T U M

I N S T I T U T F Ü R I N F O R M A T I K

The SDL Specification of the Sliding Window Protocol Revisited

Christian Facchi, Markus Haubner, Ursula Hinkel

TUM-I9614
März 1996

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

The SDL Specification of the Sliding Window Protocol Revisited

Christian Facchi[*], Markus Haubner, Ursula Hinkel
Institut für Informatik
Technische Universität München
D-80290 München
{facchi,haubnerm,hinkel}@informatik.tu-muenchen.de

March 31, 1996

Abstract

We present the results of a case study in which the use of SDL tools was analysed on the basis of the sliding window protocol. We chose the SDL specification of the protocol which was first published by the ISO. While editing and simulating the SDL specification we found out that the specification contains significant errors and does not meet the informal description of the protocol. We describe these errors and give a correct version of the SDL specification.

1 Introduction

CCITT[1] and ISO have standardized the formal description techniques (*FDT*) Estelle, LOTOS, SDL and MSC for introducing formal methods in the area of distributed systems in order to improve their quality. The specification and description language SDL is one of them. SDL is a widespread specification language, which, in our opinion due to its graphical notation and structuring concepts, is well-suited for the formulation of large and complicated specifications of distributed systems.

We will present some results of [Hau95] in which the use of SDL tools is analysed. Because of its practical relevance and simplicity we chose the sliding window protocol as a case study for specification. An SDL description of the sliding window protocol is given in [ISO91, Tur93]. The goal of [Hau95] was the transformation of this description

[*]New address: Siemens AG, PN KE TCP31, D-81359 München, Christian.Facchi@pn.siemens.de

[1]In 1993 the CCITT became the Telecommunication Standards Sector of the International Telecommunication Union (ITU-T). If a document has been published by CCITT, this organization name is used instead of ITU-T in the sequel.

to the representation of the tools examining the functionality and the user adequacy of the tools. This was followed by an examination what features the tools offer for simulation and testing of SDL specifications. While editing and simulating the protocol we found some incorrect parts within the specification of the sliding window protocol [ISO91, Tur93]. The specification does not meet the informal description of the protocol in [ISO91, Tur93]. We will explain these discrepancies by examples which we drew of the simulation. Then we will present a corrected specification with respect to the previously found errors.

This paper is organized as follows. In Section 2 we will give an informal introduction to the sliding window protocol. Section 3 contains an overview of SDL. The main part of this paper describes the errors that we found and their correction in Section 4. Section 5 summarizes the results and draws a conclusion.

2 The Sliding Window Protocol

The sliding window protocol is a widespread protocol describing one possibility of the reliable information exchange between components. The sliding window protocol can be used within the data link layer of the ISO/OSI basic reference model [ISO84]. Due to its purpose it describes a point to point connection of two communication partners without an intermediate relay station. The latter aspect is dealt with in higher layers of the ISO/OSI basic reference model. Note that the connection establishment and disconnection phase are not part of the sliding window protocol. It serves only to establish a bidirectional reliable and order preserving data transfer within an existing connection.

The basic principle of a sliding window protocol is the usage of a sending and receiving buffer. For the sender it is possible to transmit more than one message while awaiting an acknowledgement for messages which have been transmitted before. In hardware description an equivalent property is called *pipelining*.

The protocol can be described as follows ([Stø95]): The sender and the receiver communicate via channels that are lossy in the sense that messages may disappear. Messages may also be corrupted which has to be detectable by the protocol entity. Each message is tagged with a sequence number. The sender is permitted to dispatch a bounded number of messages with consecutive tags while awaiting their acknowledgements. The messages are said to fall within the sender's window. At the other end, the receiver maintains a receiver's window, which contains messages that have been received but which to this point in time cannot be output because some message with a lower sequence number is still to be received. The receiver repeatedly acknowledges the last message it has successfully transferred to the receiving user by sending the corresponding sequence number back to the sender.

We demonstrate the advantages of the sliding window protocol by an example: Station A wants to transmit 3 frames to its peer station B. Station A sends the frames 1, 2 and 3 without waiting for an acknowledgement between the frames. Having received the three frames station B responds by sending an acknowledgement for frame 3 to station

A.

Due to its practical relevance there exist a number of formal descriptions of the sliding window protocol [ISO91, MV93, Tur93, vdS95] in different specification techniques e.g. SDL, LOTOS, Estelle and process algebras.

For evaluating the SDL tools we selected the SDL specification of the sliding window protocol [ISO91, Tur93]. It is based on a *sliding window protocol using "go back n"* according to [Tan88]. For simplicity we describe only a unidirectional flow of data. Thus it is possible to distinguish two components: *transmitter* and *receiver*. Note that the flow of acknowledgements is in the opposite direction to the data flow. Each frame is identified by a unique sequence number. As an abstraction of real protocols in which a wrap around may occur an unbounded range of sequence numbers is used in [ISO91, Tur93]. The sequence number is attached to each data frame by the transmitter and it is later used for the acknowledgement and for the determination of the frame's sequential order. The transmitter increments the sequence number for each new data element.

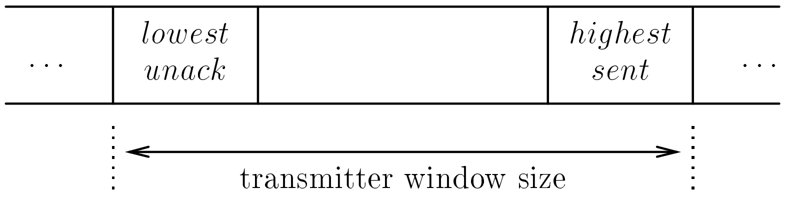


Figure 1: Transmitter window

The transmitter window shown in Figure 1 is used for buffering the unacknowledged frames. The variable *lowestunack* is used as an indicator for the lowest sequence number of an unacknowledged frame which has not necessarily been sent. Initially it is set to 1. The variable *highestsent* indicates the sequence number of the last sent frame and is initialized by 0. Both values determine the size of the transmitting window bounded by the constant *tws*.

If the transmitter wants to send a data frame, then it has to check first whether the actual window size ($highestsent - lowestunack$) is less than *tws*. If this condition is not fulfilled the data frame is not sent until it is possible. In the other case the transmitter increments *highestsent* by one, emits the data combined with *highestsent* as sequence number and starts a timer for that sequence number. Whenever a correct acknowledgement (not corrupted and with a sequence number greater or equal than *lowestunack*) is received, then all timers for frames with lower sequence numbers beginning by the received one down to *lowestunack* are cancelled. Then *lowestunack* is set to the received sequence number incremented by one. When a timeout occurs all timers according to the sequence number of the message for which the timeout has occurred up to *highestsent* are reset and the corresponding frames are retransmitted in a sequential order starting with the message for which the timeout occurred. This includes also the repeated starting of the timers.

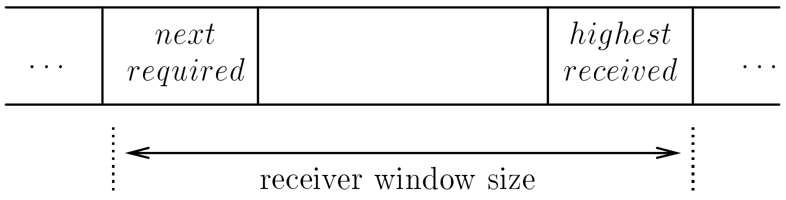


Figure 2: Receiver window

In Figure 2 the second window which is located at the receiver is presented. The receiver window is used to buffer the received frames which can not yet be handed out to the user because some frame with a lower sequence number has not been received. The variable *nextrequired*, whose initial value is 1, is used to indicate the sequence number of the next expected frame. The maximum size of the receiver window is described by the constant *rws*. If a noncorrupted frame is received with a sequence number in the range of $[nextrequired..nextrequired + rws - 1]$ all messages starting by *nextrequired* up to the first not received message are delivered to the user. Then *nextrequired* is set to the number of the first not received message and $nextrequired - 1$ is sent as an acknowledgement to the transmitter.

3 The Specification and Description Language SDL

SDL (Specification and Description Language) is a formal language for the specification of interactive, distributed systems. It provides both a graphical and a textual notation. SDL is intended for the specification of protocols in telecommunication applications, but is now increasingly used in other application areas. SDL is recommended by the CCITT and the ITU-T. Its first version was issued in 1976, the most recent version known as SDL 92 is published as Z.100 ([CCI93]) and includes object oriented extensions. We will give a short overview of SDL which explains the language constructs used in the later description of the Sliding Window Protocol. Detailed introductions of SDL are given in [BH93, OFMP$^+$94].

SDL is used to describe the internal structure, the behaviour and the data of a system. An SDL system description is composed of blocks which are connected with each other and with the environment by channels. A block is a set of processes which are connected with each other and the block environment by signalroutes. Blocks describe the internal structure of a system whereas processes represent the behaviour of a system.

A process is a communicating extended finite state-machine, that is, a communicating finite state automaton with the additional use of data variables. It consists of a finite number of states and transitions connecting these states. A process reacts to stimuli represented by signal inputs. A process is either in a state waiting for input signals or active, performing a transition.

Each process is associated with an input port in which arriving signals are inserted in the order of arrival. Signals which are received by the process at the same time are placed in random order. The input port acts as an unbounded FIFO-queue which holds the signals until they are consumed by the process. Whenever a process is in a state it accepts stimuli from its input port. It removes the first signal from the input port. The consumption of this signal initiates a transition in which the process may execute some actions. The transition is terminated by a state or a stop symbol. Signals which are not explicitly mentioned in a state as stimuli will implicitly be consumed without effect. This results in an empty transition leading back to the same state.

A process may use local data variables which represent its data state. Values of variables are manipulated in tasks. Signals may carry data values. The data concept is based on abstract data types. SDL offers some predefined data types like Boolean, Integer, Real or Charstring. The recommendation Z.105 ([IT95]) defines how the abstract syntax notation ASN.1 ([CCI88]) can be used to describe data and messages in SDL specifications.

SDL provides a timer mechanism. A timer is set with an expiration time during a transition and runs independently from the process. Processes have access to the global system time using the expression NOW. When a timer expires a timer signal is put into the input port. A timer may be reset before its expiration.

A procedure represents a self-contained part of a process. It can be parameterized in the usual way by means of formal parameters. Procedures are useful when the same sequence of states and transitions appears repeatedly in a process specification. A procedure may be called during a transition of a process. The execution of the transition is suspended until the termination of the procedure call.

4 An Analysis of the Sliding Window Protocol

In this section we present the errors that we found in the SDL specification of the sliding window protocol ([ISO91, Tur93]). We will first describe each error in an abstract way and then we will show a scenario in which it occurs.

We give only a short description of the structure of the SDL specification which is presented in full details in [Tur93]. The specification is based on SDL 88. Figure 3 gives an overview of the structure of the specification but omits signals, channel identifiers and data declarations.

The SDL specification of the protocol is composed of three blocks: TransmitterEntity, ReceiverEntity and Medium. The users of the transmitter and the receiver are part of the environment and interact with the system by signals. The two blocks TransmitterEntity and ReceiverEntity communicate via channels with the block Medium. The block Medium models an unreliable medium and is described in Section 4.4.1. The block TransmitterEntity consists of the process Transmitter which includes two proce-

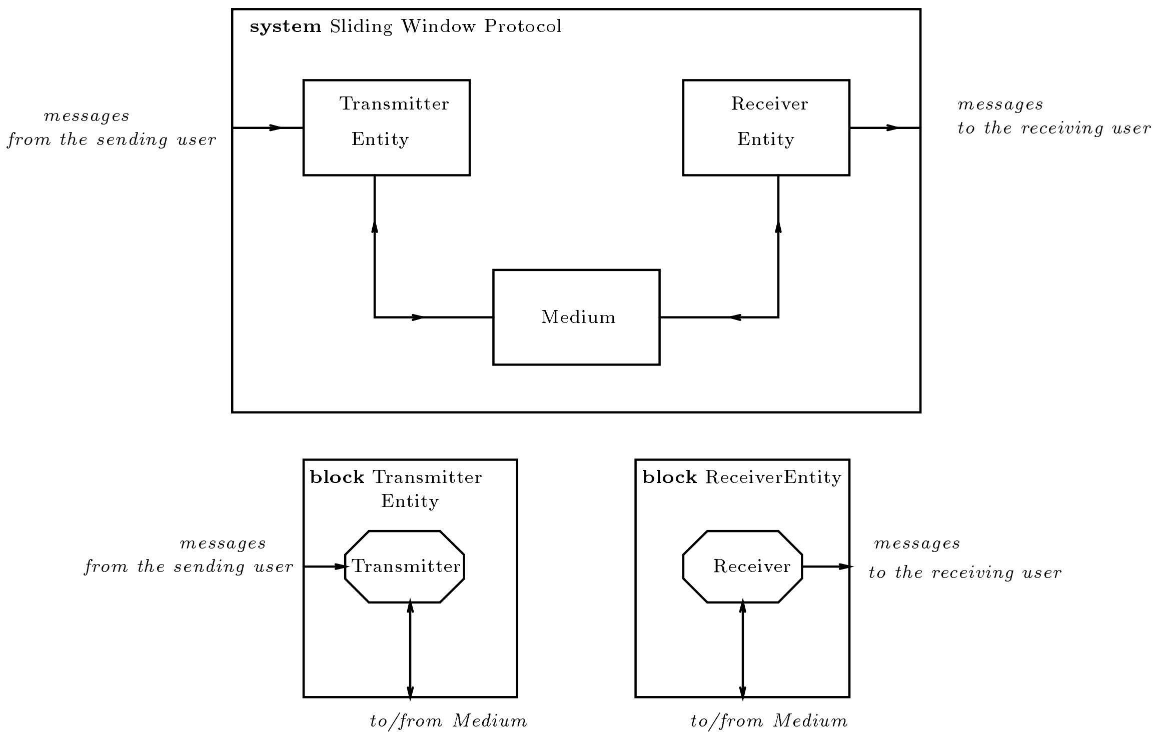

system Sliding Window Protocol
messages from the sending user
Transmitter Entity
Receiver Entity
messages to the receiving user
Medium
block Transmitter Entity
messages from the sending user
Transmitter
to/from Medium
block ReceiverEntity
Receiver
messages to the receiving user
to/from Medium

Figure 3: The structure of the SDL specification

dures: ReleaseTimers and Retransmit. The block ReceiverEntity consists of the process Receiver which includes the procedure DeliverMessages.

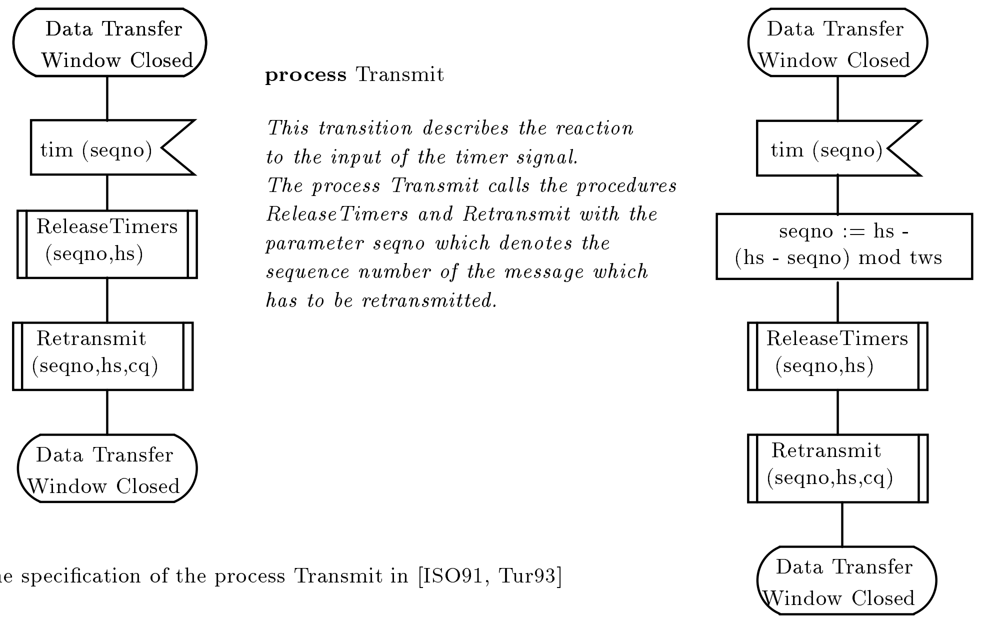
4.1 Errors Concerning the Sequence Number

In the formal description of the sliding window protocol ([ISO91, Tur93]) unbounded sequence numbers are attached to the messages. When it sends a message, the transmitter has to start a timer for that message. Each message is related to an individual timer. However, the number of timers existing at the same time is bounded. In the following we describe an error which is based on this discrepancy.

4.1.1 Description of the Error

In the process Transmitter, after a new message was sent, the timer is set to the sequence number of the message modulo *tws* by the statement "$set(now + delta, tim(hs \bmod tws))$" (*highestsent* is abbreviated by *hs*). However, after a timeout, the parameter of the timer is treated as if it contained the sequence number itself and not the modulo number (see left diagram in Figure 4).

In the procedure Retransmit the same error occurs. Instead of the sequence number


Data Transfer Window Closed
tim (seqno)
ReleaseTimers (seqno,hs)
Retransmit (seqno,hs,cq)
Data Transfer Window Closed
process Transmit
This transition describes the reaction to the input of the timer signal. The process Transmit calls the procedures ReleaseTimers and Retransmit with the parameter seqno which denotes the sequence number of the message which has to be retransmitted.
Data Transfer Window Closed
tim (seqno)
seqno := hs - (hs - seqno) mod tws
ReleaseTimers (seqno,hs)
Retransmit (seqno,hs,cq)
Data Transfer Window Closed

Part of the specification of the process Transmit in [ISO91, Tur93]

Corrected version of the specification

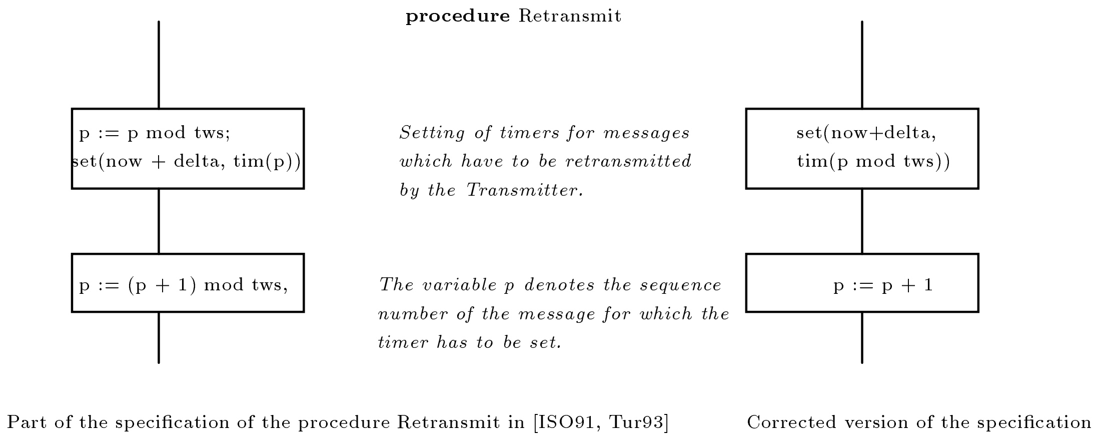
Figure 4: The use of sequence numbers in Process Transmitter

of the retransmitted message the sequence number modulo *tws* is sent and used to set the timer (see left diagram in Figure 5).

The procedure Retransmit calculates the sequence numbers of the messages to retransmit modulo *tws*, so the receiver will not accept retransmitted messages that have sequence numbers which differ from the modulo sequence number.

4.1.2 Erroneous Scenario

Suppose the transmitter window size is 5 and the value of *highestsent* (abbreviated by *hs*) is 12. Suppose further the receiver is waiting for a retransmission of message 11, because message 11 was corrupted. Having received the timer signal, the transmitter will retransmit the messages 11 and 12 with the sequence numbers 11 mod $tws = 1$ and 12 mod $tws = 2$. The receiver already got the messages 1 and 2 , so it will ignore the newly transmitted messages and will still be waiting for message 11. Now the sliding window protocol is in a livelock, where the transmitter will retransmit messages 11 and 12 with sequence numbers 1 and 2 forever and the receiver will never accept them, because their sequence numbers are lower than *nextrequired*.



procedure Retransmit
p := p mod tws;
set(now + delta, tim(p))
Setting of timers for messages which have to be retransmitted by the Transmitter.
set(now+delta, tim(p mod tws))
p := (p + 1) mod tws,
The variable p denotes the sequence number of the message for which the timer has to be set.
p := p + 1
Part of the specification of the procedure Retransmit in [ISO91, Tur93]
Corrected version of the specification

Figure 5: The setting of timers in the procedure Retransmit

4.1.3 Correction of the Specification

In order to solve this problem and to keep the changes to the specification minimal, concerning the process Transmitter we insert the assignment $seqno := hs - (hs - seqno) \bmod tws$ in a task after the input symbol of the timeout signal (see right diagram in Figure 4). It calculates the correct sequence number from the modulo sequence number and *highestsent*, so the correct sequence number will be passed to the procedures ReleaseTimers and Retransmit. In the procedure Retransmit the line "$p := (p + 1) \bmod tws$" is changed into "$p := p + 1$" and in the task "$p := p \bmod tws$; $set(now + delta, tim(p))$" the assignment is removed and the set statement is changed into "$set(now + delta, tim(p \bmod tws))$" (see right diagram in Figure 5).

4.2 Errors Concerning the Closing of the Transmitter Window

The transmitter has only a limited buffer for messages which have been received but have not yet been acknowledged. If this buffer is filled up, the transmitter does not accept any more messages and the transmitter window is closed, as shown in Figure 6.

4.2.1 Description of the Error

In the process Transmitter the transmitter window is closed too late. Even if there are tws unacknowledged messages, $lowestunack + tws$ is greater than $highestsent$ and the window is still open. As a consequence the next message that is sent will also use the timer of the lowest unacknowledged message, although it is still in use. Therefore one timer is used for two different messages. If the lowest unacknowledged message is not received correctly by the receiver the transmitter will not get a timeout for this message. The transmitter will not retransmit the message and the receiver will not pass

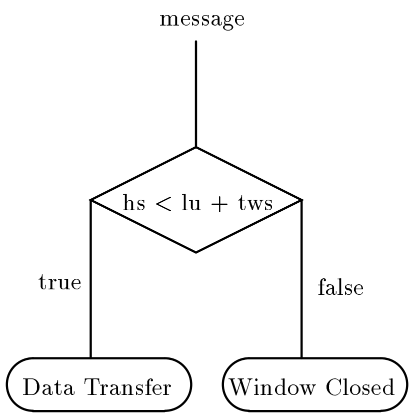


Figure 6: Closing the transmitter window

on any messages until it will have received the missing message. Thus the sliding window protocol is in a livelock.

4.2.2 Erroneous Scenario

Suppose $tws = 5$, $highestsent(hs) = 5$, $lowestunack(lu) = 1$ and the queue is set to $< 1, 2, 3, 4, 5 >$ (five messages have been sent, they are all still unacknowledged)[2]. The transmitter window is full and should have been closed after message 5 had been sent. However, the evaluation of the condition $hs < lu + tws$ $(5 < 1+5)$ in the decision symbol returns true, so the window is not closed. Suppose the transmitter sends message 6. Now the queue $< 1, 2, 3, 4, 5, 6 >$ keeps more than tws elements. As a consequence the timer for message 1 is overwritten with the timer for message 6, because in the set statement $set\,(now + delta,\ tim\,(hs \bmod tws))$ the timer instance 1 is attached to both messages. One message later than expected the condition $hs < lu + tws$ $(6 < 1 + 5)$ evaluates to false and the transmitter window is closed.

4.2.3 Correction of the Specification

The condition $hs < lu + tws$ is changed into $hs < lu + tws - 1$, so the transmitter window will be closed one message earlier, just in time.

[2]Note that the messages are represented only by their sequence numbers. For simplicity we have omitted their content.

4.3 Errors Concerning the Spooling of the Transmitter Queue in the Retransmit Process

During the retransmission of messages spooling of the messages stored in the transmitter queue is necessary, because the message that got the timeout has to be retransmitted first. In the following we describe an error which occurs during this spooling process (see Figure 7).

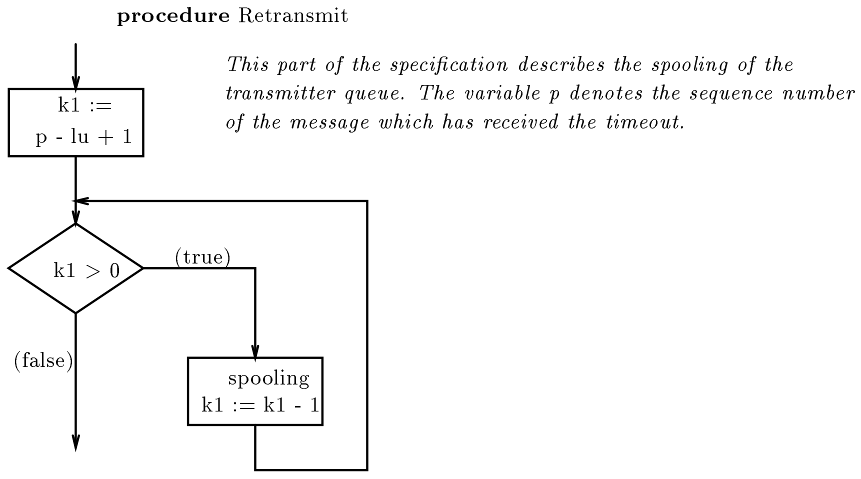

procedure Retransmit
This part of the specification describes the spooling of the transmitter queue. The variable p denotes the sequence number of the message which has received the timeout.
k1 := p - lu + 1
k1 > 0
(true)
(false)
spooling
k1 := k1 - 1

Figure 7: The spooling of the transmitter queue

4.3.1 Description of the Error

In the procedure Retransmit the message queue is spooled to the first message to be retransmitted. However, the calculation of the messages that have to be spooled is incorrect, because the queue is always spooled one message further than it should be. As a result, when the messages are retransmitted the message bodies will not fit to their sequence numbers.

4.3.2 Erroneous Scenario

Suppose a scenario in which four messages are in the transmitter window,
$queue =< 1, 2, 3, 4 >$, $lu = 1$.
Now message 2 receives a timeout, so $p = 2$.
The spooling of the messages in the queue starts:
$k1 := p - lu + 1 = 2 - 1 + 1 = 2$
$k1 = 2 > 0$
The queue is rotated once: $queue =< 2, 3, 4, 1 >$
Despite the fact that the messages are in the correct order the spooling of the messages

continues.
$k1 := k1 - 1 = 1$
$k1 = 1 > 0:$
The queue is rotated a second time: $queue =< 3, 4, 1, 2 >$
$k1 := k1 - 1 = 0$
Now the value of $k1 \ > 0$ is false, the spooling is finished and the retransmission starts. Message 2 is retransmitted with the first element in the queue so the new message has the sequence number 2, but the body of message 3. Sequence number 3 will be combined with message body 4 and sequence number 4 will be sent with the message body 1. As the checksums are calculated after the new combinations the receiver will not notice the altered sequence of the message bodies and the message is corrupted.

4.3.3 Correction of the Specification

To correct the spooling in the procedure Retransmit the calculation of $k1$ has to be $k1 := p - lu$ instead of $k1 := p - lu + 1$ in Figure 7.

4.4 Errors Concerning the Medium

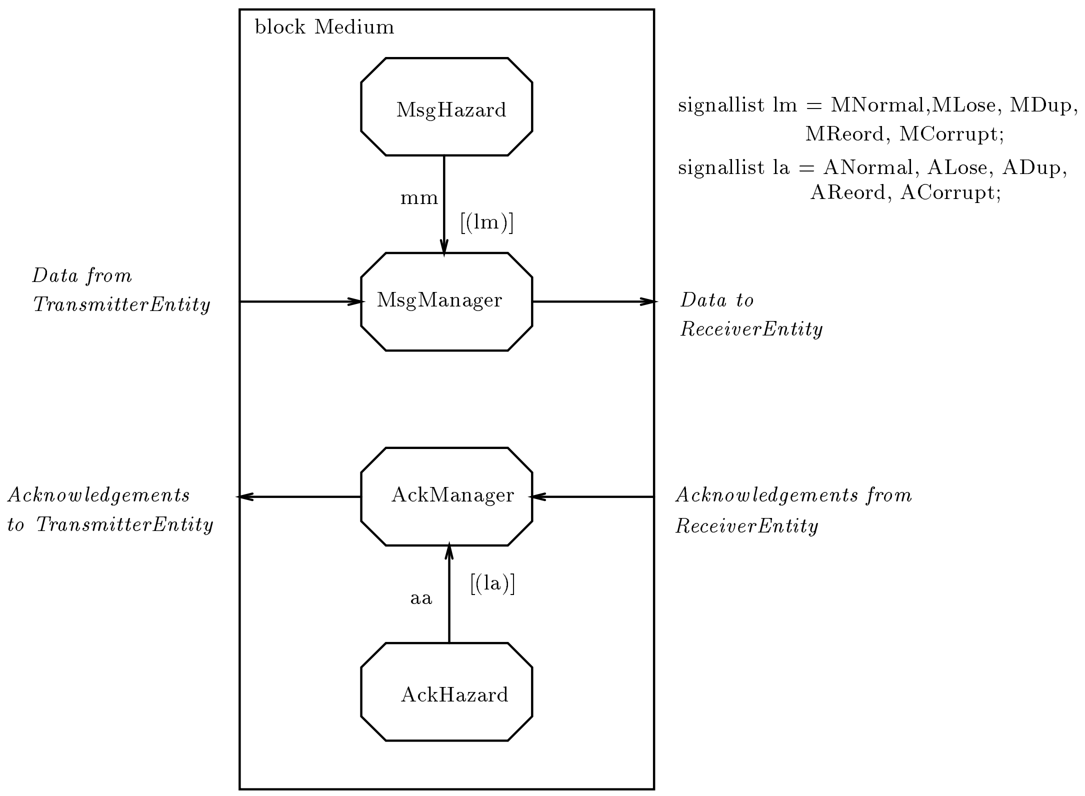
Transmitter and receiver exchange their data and acknowledgements over a medium. This medium models an unreliable channel, which can nondeterministically lose, corrupt, duplicate or re-order messages. However, in SDL 88 there exists no means for expressing nondeterminism. Therefore, in [ISO91, Tur93] hazards are introduced, as shown in Figure 8. The process MsgManager is responsible for the treatment of the data within the medium. Its nondeterministic behaviour is modelled by introducing the guard process MsgHazard. This process sends hazard signals to the MsgManager suggesting which operations are to be carried out by the MsgManager on the data which are stored in a queue: normal delivery (MNormal), loss (MLose), duplication (MDup), corruption (MCorrupt) or reordering (MReord) of messages.
The treatment of acknowledgements within the medium is handled by the process AckManager. For modelling its nondeterministic behaviour the process AckHazard is introduced and specified similar to MsgHazard.

4.4.1 Description of the Error

A hazard may send signals to its manager, although its manager's queue is empty. Some operations performed by the manager on the queue after having received a signal produce an error if the queue is empty.

4.4.2 Erroneous Scenario

Suppose message 3 waits in the queue mq to be transmitted:
$MediumMessageQueue : mq =< 3 >$
Suppose that the hazard signal $MNormal$ appears:



block Medium
MsgHazard
mm
[(lm)]
signallist lm = MNormal,MLose, MDup, MReord, MCorrupt;
signallist la = ANormal, ALose, ADup, AReord, ACorrupt;
Data from TransmitterEntity
MsgManager
Data to ReceiverEntity
Acknowledgements to TransmitterEntity
AckManager
Acknowledgements from ReceiverEntity
aa
[(la)]
AckHazard

Figure 8: Structure of the block Medium

$qitem := qfirst(mq) = 3$
Now the queue is empty:
$mq := qrest(mq) = qnew$
Message 3 is sent to the receiver.
Suppose the hazard signal $MNormal$ appears again:
Then $qitem$ is set to $qfirst(mq) = qfirst(qnew)$
According to the axiom $qfirst(qnew) == error!$ the execution of the SDL system will stop and an error message will be displayed.

4.4.3 Correction of the Specification

To prevent these errors the manager always checks if its message queue is empty when it gets a signal from its hazard. Only if the message queue is not empty the hazard signal will be processed, otherwise the manager will not do anything.

5 Conclusion

During the early stages of system development system designers interact with users to capture the problem domain and to analyse the system's requirements. This results in an informal description of the behavior of the system usually based on natural language. If formal methods are used this requirement specification is achieved by formal notations like SDL, LOTOS, Estelle or MSC.

The transition from an informal to a formal specification is a crucial point in the development of systems. The formal requirement specification has to be validated to ensure that the formal description corresponds to the specifier's intuition. This process is called *validation*. The validation of a specification is very important, because later formal (or maybe informal) development steps are based on this requirement specification. If in later development steps an inadequacy of the requirement specification is found, an expensive redefinition and reimplementation of the existing specifications and implementations has to be done. Furthermore, based on a formal requirement specification the following steps can be carried out in a purely formal way ([BDD$^+$93]).

The use of formal methods forces a system developer to write precise and unambiguous specifications. That is the basis of the application of tools that offer a syntactic check, validation and simulation of the specification. Note that a formal requirement specification does not guarantee a correct specification. It only describes the system's requirements in an unambiguous way. By using validation techniques like e.g. simulation or proving some properties (see for instance [Jon90]), errors of the specification can be detected in early development steps. We first read the SDL specifications of the sliding window protocol without noticing the errors mentioned in Section 4. However, the simulation in [Hau95] which is indeed a testing of some specification's aspects showed these inconsistencies. Note that these specifications were published first as a technical report of ISO [ISO91] and then in [Tur93] without noticing any errors.

Our analysis of the sliding window protocol has resulted in a significant improvement of the corresponding SDL specification. However, we cannot guarantee the absence of errors in the corrected specification, because we only tested the specification by simulation and so the number of errors decreased. In [Stø95] it is outlined how Focus can be used for top-down development of SDL specifications which results in a correct SDL specification if the correctness of each refinement step has been properly verified. Especially due to a higher abstractness of a specification the validation process can be simplified.

Although a formal specification may contain errors (which of course should be avoided) it helps the designer to achieve a better understanding of the system to be built. Design inconsistencies, ambiguities and incompleteness are detected in an early stage of software development.

Acknowledgements

We thank Manfred Broy, Stephan Merz, Franz Regensburger and Ekkart Rudolph who have read earlier drafts of this paper and provided valuable feedback. We also thank Verilog and Siemens AG for their tool support. Especially the simulators of Geode from Verilog and SICAT from Siemens AG helped us to find the errors in the SDL specification.

References

[BDD⁺93] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems — An Introduction to FOCUS. SFB-Bericht 342/2/92 A, Technische Universität München, January 1993.

[BH93] R. Bræk and Ø. Haugen. *Engineering Real Time Systems.* Prentice Hall, 1993.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, 1990.

[CCI88] CCITT. X.208, Specification of Abstract Syntax Notation One (ASN.1). Blue Book, FASCICLE VIII.4, Recommendations X.200-X.219. CCITT, 1988.

[CCI93] CCITT. *Recommendation Z.100, Specification and Description Language (SDL).* ITU, 1993.

[Hau95] M. Haubner. Vergleich zweier SDL-Werkzeuge anhand des Sliding Window Protokolls, 1995. Fortgeschrittenenpraktikum, in German.

[ISO84] ISO. ISO 7498: Information Processing Systems - Open Systems Interconnection - Basic Reference Model, 1984.

[ISO91] ISO/IEC. Information technology - Open System Interconection - Guidelines for the Application of Estelle, LOTOS and SDL. Technical Report ISO/IEC/TR 10167, International Organization for Standardization Geneva, 1991.

[IT95] ITU-T. *Recommendation Z.105, SDL92 Combined with ASN.1 (SDL/ASN.1).* ITU, 1995.

[MV93] S. Mauw and G.J. Veltink. *Algebraic Specification of Communication Protocols*, volume Cambridge Tracts in Theoretical Computer Science 36. Cambridge University Press, 1993.

[OFMP⁺94] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed and J. R. W. Smith. *Systems Engineering Using SDL-92.* Elsevier Science, 1994.

[Stø95] Ketil Stølen. Development of SDL Specifications in FOCUS. In Rolv Bræk and Amardeo Sarma, editors, *SDL '95: with MSC in CASE*, pages 269–278. North-Holland, 1995.

[Tan88] Andrew S. Tanenbaum. *Computer Networks (second Edition).* Prentice Hall, 1988.

[Tur93] Kenneth J. Turner, editor. *Using Formal Description Techniques.* John Wiley & Sons, 1993.

[vdS95] Jan L.A. van de Snepscheut. The Sliding-Window Protocol Revisited. *Formal Aspects of Computing*, 7:3–17, 1995.