*, pages 37–41, Sydney, Australia, 1992.
32. D. J. Floreani, J. Billington, and A. Dadej. Designing and Verifying a Communications Gateway Using Colored Petri Nets and Design/CPN. In *Proceedings of ICATPN'96*, volume 1091 of *Lecture Notes in Computer Science*, pages 153–171. Springer-Verlag, 1996.
33. FSM Library, AT&T Research Labs.
<http://www.research.att.com/sw/tools/fsm/>.
34. S. Gordon. *Verification of the WAP Transaction Layer using Coloured Petri Nets*. PhD thesis, School of Electrical and Information Engineering, University of South Australia, 2001.
35. S. Gordon and J. Billington. Analysing the WAP class 2 Wireless Transaction Protocol using Coloured Petri nets. In *Proceedings of 21st International Conference on Application and Theory of Petri Nets*, volume 1825 of *Lecture Notes in Computer Science*, pages 207–226. Springer-Verlag, 2000.
36. S. Gordon, L.M. Kristensen, and J. Billington. Verification of a Revised WAP Wireless Transaction Protocol. In *Proceedings of 23rd International Conference on Application and Theory of Petri Nets*, volume 2360 of *Lecture Notes in Computer Science*, pages 182–202. Springer-Verlag, 2002.
37. B. Han. Formal Specification of the TCP Service and Verification of TCP Connection Management. Draft PhD Thesis, University of South Australia, April 2004.
38. B. Han and J. Billington. An Analysis of TCP Connection Management Using Coloured Petri Nets. In *Proceedings of the 5th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI'2001)*, pages 590–595, Orlando, Florida, July 2001.
39. B. Han and J. Billington. Validating TCP Connection Management. In *Proceedings of the Workshop on Software Engineering and Formal Methods, Adelaide, Australia*, volume 12 of *Conferences in Research and Practice in Information Technology*, pages 47–55, June 2002.
40. G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
41. International Telecommunication Union. <http://www.itu.int/home/>.
42. Internet Engineering Task Force. <http://www.ietf.org>.
43. The Internet Engineering Task Force. TCP Extensions for High Performance. RFC 1323, 1992.

44. ISO/IEC. *Software and Systems Engineering – High-level Petri Nets – Part 1: Concepts, Definitions and Graphical Notation*. ISO/IEC FDIS 15909-1, Final Draft International Standard, International Organisation for Standardization, February 2004.
45. ITU-T. *Recommendation Z.100: Functional Specification and Description Language (SDL)*. International Telecommunications Union, 2002.
46. ITU-T. *Recommendation X.210, Information Technology - Open Systems Interconnection - Basic Reference Model: Conventions for the Definition of OSI Services*. International Telecommunications Union, Nov. 1993.
47. K. Jensen. Coloured Petri Nets and the Invariant Method. *Theoretical Computer Science*, 14:317–336, 1981.
48. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Springer-Verlag, 2nd edition, 1997.
49. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 2, Analysis Methods*. Springer-Verlag, 2nd edition, 1997.
50. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 3, Practical Use*. Springer-Verlag, 1997.
51. J.B. Jørgensen and K. H. Mortensen. Modelling and Analysis of Distributed Program Execution in BETA Using Coloured Petri Nets. In *Proceedings of ICATPN'96*, volume 1091 of *Lecture Notes in Computer Science*, pages 249–268. Springer-Verlag, 1996.
52. D. E. Knuth. Verification of Link-Level Protocols. *BIT*, 21:31–36, 1981.
53. L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
54. R. Lazic and D. Nowak. *A Unifying Approach to Data-independence*, volume 1877 of *Lecture Notes in Computer Science*, pages 581–595. Springer-Verlag, 2000.
55. L. Liu and J. Billington. Modelling and Analysis of the CES Protocol of H.245. In *Proc. of the Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 95–114, Aarhus, Denmark, 29-31 August 2001.
56. L. Liu and J. Billington. *Tackling the Infinite State Space of a Multimedia Control Protocol Service Specification*, volume 2360 of *Lecture Notes in Computer Science*, pages 273–293. Springer-Verlag, 2002.
57. L. Liu and J. Billington. Obtaining the Service Language for H.245's Multimedia Capability Exchange Signalling Protocol: the Final Step. In *Proc. of the 10th International Multi-Media Modelling Conference*, Brisbane, Australia, 5-7 January 2004.
58. C. Ouyang. *Formal Specification and Verification of the Internet Open Trading Protocol using Coloured Petri Nets*. PhD thesis, School of Electrical and Information Engineering, University of South Australia, Australia, March 2004. submitted.
59. C. Ouyang and J. Billington. On verifying the Internet Open Trading Protocol. In *Proceedings of 4th International Conference on Electronic Commerce and Web Technologies*, volume 2738 of *Lecture Notes in Computer Science*, pages 292–302, Prague, Czech Republic, 1-5 September 2003. Springer-Verlag.
60. C. Ouyang, L. M. Kristensen, and J. Billington. A formal service specification of the Internet Open Trading Protocol. In *Proceedings of 23rd International Conference on Application and Theory of Petri Nets*, volume 2360 of *Lecture Notes in Computer Science*, pages 352–373, Adelaide, Australia, 24-30 June 2002. Springer-Verlag.
61. J. Postel. Internet Protocol - DARPA Internet Program Protocol Specification. RFC 791, IETF, September 1981.

62. J. Postel. Transmission Control Protocol. RFC 793, 1981.
63. W. Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer-Verlag, 1998.
64. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall International Series in Computer Science, 1998.
65. K. Sabnani. An Algorithmic Technique for Protocol Verification. *IEEE Transactions on Communications*, 36(8):924–931, 1988.
66. W. Stallings. *Data and Computer Communications*. Prentice Hall, 6th edition, 2000.
67. Standard ML of New Jersey. <http://cm.bell-labs.com/cm/cs/what/smlnj/>.
68. L.J. Steggle and P. Kosiuczenko. A Timed Rewriting Logic Semantics for SDL: a case study of the Alternating Bit Protocol. *Electronic Notes in Theoretical Computer Science*, 15, 1998.
69. W.R. Stevens. *TCP/IP Illustrated*, volume 1. Addison-Wesley, Reading, MA, November 1994.
70. C.A. Sunshine. Formal Techniques for Protocol Specification and Verification. *IEEE Computer*, pages 346–350, September 1979.
71. I. Suzuki. Formal Analysis of the Alternating Bit Protocol by Temporal Petri Nets. *IEEE Transactions on Software Engineering*, 16(11):1273–1281, 1990.
72. I. Suzuki. Specification and Verification of the Alternating Bit Protocol by Temporal Petri Nets. In *Proceedings of the 32nd Midwest Symposium on Circuits and Systems*, pages 157–160. IEEE Press, 1990.
73. A. Tanenbaum. *Computer Networks*. Prentice Hall, 4th edition, 2003.
74. A. Tokmakoff and J. Billington. An Approach to the Analysis of Interworking Traders. In *Proc. of ICATPN'99*, volume 1639 of *Lecture Notes in Computer Science*, pages 127–146. Springer-Verlag, 1999.
75. R.S. Tomlinson. Selecting sequence numbers. In *Proc. of SIGCOMM/SIGOPS Interprocess Commun. Workshop*, ACM, pages 11–23, 1975.
76. K. J. Turner (Ed.). *Using Formal Description Techniques: An Introduction to Estelle, Lotos and SDL*. Wiley Series in Communication and Distributed Systems. John Wiley & Sons, 1993.
77. A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.
78. M. E. Villapol. *Modelling and Analysis of the Resource Reservation Protocol*. PhD thesis, Electrical and Information Engineering, University of South Australia, Australia, November 2003.
79. M. E. Villapol and J. Billington. Generation of a Service Language for the Resource Reservation Protocol using Formal Methods. In *Proc. INCOSE2001, the 11th Annual International Symposium of the International Council on Systems Engineering*, CD-ROM, Melbourne, Australia, 1-5 July 2001.
80. M. E. Villapol and J. Billington. *Analysing Properties of the Resource Reservation Protocol*, volume 2679 of *Lecture Notes in Computer Science*, pages 377–396. Springer-Verlag, 2003.
81. WAP Forum. <http://www.wapforum.org/>.
82. P. Wolper. Expressing Interesting Properties of Programs in Propositional Temporal Logic. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 184–193. ACM, 1986.
83. J. Xu and J. Kuusela. Analyzing the Execution Architecture of Mobile Phone Software with Coloured Petri Nets. *Int. Journal on Software Tools for Technology Transfer*, 2(2):133–143, 1998.