Int J Softw Tools Technol Transfer (2008) 10:29–56
DOI 10.1007/s10009-007-0050-1

SPECIAL SECTION CPN 06

Analysis of the Datagram Congestion Control Protocol's connection management procedures using the sweep-line method

Somsak Vanit-Anunchai · Jonathan Billington · Guy Edward Gallasch

Published online: 4 September 2007

Abstract State space explosion is a key problem in the analysis of finite state systems. The sweep-line method is a state exploration method which uses a notion of progress to allow states to be deleted from memory when they are no longer required. This reduces the peak number of states that need to be stored, while still exploring the full state space. The technique shows promise but has never achieved reductions greater than about a factor of 10 in the number of states stored in memory for industrially relevant examples. This paper discusses sweep-line analysis of the connection management procedures of a new Internet standard, the Datagram Congestion Control Protocol (DCCP). As the intuitive approaches to sweep-line analysis are not effective, we introduce new variables to track progress. This creates further state explosion. However, when used with the sweep-line, the peak number of states is reduced by over two orders of magnitude compared with the original. Importantly, this allows DCCP to be analysed for larger parameter values.

Somsak Vanit-Anunchai was partially supported by an Australian Research Council Discovery Grant (DP0559927) and Suranaree University of Technology.
Guy Edward Gallasch was supported by an Australian Research Council Discovery Grant (DP0559927).

S. Vanit-Anunchai (✉) · J. Billington · G. E. Gallasch
Computer Systems Engineering Centre,
School of Electrical and Information Engineering,
University of South Australia,
Mawson Lakes Campus, SA 5095, Australia
e-mail: somsak.vanit-anunchai@postgrads.unisa.edu.au

J. Billington
e-mail: jonathan.billington@unisa.edu.au

G. E. Gallasch
e-mail: guy.gallasch@unisa.edu.au

Keywords State space methods · Sweep-line · DCCP · Coloured Petri Nets · State explosion

1 Introduction

The state space method is one of the main approaches for formally analysing and verifying the behaviour of concurrent and distributed systems. In essence, this method generates all or part of the reachable states of the system. After the state space is generated, many analysis and verification questions about the system's behaviour (such as "does the system deadlock or livelock?"), can be answered. Unlike theorem proving, state space analysis tools are easier to use because they involve less complex mathematics that is often hidden by automatic computer tools, and they can provide counter examples for debugging purposes. Despite these advantages, the state space approach suffers from the well-known *state explosion problem* [31]. Even relatively small systems can generate state spaces that cannot be stored in computer memory. Thus many attempts have been made to alleviate this problem. An excellent overview and literature review about the state explosion problem is given in [31]. The attempts to alleviate state explosion fall into three broad classes. The first considers methods that represent the state space in a condensed or compact form, such as symmetry reduction [4,14]. The second class restricts state space exploration to a subset of the reachable states and includes partial order methods [29,38] such as stubborn sets [30]. The third class involves deleting or throwing away states or state information on-the-fly during state space exploration. It includes methods such as bit-state hashing [13,39], state space caching [10,12], the sweep-line method [2,23] and the pseudo-root technique [28]. To further reduce the state space, techniques from the different classes have also been combined, e.g. sweep-line and equivalence [1].

The sweep-line method exploits a notion of *progress* exhibited in the system being analysed. *Progress mappings* are defined by the user, to map the states of the system into a set of ordered progress values. Based on progress values, this method deletes old states from memory by reasoning that states with a lower progress value will not (or are unlikely to) be reached from states with a higher progress value. During exploration, a number of system properties, such as absence of deadlocks, can be verified on-the-fly.

In the area of formal analysis and verification of computer protocols that recover from loss using retransmissions, the maximum number of retransmissions of messages plays an important role in state explosion. When the maximum number of retransmissions increases, the size of the state space tends to increase rapidly [5]. When using a small maximum number of retransmissions, errors tend to reveal themselves quickly, but this does not guarantee that the system is error free for larger values of the maximum number of retransmissions. Thus it is necessary to extend state space analysis to include the highest maximum number of retransmissions possible, up to the limit specified by the protocol's specification. Previously we have used the sweep-line method to verify termination properties of Coloured Petri Net (CPN) [15,16,25] models of various protocols such as the Wireless Transaction Protocol (WTP) [11], the Internet Open Trading Protocol (IOTP) [8], and the Transmission Control Protocol [6].

The Datagram Congestion Control Protocol (DCCP), specified in Request For Comments (RFC) 4340 [19], is a new transport protocol proposed by the Internet Engineering Task Force (IETF). The protocol is designed to support various kinds of congestion control mechanisms used by different delay sensitive applications. We use CPNs to build and analyse a formal executable model of DCCP's connection management procedures according to RFC 4340 [19]. Our analysis [33] shows that DCCP connection establishment can fail when sequence numbers wrap.[1] However, state explosion limits our analysis to a maximum of only one retransmission. We need to extend the analysis to cover two retransmissions and to determine two properties: whether the undesired deadlocks are still present; and whether any new errors have emerged as a result of the additional retransmissions.

Instead of applying the more conventional approach of trying to reduce the size of the state space, we induce state space explosion by introducing new state variables into the *specification model* in order to capture additional information during model execution. By carefully selecting new state variables, the state space of the *augmented model* has a structure which facilitates far more efficient sweep-line analysis when compared with the state space of the specification model.

This paper is a revised version of [36]. It has been expanded to include the full CPN model of DCCP's connection management procedures. The contribution of this paper is threefold. Firstly, the paper presents the first complete formal specification of the connection management procedures of RFC 4340 at a level of detail sufficient for analysing all the main procedures. Building the CPN model has allowed us to remove ambiguities so that the risk of misinterpretation is very low. The CPN specification has the additional benefit that all relevant aspects of the procedures that are defined in different parts of the RFC have been brought together. Secondly, this paper provides insight into how an effective progress mapping for the DCCP connection management CPN model is derived. This is important because an effective progress mapping is key to the performance of the sweep-line method. Thirdly, we apply sweep-line analysis to the augmented model and demonstrate that the number of peak states stored, compared with sweep-line analysis of the specification model, can be significantly reduced. This has allowed us to extend the analysis of DCCP connection establishment to scenarios which could not be obtained with conventional analysis.

This paper is organised as follows. Section 2 briefly describes DCCP's connection management procedures which are then formalised in Sect. 3. Section 4 identifies sources of progress and derives a progress mapping for our augmented model of DCCP's connection management procedures. The analysis results obtained for scenarios where sequence numbers wrap are discussed in Sect. 5. Finally, Sect. 6 presents our conclusions and future work. We assume some familiarity with CPNs [15,16,25].

2 Overview of DCCP's connection management procedure

DCCP connection management has one connection establishment procedure and five closing procedures. These procedures are defined in [19] with the state diagram shown in Fig. 1. DCCP uses eight packet types: Request, Response, Ack, DataAck, Data, CloseReq, Close and Reset to set up, transmit data and close down the connection. Two packet types, Sync and SyncAck, are used for synchronization procedures. Figure 2 shows the typical procedure for connection set up (Fig. 2a) and close down (Fig. 2b).

In brief, a connection is initiated by an application at the client issuing an "active open" command. (We assume that the application at the server has issued a "passive open" command.) After receiving the "active open" the client sends a DCCP-Request packet to the server to initialise sequence numbers, and enters the REQUEST state. The server replies

[1] Sequence number wrap occurs when the sequence number rolls over the maximum sequence number to zero.

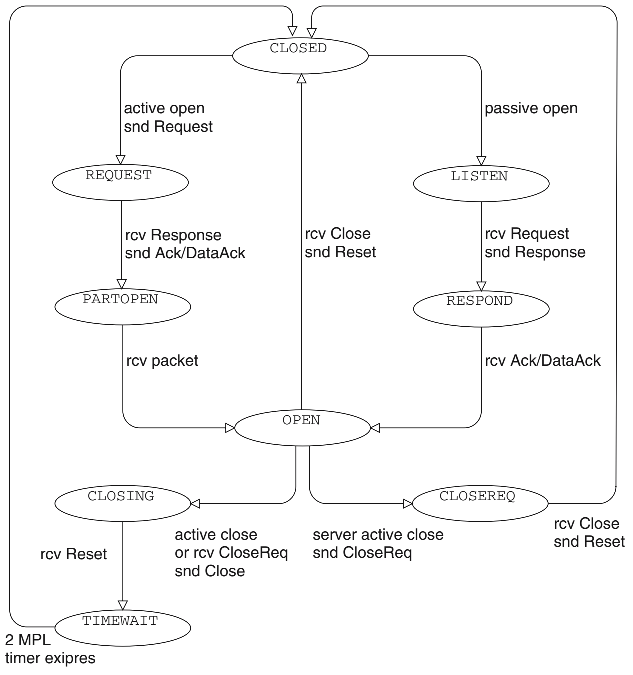


Fig. 1 DCCP state diagram redrawn from [19]

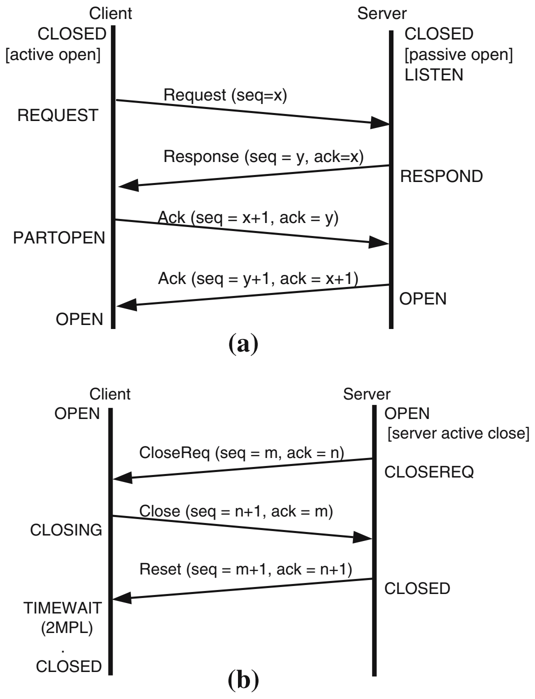


Fig. 2 Typical connection establishment and release scenarios

with a DCCP-Response packet, indicating that it is willing to communicate with the client and acknowledging the DCCP-Request. The client sends a DCCP-Ack (or DCCP-DataAck) packet to acknowledge the DCCP-Response packet and enters the PARTOPEN state. The server acknowledges the receipt of the DCCP-Ack, enters the OPEN state and is

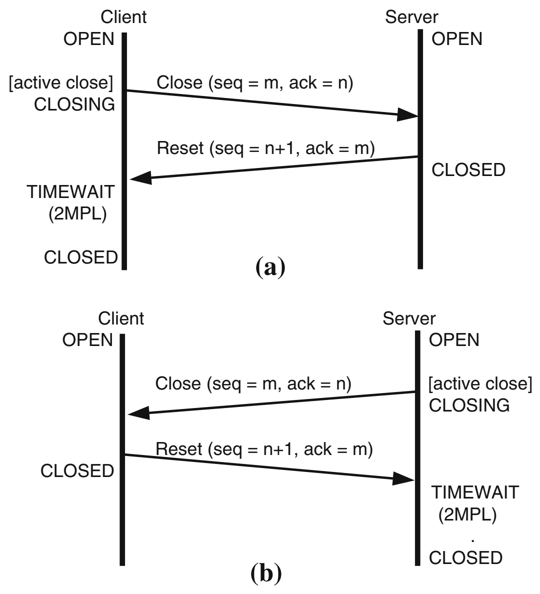


Fig. 3 Alternative close down procedures

ready for data transfer. Upon receipt of a DCCP-Ack (or DCCP-Data, DCCP-DataAck or DCCP-SyncAck) packet, the client also enters OPEN and is now also ready for data transfer.

For connection close down, the application at the server issues a "server active close" command. The server sends a DCCP-CloseReq and enters the CLOSEREQ state. The client, upon receiving the DCCP-CloseReq, enters CLOSING and generates a DCCP-Close packet in response. After the server receives the DCCP-Close packet, it responds with a DCCP-Reset packet and enters the CLOSED state. When the client receives the DCCP-Reset packet, it holds the TIMEWAIT state for 2 maximum packet lifetimes (2MPL) before also entering the CLOSED state.

Alternatively, either side may send a DCCP-Close packet to close the connection when receiving an "active close" command from the application. The end that sends the DCCP-Close packet will hold the TIMEWAIT state as shown in Fig. 3. Beside these three closing procedures, there are another two possible scenarios concerned with simultaneous closing. The first procedure is invoked when both users issue an "active close". The second occurs when the client user issues an "active close" and the application at the server issues the "server active close" command. For a detailed description of the connection set up and close down procedures, see [19].

During the connection, DCCP entities maintain a set of variables. In addition to the state and timers, the important variables are Greatest Sequence Number Sent (GSS), Greatest Sequence Number Received (GSR) and Greatest Acknowledgement Number Received (GAR). The Initial Sequence Numbers Sent and Received (ISS and ISR), the Valid Sequence Number window width (W) and the

Acknowledgement Number validity window width (AW) are also important parameters for a connection. Based on these parameters and the state variables, the valid sequence and acknowledgement number intervals are defined by Sequence Number Window Low and High [SWL,SWH], and Acknowledgement Number Window Low and High [AWL,AWH] [19].

3 CPN model of DCCP connection management

We have followed the development of DCCP Connection Management (DCCP-CM) since the publication of Internet Draft version 5 [21] in 2003. Design/CPN [27] has been used to build and maintain our DCCP connection management CPN models, with the aim of detecting errors or deficiencies in the protocol procedures. We analysed the connection establishment procedures of version 5 and discovered a deadlock [32]. When version 6 [17] was released, we revised our CPN model to reflect the changes and found similar deadlocks [22]. Version 11 of the DCCP specification was submitted to IETF for approval as a standard. We updated our model accordingly and included the synchronization mechanism. We found that the deadlocks were removed but we discovered a serious problem known as *chatter*[2] [34,37]. On updating the CPN model to RFC 4340, we investigated the connection establishment procedure when sequence numbers wrap [33] and, as indicated in the introduction, found that the attempt to set up the connection can fail. In [7,9] we defined the DCCP service and confirmed that the sequence of user observable events of the protocol conformed to its service. Most recently [35] we have reported our experience with the incremental enhancement and iterative modelling of the connection management procedures as the DCCP specification was developed. However, none of these papers have presented the full CPN model for RFC 4340, with [35] only illustrating how the specification has evolved and [33] just including results.

Thus the purpose of this section is to present the full CPN specification of the connection management procedures. We include the structure of the specification in a CPN hierarchy diagram, all of the CPN pages and some of the declarations in the rest of this section. The complete declarations including related ML functions can be found in Appendix. Insight into the decisions behind the modelling choices can be found in [35] and are thus not included, as the emphasis of this paper is (a) to provide a complete formal specification that captures DCCP's connection management and synchronization procedures and (b) to allow the development of the progress mappings for the sweep-line to be understood.

3.1 Modelling assumptions

RFC 4340 specifies a number of protocol mechanisms for DCCP: reliable connection management, synchronization of sequence numbers, reliable feature negotiation and acknowledgement/optional mechanisms. The acknowledgement/optional mechanisms are normally used during data transfer together with a procedure for congestion control which is determined using the feature negotiation proocedures. Different congestion control procedures are (or will be) specified in other RFCs (e.g. RFC 4341, RFC 4342). Our CPN model captures all of DCCP's procedures associated with connection set up, close down and synchronization as specified in RFC 4340 but does not include the data transfer and feature negotiation procedures. A DCCP packet is modelled by its packet type, and long (or short) sequence and acknowledgement numbers. Other fields in the DCCP header are omitted because they do not affect the operation of the connection management procedure. Malicious attacks are not considered. In this paper we only discuss the case when the communication channels can delay and reorder packets without loss.

3.2 Model structure

The DCCP Connection management (DCCP-CM) CPN model comprises four hierarchical levels shown in Fig. 4. It has a total of 6 *places*, 52 *executable transitions*, 15 *substitution transitions* and 18 *ML functions*. The top level page named DCCP#1 is the DCCP overview page shown in Fig. 5. Two *substitution transitions* DCCP_C and DCCP_S (rectangles with the HS tag) in Fig. 5 represent the client and the server. Both are linked to the second level page named DCCP_CM.

The DCCP_CM page comprises eight substitution transitions shown in Fig. 6. Each substitution transition is linked to the third level page. UserCommands models the actions taken when receiving commands from users. The IdleState page combines similar processing actions required in the CLOSED, LISTEN and TIMEWAIT states. The Request, Respond and PartOpen pages define the major processing actions in each of the corresponding states. The actions taken when receiving the DCCP-Data, DCCP-Ack and DCCP-DataAck packets in the OPEN, CLOSEREQ and CLOSING states are modelled in the DataTransfer page. The ClosingDown page defines the procedures undertaken on receiving a DCCP-Close in the states: RESPOND, PARTOPEN, OPEN, CLOSEREQ and CLOSING and when the client receives a DCCP-CloseReq in the PARTOPEN, OPEN and CLOSING states. Finally, CommonProcessing shown in Fig. 7

[2] The chatter scenario comprised undesired interactions between Reset and Sync packets that could involve long but finite exchanges of these packets where no progress was made, until finally the system corrected itself.

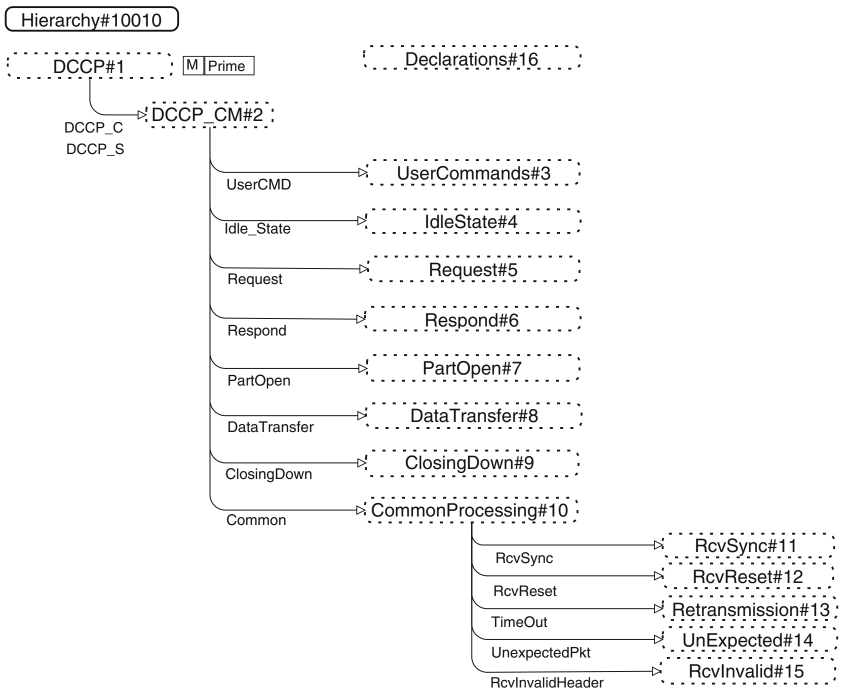


Fig. 4 The DCCP hierarchy page

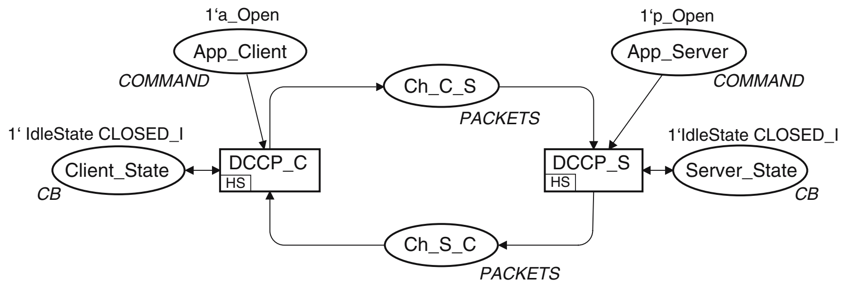


Fig. 5 The DCCP overview page

comprises five substitution transitions which are linked to fourth level pages. They model the procedures that are common to various states including: receiving DCCP-Sync and DCCP-SyncAck packets, the receipt of DCCP-Reset packets, retransmissions and timer expiry and processing unexpected packets and those with an invalid header.

3.3 The DCCP overview page

The top level page (DCCP#1) shown in Fig. 5 comprises two substitution transitions (for the client and server as mentioned above) and six places. There are two places that allow communication with the applications that use DCCP, one for the client, App_Client and the other for the server, App_Server. The states of the client and server protocol entities are stored in the places Client_State and Server_State. The client and server communicate via two channel places, Ch_C_S and Ch_S_C, shown in the middle of Fig. 5. Each models a unidirectional reordering channel, from the client to the server and the server to the client respectively.

Each place has a type (or colour set) which is defined in a set of declarations. The declarations define the data structures and any associated variables and functions used in the model. The declarations are written in CPN ML [3], a variant of Standard ML [26].

In Fig. 5, the channel places, Ch_C_S and Ch_S_C, are typed by *PACKETS*. Figure 8 defines *PACKETS* (lines 27–30) as the union of four colour sets: Type1LongPkt, Type2 LongPkt, Type1ShortPkt and Type2ShortPkt. Long sequence numbers (SN48) in line 15 are represented using the ML *infinite integer* type and range from zero to $2^{48} - 1$. Short sequence numbers (SN24) in line 17 are represented by integers ranging from zero to $2^{24} - 1$. Each packet (lines 21–26) is defined by a product comprising the Packet Type (lines 3–9), X (Extended Sequence Number bit, called 'X' in [19], line 12) and the sequence number (or a record of sequence and acknowledgement numbers). Short sequence numbers are allowed only for DCCP-Data, DCCP-Ack and DCCP-DataAck packets. Thus line 7 defines a packet type for DCCP-Data with short sequence numbers, while lines 8–9 define a different packet type for DCCP-Ack and DCCP-DataAck

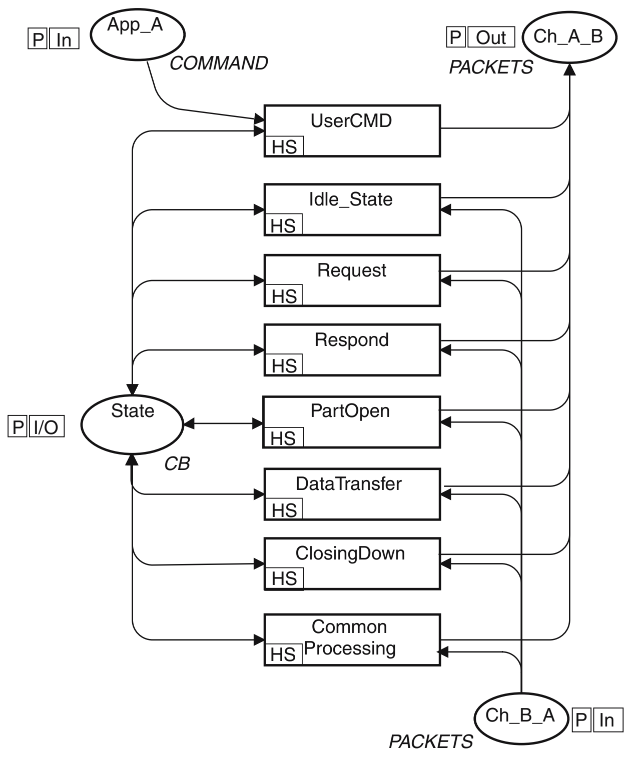


Fig. 6 DCCP_CM page

```
(* Define the Packet Structure *)
(* Packet Types *)
color PktType1 = with Request | Data;
color PktType2 = with Sync | SyncAck | Response
                      | Ack | DataAck | CloseReq
                      | Close | Rst;
color DATA = subset PktType1 with [Data];
color ACK_DATAACK = subset PktType2
                      with [Ack, DataAck];

 (* Extended Sequence Number Bit*)
 color X = with LONG | SHORT;

 (* Sequence and Acknowledgement Number (Long/Short) *)
 color SN48 = IntInf with ZERO..MaxSeqNo48;
 color SN48_AN48 = record SEQ:SN48*ACK:SN48;
 color SN24 = int with 0..max_seq_no24;
 color SN24_AN24 = record SEQ:SN24*ACK:SN24;

 (* Four Kinds of Packet *)
 color Type1LongPkt = product PktType1*X*SN48;
 color Type2LongPkt = product PktType2*X
                              *SN48_AN48;
 color Type1ShortPkt= product DATA*X*SN24;
 color Type2ShortPkt= product ACK_DATAACK*X
                              *SN24_AN24;
 color PACKETS = union PKT1:Type1LongPkt
                     + PKT2:Type2LongPkt
                     + PKT1s:Type1ShortPkt
                     + PKT2s:Type2ShortPkt;
```

Fig. 8 Definition of DCCP PACKETS

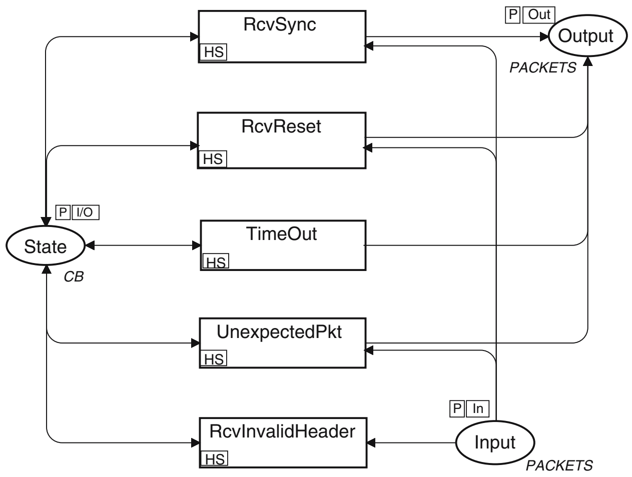


Fig. 7 CommonProcessing page

```
 1: (* DCCP variables: Counter; Greatest Sequence Numbers;
 2:    Initial Sequence Numbers *)
 3: color RCNT = int; (* Retransmission Counter *)
 4: color GS = record GSS:SN48*GSR:SN48*GAR:SN48;
 5: color ISN = record ISS:SN48*ISR:SN48;
 6:
 7: (* Major States *)
 8: color IDLE = with CLOSED_I | LISTEN
 9:                | CLOSED_F | TIMEWAIT;
10: color REQUEST = product RCNT*SN48*SN48;
11: color ACTIVE = with RESPOND_tmp | RESPOND
12:            | PARTOPEN | S_OPEN | C_OPEN
13:            | CLOSEREQ | C_CLOSING |S_CLOSING;
14: color ACTIVExRCNTxGSxISN = product ACTIVE*RCNT
15:                                   *GS*ISN;
16:
17: (* DCCP's Control Block  *)
18: color CB = union IdleState:IDLE+ReqState:REQUEST
19:              + ActiveState:ACTIVExRCNTxGSxISN;
20: (* Application Commands *)
21: color COMMAND = with p_Open | a_Open
22:            | server_a_Close | a_Close;
```

Fig. 9 DCCP's control block and user commands

because they also include acknowledgements. Strong typing of packets is very useful for developing and debugging the model.

DCCP states and state variables are stored in a structure called the *Control Block*. The places Client_State and Server_State in Fig. 5, are thus typed by CB, for Control Block. We classify DCCP states into three groups according to their functional behaviour: idle, request and active states. The differences are mainly related to how an entity responds to DCCP-Reset and DCCP-Sync packets. When in the CLOSED, LISTEN and TIMEWAIT states, the GSS, GSR, GAR, ISS and ISR state variables do not exist, while the client in the REQUEST state has only GSS and ISS instantiated. Thus we define each group with a different set of state variables.

Figure 9 defines CB (lines 18–19) as a union of three colour sets: IDLE, REQUEST and ACTIVExRCNTx-GSxISN. IDLE (lines 8–9 of Fig. 9) defines three idle states: CLOSED, LISTEN and TIMEWAIT. The CLOSED state is split into CLOSED_I to represent the initial CLOSED state

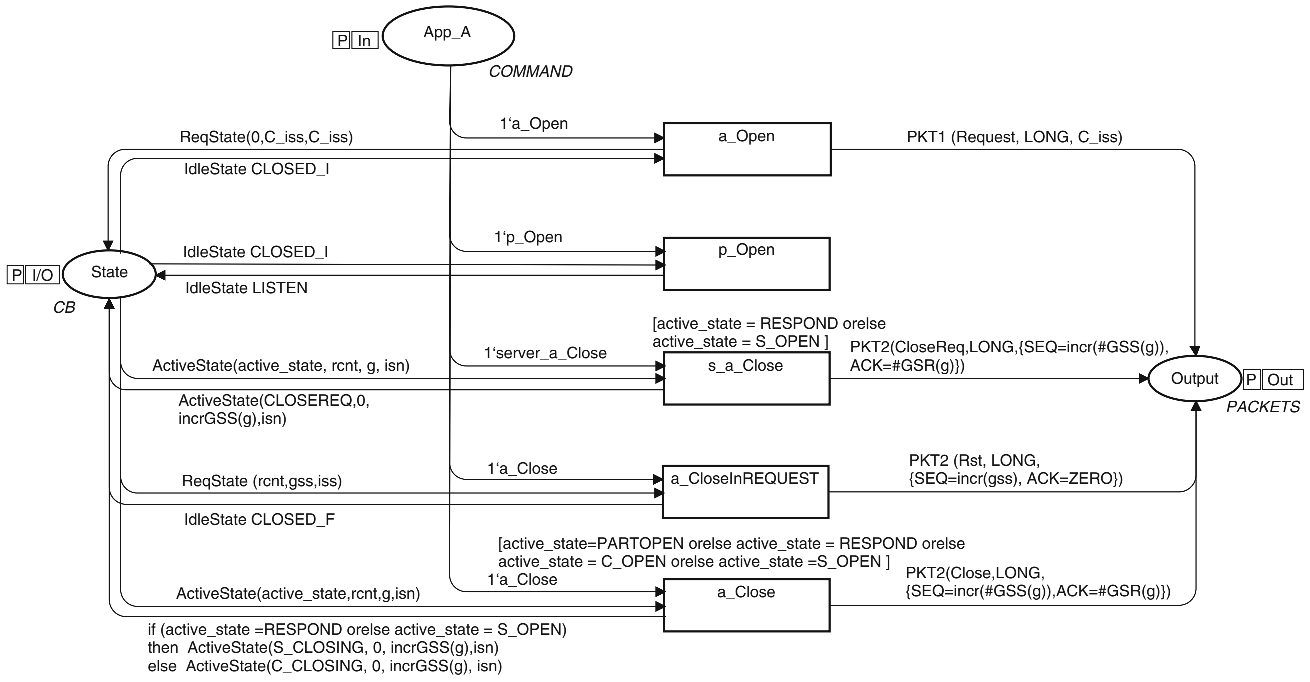

App_A
P In
COMMAND
1'a_Open
a_Open
ReqState(0,C_iss,C_iss)
IdleState CLOSED_I
PKT1 (Request, LONG, C_iss)
1'p_Open
p_Open
IdleState CLOSED_I
IdleState LISTEN
State
P I/O
CB
[active_state = RESPOND orelse active_state = S_OPEN]
1'server_a_Close
s_a_Close
ActiveState(active_state, rcnt, g, isn)
ActiveState(CLOSEREQ,0, incrGSS(g),isn)
PKT2(CloseReq,LONG,{SEQ=incr(#GSS(g)), ACK=#GSR(g)})
Output
P Out
PACKETS
1'a_Close
a_CloseInREQUEST
ReqState (rcnt,gss,iss)
IdleState CLOSED_F
PKT2 (Rst, LONG, {SEQ=incr(gss), ACK=ZERO})
[active_state=PARTOPEN orelse active_state = RESPOND orelse active_state = C_OPEN orelse active_state =S_OPEN]
1'a_Close
a_Close
ActiveState(active_state,rcnt,g,isn)
PKT2(Close,LONG, {SEQ=incr(#GSS(g)),ACK=#GSR(g)})
if (active_state =RESPOND orelse active_state = S_OPEN) then ActiveState(S_CLOSING, 0, incrGSS(g),isn) else ActiveState(C_CLOSING, 0, incrGSS(g), isn)

Fig. 10 The UserCommands page

and CLOSED_F to represent the final CLOSED state. Differentiating the initial CLOSED state from the final CLOSED state helps increase the effectiveness of the sweep-line method when using the client and server states as a measure of progress. Because we consider only one connection instance, splitting the CLOSED state into CLOSED_I and CLOSED_F does not affect the protocol's behaviour. The colour set REQUEST (line 10) is a product comprising RCNT (Retransmission Counter, line 3), GSS and ISS. Because there is only one state in this group, the REQUEST state is already distinguished from other states by using the ML selector, ReqState, in the union defined in lines 18–19. ACTIVE (lines 11–13) defines five DCCP states: RESPOND, PARTOPEN, OPEN, CLOSEREQ and CLOSING. Because the client and server respond to the CloseReq packet differently in the OPEN and CLOSING states, we differentiate these states for the client and server by prefixing them by "C" for the client; and "S" for the server. This allows us to use DCCP_CM as a page instance for the client and server, greatly reducing redundancy in the model. This was of great benefit when upgrading the model as DCCP was developed. ACTIVExRCNTxGSxISN (line 14) is a product comprising ACTIVE (lines 11–13), RCNT, GS (Greatest Sequence Numbers and Greatest Acknowledgement Numbers, line 4) and ISN (Initial Sequence Numbers, line 5).

The places App_Client and App_Server at the top of Fig. 5, typed by COMMAND, model DCCP user commands (i.e., commands that can be issued by the applications that use DCCP). Figure 9 defines a colour set COMMAND on lines 21–22. COMMAND comprises user commands for opening and closing connections.

The distribution of tokens in the places of a CPN is called a *marking* and represents a state of the system. The initial marking represents the initial state. In Fig. 5, the initial markings are written above each place. Client_State and Server_State have an initial marking of one CLOSED_I idle state token. The user command 1'a_Open is the initial marking of App_Client indicating that the client's application desires to open a connection. Similarly, the initial marking of App_Server is 1'p_Open requesting the server to enter the LISTEN state so that it is ready to receive requests from clients. The initial markings of Ch_C_S and Ch_S_C are empty, representing that no packets are currently in either channel.

3.4 The third and fourth level pages

The UserCommands page in Fig. 10 specifies the actions when the entity receives commands from its application. Transition a_Open models sending a Request when the client receives an active open command in CLOSED. Transition p_Open models the server entering the LISTEN state on receiving a passive open command in CLOSED. When the server receives a server active close command in the RESPOND or OPEN state, it sends a CloseReq packet and enters CLOSEREQ. This is modelled by transition s_a_Close. Transitions a_CloseInREQUEST and a_Close model the actions taken when the entity receives an

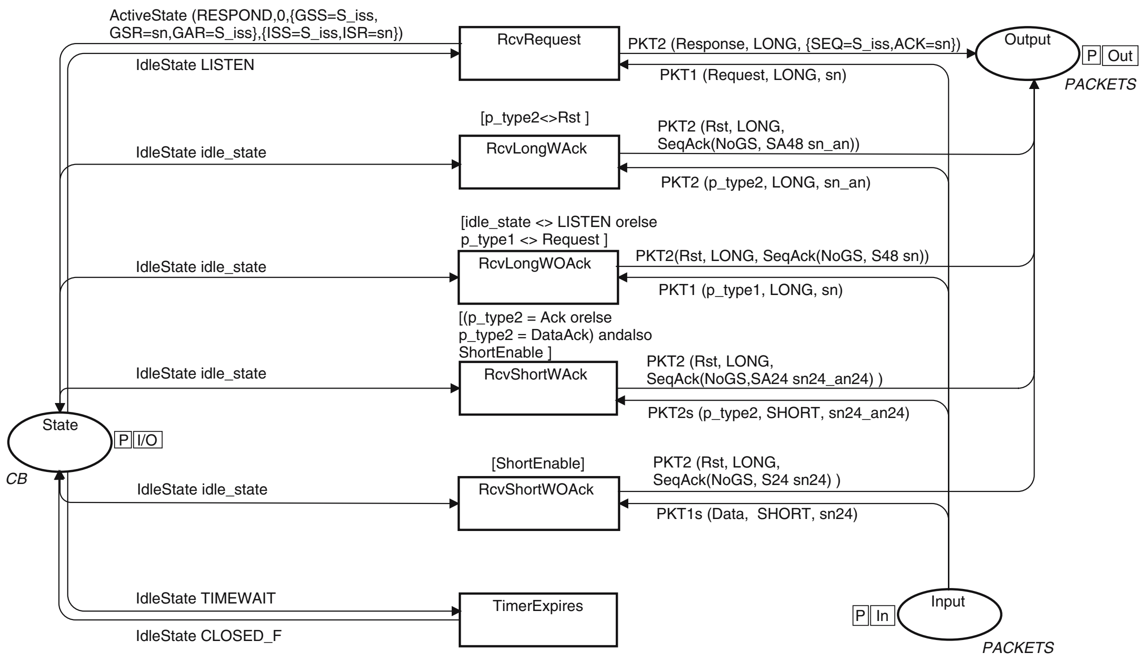

ActiveState (RESPOND,0,{GSS=S_iss, GSR=sn,GAR=S_iss},{ISS=S_iss,ISR=sn})
IdleState LISTEN
RcvRequest
PKT2 (Response, LONG, {SEQ=S_iss,ACK=sn})
PKT1 (Request, LONG, sn)
Output
P Out
PACKETS
[p_type2<>Rst]
RcvLongWAck
IdleState idle_state
PKT2 (Rst, LONG, SeqAck(NoGS, SA48 sn_an))
PKT2 (p_type2, LONG, sn_an)
[idle_state <> LISTEN orelse p_type1 <> Request]
RcvLongWOAck
IdleState idle_state
PKT2(Rst, LONG, SeqAck(NoGS, S48 sn))
PKT1 (p_type1, LONG, sn)
[(p_type2 = Ack orelse p_type2 = DataAck) andalso ShortEnable]
RcvShortWAck
IdleState idle_state
PKT2 (Rst, LONG, SeqAck(NoGS,SA24 sn24_an24))
PKT2s (p_type2, SHORT, sn24_an24)
State
P I/O
CB
[ShortEnable]
RcvShortWOAck
IdleState idle_state
PKT2 (Rst, LONG, SeqAck(NoGS, S24 sn24))
PKT1s (Data, SHORT, sn24)
IdleState TIMEWAIT
TimerExpires
IdleState CLOSED_F
Input
P In
PACKETS

Fig. 11 The IdleState page

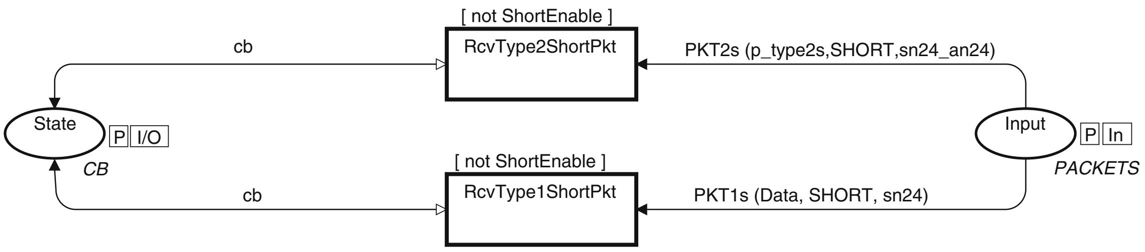


Fig. 12 The RcvInvalid page

active close command in the REQUEST, PARTOPEN, RESPOND and OPEN states.

The IdleState page in Fig. 11 specifies the actions of the entity when it is in CLOSED, LISTEN or TIMEWAIT. The transition named RcvRequest models the server receiving a Request in the LISTEN state. It initializes all state variables, enters the RESPOND state and replies with a Response packet. TimerExpires abstractly models the maximum packet lifetime timer expiring in the TIMEWAIT state.

The remaining four transitions model that when any other packets, including Sync and SyncAck, are received (in any idle state), the entity responds with a Reset. Because no sequence number variables (GSS, GSR and GAR) exist in an idle state, according to Sect. 8.3.1 of [19], the acknowledgement number of the Reset being sent is equal to the sequence number of the received packet. The sequence number of the Reset is equal to the acknowledgement number of the received packet plus one. If the received packet has no acknowledgement number, the sequence number of the Reset is equal to zero. These requirements are captured by the function SeqAck which is used to calculate the sequence and acknowledgement numbers of the Reset packet.

ShortEnable in a guard's expression (e.g. see transition RcvShortWOAck) disables or enables the use of short sequence numbers. If short sequence numbers are not allowed, a packet received with a short sequence number is discarded. This behaviour is captured in the RcvInvalid page shown in Fig. 12.

The Request page. In the REQUEST state only Response and Reset packets are expected. Upon receiving a valid Response packet, the client replies with either an Ack or a DataAck packet, initializes ISR, GSR and GAR and enters the PARTOPEN state. These actions are represented by the transitio RcvResponseSndAck in Fig. 13. Because there is no record of ISR and GSR in REQUEST, only the acknowledgment number in the Response can be validated according to step 4 of the pseudo code in Sect. 8.5 of [19]. This action is modelled by function AckValid. If short sequence numbers are allowed, the outgoing packet could include either a long or a short sequence number. Two variables, ack_dataack and

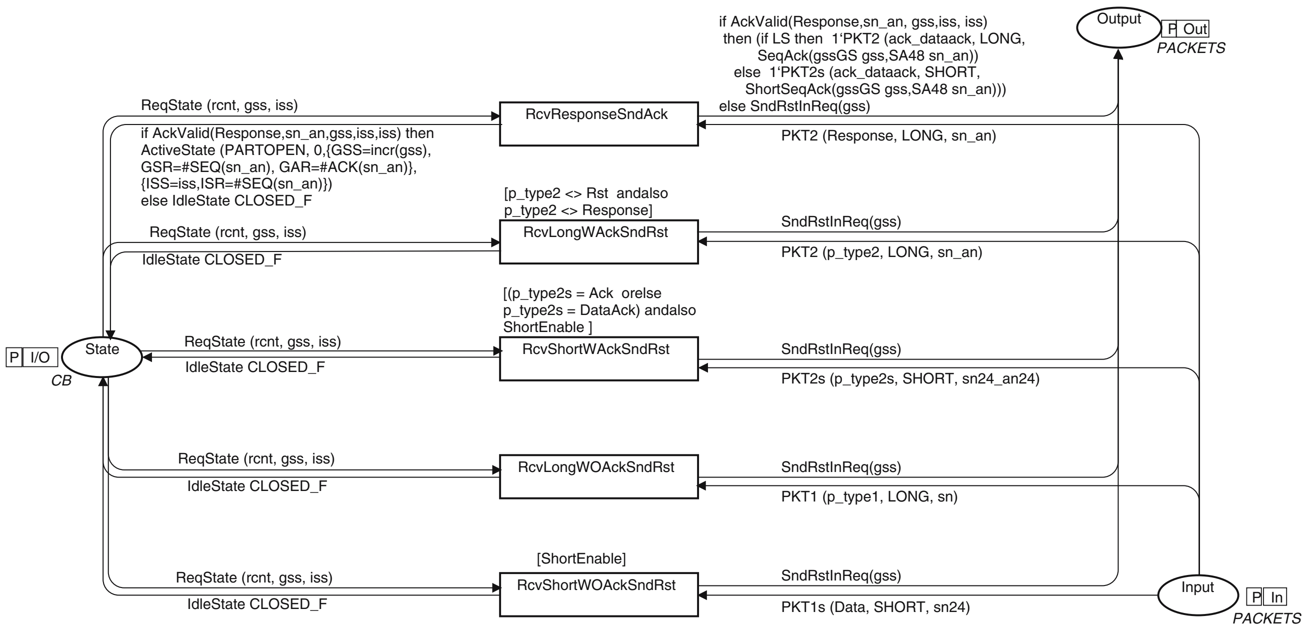

if AckValid(Response,sn_an, gss,iss, iss)
then (if LS then 1'PKT2 (ack_dataack, LONG,
SeqAck(gssGS gss,SA48 sn_an))
else 1'PKT2s (ack_dataack, SHORT,
ShortSeqAck(gssGS gss,SA48 sn_an)))
else SndRstInReq(gss)
Output
P Out
PACKETS
ReqState (rcnt, gss, iss)
RcvResponseSndAck
if AckValid(Response,sn_an,gss,iss,iss) then
ActiveState (PARTOPEN, 0,{GSS=incr(gss),
GSR=#SEQ(sn_an), GAR=#ACK(sn_an)},
{ISS=iss,ISR=#SEQ(sn_an)})
else IdleState CLOSED_F
PKT2 (Response, LONG, sn_an)
[p_type2 <> Rst andalso
p_type2 <> Response]
ReqState (rcnt, gss, iss)
RcvLongWAckSndRst
SndRstInReq(gss)
IdleState CLOSED_F
PKT2 (p_type2, LONG, sn_an)
[(p_type2s = Ack orelse
p_type2s = DataAck) andalso
ShortEnable]
P I/O
State
CB
ReqState (rcnt, gss, iss)
RcvShortWAckSndRst
SndRstInReq(gss)
IdleState CLOSED_F
PKT2s (p_type2s, SHORT, sn24_an24)
ReqState (rcnt, gss, iss)
RcvLongWOAckSndRst
SndRstInReq(gss)
IdleState CLOSED_F
PKT1 (p_type1, LONG, sn)
[ShortEnable]
ReqState (rcnt, gss, iss)
RcvShortWOAckSndRst
SndRstInReq(gss)
Input
P In
IdleState CLOSED_F
PKT1s (Data, SHORT, sn24)
PACKETS

Fig. 13 The Request page

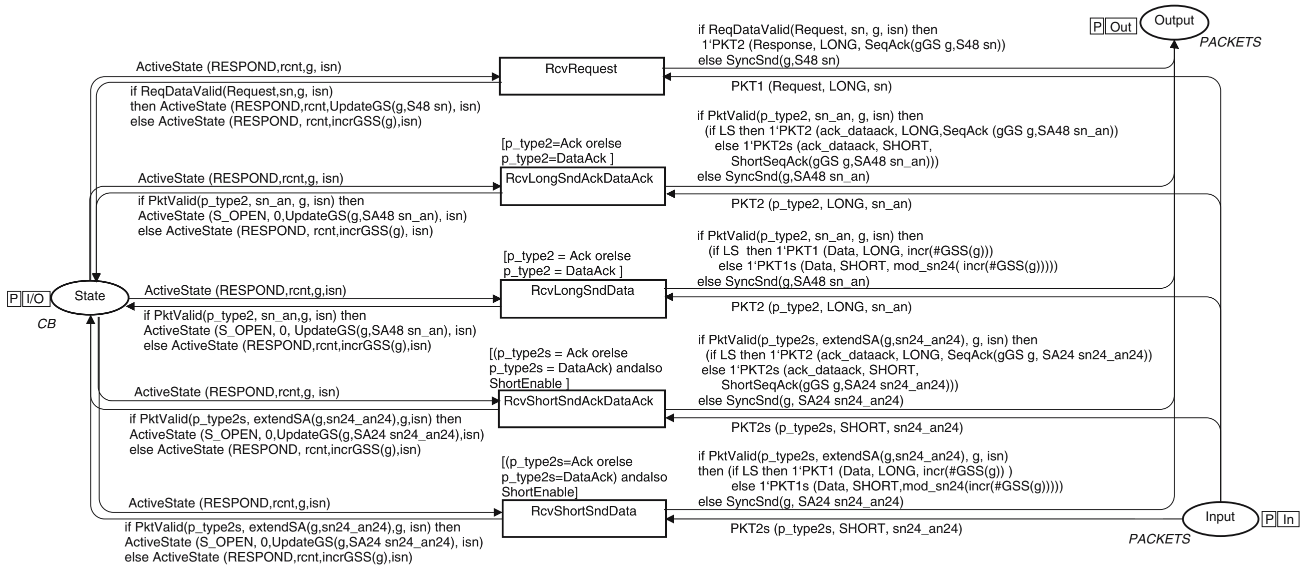

if ReqDataValid(Request, sn, g, isn) then
1'PKT2 (Response, LONG, SeqAck(gGS g,S48 sn))
else SyncSnd(g,S48 sn)
P Out
Output
PACKETS
ActiveState (RESPOND,rcnt,g, isn)
RcvRequest
if ReqDataValid(Request,sn,g, isn)
then ActiveState (RESPOND,rcnt,UpdateGS(g,S48 sn), isn)
else ActiveState (RESPOND, rcnt,incrGSS(g),isn)
PKT1 (Request, LONG, sn)
if PktValid(p_type2, sn_an, g, isn) then
(if LS then 1'PKT2 (ack_dataack, LONG,SeqAck (gGS g,SA48 sn_an))
else 1'PKT2s (ack_dataack, SHORT,
ShortSeqAck(gGS g,SA48 sn_an)))
else SyncSnd(g,SA48 sn_an)
[p_type2=Ack orelse
p_type2=DataAck]
ActiveState (RESPOND,rcnt,g, isn)
RcvLongSndAckDataAck
if PktValid(p_type2, sn_an, g, isn) then
ActiveState (S_OPEN, 0,UpdateGS(g,SA48 sn_an), isn)
else ActiveState (RESPOND, rcnt,incrGSS(g), isn)
PKT2 (p_type2, LONG, sn_an)
if PktValid(p_type2, sn_an, g, isn) then
(if LS then 1'PKT1 (Data, LONG, incr(#GSS(g)))
else 1'PKT1s (Data, SHORT, mod_sn24(incr(#GSS(g)))))
else SyncSnd(g,SA48 sn_an)
[p_type2 = Ack orelse
p_type2 = DataAck]
P I/O
State
CB
ActiveState (RESPOND,rcnt,g,isn)
RcvLongSndData
if PktValid(p_type2, sn_an,g, isn) then
ActiveState (S_OPEN, 0, UpdateGS(g,SA48 sn_an), isn)
else ActiveState (RESPOND,rcnt,incrGSS(g),isn)
PKT2 (p_type2, LONG, sn_an)
if PktValid(p_type2s, extendSA(g,sn24_an24), g, isn) then
(if LS then 1'PKT2 (ack_dataack, LONG, SeqAck(gGS g, SA24 sn24_an24))
else 1'PKT2s (ack_dataack, SHORT,
ShortSeqAck(gGS g,SA24 sn24_an24)))
else SyncSnd(g, SA24 sn24_an24)
[(p_type2s = Ack orelse
p_type2s = DataAck) andalso
ShortEnable]
ActiveState (RESPOND,rcnt,g, isn)
RcvShortSndAckDataAck
if PktValid(p_type2s, extendSA(g,sn24_an24),g,isn) then
ActiveState (S_OPEN, 0,UpdateGS(g,SA24 sn24_an24),isn)
else ActiveState (RESPOND, rcnt,incrGSS(g),isn)
PKT2s (p_type2s, SHORT, sn24_an24)
if PktValid(p_type2s, extendSA(g,sn24_an24), g, isn)
then (if LS then 1'PKT1 (Data, LONG, incr(#GSS(g)))
else 1'PKT1s (Data, SHORT,mod_sn24(incr(#GSS(g)))))
else SyncSnd(g, SA24 sn24_an24)
[(p_type2s=Ack orelse
p_type2s=DataAck) andalso
ShortEnable]
ActiveState (RESPOND,rcnt,g,isn)
RcvShortSndData
Input
P In
if PktValid(p_type2s, extendSA(g,sn24_an24),g, isn) then
ActiveState (S_OPEN, 0,UpdateGS(g,SA24 sn24_an24), isn)
else ActiveState (RESPOND,rcnt,incrGSS(g),isn)
PKT2s (p_type2s, SHORT, sn24_an24)
PACKETS

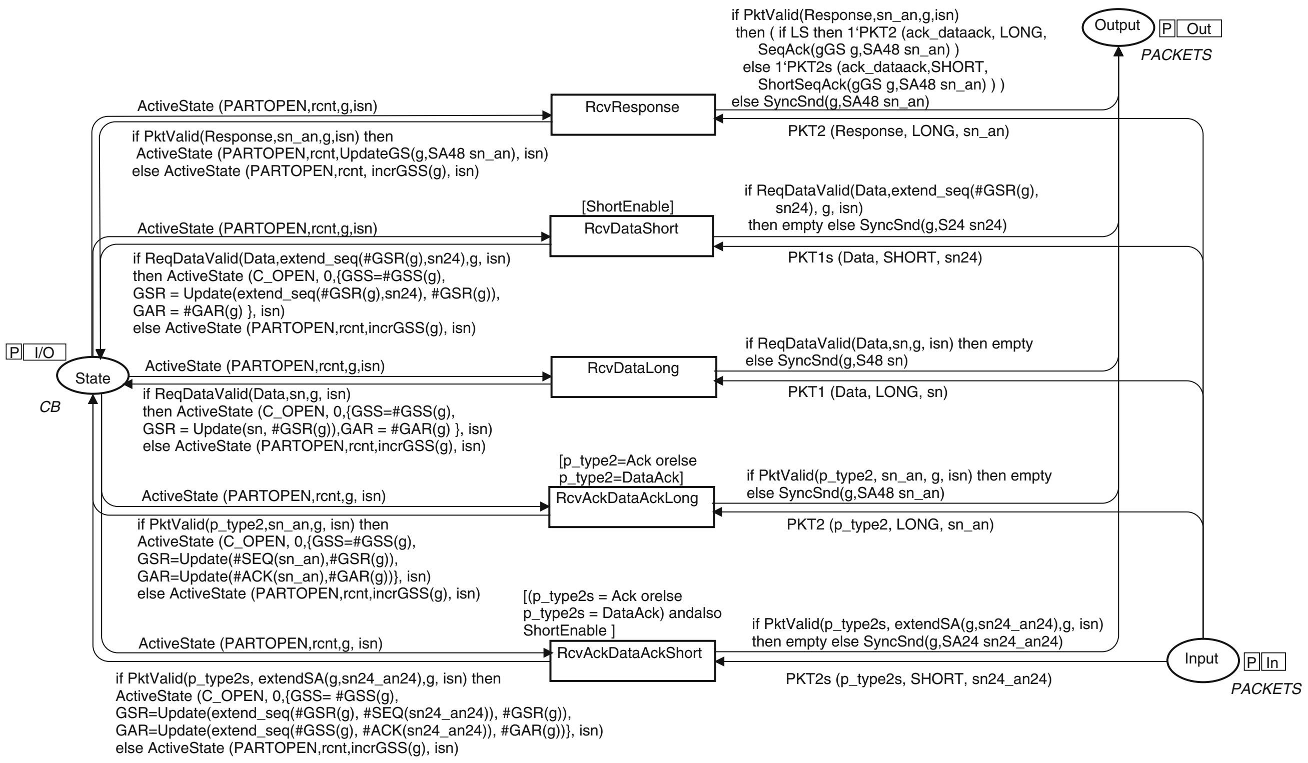
Fig. 14 The Respond page

LS, are used to select an Ack or DataAck packet with long or short sequence number. If the Response is invalid, the entity resets the connection. The reset sent is built by the function SndRstInReq. Because the client has not yet recorded ISR, the acknowledgement number of the outgoing Reset is equal to zero according to Sect. 8.1.1 of [19]. The other transitions model that if the client receives any packets other than Response or Reset, it resets the connection.

The Respond page. This page (Fig. 14) models how the server responds when it receives Request, Ack or DataAck packets. When receiving a Request retransmitted from the client, the server replies with a Response by the transition RcvRequest. The transitions RcvLongSndAckDataAck and RcvLongSndData model the server replying with either an Ack or DataAck or Data packet and entering the OPEN state when it receives an Ack or DataAck having a long sequence number. The transitions RcvShortSndAckDataAck and RcvShortSndData model similar actions when an Ack or DataAck with a short sequence number is received. All outgoing Ack, DataAck and Data packets can have either long or short sequence numbers depending on the variable LS. If LS is true, function SeqAck computes the long sequence

```
(*** Sending a Sync packet in an ACTIVE state ***)
 fun SyncSnd(g:GS,SA48 sn_an) = 1`PKT2 (Sync,LONG,{SEQ=incr(#GSS(g)),ACK= #SEQ(sn_an)})
   | SyncSnd(g:GS,S48 sn)     = 1`PKT2 (Sync,LONG,{SEQ=incr(#GSS(g)),ACK=sn})
   | SyncSnd(g:GS,SA24 sn24_an24)
                              = 1`PKT2(Sync,LONG,{SEQ=incr(#GSS(g)),ACK=extend_seq(#GSR(g),#SEQ(sn24_an24))})
   | SyncSnd(g:GS,S24 sn24)   = 1`PKT2(Sync,LONG,{SEQ=incr(#GSS(g)),ACK=extend_seq(#GSR(g),sn24)});
```

Fig. 15 Function SyncSnd sending a Sync packet



if PktValid(Response,sn_an,g,isn) then (if LS then 1`PKT2 (ack_dataack, LONG, SeqAck(gGS g,SA48 sn_an) ) else 1`PKT2s (ack_dataack,SHORT, ShortSeqAck(gGS g,SA48 sn_an))) else SyncSnd(g,SA48 sn_an)
Output
P Out
PACKETS
ActiveState (PARTOPEN,rcnt,g,isn)
RcvResponse
PKT2 (Response, LONG, sn_an)
if PktValid(Response,sn_an,g,isn) then ActiveState (PARTOPEN,rcnt,UpdateGS(g,SA48 sn_an), isn) else ActiveState (PARTOPEN,rcnt, incrGSS(g), isn)
if ReqDataValid(Data,extend_seq(#GSR(g), sn24), g, isn) then empty else SyncSnd(g,S24 sn24)
[ShortEnable]
ActiveState (PARTOPEN,rcnt,g,isn)
RcvDataShort
PKT1s (Data, SHORT, sn24)
if ReqDataValid(Data,extend_seq(#GSR(g),sn24),g, isn) then ActiveState (C_OPEN, 0,{GSS=#GSS(g), GSR = Update(extend_seq(#GSR(g),sn24), #GSR(g)), GAR = #GAR(g) }, isn) else ActiveState (PARTOPEN,rcnt,incrGSS(g), isn)
P I/O
State
CB
if ReqDataValid(Data,sn,g, isn) then empty else SyncSnd(g,S48 sn)
ActiveState (PARTOPEN,rcnt,g,isn)
RcvDataLong
PKT1 (Data, LONG, sn)
if ReqDataValid(Data,sn,g, isn) then ActiveState (C_OPEN, 0,{GSS=#GSS(g), GSR = Update(sn, #GSR(g)),GAR = #GAR(g) }, isn) else ActiveState (PARTOPEN,rcnt,incrGSS(g), isn)
[p_type2=Ack orelse p_type2=DataAck]
if PktValid(p_type2, sn_an, g, isn) then empty else SyncSnd(g,SA48 sn_an)
ActiveState (PARTOPEN,rcnt,g, isn)
RcvAckDataAckLong
PKT2 (p_type2, LONG, sn_an)
if PktValid(p_type2,sn_an,g, isn) then ActiveState (C_OPEN, 0,{GSS=#GSS(g), GSR=Update(#SEQ(sn_an),#GSR(g)), GAR=Update(#ACK(sn_an),#GAR(g))}, isn) else ActiveState (PARTOPEN,rcnt,incrGSS(g), isn)
[(p_type2s = Ack orelse p_type2s = DataAck) andalso ShortEnable]
if PktValid(p_type2s, extendSA(g,sn24_an24),g, isn) then empty else SyncSnd(g,SA24 sn24_an24)
ActiveState (PARTOPEN,rcnt,g, isn)
RcvAckDataAckShort
Input
P In
PACKETS
PKT2s (p_type2s, SHORT, sn24_an24)
if PktValid(p_type2s, extendSA(g,sn24_an24),g, isn) then ActiveState (C_OPEN, 0,{GSS= #GSS(g), GSR=Update(extend_seq(#GSR(g), #SEQ(sn24_an24)), #GSR(g)), GAR=Update(extend_seq(#GSS(g), #ACK(sn24_an24)), #GAR(g))}, isn) else ActiveState (PARTOPEN,rcnt,incrGSS(g), isn)

Fig. 16 The PartOpen page

and acknowledgement numbers of the outgoing packet. Similar to SeqAck, function ShortSeqAck generates short sequence and acknowledgement numbers of the outgoing packet, when LS is false. In active states, the sequence (and acknowledgement) numbers validity check is done by the functions ReqDataValid and PktValid. ReqDataValid is used to check the sequence number of type1 packets (Request or Data). PktValid is used to check both sequence and acknowledgement numbers of type2 packets (see Fig. 8). If the packet is sequence-invalid, DCCP sends a Sync packet defined by the function SyncSnd in Fig. 15.

The PartOpen page. The PartOpen page (Fig. 16) is very similar to the Respond page. It models how the client responds when it receives Response, Ack, DataAck or Data packets. When receiving a Response retransmitted from the server, the client replies with an Ack or DataAck (transition RcvResponse). The transitions RcvDataShort and RcvDataLong model that when the client receives a Data packet with either a short or long sequence number, it enters the OPEN state. The transitions RcvAckDataAckLong and RcvAckDataAckShort model similar actions when an Ack or DataAck with a long or short sequence number is received. If the received packet is sequence-invalid, DCCP sends a Sync packet instead, similar to the Respond page. (Since we do not model DCCP's data transfer phase, when the client enters the OPEN state, it does not reply.)

The DataTransfer page. (Fig. 17) model DCCP's behaviour when receiving a Data, Ack or DataAck packet in a state after the connection is established (i.e., in OPEN, CLOSING and CLOSEREQ). It comprises four transitions: RcvDataShort, RcvDataLong, RcvAckDataAckShort and RcvAckDataAckLong. If the packet received is invalid, these transitions reply with a Sync packet. When receiving the valid packet, they do not reply but update the state variables accordingly.

The ClosingDown page (Fig. 18) defines DCCP's behaviour when receiving a CloseReq or Close packet in an *active* state. The transition RcvCloseReqClient models that when the client receives a valid CloseReq, it enters the CLOSING state and replies with a Close packet. The transition RcvClose models the client or server receiving a Close packet.

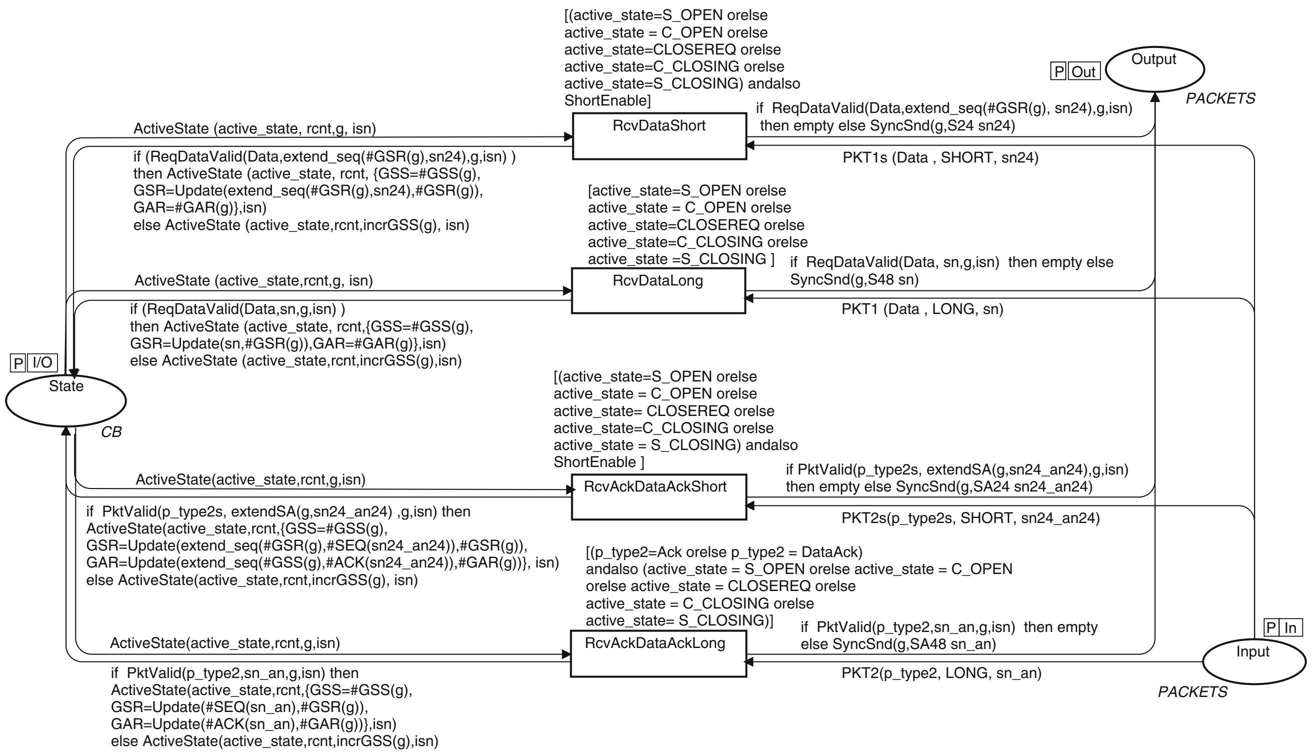

[(active_state=S_OPEN orelse
active_state = C_OPEN orelse
active_state=CLOSEREQ orelse
active_state=C_CLOSING orelse
active_state=S_CLOSING) andalso
ShortEnable]
RcvDataShort
ActiveState (active_state, rcnt,g, isn)
if (ReqDataValid(Data,extend_seq(#GSR(g),sn24),g,isn))
then ActiveState (active_state, rcnt, {GSS=#GSS(g),
GSR=Update(extend_seq(#GSR(g),sn24),#GSR(g)),
GAR=#GAR(g)},isn)
else ActiveState (active_state,rcnt,incrGSS(g), isn)
if ReqDataValid(Data,extend_seq(#GSR(g), sn24),g,isn)
then empty else SyncSnd(g,S24 sn24)
PKT1s (Data , SHORT, sn24)
P Out
Output
PACKETS
[active_state=S_OPEN orelse
active_state = C_OPEN orelse
active_state=CLOSEREQ orelse
active_state=C_CLOSING orelse
active_state =S_CLOSING]
RcvDataLong
ActiveState (active_state,rcnt,g, isn)
if ReqDataValid(Data, sn,g,isn) then empty else
SyncSnd(g,S48 sn)
PKT1 (Data , LONG, sn)
if (ReqDataValid(Data,sn,g,isn))
then ActiveState (active_state, rcnt,{GSS=#GSS(g),
GSR=Update(sn,#GSR(g)),GAR=#GAR(g)},isn)
else ActiveState (active_state,rcnt,incrGSS(g),isn)
P I/O
State
CB
[(active_state=S_OPEN orelse
active_state = C_OPEN orelse
active_state= CLOSEREQ orelse
active_state=C_CLOSING orelse
active_state = S_CLOSING) andalso
ShortEnable]
RcvAckDataAckShort
ActiveState(active_state,rcnt,g,isn)
if PktValid(p_type2s, extendSA(g,sn24_an24),g,isn)
then empty else SyncSnd(g,SA24 sn24_an24)
PKT2s(p_type2s, SHORT, sn24_an24)
if PktValid(p_type2s, extendSA(g,sn24_an24) ,g,isn) then
ActiveState(active_state,rcnt,{GSS=#GSS(g),
GSR=Update(extend_seq(#GSR(g),#SEQ(sn24_an24)),#GSR(g)),
GAR=Update(extend_seq(#GSS(g),#ACK(sn24_an24)),#GAR(g))}, isn)
else ActiveState(active_state,rcnt,incrGSS(g), isn)
[(p_type2=Ack orelse p_type2 = DataAck)
andalso (active_state = S_OPEN orelse active_state = C_OPEN
orelse active_state = CLOSEREQ orelse
active_state = C_CLOSING orelse
active_state= S_CLOSING)]
RcvAckDataAckLong
ActiveState(active_state,rcnt,g,isn)
if PktValid(p_type2,sn_an,g,isn) then empty
else SyncSnd(g,SA48 sn_an)
PKT2(p_type2, LONG, sn_an)
P In
Input
PACKETS
if PktValid(p_type2,sn_an,g,isn) then
ActiveState(active_state,rcnt,{GSS=#GSS(g),
GSR=Update(#SEQ(sn_an),#GSR(g)),
GAR=Update(#ACK(sn_an),#GAR(g))},isn)
else ActiveState(active_state,rcnt,incrGSS(g),isn)

Fig. 17 The DataTransfer page

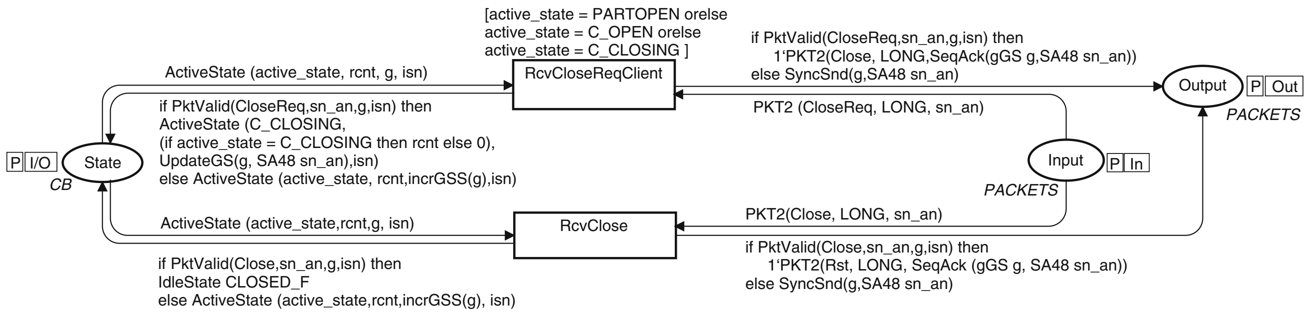

[active_state = PARTOPEN orelse
active_state = C_OPEN orelse
active_state = C_CLOSING]
RcvCloseReqClient
ActiveState (active_state, rcnt, g, isn)
if PktValid(CloseReq,sn_an,g,isn) then
1`PKT2(Close, LONG,SeqAck(gGS g,SA48 sn_an))
else SyncSnd(g,SA48 sn_an)
Output
P Out
PACKETS
PKT2 (CloseReq, LONG, sn_an)
if PktValid(CloseReq,sn_an,g,isn) then
ActiveState (C_CLOSING,
(if active_state = C_CLOSING then rcnt else 0),
UpdateGS(g, SA48 sn_an),isn)
else ActiveState (active_state, rcnt,incrGSS(g),isn)
P I/O
State
CB
Input
P In
PACKETS
ActiveState (active_state,rcnt,g, isn)
RcvClose
PKT2(Close, LONG, sn_an)
if PktValid(Close,sn_an,g,isn) then
1`PKT2(Rst, LONG, SeqAck (gGS g, SA48 sn_an))
else SyncSnd(g,SA48 sn_an)
if PktValid(Close,sn_an,g,isn) then
IdleState CLOSED_F
else ActiveState (active_state,rcnt,incrGSS(g), isn)

Fig. 18 The ClosingDown page

If the Close is valid, DCCP resets the connection, otherwise it sends a Sync packet.

The RcvSync page. The RcvSync page (Fig. 19) models DCCP receiving a Sync or SyncAck packet in an active state. If the received Sync or SyncAck packet is invalid, the packet is ignored. On receiving a valid Sync packet (see transition RcvSync), GSR is updated but GAR is not because the acknowledgement number in the Sync packet could be sequence-invalid. When a valid SyncAck is received, both GSR and GAR are updated. Transition RcvSyncAckBeforeOpen models that when a valid SyncAck is received in PARTOPEN or RESPOND, the endpoint enters C_OPEN or S_OPEN, respectively, using function go_open (defined at the end of Appendix). Transition RcvSyncAckAfterOpen models DCCP receiving a SyncAck after it enters OPEN (including the CLOSEREQ and CLOSING states).

The RcvReset page (Fig. 20) models the receipt of a Reset packet. Transition InIdleState models that the Reset packet is always discarded in an idle state. In an active state or the REQUEST state if the Reset is valid, DCCP enters TIME-WAIT. If the client receives an invalid Reset while in REQUEST, it resets the connection using function SndRstInReq. Because the client has not yet recorded ISR, the acknowledgement number of the outgoing Reset is equal to zero according to Sect. 8.1.1 of [19]. Receiving an invalid Reset in an active state results in DCCP replying with a Sync packet (see the transition InActiveState). However, the Sync packet has an acknowledgement number equal to

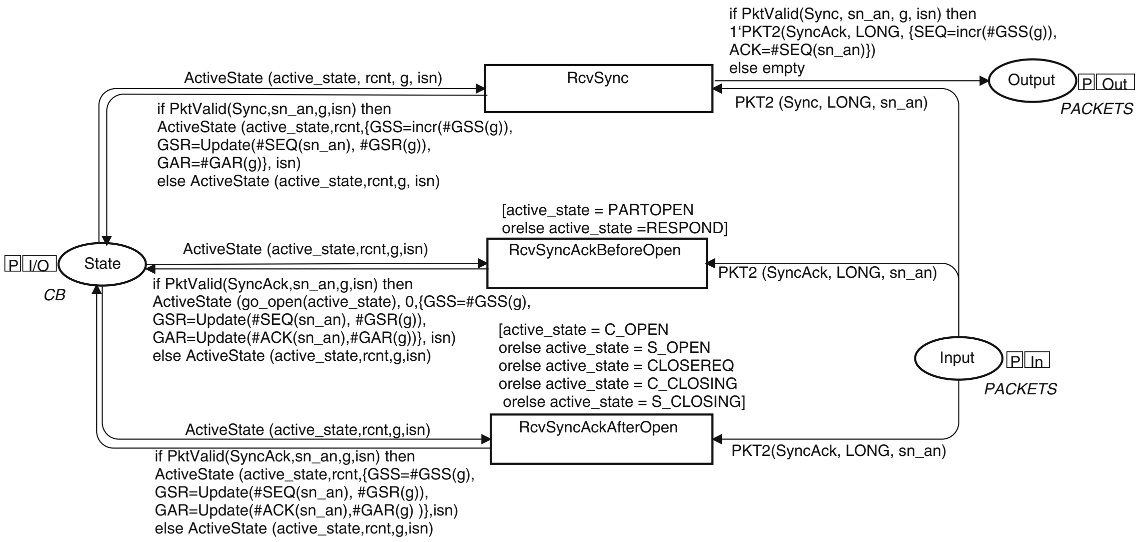

if PktValid(Sync, sn_an, g, isn) then
1‘PKT2(SyncAck, LONG, {SEQ=incr(#GSS(g)),
ACK=#SEQ(sn_an)})
else empty
ActiveState (active_state, rcnt, g, isn)
RcvSync
Output
P Out
PACKETS
PKT2 (Sync, LONG, sn_an)
if PktValid(Sync,sn_an,g,isn) then
ActiveState (active_state,rcnt,{GSS=incr(#GSS(g)),
GSR=Update(#SEQ(sn_an), #GSR(g)),
GAR=#GAR(g)}, isn)
else ActiveState (active_state,rcnt,g, isn)
[active_state = PARTOPEN
orelse active_state =RESPOND]
P I/O
State
CB
ActiveState (active_state,rcnt,g,isn)
RcvSyncAckBeforeOpen
PKT2 (SyncAck, LONG, sn_an)
if PktValid(SyncAck,sn_an,g,isn) then
ActiveState (go_open(active_state), 0,{GSS=#GSS(g),
GSR=Update(#SEQ(sn_an), #GSR(g)),
GAR=Update(#ACK(sn_an),#GAR(g))}, isn)
else ActiveState (active_state,rcnt,g,isn)
[active_state = C_OPEN
orelse active_state = S_OPEN
orelse active_state = CLOSEREQ
orelse active_state = C_CLOSING
orelse active_state = S_CLOSING]
Input
P In
PACKETS
ActiveState (active_state,rcnt,g,isn)
RcvSyncAckAfterOpen
PKT2(SyncAck, LONG, sn_an)
if PktValid(SyncAck,sn_an,g,isn) then
ActiveState (active_state,rcnt,{GSS=#GSS(g),
GSR=Update(#SEQ(sn_an), #GSR(g)),
GAR=Update(#ACK(sn_an),#GAR(g))},isn)
else ActiveState (active_state,rcnt,g,isn)

Fig. 19 The RcvSync page

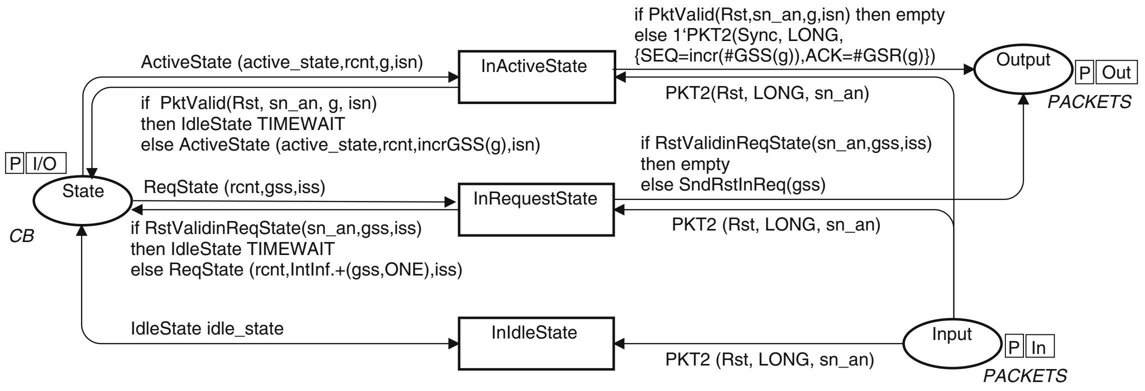

if PktValid(Rst,sn_an,g,isn) then empty
else 1‘PKT2(Sync, LONG,
{SEQ=incr(#GSS(g)),ACK=#GSR(g)})
ActiveState (active_state,rcnt,g,isn)
InActiveState
Output
P Out
PACKETS
PKT2(Rst, LONG, sn_an)
if PktValid(Rst, sn_an, g, isn)
then IdleState TIMEWAIT
else ActiveState (active_state,rcnt,incrGSS(g),isn)
P I/O
if RstValidinReqState(sn_an,gss,iss)
then empty
else SndRstInReq(gss)
State
CB
ReqState (rcnt,gss,iss)
InRequestState
PKT2 (Rst, LONG, sn_an)
if RstValidinReqState(sn_an,gss,iss)
then IdleState TIMEWAIT
else ReqState (rcnt,IntInf.+(gss,ONE),iss)
IdleState idle_state
InIdleState
Input
P In
PACKETS
PKT2 (Rst, LONG, sn_an)

Fig. 20 The RcvReset page

GSR rather than the sequence number received (Sect. 7.5.4 of [19]).

The Retransmission page. During connection set up and close down retransmissions occur when DCCP holds a state for too long. DCCP retransmissions are modelled in Fig. 21. Retransmission in each state is modelled by the transition corresponding to the state’s name: Retrans_REQUEST, RetransShort_PARTOPEN, RetransLong_PARTOPEN, Retrans_CLOSEREQ and Retrans_CLOSING. In PARTOPEN the client can retransmit either an Ack or a DataAck. Each retransmission increases the retransmission counter (rcnt) by one. When the counter reaches the maximum retransmission value (lines 1–4 of Fig. 22), DCCP resets the connection and enters the CLOSED state. These actions are modelled by the transitions BackOff_REQUEST and BackOff_ActiveState. The guard function BackOff (lines 5–12 of Fig. 22) checks whether the number of retransmissions has reached the maximum value or not. The server in RESPOND returns a Response packet on receiving a valid Request packet. It does not retransmit Response packets but backs off to CLOSED after holding the RESPOND state for 4 MPL.

The Unexpected page in Fig. 23 models the receipt of unexpected events as defined in the pseudo code of step 7 of Sect. 8.5 of [19]. Firstly, DCCP does not expect to receive a Request or Response after the connection is established. Secondly, the client does not expect to receive a Request and the server never expects a Response packet. Thirdly, the server in RESPOND does not expect to receive a Data packet. Fourthly the server never expects to receive a CloseReq. If an unexpected packet is received, the entity replies with a Sync packet. Transitions RcvRequest and RcvResponse model DCCP receiving the unexpected Request and Response packets, respectively. The transitions RcvDataShort_inRESPOND and RcvDataLong_inRESPOND model the actions taken by the server on receipt of a short or long Data packet in the RESPOND state. The last transition ServerRcvCloseReq

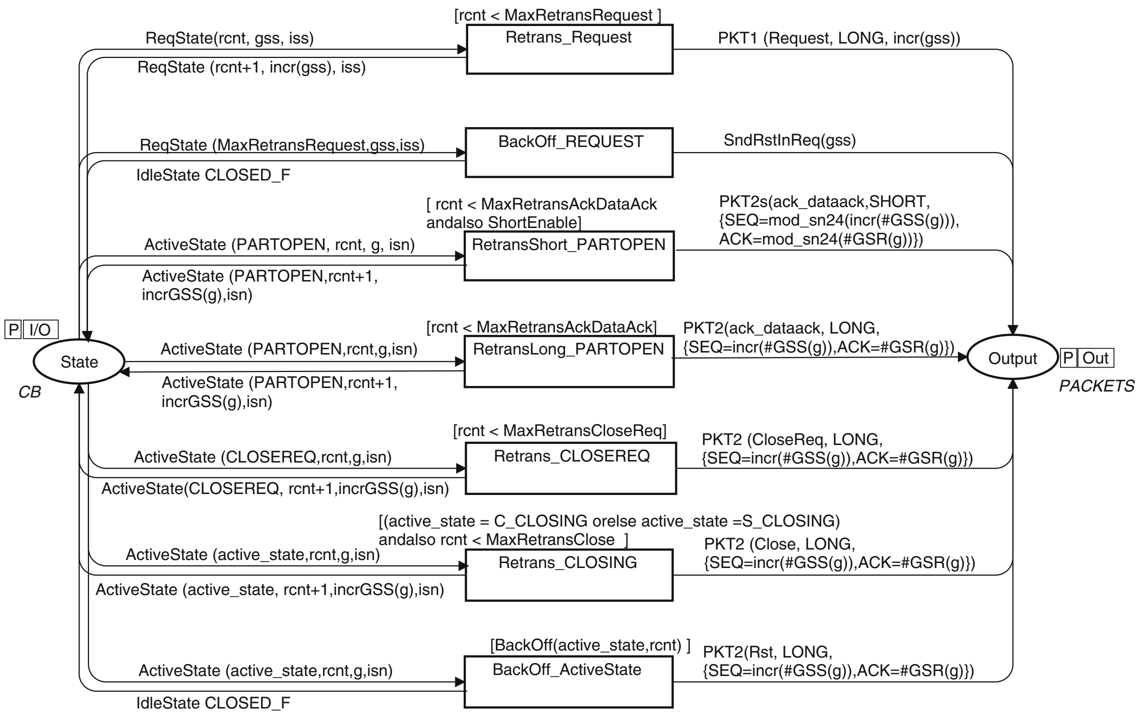

[rcnt < MaxRetransRequest]
Retrans_Request
ReqState(rcnt, gss, iss)
ReqState (rcnt+1, incr(gss), iss)
PKT1 (Request, LONG, incr(gss))
BackOff_REQUEST
ReqState (MaxRetransRequest,gss,iss)
IdleState CLOSED_F
SndRstInReq(gss)
[rcnt < MaxRetransAckDataAck andalso ShortEnable]
RetransShort_PARTOPEN
ActiveState (PARTOPEN, rcnt, g, isn)
ActiveState (PARTOPEN,rcnt+1, incrGSS(g),isn)
PKT2s(ack_dataack,SHORT, {SEQ=mod_sn24(incr(#GSS(g))), ACK=mod_sn24(#GSR(g))})
P I/O
State
CB
[rcnt < MaxRetransAckDataAck]
RetransLong_PARTOPEN
ActiveState (PARTOPEN,rcnt,g,isn)
ActiveState (PARTOPEN,rcnt+1, incrGSS(g),isn)
PKT2(ack_dataack, LONG, {SEQ=incr(#GSS(g)),ACK=#GSR(g)})
Output
P Out
PACKETS
[rcnt < MaxRetransCloseReq]
Retrans_CLOSEREQ
ActiveState (CLOSEREQ,rcnt,g,isn)
ActiveState(CLOSEREQ, rcnt+1,incrGSS(g),isn)
PKT2 (CloseReq, LONG, {SEQ=incr(#GSS(g)),ACK=#GSR(g)})
[(active_state = C_CLOSING orelse active_state =S_CLOSING) andalso rcnt < MaxRetransClose]
Retrans_CLOSING
ActiveState (active_state,rcnt,g,isn)
ActiveState (active_state, rcnt+1,incrGSS(g),isn)
PKT2 (Close, LONG, {SEQ=incr(#GSS(g)),ACK=#GSR(g)})
[BackOff(active_state,rcnt)]
BackOff_ActiveState
ActiveState (active_state,rcnt,g,isn)
IdleState CLOSED_F
PKT2(Rst, LONG, {SEQ=incr(#GSS(g)),ACK=#GSR(g)})

Fig. 21 The Retransmission page

```
 val MaxRetransRequest       = 1;
 val MaxRetransAckDataAck    = 1;
 val MaxRetransCloseReq      = 1;
 val MaxRetransClose         = 1;
fun BackOff(state,rcnt):bool =
    case state of
   RESPOND    =>  true
 | PARTOPEN   => (rcnt=MaxRetransAckDataAck)
 | C_CLOSING  => (rcnt=MaxRetransClosing)
 | CLOSEREQ   => (rcnt=MaxRetransCloseReq)
 | S_CLOSING  => (rcnt=MaxRetransClosing)
 |_           => false;
```

Fig. 22 Declaration for the Retransmission page

models that when the server receives a CloseReq, it replies with a Sync packet.

4 Sweep-line analysis

The success of applying sweep-line relies on the notion of a *progress measure* which is defined [2] as a tuple $\mathcal{P} = (\mathcal{O}, \sqsubseteq, \varphi)$, where $\mathcal{O}$ is a set of progress values, $\sqsubseteq \subseteq \mathcal{O} \times \mathcal{O}$ is a partial order on the progress values, and $\varphi : \mathbb{M} \to \mathcal{O}$ is a progress mapping function from markings of the CPN model, $\mathbb{M}$, to progress values. Gordon et al. [11] point out that protocols can exhibit more than one source of progress and suggest the use of a vector of progress values that we shall call a progress vector. In [6] the progress vector is used in conjunction with lexicographical ordering. We use this approach to analyse the DCCP-CM CPN model.

In this paper $\sqsubseteq$ is a total order rather than partial order. The sweep-line algorithm generates the successors of all unexplored states with the lowest progress value first. Once all states with this lowest progress value have been explored, they will be deleted from memory and the conceptual "sweep-line" will move on to states with the new lowest progress value. A progress mapping is said to be *monotonic* if, for every reachable state, it has a progress value equal to or less than all of its successors. When this is not the case, i.e., at least one successor of one state has a progress value that is less than its predecessor (representing *regress* rather than progress), the sweep-line must conduct additional sweeps of (part of) the state space, using the destinations of these so-called *regress edges* as roots of a new sweep. Thus, some parts in the state space may be explored more than once. However, the sweep-line still guarantees full exploration of a state space and is guaranteed to terminate. Detailed explanations of the sweep-line method can be found in [2,23].

4.1 Notation

We define some notation used for identifying sources of progress in a CPN model. Let $M \in \mathbb{M}$ be a marking of the CPN model. $M(p)$ is the marking of place p (the multiset of tokens on place p) and $|M(p)|$ is the number of tokens on place p.

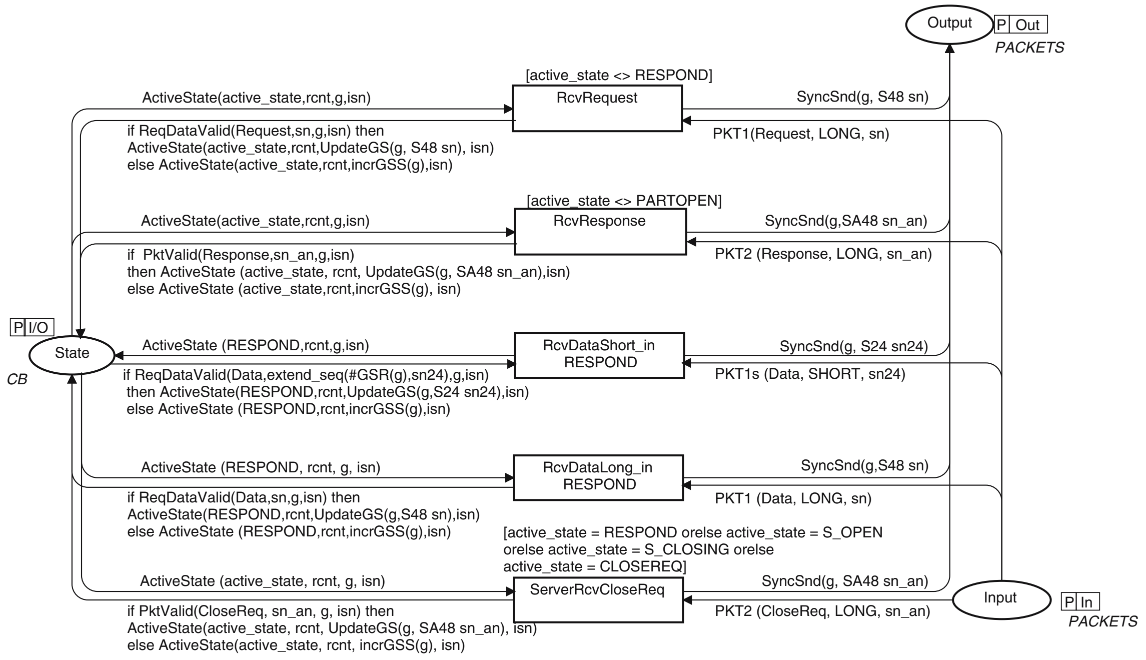

Output
P Out
PACKETS
[active_state <> RESPOND]
ActiveState(active_state,rcnt,g,isn)
RcvRequest
SyncSnd(g, S48 sn)
PKT1(Request, LONG, sn)
if ReqDataValid(Request,sn,g,isn) then
ActiveState(active_state,rcnt,UpdateGS(g, S48 sn), isn)
else ActiveState(active_state,rcnt,incrGSS(g),isn)
[active_state <> PARTOPEN]
ActiveState(active_state,rcnt,g,isn)
RcvResponse
SyncSnd(g,SA48 sn_an)
PKT2 (Response, LONG, sn_an)
if PktValid(Response,sn_an,g,isn)
then ActiveState (active_state, rcnt, UpdateGS(g, SA48 sn_an),isn)
else ActiveState (active_state,rcnt,incrGSS(g), isn)
P I/O
State
CB
ActiveState (RESPOND,rcnt,g,isn)
RcvDataShort_in RESPOND
SyncSnd(g, S24 sn24)
PKT1s (Data, SHORT, sn24)
if ReqDataValid(Data,extend_seq(#GSR(g),sn24),g,isn)
then ActiveState(RESPOND,rcnt,UpdateGS(g,S24 sn24),isn)
else ActiveState (RESPOND,rcnt,incrGSS(g),isn)
ActiveState (RESPOND, rcnt, g, isn)
RcvDataLong_in RESPOND
SyncSnd(g,S48 sn)
PKT1 (Data, LONG, sn)
if ReqDataValid(Data,sn,g,isn) then
ActiveState(RESPOND,rcnt,UpdateGS(g,S48 sn),isn)
else ActiveState (RESPOND,rcnt,incrGSS(g),isn)
[active_state = RESPOND orelse active_state = S_OPEN
orelse active_state = S_CLOSING orelse
active_state = CLOSEREQ]
ActiveState (active_state, rcnt, g, isn)
ServerRcvCloseReq
SyncSnd(g, SA48 sn_an)
PKT2 (CloseReq, LONG, sn_an)
Input
P In
PACKETS
if PktValid(CloseReq, sn_an, g, isn) then
ActiveState(active_state, rcnt, UpdateGS(g, SA48 sn_an), isn)
else ActiveState(active_state, rcnt, incrGSS(g), isn)

Fig. 23 The Unexpected page

Measures of progress, such as the greatest sequence number sent, are often included as a component in a product token residing on a state place, e.g. Client_State. To extract information from a product token we shall use projection functions. Our progress mappings operate on markings, which are multisets of tokens rather than tokens themselves. However, when p is a state place, it only contains one token ($\forall\ M \in [M_0\rangle, |M(p)| = 1$, where $[M_0\rangle$ is the set of reachable markings), and hence $M(p)$ is a *singleton multiset*. If the colour set of place p, is Type(p) then for $|M(p)| = 1$, we can represent $M(p)$ as $1`a$ where $a \in Type(p)$. We can then convert a singleton multiset, into its basis element (i.e., the token), with a function $elem$, so that when $|M(p)| = 1$, $M(p) = 1`a$ and $elem(M(p)) = a$.

4.2 Usual sources of progress

The following describes three usual sources of progress.

4.2.1 Sequence number variables

Firstly, consider GSS. Every time the client or server sends a packet, the state variable GSS (the Greatest Sequence number Sent) increases by one. GSS is stored within a single product token in the places Client_State and Server_State. However, when an entity is in an idle state (CLOSED, LISTEN and TIMEWAIT), there is no GSS. To capture the progress of GSS from the marking of the place Client_State we define a progress mapping for the client $\varphi^c_{gss} : \mathbb{M} \to \mathbb{N}$, where the superscript "c" refers to the Client entity and

$$\varphi^c_{gss}(M) = Proj_{gss}(elem(M(Client\_State))) \tag{1}$$

and $Proj_{gss}$ takes a state variable of type CB and returns GSS for active states and ISS otherwise. During connection establishment and close down, the number of transmitted packets is small. We consider that the sequence number may wrap only once. If the GSS value is less than ISS, it means that the GSS value has wrapped. In this situation, $Proj_{gss}$ returns GSS plus 2^{48} in order to maintain increasing progress values. If there is no GSS value in the state variable, $Proj_{gss}$ returns ISS because it is the starting value of GSS for each connection. The progress mapping, $\varphi^s_{gss}, \varphi^c_{gsr}, \varphi^s_{gsr}, \varphi^c_{gar}$ and φ^s_{gar} for other client and server sequence number variables can also be defined in a similar way.

4.2.2 Major states

Both entities progress through the states defined by colour sets IDLE, REQUEST and ACTIVE. To capture this progress from the token in the place Client_State, we define a progress mapping $\varphi^c_{state} : \mathbb{M} \to \mathbb{N}$ where

$$\varphi^c_{state}(M) = State2Num(Proj_{state}(elem(M(Client\_State)))) \tag{2}$$

The projection function, $Proj_{state}$, takes a state variable of type CB and returns the DCCP state. Function $State2Num$ maps this state to an integer according to the ordering of

Table 1 An ordering and corresponding mapping for DCCP state

Client state	$State2Num$ (state)	Server state	$State2Num$ (state)
CLOSED_I	1	CLOSED_I	1
REQUEST	2	LISTEN	2
PARTOPEN	3	RESPOND	3
C_OPEN	4	S_OPEN	4
C_CLOSING	6	C_CLOSEREQ	5
TIMEWAIT	7	S_CLOSING	6
CLOSED_F	8	TIMEWAIT	7
		CLOSED_F	8

DCCP states shown in Column 1 of Table 1. The server's progress mapping, φ^s_{state}, is defined analogously, using the ordering of DCCP states shown in Column 3 of Table 1.

Because of lost and delayed packets, an entity may retransmit. The progress exhibited by the retransmission counters (RCNTs) has already been covered by GSS because GSS is increased for every packet sent, including retransmissions. Thus the progress captured by RCNTs is not needed. However, when retransmission occurs, the state is not changed, and hence, intuitively, it is useful to include both major state and GSS in progress mappings.

4.2.3 Application commands

During connection set up and close down, the applications at the client and server will issue commands. The progress of issuing commands can be captured by the decrease in the total number of command tokens in both the App_Client and App_Server places. This can help to differentiate between the CLOSED_F state caused by timer expiry in the TIMEWAIT state and the CLOSED_F state resulting from an application close command. Thus we define $\varphi_{cmd} : \mathbb{M} \rightarrow \mathbb{N}$ where

$$\varphi_{cmd}(M) = -|M(App\_Client)| - |M(App\_Server)| \quad (3)$$

Every application command issued corresponds to a change of state. However, other DCCP state changes also occur due to internal behaviour. Hence we consider a state change due to an application command to be a more significant event. Thus we give φ_{cmd} greater weighting than φ^c_{state} and φ^s_{state}.

4.3 More subtle measures of progress

From our experience with DCCP, more than 90% of the state space has at least one entity in CLOSED, which has no sequence number variables. φ^c_{state} and φ^c_{gss} provide no differentiation when the client is CLOSED, and similarly for the server. Thus some measure of progress is needed for when an entity is in the CLOSED state. When an entity is in an idle state (CLOSED,LISTEN and TIMEWAIT), there are two ways in which progress is exhibited:

(a) Receiving DCCP-Reset Packets. When either end receives a DCCP-Reset Packet, the total number of packets in both channel places will decrease by one. This is simply because in CLOSED, LISTEN and TIMEWAIT, the entity discards the DCCP-Reset and does not send a packet in response. To capture this measure of progress we define $\varphi_{ch\_num} : \mathbb{M} \rightarrow \mathbb{N}$ where

$$\varphi_{ch\_num}(M) = -|M(Ch\_C\_S)| - |M(Ch\_S\_C)| \quad (4)$$

Note that this progress mapping initially decreases as the first packets are sent into the channel. However, when component progress measures are combined using lexicographical ordering (as will be done shortly) this *regress* is more than offset by *progress* captured in other "more significant" component progress measures when the number of messages in the channel are increasing.

(b) Receiving non-DCCP-Reset Packets. When any packet but DCCP-Reset is received from one channel by an entity in an idle state, it will send a DCCP-Reset packet into the other channel in response, so that the total number of packets over both channels remains the same. Owing to no GSS, GSR or GAR in these states, the DCCP-Reset packet sent will have a sequence number set to the acknowledgement number of the received packet plus one, and an acknowledgement number set to the sequence number of the received packet. Thus the summation of all sequence numbers and acknowledgement numbers in all packets over both channel places will be increased by one. Thus we define a progress mapping $\varphi_{ch\_sum} : \mathbb{M} \rightarrow \mathbb{N}$ to capture the progress when an entity in an idle state replies with a Reset packet, where

$$\varphi_{ch\_sum}(M) = Sum\_Seq\_Ack(M(Ch\_C\_S)) + Sum\_Seq\_Ack(M(Ch\_S\_C)) \quad (5)$$

where the function Sum_Seq_Ack takes the multiset of packets in a channel and returns the summation of sequence and acknowledgement numbers of every packet. However, there are still two problems. The first is that when an entity in an idle state receives a packet with no acknowledgement number, the sequence number of the Reset packet (sent in reply) is set to zero and its acknowledgement number to the sequence number of the packet received. Thus the markings before and after this action have the *same* progress value. To overcome this problem, when computing Sum_Seq_Ack, we consider that the packets with no acknowledgement number have an acknowledgement number $= -1$. The second problem is that when sequence numbers wrap $\varphi_{ch\_sum}(M)$ will decrease and degrade the performance of the sweep-line. Thus we add 2^{48} to a sequence or acknowledgement number if it is less than ISS, as was the case with $Proj_{gss}$.

φ_{ch_num} has higher significance than φ_{ch_sum} because the effect of φ_{ch_sum} is only important when φ_{ch_num} is constant. Both φ_{ch_num} and φ_{ch_sum} have lower significance than other progress mappings because they are only effective when an entity is in an idle state.

In conclusion, we have identified several sources of progress of the specification model. The progress vector φ_s when considering only GSS has seven dimensions, where

$$\varphi_s(M) = (\varphi_{cmd}(M), \varphi^c_{state}(M), \varphi^s_{state}(M), \varphi^c_{gss}(M), \varphi^s_{gss}(M), \varphi_{ch\_num}(M), \varphi_{ch\_sum}(M)) \quad (6)$$

and the subscript "s" refers to the specification model.

Our experiments show that this progress vector is monotonic for all scenarios analysed in this paper, and swapping the order of the client and server progress mappings has no effect on the performance of the sweep-line. Moreover, the performance is improved slightly by including progress mappings for the GSR and GAR variables in the vector. Thus the full progress vector φ_s is given by

$$\varphi_s(M) = (\varphi_{cmd}(M), \varphi^c_{state}(M), \varphi^s_{state}(M), \varphi^c_{gss}(M), \varphi^s_{gss}(M), \varphi^s_{gsr}(M), \varphi^c_{gsr}(M), \varphi^c_{gar}(M), \varphi^s_{gar}(M), \varphi_{ch\_num}(M), \varphi_{ch\_sum}(M)) \quad (7)$$

4.4 Searching for a better progress measure

Although the sweep-line method helps us to analyse protocols for scenarios that could not be reached before, most results of protocol verification (see for example [6,8,11,24]) show that the peak states stored are around 20–30% of the full state space. The best reduction is shown in [6] where the number of peak states is around 10% (i.e., a reduction in states stored of a factor of 10). The sweep-line method was also applied to compare the service language with the protocol language (language inclusion) of DCCP connection management in [7,9]. It used the same progress measures as described in Sects. 4.2 and 4.3. However, the reduction in peak states stored [7,9] is about a factor of 3–4. Because the state space grows very rapidly with respect to the number of retransmissions, the progress measures used in [7,9] allowed only a few additional scenarios to be analysed (that could not be analysed by conventional state space analysis). Further reduction is thus required to verify DCCP's connection management behaviour.

We learnt from [6,7,9] that although the state variables (states and sequence number sent) in the entities seem to be intuitively good progress measures, there are three fundamental problems. Firstly, more than 90% of the state space has either the server or the client in an idle state. Secondly, in these idle states there are no sequence number state variables. Thirdly, sequence numbers can wrap, leading to regress rather than progress. In other words, the efficiency of the sweep-line largely depends on the progress we can capture from the channel places. Channel places are not a good source of progress, partly because of the overtaking property of the channels. This explains why we only get a factor of 3–4 reduction in peak states stored.

From experiments we have noticed that the initial sequence number received (ISR) plays a crucial role in progress. S_ISR is the first sequence number the server receives (in a Request packet). The client's ISR (C_ISR) is in the first Response packet the client receives before moving to PARTOPEN. If the model is modified to store the values of ISR in the places Server_State and Client_State throughout the connection, including when the entity is in TIMEWAIT and CLOSED_F, we find something interesting. For example, when the initial sequence numbers sent (ISS) by both the client and the server are equal to five and one retransmission is allowed for the Request packet, the state space (total 19,602 nodes) is roughly divided into two large groups: group A, 11,930 nodes with S_ISR = 5 and group B, 7,612 nodes with S_ISR = 6; and one small group of 60 nodes without S_ISR. The size of group A is 60.86% of the total state space and the size of the group B is 38.83% of the total state space. Using only S_ISR as the progress measure, the sweep-line will finish working on group A before it starts working on group B. In this case the peak states stored can be reduced to 60.91% (11,939 nodes), where some states from group B are generated (but not explored) while exploring the states of group A.

Using [S_ISR, C_ISR] as the progress measure, group A is divided again into another two large groups (6,686 nodes with C_ISR = 5 and 4,684 nodes with C_ISR = 6) and one small group (560 nodes) with no C_ISR. However, because there is no loss and the server cannot retransmit the Respond packet, group B is divided into one large group with C_ISR = 5 (7,158 nodes) and a small group (454 nodes) with no C_ISR. The peak states stored is further reduced to 36.52% (7,158 nodes). The clue is that every new progress measure added reduces the number of states that have the same progress vector.

We can continue to successively divide each group again by recording the sequence numbers of the Ack packets (S_IACK, C_IACK) that cause the server and the client to enter OPEN. Unfortunately, while S_ISR and C_ISR are the parameters of the specification model, S_IACK and C_IACK are not. By adding these two parameters, the specification model is modified to produce a different model, raising two issues. Firstly, does the new model (denoted the *augmented* model) have the same behaviour as the original specification model? Because the added parameters are solely used to measure progress and are not used in any protocol operations, both models exhibit the same behaviour. Secondly, the added parameters increase (explode) not only the total number of states but also the amount of memory used by each node in the state space. This is very harmful for state space analysis.

```
(* Modified DCCP state variables *)
color SN = IntInf;
color RCNT = int;(*Retransmission Counter*)
color GS = record GSS:SN48
                * GSR:SN48
                * GAR:SN48;
color SNV = record ISS:SN48 * SNReq_Resp:SN
                 * ISR:SN48 * SNData_L:SN
               * SNAck_L:SN * SNCloseReq:SN
               * SNClose:SN * SNSync:SN
               * SNReset:SN;
color IDLE = with CLOSED_I | LISTEN
                 | TIMEWAIT | CLOSED_F;
color IDLE_STATE = product IDLE*SNV;
 (* counter,gss,SNV *)
color RCNTxGSSxSNV = product RCNT*SN48*SNV;
color ACTIVE = with RESPOND | PARTOPEN | S_OPEN
    | C_OPEN | CLOSEREQ | C_CLOSING |S_CLOSING;
color ACTIVExRCNTxGSxSNV = product ACTIVE * RCNT
                                     * GS * SNV;
color CB = union IdleState:IDLE_STATE
               + ReqState:RCNTxGSSxSNV
               + ActiveState:ACTIVExRCNTxGSxSNV;

```

Fig. 24 Modified DCCP's control block

Although the total number of states increases, the number of states in each subset of the partition decreases. During a sweep through one subset, the progress measure described in Sects. 4.2 and 4.3 can be used to further subdivide it. Thus not only is there a reduction in the peak number of states stored but also the potential for a reduction in execution time due to less time being spent comparing newly generated states with the states currently in memory (due to a reduced number of stored states).

4.5 Progress mappings for the augmented model

Further to the observation discussed in the previous section, we experiment by adding the new variables one by one. We notice when recording the latest sequence number of the packets *sent* for each packet type (rather than *received*), the sweep-line algorithm gives better reduction in peak memory. Nevertheless ISRs are still used in the progress vector because they are already in the original model. We also exploit the progress from sequence numbers sent of Sync and Reset packets and further divide the state space to an even finer grain.

Figure 24 shows the modification to the declarations of the state variables. A new colour set called SNV (Sequence Number Vector) is defined in place of ISN. This records the additional progress variables: the latest sequence number of the packet sent for each packet type, and ISR. The colour set SNV is attached to every state, including idle states and REQUEST.

Using progress functions similar to those defined in Sect. 4.2, we can extract an additional 14 dimension progress vector (Eq. 8) to measure progress in the augmented model. The ordering of progress values is arranged according to the point when each variable first appears in the typical scenario shown in Fig. 2 (connection set up followed by closing).

$$\begin{aligned}\varphi_a(M) = (&\varphi^c_{SNReq\_Resp}(M), \varphi^s_{ISR}(M), \varphi^s_{SNReq\_Resp}(M),\\ &\varphi^c_{ISR}(M), \varphi^c_{SNAck\_L}(M), \varphi^s_{SNAck\_L}(M),\\ &\varphi^s_{SNData\_L}(M), \varphi^s_{SNCloseReq}(M),\\ &\varphi^c_{SNClose}(M), \varphi^s_{SNClose}(M), \varphi^s_{SNSync}(M),\\ &\varphi^c_{SNSync}(M), \varphi^s_{SNReset}(M), \varphi^c_{SNReset}(M)) \qquad (8)\end{aligned}$$

Thus the overall progress vector for the augmented model is:

$$\varphi_{DCCP}(M) = (\varphi_a(M)\ ,\ \varphi_s(M)) \qquad (9)$$

Although the analysis result using the proposed progress measures are very promising, there is still one drawback. The number of terminal markings increases rapidly with respect to the number of new variables added. To avoid this drawback we introduce four transitions and two places as shown in Fig. 25, to merge redundant terminal markings into the terminal markings that exist in the original state space. The places Cnt_C_S and Cnt_S_C are used to count the number of packets in channels. The clean up operation takes place when there is nothing left in both channels.

5 Experimental results

5.1 Initial configurations

We analyse two DCCP connection management models using Design/CPN version 4.0.5 with the occurrence graph tool and the prototype sweep-line library from [6] to handle progress vectors, on a Pentium-IV 2.6 GHz computer with 1 GB RAM. The first model is the specification model as described in Sect. 3. The second model is the augmented model as described in Sect. 4.5. The DCCP-CM models are initialised by distributing tokens to places App_Client, App_Server, Client_State and Server_State of the model to create the initial markings. Instead of non-deterministic use of long or short sequence number and DCCP-Ack or DCCP-DataAck, we choose to analyse the procedures when only long sequence number and DCCP-Ack are used. Table 2 shows the initial markings for the Application places for both the client and server. We present the analysis results of three cases. In all cases, the client and server are initially in CLOSED_I, both channel places are empty and ISS values on both sides are set to $2^{48} - 3$ to allow sequence numbers to wrap to zero.

Case A is for connection establishment. The client issues an "active Open" command while the server issues a "passive Open" command. When limiting the maximum number of retransmissions to one, connection establishment can fail due to a deadlock when sequence numbers wrap [33]. In this paper we use the sweep-line to extend the analysis to include a scenario when the maximum number of retransmissions is

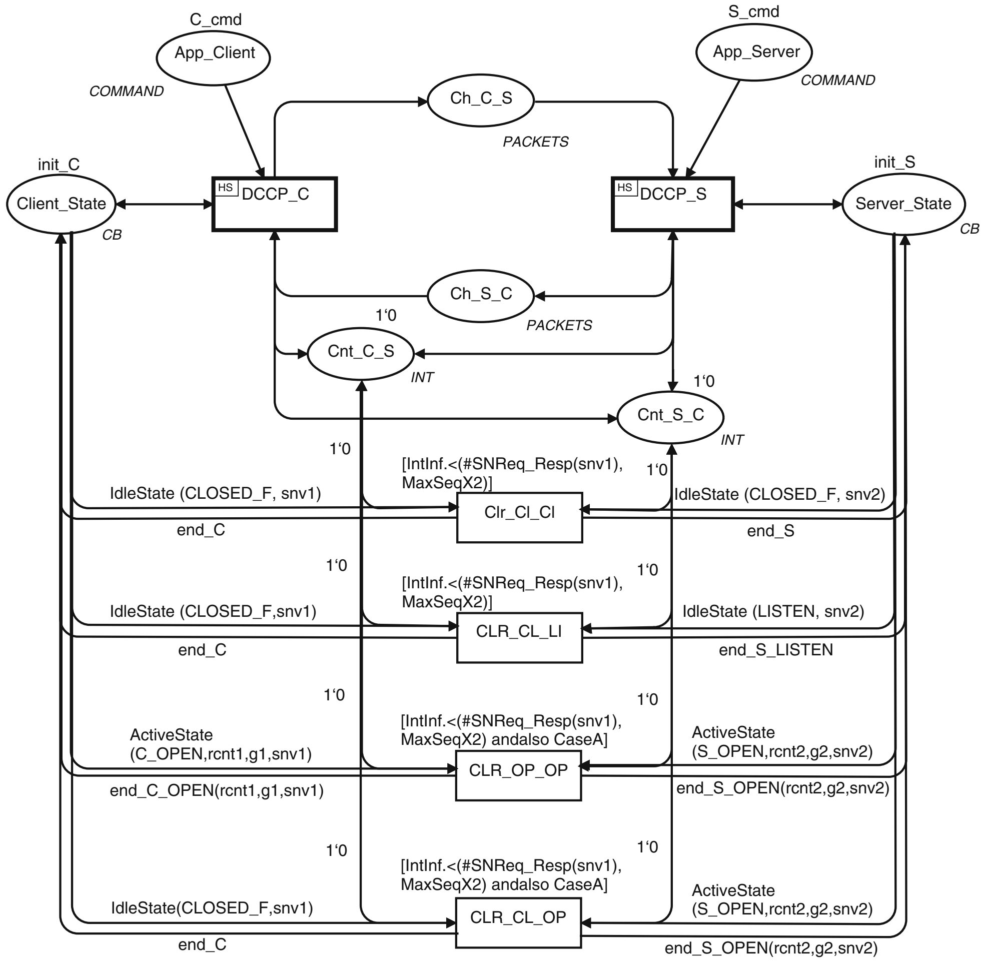

C_cmd
App_Client
COMMAND
S_cmd
App_Server
COMMAND
Ch_C_S
PACKETS
init_C
Client_State
CB
HS DCCP_C
HS DCCP_S
init_S
Server_State
CB
Ch_S_C
PACKETS
1'0
Cnt_C_S
INT
1'0
Cnt_S_C
INT
1'0
[IntInf.<(#SNReq_Resp(snv1), MaxSeqX2)]
1'0
IdleState (CLOSED_F, snv1)
Clr_Cl_Cl
IdleState (CLOSED_F, snv2)
end_C
end_S
1'0
1'0
[IntInf.<(#SNReq_Resp(snv1), MaxSeqX2)]
IdleState (CLOSED_F,snv1)
CLR_CL_LI
IdleState (LISTEN, snv2)
end_C
end_S_LISTEN
1'0
1'0
[IntInf.<(#SNReq_Resp(snv1), MaxSeqX2) andalso CaseA]
ActiveState (C_OPEN,rcnt1,g1,snv1)
CLR_OP_OP
ActiveState (S_OPEN,rcnt2,g2,snv2)
end_C_OPEN(rcnt1,g1,snv1)
end_S_OPEN(rcnt2,g2,snv2)
1'0
1'0
[IntInf.<(#SNReq_Resp(snv1), MaxSeqX2) andalso CaseA]
IdleState(CLOSED_F,snv1)
CLR_CL_OP
ActiveState (S_OPEN,rcnt2,g2,snv2)
end_C
end_S_OPEN(rcnt2,g2,snv2)

Fig. 25 Augmented model: the top level page

Table 2 Initial configurations

Case	Initial Markings	
	App_Client	App_Server
A	1‘active open	1‘passive open
B	1‘active open	1‘passive open ++ 1‘active close
C	1‘active open	1‘passive open ++ 1‘server active close

two to determine if the undesired deadlocks still exist. Cases B and C cover the case when the server issues a close command while the connection is being established. We select configurations B and C to determine if the application on the server can clear the deadlocks by issuing either an “active close” or a “server active close” command.

5.2 Analysis results

The analysis results for the DCCP connection management CPN specification and augmented models in various configurations are shown in Tables 3 and 4. All progress measures used are monotonic. The first column in Table 3 shows the configurations being analysed, where the 4-tuple is the maximum number of retransmissions allowed for Request, Ack, CloseReq and Close packet types, respectively. An “x” means the retransmission of those packet types never happens in that configuration. Columns 2–3 show the analysis results of the specification model when using a constant progress value (Sweep-Line$_C$) which simulates conventional reachability analysis. Columns 4–6 show the results of the specification model using the progress measure φ_s described in Sects. 4.2 and 4.3 (Sweep-Line$_S$). Columns 7–9 show the result of the augmented model using the progress measure φ_{DCCP} described in Sect. 4.5 (Sweep-Line$_A$). Comparisons of space[3] and time[4] used between Sweep-Line$_S$ and Sweep-Line$_C$ are shown in columns 10–11. Comparison of space and time used between Sweep-Line$_A$ and Sweep-Line$_C$ are shown in columns 12–13. A “–” means the full state space cannot be generated due to computer memory limits. Table 3 shows *five* cases where Sweep-Line$_S$ cannot generate the state space.

[3] (the number of peak states/the total number of states in the original state space)$\times$ 100.

[4] (the exploration time using progress vector φ_s/the exploration time using constant progress value)$\times$ 100.

Table 3 Sweep-line analysis results of DCCP connection management

Config.	value Sweep-Line$_C$ constant progress value		Sweep-line$_S$ specification model			Sweep-line$_A$ augmented model			(S/C)*100		(A/C)*100	
	nodes	hh:mm:ss	total nodes	peak nodes	hh:mm:ss	total nodes	peak nodes	hh:mm:ss	% space	% time	% space	% time
A-(0,0,x,x)	102	00:00:00	102	34	00:00:00	140	15	00:00:00	33.33	–	14.71	–
A-(0,1,x,x)	209	00:00:00	209	71	00:00:00	314	34	00:00:00	33.97	–	16.27	–
A-(0,2,x,x)	602	00:00:00	602	218	00:00:00	983	119	00:00:01	36.21	–	19.77	–
A-(1,0,x,x)	2,397	00:00:01	2,397	918	00:00:02	4,870	87	00:00:03	38.30	200	3.63	300
A-(1,1,x,x)	11,870	00:00:08	11,870	4,435	00:00:10	29,212	288	00:00:23	37.36	125	2.43	288
A-(1,2,x,x)	61,239	00:01:05	61,239	24,289	00:01:22	172,307	1,096	00:02:46	39.66	126	1.79	255
A-(2,0,x,x)	116,745	00:03:03	116,745	42,486	00:03:24	362,528	1,263	00:05:09	36.39	111	1.08	169
A-(1,3,x,x)	296,961	00:10:35	296,961	123,463	00:12:51	934,049	4,167	00:17:58	41.58	121	1.40	170
A-(2,1,x,x)	964,862	01:41:29	964,862	354,710	01:59:47	3,970,455	6,142	01:09:03	36.76	118	0.64	68
A-(2,2,x,x)	–	–	–	–	–	31,872,051	34,059	11:46:59	–	–	–	–
A-(2,3,x,x)	–	–	–	–	–	219,200,989	161,461	120:07:23	–	–	–	–
B-(0,0,x,0)	476	00:00:00	476	153	00:00:00	745	45	00:00:00	32.14	–	9.45	–
B-(0,0,x,1)	1,505	00:00:01	1,505	515	00:00:01	2,875	129	00:00:02	34.22	100	8.57	200
B-(0,1,x,0)	1,636	00:00:01	1,636	532	00:00:01	2,962	152	00:00:02	32.52	100	9.29	200
B-(0,1,x,1)	7,479	00:00:05	7,479	2,874	00:00:06	15,580	518	00:00:14	38.43	120	6.93	280
B-(1,0,x,0)	24,927	00:00:20	24,927	9,093	00:00:23	72,380	394	00:00:57	36.48	115	1.58	285
B-(1,0,x,1)	203,168	00:04:38	203,168	74,815	00:05:33	723,963	2,743	00:11:43	36.82	120	1.35	253
B-(1,1,x,0)	218,951	00:06:27	218,951	77,131	00:06:59	876,163	1,559	00:13:45	35.23	108	0.71	213
B-(2,0,x,0)	–	–	–	–	–	12,568,700	9,154	03:39:20	–	–	–	–
B-(1,1,x,1)	–	–	–	–	–	13,241,057	21,479	04:17:28	–	–	–	–
C-(0,0,0,0)	666	00:00:00	666	235	00:00:00	1,089	45	00:00:01	35.29	–	6.76	–
C-(0,0,0,1)	1,245	00:00:01	1,245	449	00:00:01	2,156	92	00:00:02	36.06	100	7.39	200
C-(0,1,0,0)	3,270	00:00:02	3,270	1,148	00:00:02	6,244	146	00:00:05	35.11	100	4.46	250
C-(0,1,0,1)	9,080	00:00:06	9,080	3,321	00:00:08	20,150	430	00:00:18	36.57	133	4.74	300
C-(0,0,1,0)	8,890	00:00:05	8,890	3,550	00:00:07	17,536	233	00:00:14	39.93	140	2.62	280
C-(1,0,0,0)	45,368	00:00:40	45,368	17,214	00:00:46	159,818	394	00:02:05	37.94	115	0.87	312
C-(0,1,1,0)	79,320	00:01:10	79,320	30,774	00:01:30	169,728	1,341	00:02:44	38.80	129	1.69	234
C-(0,0,1,1)	127,195	00:02:03	127,195	49,737	00:02:40	289,062	2,573	00:04:54	39.10	130	2.02	239
C-(1,0,0,1)	305,807	00:08:12	305,807	110,955	00:08:48	1,441,029	2,798	00:24:25	36.28	107	0.91	298
C-(1,1,0,0)	477,764	00:19:45	477,764	175,913	00:21:10	2,058,949	1,727	00:33:52	36.82	107	0.36	171
C-(1,0,1,0)	–	–	1,493,946	569,749	03:16:27	8,141,588	6,719	02:26:25	–	–	–	–
C-(1,1,0,1)	–	–	–	–	–	17,594,060	18,606	05:50:50	–	–	–	–

Table 4 Terminal markings

Config.	Terminal Markings			
	Type-I	Type-II	Type-III	Type-IV
A-(0,0,x,x)	2	1	1	0
A-(1,0,x,x)	15	1	1	1
A-(2,0,x,x)	83	1	1	8
A-(0,1,x,x)	5	1	1	0
A-(0,2,x,x)	9	1	1	0
A-(1,1,x,x)	38	1	1	2
A-(1,2,x,x)	68	1	1	3
A-(1,3,x,x)	105	1	1	4
A-(2,1,x,x)	198	1	1	15
A-(2,2,x,x)	343	1	1	22
A-(2,3,x,x)	519	1	1	29

5.2.1 Reduction of the peak number of stored states

While Sweep-Line$_S$ reduces the peak number of stored states to about 30–40%, Sweep-Line$_A$ gives a more promising result. The larger the state spaces, the greater the reduction. In most cases the reduction is better than a factor of 10 (10%). The best result shown in Table 3 is 0.36% (277 times smaller than the original state space). However, it is not immediately clear how much this translates to a reduction in real memory usage, because each state in the augmented model stores slightly more information. However, Sweep-Line$_A$ can finish the exploration in some configurations, such as A-(2,2,x,x) and A-(2,3,x,x), while Sweep-Line$_S$ and Sweep-Line$_C$ cannot. This leads us to believe that, pragmatically, Sweep-Line$_A$ also provides a significant reduction in memory usage when the state spaces are large.

5.2.2 Execution time comparison

Generally Sweep-Line$_S$ and Sweep-Line$_A$ have longer exploration times than Sweep-Line$_C$ because of the overhead computing the progress mappings (11 functions for Sweep-Line$_S$ and 25 functions for Sweep-Line$_A$). When the original state space is small, Sweep-Line$_A$ has the longest exploration time because the state space of the augmented model is bigger and Sweep-Line$_A$ has more progress mappings to calculate. As the state space grows bigger, Sweep-Line$_A$ gradually becomes more efficient. When the size of the original state space is large, for example in configurations A-(2,1,x,x) and C-(1,0,1,0), Sweep-Line$_A$ is faster than Sweep-Line$_S$ even though the augmented model has a bigger state space. This is because Sweep-Line$_A$ spends less time comparing new states to existing states (due to storing fewer states in memory at any one time).

5.3 Terminal marking classification

Terminal markings (dead markings) are states from which no action can occur. Undesired terminal markings are called deadlocks. Table 4 shows the terminal markings of Configuration A. All terminal markings have no packets left in the channels and hence there are no unspecified receptions. The terminal markings are classified into four types. Type I terminal markings arise when both the client and server are in the OPEN state indicating that the connection has been successfully established. Terminal markings Types II and III arise in situations when the connection attempt fails because the back-off timer[5] expires. Both types of terminal markings are acceptable. In Type II terminal markings both sides are CLOSED. For Type III terminal markings, the client is CLOSED, but the server is in the LISTEN state. This can happen when the server is initially CLOSED (down for maintenance or busy) and rejects the connection request. The server then recovers and moves to the LISTEN state while the client finishes in CLOSED on receipt of the reset. These three types of terminal markings are expected. However, Type IV terminal markings are undesired deadlocks where the client is CLOSED but the server stays in OPEN. Hence, allowing up to two retransmissions of each packet type has not eliminated the deadlocks for the case where ISS is $2^{48} - 3$.

Every scenario of configuration B and C has two Type II and one Type III terminal marking. One terminal marking of Type II has a close command left in the place App_Server while the other has no token left in this place. There are no Type IV terminal markings in configurations B and C. This shows that these deadlocks can be overcome by the server closing the connection.

These experiments were conducted initially for an ISS of $2^{48} - 3$. It is infeasible to generate the state space for every one of the 2^{48} values of ISS (0 to $2^{48} - 1$), however, the initial experiments have been followed up with further experiments for other ISS values that cause sequence numbers to wrap. All give similar deadlock results. Thus we conjecture that when sequence numbers wrap: (1) when the maximum number of retransmissions is two, connection establishment still has an undesired deadlock, and (2) when the maximum number of retransmissions is one, the deadlocks do not occur when the application on the server issues a close command (either "active close" or "server active close").

6 Conclusions and future work

In the first part of this paper we provided the first formal specification of DCCPs connection management procedures. Significant effort was spent in ensuring that it truly reflects the procedures defined in RFC 4340. The complexity of the specification is managed using the hierarchical constructs of CPNs. The structure of the specification has been refined during iterative development of the model. In addition to the main procedures for connection establishment and release, our CPN specification takes into account DCCPs synchronization procedures, non-deterministic choice of Ack, DataAck or Data packets, the use of long and short sequence numbers and modulo arithmetic to correctly model sequence and acknowledgement numbers and their operations, and the use of a back-off timer in conjunction with retransmissions. The specification provides sufficient detail to allow all the connection management procedures to be analysed. Our CPN model has been used to find various ambiguities and technical errors [18,20] in various Internet drafts as DCCP was being developed. Those deficiencies were reported to the Internet Engineering Task Force and rectified in RFC 4340.

This CPN specification is used in the second part of this paper to illustrate how the sweep-line method can be used to analyse an industrially relevant protocol. We give some insight into how to determine sources of progress for our DCCP-CM CPN model. We believe this approach could be applied to other transport protocols. While the main stream approaches try to reduce the size of the state space, we explode it by augmenting the model with additional state information in such a way that the exploded state space has a structure which is easier for the sweep-line to explore. This gives the very promising result of significant reduction in both peak states stored and exploration time when analysing large state spaces. The efficiency of this method increases as the original state space gets bigger.

We also successfully extended the analysis of DCCP connection management to *five* configurations (see Table 3) that were previously out of reach. In this paper we confirmed for an ISS value of $2^{48} - 3$ that when the maximum number of retransmissions is two, that connection establishment still has an undesired deadlock. We also confirmed for this ISS that when the maximum number of retransmissions is one, the deadlocks do not occur when the application on the server issues a close command (either "active close" or "server active close"). It is infeasible to perform this kind of analysis for all values of ISS, however, our experiments so far suggest that this behaviour holds for values of ISS where sequence numbers wrap.

The work presented in this paper provides some evidence that it may be possible to exploit the full potential of the sweep-line method. We hope it will be possible in future to craft progress mappings that will only need several thousand states to be stored in memory, even when the state space is huge (i.e., greater than 10^9 states), and so make the sweep-line a practical verification technique. The development of a structured approach to obtain such progress mappings and to perform the necessary model transformations is an open research question.

[5] After retrying for a period (measured by a "back-off" timer), the client will send a DCCP-Reset and will "back off" to the CLOSED state [19].

Acknowledgments The authors are grateful to the anonymous reviewers whose comments have helped to improve the quality of the paper, and who have provided some interesting challenges for future work. The comments of the editor have also improved the quality of this paper and are gratefully acknowledged.

Appendix: Declaration of DCCP-CM specification model

```
(*******************************************************************************)
(*  Program : Declaration Page of DCCP-CM CPN Model - RFC-4340                 *)
(*  Author  : Somsak Vanit-Anunchai                                            *)
(*            School of Electrical and Information Engineering                 *)
(*            Computer Systems Engineering Center                              *)
(*  Date    : 23/01/07                                                         *)
(*  Copyright (c) University of South Australia 2005-2007                      *)
(*******************************************************************************)
(******************* Define constant values ************************************)
val ZERO               = IntInf.fromInt(0);
val ONE                = IntInf.fromInt(1);
val max_seq_no24       = 16777215; (* 24-bit sequence number *)
val MaxSeqNo48plus1    = IntInf.pow(IntInf.fromInt(2),48);
val MaxSeqNo24plus1    = IntInf.pow(IntInf.fromInt(2),24);
val MaxSeqNo48         = IntInf.-( MaxSeqNo48plus1,IntInf.fromInt(1));
val MaxSeqNo24         = IntInf.-( MaxSeqNo24plus1,IntInf.fromInt(1));
                                   (* required for function Wrap *)
color BOOL             = bool;
color SN               = IntInf;   (* required for function Wrap *)

(*********************** Defining the packet structure *************************)
(*                 Packet types                                                *)
color PktType1      = with Request | Data;
color PktType2      = with Sync | SyncAck | Response | Ack | DataAck
                                | CloseReq | Close | Rst;
color DATA          = subset PktType1 with [Data];
color ACK_DATAACK   = subset PktType2 with [Ack, DataAck];

(********************** Extended sequence number bit ***************************)
color X             = with LONG | SHORT;

(************ Sequence and Acknowledgement Numbers (Long/Short) ****************)
color SN48          = IntInf with ZERO..MaxSeqNo48;
color SN48_AN48     = record SEQ:SN48*ACK:SN48;
color SN24          = int with 0..max_seq_no24;
color SN24_AN24     = record SEQ:SN24*ACK:SN24;

(************************ Four Kinds of Packet *********************************)
color Type1LongPkt  = product PktType1*X*SN48;                        (* Type1*)
color Type2LongPkt  = product PktType2*X*SN48_AN48;                   (* Type2*)
color Type1ShortPkt = product DATA*X*SN24;                            (*Type1s*)
color Type2ShortPkt = product ACK_DATAACK*X*SN24_AN24;                (*Type2s*)
color PACKETS       = union PKT1:Type1LongPkt
                          + PKT2:Type2LongPkt
                          + PKT1s: Type1ShortPkt
                          + PKT2s:Type2ShortPkt
            declare of_PKT1, of_PKT2, of_PKT1s, of_PKT2s;

(******************** Declare Packet Variables *********************************)
var p_type1:PktType1;
var p_type2:PktType2;
var p_type2s:ACK_DATAACK;
var ack_dataack:ACK_DATAACK;
var sn:SN48;
var sn_an:SN48_AN48;
var sn24:SN24;
var sn24_an24:SN24_AN24;
var packet:PACKETS;
```

```
(****************************************************************************)
(* DCCP state variables                                                    *)
(* Retransmission Counter; Greatest Sequence Numbers; Initial Sequence Numbers***)
color RCNT      = int;
color GS        = record GSS:SN48*GSR:SN48*GAR:SN48;
color ISN       = record ISS:SN48*ISR:SN48;

(* Major States                                                            *)
color IDLE      = with CLOSED_I | LISTEN | TIMEWAIT | CLOSED_F;
color REQUEST   = product RCNT*SN48*SN48;              (* Counter x GSS x ISS  *)
color ACTIVE    = with  RESPOND | PARTOPEN | S_OPEN | C_OPEN | CLOSEREQ
                              | C_CLOSING |S_CLOSING;

color ActiveStatexRCNTxGSxISN = product ACTIVE*RCNT*GS*ISN;
(* Control Block                                                           *)
color CB        = union  IdleState:IDLE
                       + ReqState:REQUEST
                       + ActiveState:ActiveStatexRCNTxGSxISN
        declare of_IdleState, of_ReqState, of_ActiveState;
(* User Commands                                                           *)
color COMMAND = with p_Open | a_Open | server_a_Close | a_Close;

(******************** Declare State Variables *********************************)
var rcnt:RCNT;
var gss,gsr,gar,iss,isr:SN48;
var g:GS;
var isn:ISN;
var id_state:IDLE;
var active_state:ACTIVE;
var cb:CB;
var LS:BOOL;
(****************************************************************************)

(* Redefines two data types (seq_ack and state variable) to allow ML functions
   to operate on different types of variable (different color sets)         *)

datatype seq_ack        = NoGS | S48 of SN48 | SA48 of SN48_AN48
                               | S24 of SN24 | SA24 of SN24_AN24;
datatype state_variable = NoGS | gssGS of SN48 | gGS of GS;

(* Remark:  functions that operate while the sequence number wraps are      *)
(* incr, Wrap, Update, extend_seq, PktValid, ReqDataValid, RstValidinReqState  *)

(****************************************************************************)
(* Function : incr(snl)                                                    *)
(* Purpose  : Increasing a 48-bit sequence number by one                   *)
(*            if sequence number equals 2^48-1 then go back to 0           *)
(* Input    : snl is a 48-bit sequence number                              *)
(* Output   : snl+1 and if sequence number equals 2^48-1 then rolls to 0   *)
(*                                                                         *)
fun incr(snl):SN48 =
        if (snl = MaxSeqNo48)  then ZERO  else (IntInf.+(snl, ONE));

(****************************************************************************)
(* Function : Wrap(s,Max) = modulo(s,max)                                  *)
(* Purpose  : Folding the number outside [0..Max] back into the range      *)
(* Input    : When MaxValue = 2^48, s = an integer in [-2^47..2^48+2^47-1]   *)
(*             MaxValue = 2^24, s = an integer in [-2^23..2^24+2^23-1]     *)
(* Output   : an integer in [0..2^48-1]                                    *)

fun Wrap(seq_b:SN, MaxValue):SN48 =
    if IntInf.<(seq_b,ZERO) then IntInf.+(seq_b,MaxValue)
       else (if IntInf.>(seq_b,IntInf.-(MaxValue,ONE))
               then IntInf.-(seq_b,MaxValue)
               else seq_b);

(****************************************************************************)
```

```
(*  Pre-defined constants for sequence number validity check                   *)
(*      when sequence number wraps                                              *)
(* client's/server's sequence/ack window size                                   *)
val w  = 100.0;         val aw = 100.0;
val center          = IntInf.pow(IntInf.fromInt(2),47);
val quarter         = RealToIntInf 0 (Real.realFloor(w/4.0));
val swl             = IntInf.-(IntInf.+(center,ONE),quarter);
val three_quarter   = RealToIntInf 0 (Real.realCeil((w*3.0)/4.0));
val swh             = IntInf.+(center,three_quarter);
val awl             = IntInf.-(IntInf.+(center,ONE),RealToIntInf 0 aw);
val awh             = center;

(*******************************************************************************)
(* Function : Update(new,old)                                                   *)
(* Purpose  : comparing received seq(ack) with GSR(GAR)                         *)
(*            even when sequence number wraps.                                  *)
(*            center   = new - bias                                             *)
(*            old_bias = old - bias (modulo 2^48)                               *)
(*            if center > old_bias then new else old                            *)
(*                                                                              *)
(* Input    : seq(ack), GSR(GAR)                                                *)
(* Output   : GSR(GAR)                                                          *)
(*                                                                              *)
fun Update(new:SN48, old:SN48):SN48 =
let
    val bias        = IntInf.-(new,center);             (* center = new - bias *)
    val old_bias    = Wrap(IntInf.-(old,bias),MaxSeqNo48plus1);
in
    if IntInf.>(center,old_bias) then new else old
end;

(*******************************************************************************)
(* Function : extend_seq(REF,S)                                                 *)
(* Purpose  : Extending 24-bit to 48-bit sequence numbers                       *)
(*        according to section 7.6 page 55 of RFC 4340                          *)
(* Input    : REF is GSR(GSS), S is a 24-bit seq (ack) number                   *)
(* Output   : A 48-bit seq (ack) number                                         *)
(* Description: comparing 24 lower bit of REF with S linearly and circularly    *)
(*          (modulo 2^24).                                                      *)
(* 1. If S is less than REF_low due to S roll over 2^24                         *)
(*                    ____________________                                      *)
(*         then SN48 = | REF_hi+1 |    S    |                                   *)
(*                    |__________|_________|                                    *)
(* 2. If REF_low is less than S due to REF_low roll over 2^24                   *)
(*                    ____________________                                      *)
(*         then SN48 = | REF_hi-1 |    S    |                                   *)
(*                    |__________|_________|                                    *)
(*                    ____________________                                      *)
(* 3. Otherwise SN48 = | REF_hi   |    S    |                                   *)
(*                    |__________|_________|                                    *)
(*                                                                              *)
fun extend_seq(REF:SN48, sn24:SN24):SN48 =
let
    val S           = IntInf.fromInt(sn24);
    val center24    = IntInf.pow(IntInf.fromInt(2),23);
    val REF_hi      = IntInf.quot(REF,MaxSeqNo24plus1);
    val REF_lo      = IntInf.mod(REF,MaxSeqNo24plus1);
    val bias        = IntInf.-(REF_lo,center24);
    val s_bias      = Wrap(IntInf.-(IntInf.fromInt(sn24),bias),MaxSeqNo24plus1);
in
     if IntInf.<(S,REF_lo)  andalso IntInf.<(center24,s_bias)
          then IntInf.+(IntInf.*(Wrap(IntInf.+(REF_hi,ONE),MaxSeqNo24plus1)
                              ,MaxSeqNo24plus1),S)
          else (if IntInf.>(S,REF_lo) andalso IntInf.>(center24,s_bias)
                    then IntInf.+(IntInf.*(Wrap(IntInf.-(REF_hi,ONE)
                               ,MaxSeqNo24plus1)
                                    ,MaxSeqNo24plus1)
                             ,S)
                    else IntInf.+(IntInf.*(REF_hi,MaxSeqNo24plus1),S))
end;

(***************************************************************************)
(* Function : extendSA                                                     *)
(* Purpose  : Extending both seq and ack numbers                           *)
(* Input    : {GSS,GSR,GAR} and 24-bit {seq,ack}                           *)
(* Output   : 48-bit {seq,ack}                                             *)
(*                                                                         *)
fun extendSA(g:GS,sn24_an24:SN24_AN24):SN48_AN48    =
                {SEQ=extend_seq(#GSR(g),#SEQ(sn24_an24)),
                 ACK=extend_seq(#GSS(g),#ACK(sn24_an24))};

(***************************************************************************)
(* Function : mod_sn24                                                     *)
(* Purpose  : Extracting the lower 24 bits from a 48 bit sequence number   *)
(* Input    : A 48-bit sequence number                                     *)
(* Output   : A 24-bit sequence number                                     *)
(*                                                                         *)
fun mod_sn24(sn48:SN48):SN24 = IntInf.toInt( IntInf.mod(sn48,MaxSeqNo24plus1));

(***************************************************************************)
(* Function : SeqAck                                                       *)
(* Purpose  : Computing 48-bit {seq,ack} an outgoing packet                *)
(* Input    : NoGS/gss/{GSS,GSR,GAR}, Receiving 24-bit/48-bit of seq/{seq,ack}  *)
(* Output   : 48-bit {seq,ack} for an outgoing packet                      *)
(*                                                                         *)
fun SeqAck(NoGS ,SA48 sn_an)      = {SEQ=incr( #ACK(sn_an)), ACK = #SEQ(sn_an)}
  | SeqAck(NoGS, S48 sn)          = {SEQ=ZERO, ACK=sn}
  | SeqAck(NoGS,SA24 sn24_an24)   =
        {SEQ = IntInf.mod(IntInf.fromInt(#ACK(sn24_an24)+1),MaxSeqNo24plus1),
         ACK = IntInf.fromInt(#SEQ(sn24_an24))}
  | SeqAck(NoGS,S24 sn24)         = {SEQ = ZERO, ACK = IntInf.fromInt(sn24)}
  | SeqAck(gssGS gss, SA48 sn_an) = {SEQ = incr(gss), ACK = #SEQ(sn_an)}
  | SeqAck(gGS g,SA48 sn_an)      =
        {SEQ = incr(#GSS(g)),ACK = Update(#SEQ(sn_an),#GSR(g))}
  | SeqAck(gGS g,S48 sn)          = {SEQ = incr(#GSS(g)),ACK = Update(sn,#GSR(g))}
  | SeqAck(gGS g,SA24 sn24_an24)  =
        {SEQ = incr(#GSS(g)),
         ACK = Update(#GSR(g),extend_seq(#GSR(g),#SEQ(sn24_an24)))}
 | SeqAck(gGS g,S24 sn24)=
        {SEQ = incr(#GSS(g)),ACK = Update(#GSR(g),extend_seq(#GSR(g),sn24))};
(***************************************************************************)
(* Function : ShortSeqAck                                                  *)
(* Purpose  : Computing 24-bit {seq,ack} for an outgoing packet            *)
(* Input    : NoGS/gss/{GSS,GSR,GAR}, Receiving 24-bit/48-bit of seq/{seq,ack}  *)
(* Output   : 24-bit {seq,ack} for an outgoing packet                      *)
(*                                                                         *)
fun ShortSeqAck(gssGS gss, SA48 sn_an)  =
        {SEQ=mod_sn24(incr(gss)), ACK=mod_sn24(#SEQ(sn_an))}
  | ShortSeqAck(gGS g,SA48 sn_an)       =
        {SEQ=mod_sn24(incr(#GSS(g))),
         ACK=mod_sn24(Update(#SEQ(sn_an),#GSR(g)))}
  | ShortSeqAck(gGS g,S48 sn)           =
        {SEQ=mod_sn24(incr(#GSS(g))), ACK=mod_sn24(Update(sn,#GSR(g)))}
  | ShortSeqAck(gGS g,SA24 sn24_an24)   =
        {SEQ=mod_sn24(incr(#GSS(g))),
         ACK=mod_sn24(Update(#GSR(g),extend_seq(#GSR(g),#SEQ(sn24_an24))))}
  | ShortSeqAck(gGS g,S24 sn24)         =
        {SEQ=mod_sn24(incr(#GSS(g))),
         ACK=mod_sn24(Update(#GSR(g),extend_seq(#GSR(g),sn24)))};
```

```
(******************************************************************************)
(* Function : SndRstInReq (Send Reset pkt in the REQUEST state)               *)
(* Purpose  : Computing an outgoing Reset packet in the REQUEST state         *)
(* Input    : GSS                                                             *)
(* Output   : An outgoing Reset packet with 48-bit {seq, ack}                 *)
(*                                                                            *)
fun SndRstInReq(gss:SN48) = 1`PKT2(Rst,LONG,{SEQ=incr(gss),ACK=ZERO});

(******************************************************************************)
(* Function : SyncSnd                                                         *)
(* Purpose  : Computing an outgoing Sync packet                               *)
(* Input    : {GSS,GSR,GAR}, Receiving 24-bit/48-bit of seq/{seq,ack}         *)
(* Output   : An outgoing Sync packet with 48-bit {seq, ack}                  *)
(*                                                                            *)
fun SyncSnd(g:GS,SA48 sn_an)        =
        1`PKT2 (Sync,LONG,{SEQ=incr(#GSS(g)),ACK = #SEQ(sn_an)})
  | SyncSnd(g:GS,S48 sn)            =
        1`PKT2 (Sync,LONG,{SEQ=incr(#GSS(g)),ACK=sn})
  | SyncSnd(g:GS,SA24 sn24_an24)    =
        1`PKT2 (Sync, LONG,
               {SEQ=incr(#GSS(g)),ACK=extend_seq(#GSR(g),#SEQ(sn24_an24))})
  | SyncSnd(g:GS,S24 sn24)          =
        1`PKT2 (Sync,LONG,{SEQ = incr(#GSS(g)),ACK = extend_seq( #GSR(g),sn24)});
(***** In CLOSED, LISTEN, REQUEST and TIMEWAIT states, no Sync is sent out ******)

(******************************************************************************)
(* Function : UpdateGS                                                        *)
(* Purpose  : Computing {GSS,GSR,GAR} of the next active state                *)
(* Input    : Existing {GSS,GSR,GAR}, Receiving 24-bit/48-bit of seq/{seq,ack}  *)
(* Output   : {GSS,GSR,GAR} of the next active state                          *)
(*                                                                            *)
fun UpdateGS(g:GS, SA48 sn_an)  = {GSS=incr(#GSS(g)),
                                   GSR=Update(#SEQ(sn_an),#GSR(g)),
                                   GAR=Update(#ACK(sn_an),#GAR(g))}
   | UpdateGS(g:GS, S48 sn)     = {GSS=incr(#GSS(g)),
                                   GSR=Update(sn,#GSR(g)),
                                   GAR= #GAR(g)}
   | UpdateGS(g:GS, SA24 sn24_an24) =
                {GSS=incr(#GSS(g)),
                 GSR=Update(#GSR(g),extend_seq(#GSR(g),#SEQ(sn24_an24))),
                 GAR=Update(#GAR(g),extend_seq(#GSS(g),#ACK(sn24_an24)))}
   | UpdateGS(g:GS, S24 sn24) =
                     {GSS=incr(#GSS(g)),
                      GSR=Update(#GSR(g),extend_seq(#GSR(g),sn24)),
                      GAR= #GAR(g)};
(******************************************************************************)
(* Function : incrGSS                                                         *)
(* Purpose  : When receiving sequence-invalid pkt, the sender does not change   *)
(*        its   GSR and GAR but increments GSS.                               *)
(* Input    : Existing {GSS,GSR,GAR}, Receiving 24-bit/48-bit of seq/{seq,ack}  *)
(* Output   : {GSS,GSR,GAR} of the next active state                          *)
fun incrGSS(g:GS) =  {GSS=incr(#GSS(g)),GSR = #GSR(g),GAR = #GAR(g)};

(********************* Set up Initial markings ********************************)
val C_iss = IntInf.-(MaxSeqNo48plus1,IntInf.fromInt(3);
val S_iss = IntInf.-(MaxSeqNo48plus1,IntInf.fromInt(3);
val C_cmd = 1`a_Open;                  val S_cmd = 1`p_Open;

(* Client/Server initial state *)
val init_C =1`IdleState  CLOSED_I;      val  init_S = 1`IdleState CLOSED_I;

(* Set up parameters: MaxRetrans; Allow to use short sequence number          *)
val MaxRetransRequest       =1;         val MaxRetransAckDataAck    =1;
val MaxRetransCloseReq      =1;         val MaxRetransClose     =1;
val ShortEnable     = false;
(******************************************************************************)
```

```
(* Function : BackOff                                                      *)
(* Purpose  : Check if the retransmission counter has reached the MaxRetrans    *)
(* Input    : State, retransmission counter                                *)
(* Output   : true/false                                                   *)

fun BackOff(state,rcnt):bool =
    case state of
        RESPOND     =>  true
        | PARTOPEN  => (rcnt=MaxRetransAckDataAck)
        | C_CLOSING => (rcnt=MaxRetransClose)
        | CLOSEREQ  => (rcnt=MaxRetransCloseReq)
        | S_CLOSING => (rcnt=MaxRetransClose)
        |_          => false;

(**************************************************************************)
(* Function : SeqValid                                                     *)
(* Purpose  : Check if the received pkt has a valid sequence number        *)
(* Input    : Received pkt type, received {seq,ack}, {GSS,GSR,GAR},        *)
(*        {ISS, ISR}                                                       *)
(* Output   : true/false                                                   *)

fun SeqValid(p_type2:PktType2, s2:SN48_AN48, g:GS, isn:ISN) =
let
    val bias    = IntInf.-(#GSR(g),center);
    val seq_b   = Wrap(IntInf.-(#SEQ(s2),bias),MaxSeqNo48plus1);
    val isr_b   = Wrap(IntInf.-(#ISR(isn),bias),MaxSeqNo48plus1);
    val SWL     = IntInf.max(swl,isr_b);
    val SWH     = swh;
in
    case p_type2 of
    Response    => IntInf.>=(seq_b,SWL) andalso IntInf.<=(seq_b,SWH)
    | Ack       => IntInf.>=(seq_b,SWL) andalso IntInf.<=(seq_b,SWH)
    | DataAck   => IntInf.>=(seq_b,SWL) andalso IntInf.<=(seq_b,SWH)
    | CloseReq  => IntInf.>(seq_b,center) andalso IntInf.<=(seq_b,SWH)
    | Close     => IntInf.>(seq_b,center) andalso IntInf.<=(seq_b,SWH)
    | Rst       => IntInf.>(seq_b,center) andalso IntInf.<=(seq_b,SWH)
    | Sync      => IntInf.>=(seq_b,SWL)
    | SyncAck   =>  IntInf.>=(seq_b,SWL)
end;
(**************************************************************************)
(* Function : AckValid                                                     *)
(* Purpose  : Check if the received pkt has a valid acknowledgement number      *)
(* Input    : Received pkt type, received {seq,ack}, GSS, GAR, ISS         *)
(* Output   : true/false                                                   *)
(*                                                                         *)
fun AckValid(p_type2:PktType2, s2:SN48_AN48, gss:SN48,gar:SN48, iss:SN48) =
let
    val bias    = IntInf.-(gss,center);
    val gar_b   = Wrap(IntInf.-(gar,bias),MaxSeqNo48plus1);
    val ack_b   = Wrap(IntInf.-(#ACK(s2),bias),MaxSeqNo48plus1);
    val iss_b   = Wrap(IntInf.-(iss,bias),MaxSeqNo48plus1);
    val AWL     = IntInf.max(awl,iss_b);
    val AWH     = center;
in
      case p_type2 of
      Response  =>   IntInf.>=(ack_b,AWL) andalso IntInf.<=(ack_b,AWH)
    | Ack       =>  IntInf.>=(ack_b,AWL) andalso IntInf.<=(ack_b,AWH)
    | DataAck   =>  IntInf.>=(ack_b,AWL) andalso IntInf.<=(ack_b,AWH)
    | CloseReq  =>  IntInf.>=(ack_b,gar_b) andalso IntInf.<=(ack_b,AWH)
    | Close     =>  IntInf.>=(ack_b,gar_b) andalso IntInf.<=(ack_b,AWH)
    | Rst       =>  IntInf.>=(ack_b,gar_b) andalso IntInf.<=(ack_b,AWH)
    | Sync      =>  IntInf.>=(ack_b,AWL) andalso IntInf.<=(ack_b,AWH)
    | SyncAck   =>  IntInf.>=(ack_b,AWL) andalso IntInf.<=(ack_b,AWH)
end;
(**************************************************************************)
```

```
(* Function : ReqDataValid                                                 *)
(* Purpose  : Check if the received Request/Data pkt has a valid seq number     *)
(* Input    : Received pkt type, received seq, {GSS,GSR,GAR}, {ISS,ISR}          *)
(* Output   : true/false                                                   *)
(*                                                                         *)
fun ReqDataValid(p_type1:PktType1, s1:SN48, g:GS, isn:ISN) =
let
    val bias    = IntInf.-(#GSR(g),center);
    val seq_b   = Wrap(IntInf.-(s1,bias),MaxSeqNo48plus1);
    val isr_b   = Wrap(IntInf.-(#ISR(isn),bias),MaxSeqNo48plus1);
    val SWL     = IntInf.max(swl,isr_b);
    val SWH     = swh;
in
    case p_type1 of
      Request => IntInf.>=(seq_b, SWL) andalso IntInf.<=(seq_b,SWH)
    | Data    => IntInf.>=(seq_b ,SWL)  andalso IntInf.<=(seq_b,SWH)
end;

(*******************************************************************************)
(* Function : PktValid                                                     *)
(* Call : SeqValid, AckValid                                               *)
(* Purpose  : Check if the received pkt has both  seq and ack numbers valid     *)
(* Input    : Received pkt type, received {seq,ack}, {GSS,GSR,GAR}, {ISS,ISN}   *)
(* Output   : true/false                                                   *)
(*                                                                         *)
fun PktValid(p_type2:PktType2, s2:SN48_AN48, g:GS, isn:ISN) =
let
    val check_seq = SeqValid(p_type2:PktType2,s2:SN48_AN48, g:GS, isn:ISN);
    val check_ack = AckValid(p_type2:PktType2,s2:SN48_AN48,
                             #GSS(g),#GAR(g),#ISS(isn));
in
        (check_seq) andalso (check_ack)
end;
(*******************************************************************************)
(* Function : RstValidinReqState                                           *)
(* Purpose  : Check if the received Rst pkt (in REQUEST state)             *)
(*            has a valid seq number                                       *)
(* Input    : Received {seq,ack}, GSS, ISS                                 *)
(* Output   : true/false                                                   *)
(*                                                                         *)
fun RstValidinReqState(s2:SN48_AN48, gss:SN48, iss:SN48) =
let
    val bias    = IntInf.-(gss,center);
    val ack_b   = Wrap(IntInf.-(#ACK(s2),bias),MaxSeqNo48plus1);
    val iss_b   = Wrap(IntInf.-(iss,bias),MaxSeqNo48plus1);
    val AWL     = IntInf.max(awl,iss_b);
    val AWH     = center;
in
        IntInf.>=(ack_b, AWL) andalso IntInf.<=(ack_b,AWH)
end;

(****** Misc function used in Sync page *******)
fun go_open(state) = if state = PARTOPEN then C_OPEN else S_OPEN;
(*******************************************************************************)
```

References

1. Billington, J., Gallasch, G.E., Kristensen, L.M., Mailund, T.: Exploiting equivalence reduction and the sweep-line method for detecting terminal states. IEEE Trans. Systems, Man Cybernetics, Part A: Systems Humans **34**(1), 23–37 (2004)
2. Christensen, S., Kristensen, L.M., Mailund, T.: A sweep-line method for state space exploration. In Proceedings of TACAS 2001. Lecture Notes in Computer Science, vol. 2031, pp. 450–464. Springer, Heidelberg (2001)
3. CPN ML: An extension of standard ML. http://www.daimi.au.dk/designCPN/sml/cpnml.html
4. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Formal Methods System Design **9**(1/2), 105–131 (1996)
5. Gallasch, G.E., Billington, J.: Using parametric automata for the verification of the stop-and-wait class of protocols. Proceedings

of ATVA'05. Lecture Notes in Computer Science, vol. 3707, pp. 457–473. Springer, Heidelberg (2005)
6. Gallasch, G.E., Han, B., Billington, J.: Sweep-line analysis of TCP connection management. In Proceedings of ICFEM'05. Lecture Notes Computer Science, vol. 3785 pp. 156–172. Springer, Heidelberg (2005)
7. Gallasch, G.E., Billington, J., Vanit-Anunchai, S., Kristensen, L.M.: Checking safety properties on-the-fly with the sweep-line method. Int. J. Software Tools Technol. Transfer **9**(3–4), 371–391 (2007) (special section on material from CPN'04 and CPN'05)
8. Gallasch, G.E., Ouyang, C., Billington, J., Kristensen, L.M.: Experimenting with progress mappings for the sweep-line analysis of the Internet Open Trading Protocol. In: Fifth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, DAIMI PB 570, pp. 19–38. Department of Computer Science, University of Aarhus, Aarhus October 8–11, (2004). Available via http://www.daimi.au.dk/CPnets/workshop04/cpn/papers/
9. Gallasch, G.E., Vanit-Anunchai, S., Billington, J., Kristensen, L.M.: Checking language inclusion on-the-fly with the sweep-line method. In: Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, DAIMI PB 576, pp. 1–20. Department of Computer Science, University of Aarhus, Aarhus October 24–26, (2005). Available via http://www.daimi.au.dk/CPnets/workshop05/cpn/papers/
10. Godefroid, P., Holzmann, G.J., Pirottin, D.: State-space caching revisited. Formal Methods System Design **7**(3), 227–241 (1995)
11. Gordon, S., Kristensen, L.M., Billington, J.: Verification of a revised WAP wireless transaction protocol. In: Proceedings of 23rd International Conference on Application and Theory of Petri Nets, Lecture Notes Computer Science, vol. 2360, pp. 182–202. Springer, Heidelberg (2002)
12. Holzmann, G.J.: Design and Validation of Computer Protocols. Prentice Hall, New York (1990)
13. Holzmann, G.J.: An analysis of bitstate hashing. Formal Methods System Design **13**(3), 287–305 (1998)
14. Jensen, K.: Condensed state spaces for symmetrical coloured Petri Nets. Formal Methods System Design **9**(1/2), 7–40 (1996)
15. Jensen, K.: Coloured Petri Nets: basic concepts, analysis methods and practical use. Vol. 1, basic concepts. Monographs in Theoretical Computer Science. Springer, Heidelberg, 2nd edn. (1997)
16. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN tools for modelling and validation of concurrent systems. Int. J. Software Tools Technol. Transfer **9**(3–4), 213–254 (2007) (special section on material from CPN04/05)
17. Kohler, E., Handley, M., Floyd, S.: Datagram Congestion Control Protocol, draft-ietf-dccp-spec-6. Available via http://www.read.cs.ucla.edu/dccp/draft-ietf-dccp-spec-06.txt, February (2004)
18. Kohler, E., Handley, M., Floyd, S.: Substantive differences between draft-ietf-dccp-spec-11 and draft-ietf-dccp-spec-12. Available via http://www.read.cs.ucla.edu/dccp/diff-spec-11-12-explain.txt, December (2005)
19. Kohler, E., Handley, M., Floyd, S.: Datagram Congestion Control Protocol, RFC 4340. Available via http://www.rfc-editor.org/rfc/rfc4340.txt, March (2006)
20. Kohler, E., Handley, M., Floyd, S.: Substantive differences between draft-ietf-dccp-spec-13 and RFC 4340. Available via http://www.read.cs.ucla.edu/dccp/diff-spec-13-rfc-explain.txt, March 2006
21. Kohler, E., Handley, M., Floyd, S., Padhye, J.: Datagram Congestion Control Protocol, draft-ietf-dccp-spec-5. Available via http://www.read.cs.ucla.edu/dccp/draft-ietf-dccp-spec-05.txt, October 2003
22. Kongprakaiwoot, T.: Verification of the Datagram Congestion Control Protocol using coloured petri nets. Master's thesis, Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, South Australia (2004)
23. Kristensen, L.M., Mailund, T.: A generalised sweep-line method for safety properties. In: Proceedings of FME'02. Lecture Notes in Computer Science, vol. 2391, pp. 549–567. Springer, Heidelberg (2002)
24. Kristensen, L.M., Mailund, T.: Efficient path finding with the sweep-line method using external storage. In: Proceedings of ICFEM'03. Lecture Notes in Computer Science, vol. 2885, pp. 319–337. Springer, Heidelberg (2003)
25. Kristensen, L.M., Christensen, S., Jensen, K.: The practitioner's guide to Coloured Petri Nets. Int. J. Software Tools Technol. Transfer **2**(2), 98–132 (1998)
26. Milner, R., Harper, R., Tofte, M.: The definition of standard ML. MIT Press, New York (1990)
27. Design/CPN Online. http://www.daimi.au.dk/designCPN/
28. Parashkevov, A.N., Yantchev, J.: Space efficient reachability analysis through use of pseudo-root states. In: Proceedings of TACAS'97, Lecture Notes in Computer Science, vol. 1217, pp. 50–64. Springer, Heidelberg (1997)
29. Peled, D.: All from one, one for all: on Model checking using representatives. In: Proceedings of CAV'93, Lecture Notes in Computer Science, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
30. Valmari, A.: A stubborn attack on state explosion. In: Proceedings of CAV'90, Lecture Notes in Computer Science, vol. 531, pp. 156–165. Springer, Heidelberg (1990)
31. Valmari, A.: The state explosion problem. In: Lectures on Petri Nets I: Basic Models, Lecture Notes in Computer Science, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
32. Vanit-Anunchai, S., Billington, J.: Initial result of a formal analysis of DCCP connection management. In: Proceedings of Fourth International Network Conference (INC 2004), pp. 63–70, University of Plymouth, Plymouth, 6–9 July 2004
33. Vanit-Anunchai, S., Billington, J.: Effect of sequence number wrap on DCCP connection establishment. In: Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 345–354 IEEE Computer Society Press, Monterey, 11–13 September 2006
34. Vanit-Anunchai, S., Billington, J.: Chattering behaviour in the Datagram Congestion Control Protocol. IEE Electron. Lett. **41**(21), 1198–1199 (2005)
35. Vanit-Anunchai, S., Billington, J.: Modelling the Datagram Congestion Control Protocol's connection management and synchronization procedures. In: Proceedings of 28th International Conference on Application and Theory of Petri Nets and other models of concurrency (ICATPN'07), Lecture Notes in Computer Science, vol. 4546, pp. 423–444. Springer, Heidelberg (2007)
36. Vanit-Anunchai, S., Billington, J., Gallasch, G.E.: Sweep-line analysis of DCCP connection management. In: Seventh workshop and tutorial on practical use of coloured petri nets and the CPN tools, DAIMI PB-579, pp. 157–175. Department of Computer Science, University of Aarhus, Aarhus 24–26 October (2006). Available via http://www.daimi.au.dk/CPnets/workshop06/cpn/papers/
37. Vanit-Anunchai, S., Billington, J., Kongprakaiwoot, T.: Discovering chatter and incompleteness in the Datagram Congestion Control Protocol. In: Proceedings of FORTE'05, Lecture Notes in Computer Science, vol. 3731, pp. 143–158. Springer, Heidelberg (2005)
38. Wolper, P., Godefroid, P.: Partial order methods for temporal verification. In: Proceedings of CONCUR'93, Lecture Notes in Computer Science, vol. 715, pp. 233–246. Springer, Heidelberg (1993)
39. Wolper, P., Leroy, D.: Reliable hashing without collision detection. In: Proceedings of CAV'93, Lecture Notes in Computer Science, vol. 697, pp. 59–70. Springer, Heidelberg (1993)