# The SDL Specification of the Sliding Window Protocol Revisited

Christian Facchi[a], Markus Haubner, Ursula Hinkel[b]

[a]Siemens AG, PN KE CPT 51, D-81359 München, Christian.Facchi@pn.siemens.de

[b]Institut für Informatik, Technische Universität München, D-80290 München,
{haubnerm,hinkel}@informatik.tu-muenchen.de

This paper is a corrigendum to the SDL specification of the sliding window protocol which was first published by the ISO/IEC as a technical report. We present some results of a tool supported simulation of the SDL specification of the sliding window protocol. We found out that the specification contains significant errors and does not meet the informal description of the protocol. In this paper 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. 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 a case study [3], in which we examined the tool supported development of protocols. Because of its practical relevance and simplicity we chose the SDL specification of the sliding window protocol as an example for this examination. An SDL description of the sliding window protocol is given in [7,11]. Working with the tools we did not follow any method for the testing of SDL systems but did just use the various facilities of the tools like the graphical editor and the simulator. When editing and simulating the SDL specification of the sliding window protocol, we found some incorrect parts within this specification. The specification does not meet the informal description of the protocol which is also given in [7,11]. 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. Therefore, this paper can be regarded as a corrigendum to some parts of [7,11].

Based on our experience we propose the use of formal methods with tool assistance. Although a formal specification may contain errors (which of course should be avoided), it

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

helps the designer to achieve a better understanding of the system to be built. Tools are extremely useful in achieving a correct specification. Design inconsistencies, ambiguities and incompleteness are detected in an early stage of software development.

This paper is organized as follows. In Section 2 we will give an informal introduction to the sliding window protocol. The main part of this paper, Section 3, describes the errors that we found and their correction. Moreover, we will explain how we discovered the errors using SDL tools. Section 4 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 [6]. Due to its purpose it describes a point to point connection of two communication partners (a transmitter and a receiver) 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 the sliding window protocol is the usage of a sending and receiving buffer. For the transmitter 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*.

According to [9], the protocol can be described as follows: The transmitter 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 transmitter is permitted to dispatch a bounded number of messages with consecutive tags while awaiting their acknowledgements. The messages are said to fall within the transmitter'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 transmitter.

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.

The SDL specification of the sliding window protocol [7,11] is based on a *sliding window protocol using "go back n"* according to [10]. For simplicity only a unidirectional flow of data is described. 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 [7,11]. 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.

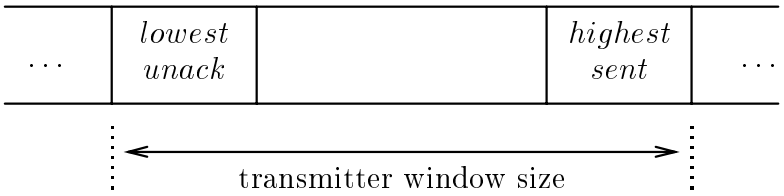



Figure 1. Transmitter window

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, 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.

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.

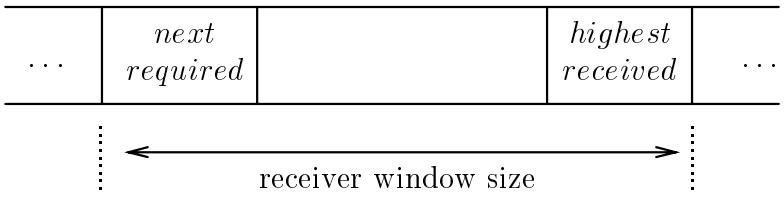



Figure 2. Receiver window

## 3. 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 ([7,11]). We will first describe each error in an abstract way and then show a scenario in which it occurs followed by a corrected specification.

### 3.1. Tracing the errors

For our case study, in which we evaluated the facilities of SDL tools, we chose the sliding window protocol as an example, because it is a well known, simple protocol. We did not follow any systematic testing method (like e.g. using TTCN or a test case generation) but concentrated on the evaluation of the various facilities of SDL tools.

One way to check the behaviour of the protocol is to use the simulation as it is offered by some SDL tools. By simulating the SDL specification the behaviour of the specified system can be debugged. We started with executing a single step simulation. We sent signals from the environment to the system and observed the reaction of the SDL processes. The exchange of signals as well as the internal status of the system like the values of variables and the input ports are displayed and can be observed during the simulation. We immediately recognized that there was something wrong with the behaviour of the protocol.

Thinking that the problem might have its cause in our specification of the protocol which we used as input to the tools, or that we might have made some mistakes during the simulation, we generated Message Sequence Charts of the simulation. We analysed the MSCs and checked the corresponding parts of the SDL specification with paper and pencil. Thus, we found the errors described further below.

The advantage of using tools is the visualization of the dynamic behaviour of the system specified by SDL. The interaction of the processes and the exchange of signals as well as the changing of the data values within the processes are difficult to imagine without tool support. Especially the display of the values of the variables of the processes and the message flow with the values of parameters transmitted by the signals were very helpful for the detection of the errors.

### 3.2. A short overview of the SDL specification

We give only a short description of the structure of the SDL specification which is presented in full details in [11]. 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.

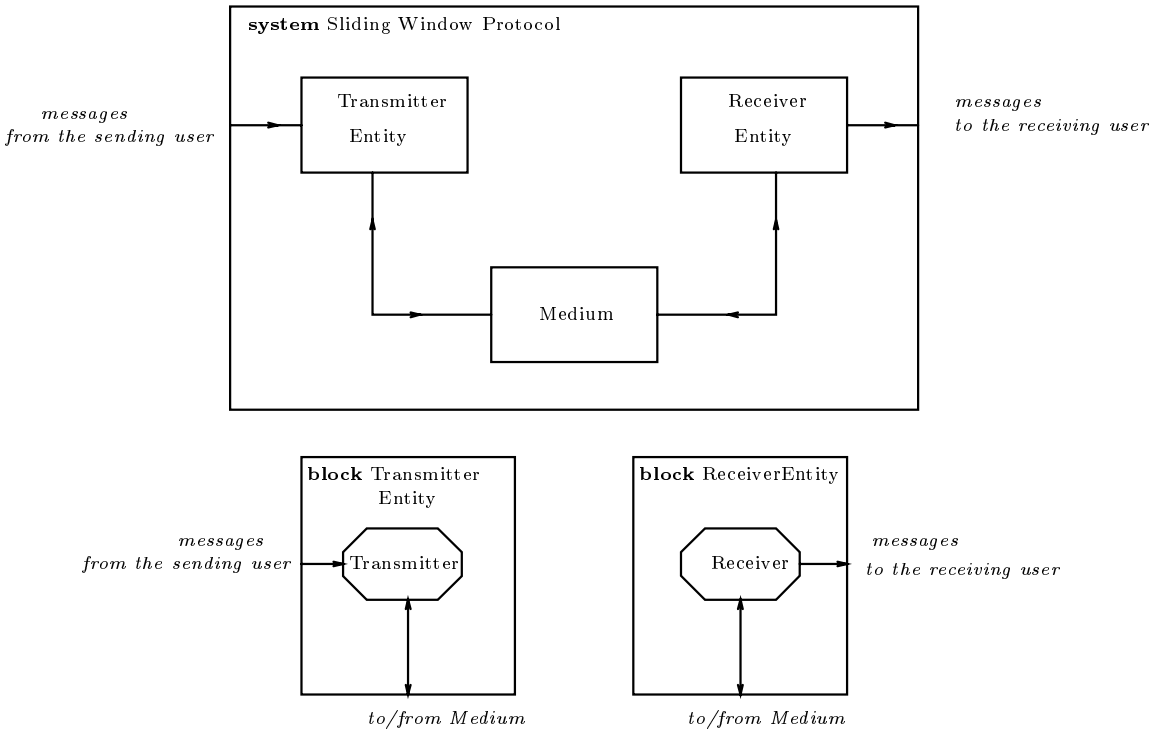


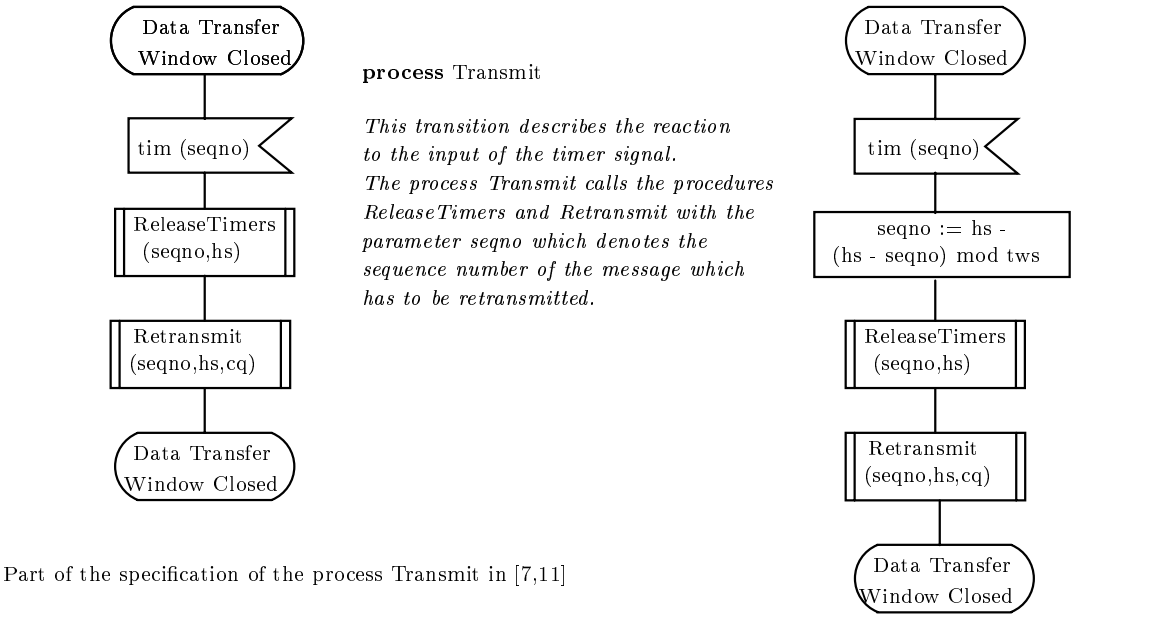

Figure 3. The structure of the SDL specification

The SDL specification of the protocol is composed of three blocks: TransmitterEntity, ReceiverEntity and Medium. The sending and the receiving users are part of the environment and interact with the corresponding protocol entities by signals. The two blocks TransmitterEntity and ReceiverEntity communicate via channels with the block Medium. The block Medium models an unreliable medium, which can nondeterministically lose, corrupt, duplicate or re-order messages. Its behaviour is described in Section 3.6. The behaviour of the block Medium is not part of the SDL specification of the sliding window protocol itself. However, there is an SDL specification of an unreliable medium in another chapter of [7,11]. Thus, we took this specification for the medium of the Sliding Window Protocol.
The TransmitterEntity sends data, which it gets as input from the sending user, via the medium. The ReceiverEntity gets data from the medium and sends acknowledgements over the medium to the TransmitterEntity. Data, which have been correctly transmitted, are given to the receiving user.
The block TransmitterEntity consists of the process Transmitter which includes two procedures: ReleaseTimers and Retransmit. The block ReceiverEntity consists of the process

Receiver which includes the procedure DeliverMessages.

### 3.3. Errors Concerning the Sequence Number

In the formal description of the sliding window protocol ([7,11]) unbounded sequence numbers are attached to the messages. When the transmitter sends a message, it has to start a timer for that message. Each message is related to an individual timer. However, due to a constraint of the informal description of the protocol, the number of timers existing at the same time is bounded. In the following we describe an error which is based on dealing with this discrepancy in the SDL specification.




Part of the specification of the process Transmit in [7,11]

Corrected version of the specification

Figure 4. The use of sequence numbers in Process Transmitter

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

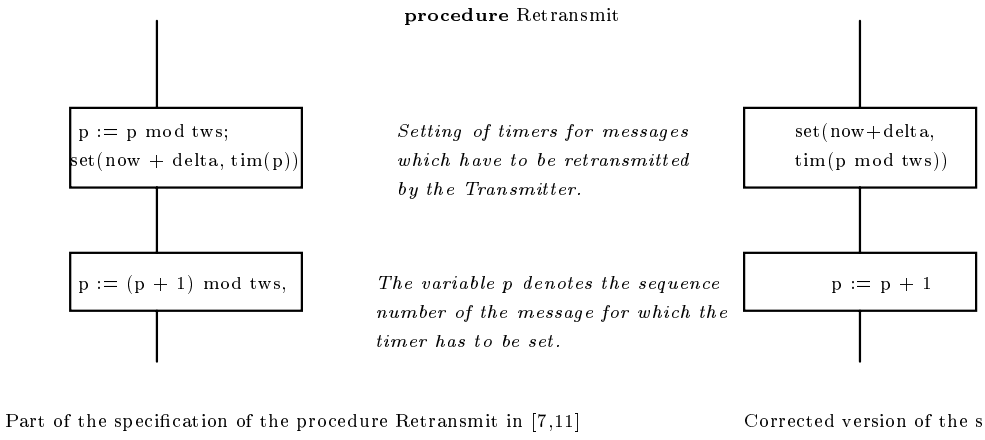



Figure 5. The setting of timers in the procedure Retransmit

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.

### 3.3.2. 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 $11 \bmod tws = 1$ and $12 \bmod tws = 2$. 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*.

### 3.3.3. 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).

### 3.4. Errors Concerning the Closing of the Transmitter Window

The transmitter has 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.

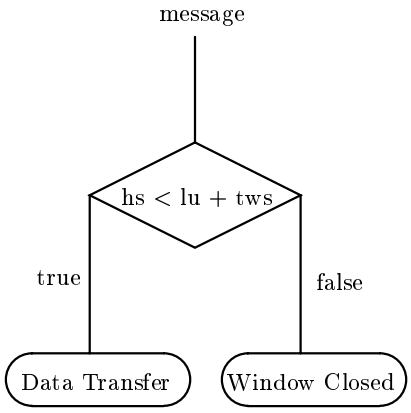



Figure 6. Closing the transmitter window

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

#### 3.4.2. 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

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

$set\,(now + delta,\, tim\,(hs \bmod tws))$ 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.

### 3.4.3. 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.

## 3.5. Errors Concerning the Spooling of the Transmitter Queue in the Retransmit Process

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

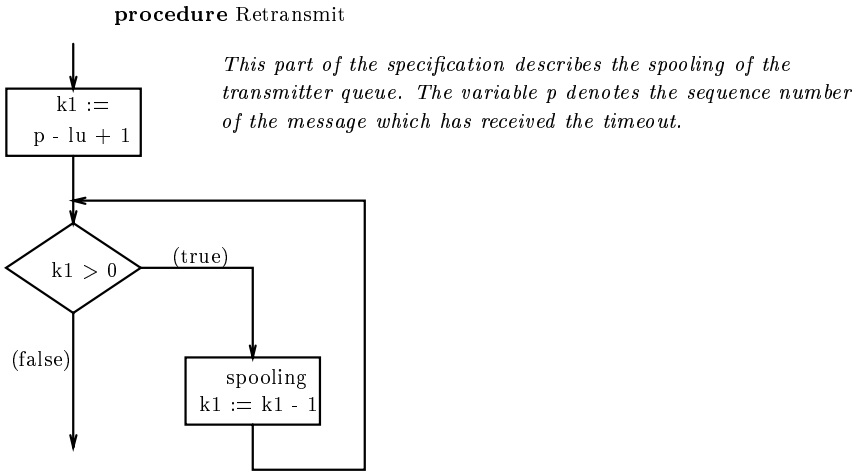



Figure 7. The spooling of the transmitter queue

### 3.5.1. Description of the Error

In the procedure Retransmit the message queue should be rotated until the first message to be retransmitted is at the beginning of the queue. However, the calculation of the messages that have to be rotated is incorrect, because the queue is always rotated 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.

### 3.5.2. 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 rotation 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 rotation 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 rotation is finished and the retransmission starts. Message 2 is retransmitted with the first element in the queue as content. Thus, 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.

### 3.5.3. Correction of the Specification

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

## 3.6. 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 [7,11] 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: normal delivery (MNormal), loss (MLose), duplication (MDup), corruption (MCorrupt) or reordering (MReord) of messages. The data within the medium are stored in a queue called Medium Message Queue $mq$, which is a local variable of the SDL process MsgManager.
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.

### 3.6.1. Description of the Error

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

### 3.6.2. Erroneous Scenario

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

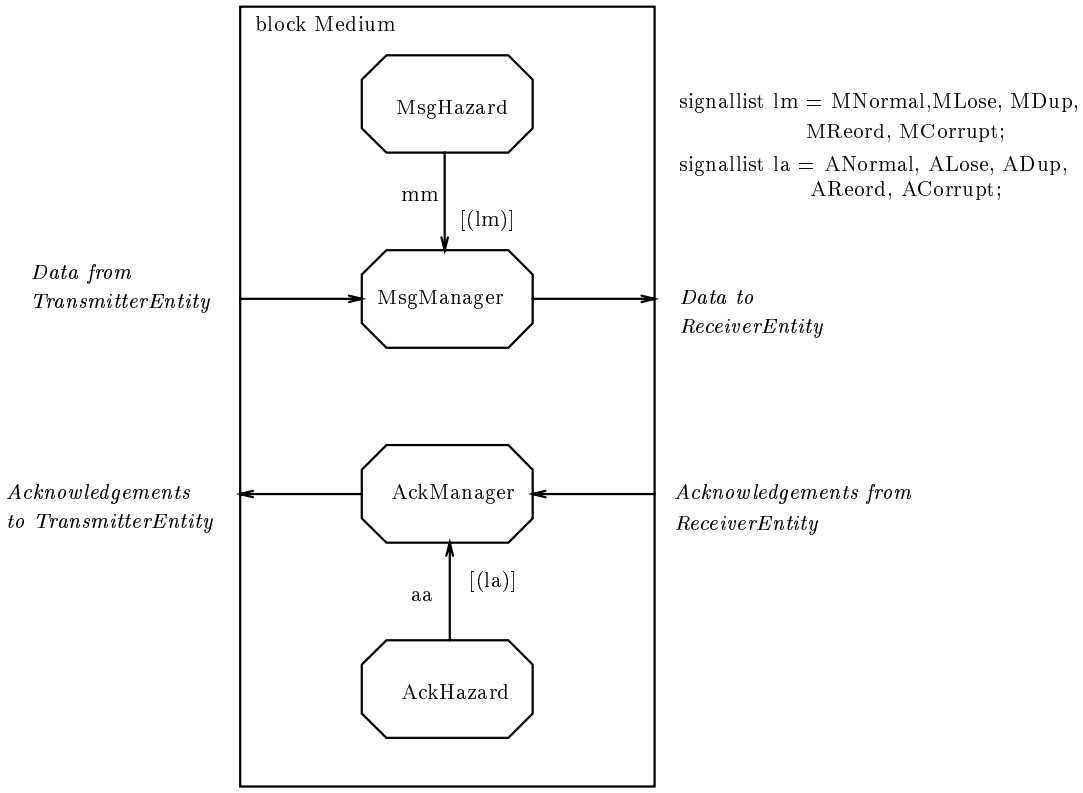



Figure 8. Structure of the block Medium

$qitem := qfirst(mq) = 3$
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.

### 3.6.3. Correction of the Specification

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

Note that this error would not have occurred if SDL 92 was used which includes explicit language constructs for nondeterminism, because the usage of the SDL processes *MsgManager* and *AckHazard* is not necessary.

## 4. CONCLUSION

Our analysis of the sliding window protocol with tool assistance resulted in a significant improvement of the corresponding SDL specification. First, we were suprised that errors, which are typical for programming, are found in an SDL specification. But taking a closer look at the specification, we recognized that some parts of the specification concerning data types and operations on data are very complex and have been specified by programming concepts like procedures. It is not surprising that using concepts for programming results in simple programming errors. Based on our experience we propose the use of formal methods with tool assistance for the development of SDL systems. Thus, typical programming errors are detected in the early stages of system development.

The use of formal methods forces a system developer to write precise and unambiguous specifications. Note that a formal requirement specification does not guarantee a correct specification. It only describes the system's requirements in an unambiguous way. The formal requirement specification has to be checked to ensure that it corresponds to the specifier's intuition. This process is called *validation*. By using validation techniques like e.g. simulation or proving some properties errors of the specification can be detected in early development steps. We first read the SDL specification of the sliding window protocol without noticing the errors presented in Section 3. The SDL specification describes a complex behaviour and is hard to overlook. However, the simulation in [3], which is a testing of some specification's aspects, showed these inconsistencies immediately. Indeed we found these errors during the simulation without using any systematic methodology. We think that a systematic approach for the simulation might yield even more errors. Moreover, writing a specification without a later simulation is quite similar to programming without testing the program. A programmer would not rely on an untested program, except the program had been formally verified. However, even a formally verified program should be tested in order to check whether it meets the requirements. The validation of a specification should be tool supported, because in most cases a manual approach is very time consuming so that the validation will be omitted or be done only for some parts of the specifications. We think that in practice a simulation should be chosen rather than a formal verification. A simulation can be carried out without having mathematical knowledge which is essential for formal proofs. There exists a variety of tools for SDL which are of great assistance in editing and checking syntactically and semantically SDL specifications. During the tutorials of the last SDL Forum ([1]) methods for the testing and the validation of SDL systems were presented (see [5,2]).

The importance of simulating SDL specifications is demonstrated by the fact that, although the authors of the specification of the sliding window protocol are SDL experts and did a detailed analysis of the informal description of the protocol, they did not succeed in giving a correct SDL specification.

Although large programs and specifications or text are likely to contain some errors, we have been surprised that these fundamental errors of the specification of the sliding window protocol have not been noticed before. If an implementation of the protocol had been based on the SDL specification, the errors should have been discovered immediately.

In another case study we found some errors in the specification of the Abracadabra protocol, too [8,4]. Therefore, we suggest that all SDL specifications which are part of

ITU-T standards or ISO technical reports, like [7,11], should be checked by tools and if necessary be corrected. This could result in a corrigendum to [7,11]. It might also be interesting to have a closer look at the Estelle and LOTOS specifications included in [7,11] and to analyse whether these specifications are correct or include some errors not yet known.

## ACKNOWLEDGEMENTS

We thank Manfred Broy, Øystein Haugen, Stephan Merz, Franz Regensburger and Ekkart Rudolph who read earlier drafts of this paper and provided valuable feedback. For analysing and simulating the SDL specification of the sliding window protocol we used the SDL tools ObjectGeode by Verilog and SICAT by Siemens AG.

## REFERENCES

1. R. Bræk and A. Sarma. *SDL '95: with MSC in CASE*. North-Holland, 1995.
2. A.R. Cavalli, B.-M. Chin, and K. Chon. Testing Methods for SDL Systems. *Computer Networks and ISDN Systems*, 28(12):1669 – 1683, 1996.
3. M. Haubner. Vergleich zweier SDL-Werkzeuge anhand des Sliding Window Protokolls. Fortgeschrittenenpraktikum, Technische Universität München, 1995. in German.
4. U. Hinkel. An Analysis of the Abracadabra-Protocol, 1996. Internal report, in German.
5. D. Hogrefe. Validation of SDL Systems. *Computer Networks and ISDN Systems*, 28(12):1659 – 1667, 1996.
6. ISO. ISO 7498: Information Processing Systems - Open Systems Interconnection - Basic Reference Model, 1984.
7. ISO/IEC. Information Technology - Open System Interconection - guidelines for the application of Estelle, LOTOS and SDL. Technical Report ISO/IEC/TR 10167, 1991.
8. C. Klein. Spezifikation eines Dienstes und Protokolls in FOCUS – Die Abracadabra Fallstudie, 1995. in German.
9. K. Stølen. Development of SDL Specifications in FOCUS. In R. Bræk and A. Sarma, editors, *SDL '95: with MSC in CASE*, pages 269–278. North-Holland, 1995.
10. A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 1988.
11. K. J. Turner. *Using Formal Description Techniques - An Introduction to Estelle, Lotos and SDL*. John Wiley & Sons, 1993.