

# Debugging a Real-life Protocol with CFFD-Based Verification Tools

Antti Kervinen<sup>1</sup>, Antti Valmari<sup>1</sup>, and Risto Järnström<sup>2</sup>

<sup>1</sup> Tampere University of Technology, Software Systems Laboratory  
PO Box 553, FIN-33101 Tampere, FINLAND

{ask, ava}@cs.tut.fi

<sup>2</sup> Instrumentointi Oy

Sarankulmankatu 20, FIN-33900 Tampere, FINLAND

risto.jarnstrom@insta.fi

**Abstract.** In this paper we describe how verification tools, which are based on model checking, were used in a real-life communication protocol design project. Parallel composition, abstraction, reduction and visualisation tools were used to examine the behaviour of the protocol. We performed all verification and debugging visually with the figures that the tools produced. A figure represents the behaviour of the system in a certain point of view, which is selected by choosing a set of system's actions to be externally observable. Visualisation is a user-friendly approach to verifying and validating systems, which does not compromise the completeness of verification. We present how the protocol was modelled and how both safety and liveness failures in the model were found.

## 1 Introduction

This case study describes the use of automatic verification during the development of a real-life communication protocol. The case study was performed jointly by the Software Systems Laboratory of Tampere University of Technology and Instrumentointi Oy. The latter is a privately owned Finnish company specialising in industrial automation, defence and information technology. It has about 500 employees.

In 1999 Instrumentointi Oy decided to develop a small communication protocol to be used within its products. The purpose of the protocol is to transmit data in both directions between a client and a server in two different modes: *terminal mode* and *PC mode*. Both modes must work efficiently, and the PC mode must provide reliable service even if the channels are unreliable.

In autumn 1999 Instrumentointi Oy asked if the correctness of the protocol could be checked with the verification methods and tools that are being developed at the Software Systems Laboratory. The Laboratory took the challenge.

To perform its tasks in the verification project, the Software Systems Laboratory employed in the beginning of the year 2000 an undergraduate student, A. Kervinen. He worked part-time on the project, while simultaneously continuing his studies. His work was supervised by A. Valmari.

From January to March 2000 Kervinen spent most of his project time learning verification theory and the available tools, and familiarising himself with the

protocol on the basis of the design documentation provided by Instrumentointi Oy. Also his supervisor studied the protocol documentation. As seems to be the norm in formal methods case studies, some unclear points and potential problems were found during this stage. These and their potential solutions were discussed in a meeting that was held between the Laboratory and Instrumentointi Oy in early April. Instrumentointi Oy then fixed the design and delivered a new version of the documentation. It is this version of the protocol that was verified.

The verification runs were performed mainly during April, May and June. In June Kervinen worked full-time on the project. First, the high-level operation of the protocol was investigated with the aid of the “**H** model” that Kervinen wrote for the purpose. In the beginning, all problems that were found were only problems of the **H** model, not of the real protocol. However, in the middle of May an important error was found in the protocol. This error will be described in Section 4.4. The protocol and **H** model were corrected and verification continued.

In early June the coverage of the verification of the latest version of the **H** model was deemed sufficient and no errors had been found in it. Attention was turned to the byte-level operation of the protocol for a couple of weeks. By the end of June the “**L** model” was written, verified and debugged until it seemed reliable.

Verification methods and tools used in this study are described in Section 2. In Section 3 we briefly present the structure of the protocol. The verification of **H** model and **L** model is described in Sections 4 and 5, respectively. The conclusions are presented in Section 6.

## 2 Methods and Tools

The state space of a system consists of all the global states that a system can reach during any of its executions, and of all transitions that the system can ever make between those states.

State spaces of systems, their component processes and environments can be represented by *LTSs*. An *LTS* is a directed graph, one of whose vertices is distinguished as the *initial state*, and whose edges are labelled with names of actions that the object modelled by the *LTS* can do, or with a special symbol “ $\tau$ ”.  $\tau$ -edges represent system’s state transitions that are *invisible* for all objects outside the system. All other edges are *visible* and by them the system interacts with its environment.

### 2.1 Compositional *LTS* construction

*LTSs* can be connected in parallel and the result is an *LTS*. Parallel *LTSs* communicate *synchronously*—that is, by executing the same action simultaneously. This does not mean loss of generality, however, because asynchronous communication can be constructed by representing the communication medium as an

LTS that first communicates synchronously with the sending agent and then with the receiving agent.

It is often the case that some actions that are used to represent the communication between some system components are not directly important for the verification questions at hand. Therefore, after connecting the corresponding LTSs in parallel, those actions can be *hidden*, that is, converted to  $\tau$ . The result is an LTS that contains many  $\tau$ -edges. Such an LTS can usually be *reduced* a lot. LTS reduction means construction of an LTS that is equivalent to the input LTS in some well-defined sense, but smaller.

In this study we use a reduction tool that preserves *CFFD-equivalence* [19, 20]. “CFFD” is an abbreviation of “chaos-free failures divergences”. It is the weakest equivalence of non-deterministic LTSs that preserves all execution traces, deadlocks and livelocks and is a congruence. The preserved information allows analysing whatever can be specified with the well-known “next-state”-free *linear temporal logic* ( $LTL_{\neg X}$ ) of Manna and Pnueli [10], a logic that is used very widely in verification. Branching time properties, that can be expressed with forms of *computation tree logic*, are not preserved. It means that we cannot always see, for example, from which point on a system cannot execute certain actions anymore. However, choosing an equivalence involves a compromise: the more information is preserved, the less the state space can be reduced. In this case we put emphasis on powerful reduction and realisable clear visual representation.

An LTS for the system as a whole can be constructed by dividing the system hierarchically into subsystems, and by computing an LTS for each of the subsystems one at a time starting from the bottom by composing the LTSs in the subsystem in parallel, hiding the actions that are needed only for communication within the subsystem, and reducing. The resulting LTS for the system as a whole is often dramatically smaller than the directly computed LTS. This method is called *compositional LTS construction*. It has been in use at least since 1989 [9].

## 2.2 Visual verification

One possible way of using CFFD-equivalence is to select a small subset of the actions of a system as visible, construct a reduced LTS that represents the behaviour of the system, and let a computer tool draw the LTS onto the display. Experience has shown that when the subset is small enough, the resulting picture has approximately ten states, and can be investigated manually.

Many properties of systems can be checked from such pictures pretty easily. This makes it possible to check properties without ever encoding them formally for a model checking tool. There is a triple advantage: feedback from the behaviour of the system is obtained rapidly without having to encode requirements in  $LTL_{\neg X}$  or other logics; also users who do not know any temporal logic or similar formalism can use the approach; and the pictures occasionally exhibit undesirable features of behaviour that are so unexpected that they would not

have been taken into account in the formulation of the requirements in the logic, and thus would not have been checked during ordinary verification.

This approach to the validation of systems was systemised and given the name *visual verification* in [18]. Technically, it is not verification because of the absence of the formal requirements document. It deserves the name, however, because it uses similar algorithms and tools as real verification, and provides essentially equally reliable results. The last argument is justified since it is difficult or even impossible to ensure that formal requirements encode precisely and completely the informal and perhaps vague notion of “correct behaviour” that designers of a system have in their minds.

Our verification toolset consists of the *ARA* (Advanced Reachability Analysis) tool [17, 13] developed originally at the Technical Research Centre of Finland and later maintained at Tampere University of Technology, and of new tools developed at Tampere University of Technology [8].

We used four tools of the toolset in the project. The state space generator tool was used to generate LTSs out of process descriptions written in a variant of the Lotos language [5, 1, 15]. The LTS that represents the whole system is composed by using the parallel composition tool, which also hides desired actions, and the reduction tool in the compositional construction manner. The visualisation tool is used to represent the LTS.

### 3 The Protocol

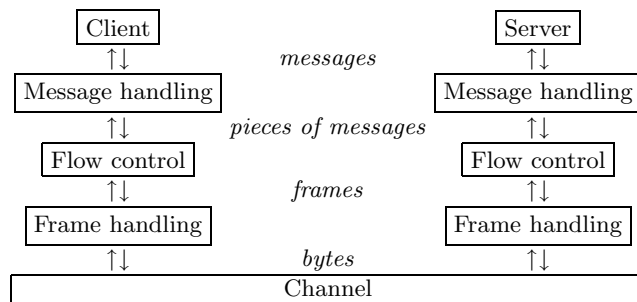
The protocol transmits data in both directions between a client and a server. It can operate in two modes: the *terminal mode* and the *PC mode*. In the terminal mode messages are sent byte by byte to the opposite site. In the PC mode flow control is used to ensure reliable data transmission. Many features of the PC mode have been adapted from the standard protocol called *HDLC* (High Level Data Link Control) protocol [6, 7, 14].

If the protocol is neither in the terminal nor the PC mode, it is in a *synchronisation* mode. When the client station wants to start communicating with the server, it first sends either a  $0D_{\text{hex}}$  byte or a  $7E_{\text{hex}}$  byte to select the terminal or the PC mode. The server never makes an initiative to start communication or to select the mode.

#### 3.1 The terminal mode

The terminal mode basically gives the server and the client direct access to the underlying channel. The channel is an asynchronous bidirectional serial connection through which bytes can be sent. The protocol is byte-oriented in contrast to the bit-oriented HDLC protocol.

In the terminal mode, whenever the server or the client gives the protocol a “legal” byte to be transmitted, the protocol simply sends it (and nothing else)



**Fig. 1.** PC mode layers.

to the opposite site through the channel. If the protocol entity, while in the terminal mode, receives a legal byte from the channel, it simply delivers it to the station. Certain 8-bit combinations are considered “illegal” and not transmitted. The protocol uses some of them for control purposes, such as  $7E_{\text{hex}}$  for signalling change to the PC mode. Both directions (from client to server and vice versa) operate independently of each other.

The terminal mode is no more reliable than the underlying channel. If the channel loses or corrupts a byte, then also the terminal mode loses or corrupts the byte, or even worse: certain channel errors can at least in principle confuse the mode selection mechanism of the protocol with unexpected consequences.

From the verification point of view, the terminal mode is not particularly interesting or challenging in itself. However, because of the possibility of confusion in the mode control mechanism, the interaction of the terminal mode with the PC mode is interesting. The **L** model is designed to investigate this interaction.

### 3.2 The PC mode

The PC mode of the protocol is based on the HDLC protocol. For simplicity and efficiency, the mode does not implement all of the functionality of the HDLC. The PC mode can be thought of to consist of four layers, as is shown in Figure 1.

The *message handling* layer divides messages to pieces short enough to be sealed into frames and again constructs whole messages out of received pieces. The first byte of a piece of a message is a length/more byte, which is added by the layer. Byte values 1–255 express the length of the piece of the message and that it is the last piece of the message being sent. (The next piece will be the first piece of the next message.) The value zero means that the piece is 255 bytes long and there are more pieces to come in the same message.

The *flow control* layer is the most complicated layer. It establishes, maintains and breaks off connection between protocol peers and recovers from channel errors with a mechanism that is based on frames, acknowledgements, timeouts and sequence numbers. The behaviour of the flow control will be clarified in the description of the first version of the **H** model.

The *frame handling* layer sends frames, byte by byte, to the channel and adds a flag byte and byte-stuffing bytes, if necessary, to distinguish the starting points of frames. The  $7E_{\text{hex}}$  byte is considered to be the flag byte if it is not followed by a byte-stuffing byte  $FF_{\text{hex}}$ . The frame handling also adds two checksum bytes to the end of every frame, which makes the frame handling layer capable to detect corrupted received frames.

The flag byte of a frame is followed by a *control* byte that defines the type of the frame. Depending on the type, it can also carry values of flow control's counters. The protocol has six frame types: *I* (information), *RR* (receive ready), *REJ* (reject), *SABM* (set asynchronous balanced mode), *UA* (unnumbered acknowledgement) and *DISC* (disconnect).

*SABM* and *UA* frames are used to create connections and to resynchronise stations. With a *DISC* frame a station expresses its will to close a connection. *I* frames carry the data of messages. *RR* frames are used to acknowledge received *I* frames and to maintain the connection so that it will not be closed because of silence. *REJ* frames are used to resynchronise flow control's counter values and to inform the opposite station of errors in received frames.

## 4 The H model

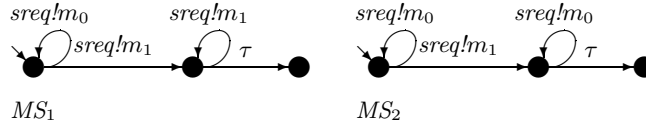
The protocol is modelled in top-down order. The first model concerns thus the highest layers of the protocol. The model is called **H** model.

Our modelling strategy is to build a very simple model first. Then the model is studied and made sure that there are no errors—neither in the model nor in the protocol according to the model. If so, some details from the specification are added to the model and it is investigated again. The goal is to extend the model to resemble the specification as much as possible within the capacity of the verification tools.

### 4.1 Message sources

A message source acts as a customer who wants to send messages to the other station. It communicates with the protocol by “send request” actions. Message sequences that are input to the protocol are composed of two different messages:  $m_0$  and  $m_1$ . The reliability of the protocol is verified visually from the resulting system by examining the output of the protocol.

We use two different message sources to examine the behaviour of the protocol. With message source 1 (see Figure 2,  $MS_1$ ) it is possible to see whether the protocol is able to change the order of messages. If message source 1 in parallel with the **H** model has a possible output where  $m_1$  appeared before  $m_0$ , there is an error in the protocol. Furthermore, if the protocol is able to change the order of messages, then the tools show at least one instance of  $m_1$  appearing before  $m_0$ . This is because the protocol is *data independent*. It means that the



**Fig. 2.** The message sources used in verification with the H model.

contents of messages do not affect the behaviour of the protocol. Therefore, if it has an execution where two messages swap, then with the data source it has an execution where the first  $m_1$  happens to be the latter of the two messages that swap. Application of data independence in verification was investigated in [21].

Similarly, message source 2 (see Figure 2,  $MS_2$ ) reveals the possibility to duplicate messages. Duplication is detected as more than one  $m_1$  messages in the output.

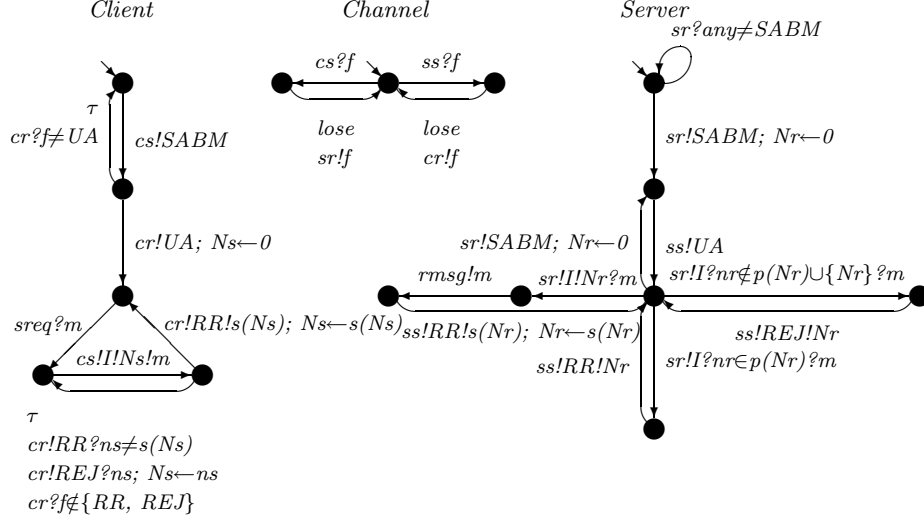
Assuming that the protocol cannot swap or duplicate messages, then also the ability to lose messages or postpone forever their delivery—or even input—is detected. Livelock or termination without outputting an  $m_1$  appears if there is such a problem. For this reason, the presence of the  $\tau$ -transition in message source 2, which gives us finite input sequences, is essential. The reasoning is omitted because of lack of space.

#### 4.2 The first version—unidirectional transmission

The **H** model does not contain the functionality of the message handling layer (see Figure 1). If the flow control is able to ensure that pieces of messages are received in the same order as they are sent without losing or duplicating any of them, then reconstructing messages is a straightforward task. On the other hand, if the flow control fails to ensure even one of these features, the message handling layer can not detect the error. Thus the message handling layer is not essential for the reliability of the protocol and it is omitted in the **H** model.

The first version of the model transfers messages only in one direction. The client, the server and the channel processes of the model are presented in Figure 3. The channel process synchronises at first with either client send ( $cs$ ) or server send ( $ss$ ) action. Then it may either lose the frame or synchronise with the opposite station with server receive ( $sr$ ) or client receive ( $cr$ ) actions (the information of the frame is in the action name). The behaviour of the frame handling layer and byte-oriented channel is abstracted into the channel process—and some of their behaviour is not modelled at all yet.

The client process establishes a connection by sending *SABM* frames until it receives a *UA* frame. Then it starts waiting a send request (*sreq*) from the message source. If it receives one, it sends an *I* frame with current  $Ns$  (next send) counter value and message  $m$ . The client resends the same *I* frame if it receives an *RR* frame that does not include the successor of  $Ns$  counter value, or



**Fig. 3.** Client, server and channel processes in the first version of the model.

it receives other than  $RR$  or  $REJ$  frame, or a timeout occurs, which is modelled by a  $\tau$ -transition in the figure. This means that the number of retransmissions is not limited in the model. If a reject frame is received then the same message is resent with a new sequence number. Finally, the client returns to wait for the next send request with an updated  $Ns$  counter value, if it receives an  $RR$  frame with the sequence number that is the successor of  $Ns$ 's current value.

The server process ignores initially all other received frames except  $SABM$ , to which it responds with a  $UA$  frame and regards the connection established. When the connection is established, the server accepts  $SABM$  and  $I$  frames. If it receives an  $I$  frame with the sequence number that equals the value of the  $Nr$  (next receive) counter, then it announces a message received and sends an  $RR$  frame with the new value of  $Nr$  counter. If the received  $I$  frame has a sequence number that is a predecessor of the  $Nr$  counter value, then the frame is regarded as duplicate, the message is ignored and an  $RR$  frame with the current  $Nr$  counter value is sent. Remaining possible  $I$  frames are incorrectly numbered. In this case a reject frame is sent with the current  $Nr$  counter value to correct the numbering of the client process.

To compose the LTS of the whole system, we compose at first a message source and the client process in parallel, hide all actions which do not synchronise with the channel process and replace the result with a minimal CFFD-equivalent LTS. Then the minimal LTS is composed in parallel with the channel process, all other actions except those which synchronise with the server process are hidden and the result is again minimised. Finally, the resulting LTS is composed in parallel with the server process, all actions except  $rmmsg!M0$  and  $rmmsg!M1$  are hidden and the result is minimised. When message source 2 is used and

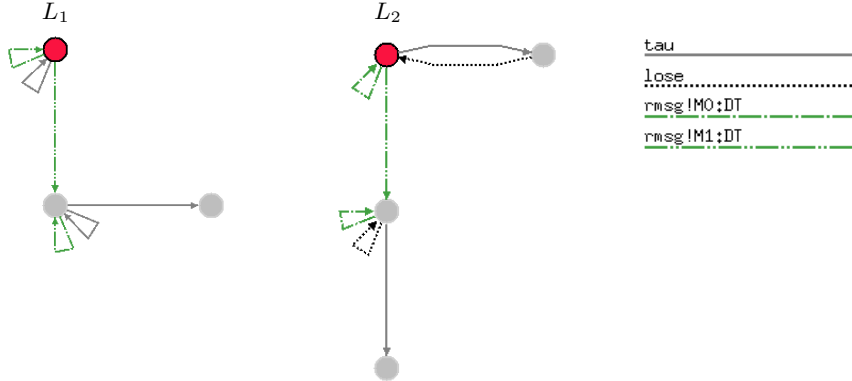


Fig. 4. The model of the first version.

the minimised result is given to the visualisation tool, it produces the LTS  $L_1$  presented in Figure 4 (LTSs in the figure are captured from screen).

There are  $\tau$ -loops in the LTS  $L_1$  in Figure 4. The  $\tau$ -loop in the initial state (the darkest circle) of  $L_1$  means that the protocol is able to execute invisible actions forever without delivering any messages. Infinite invisible activity is called *divergence*. We guess that one cause of divergences is that the channel may always lose frames and the client may always resend frames. Therefore, we re-compose the model but leave this time *lose* action visible. The result is LTS  $L_2$  in Figure 4.

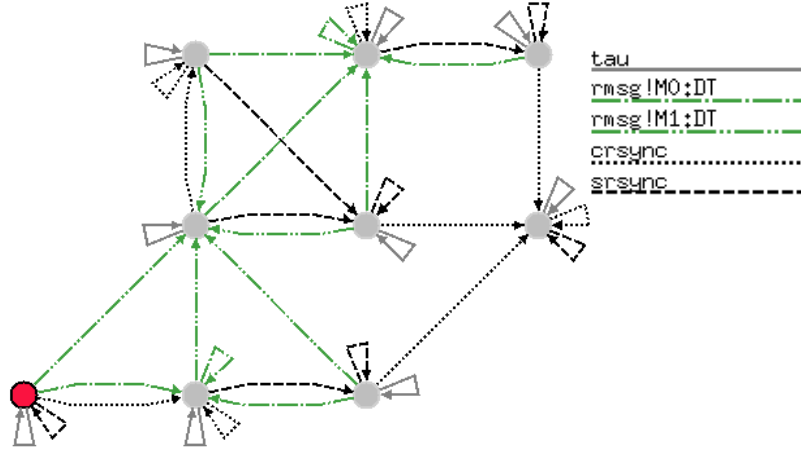
From the result we see that all divergences in  $L_1$  indeed included *lose* action. It means that if the channel cannot lose an infinite number of frames, the protocol cannot execute internal actions forever. We assume that the channel will eventually deliver frames to the receiving station, in other words, it cannot lose an infinite number of frames in a row. With this assumption the first version of the model is able to eventually deliver messages without duplicating or losing any of them or changing their order.

### 4.3 The second version—resynchronisation

The first version of the **H** model is extended with a resynchronisation mechanism. Resynchronisation is initiated after losing a frame cycle, that is after not finding a flag byte where it was expected. When a station loses the frame cycle, it starts transmitting *SABM* frames and waits for a *UA* frame to recover.

Since the frame cycle can be lost only when a frame is sent, the channel process of the model is modified so that in addition to passing and losing frames it is able to synchronise itself with the receiving station by executing a “resynchronise” action. Execution of the action corresponds to appearance of such an error in the channel that the receiver loses the frame cycle and starts resynchronising.

When we compose the model and leave actions  $rmsg!M0$ ,  $rmsg!M1$  and *lose* visible, the result has still  $\tau$ -loops. It seems that a modification of the model has



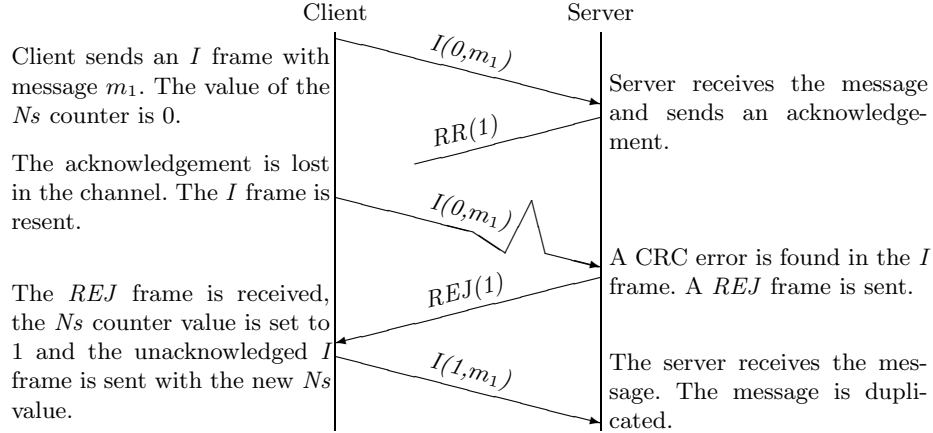
**Fig. 5.** The livelock in the second version of the model.

caused a possibility to livelock. Furthermore, the protocol seems to be able to duplicate messages. We recompose the model and leave also actions *crsync* and *srsync* visible (actions are “resynchronise the client process” and “resynchronise the server process”). The action *lose* is now hidden to keep the visualised state space small. The result of the composition is presented in Figure 5.

In the figure we can see a state (on the right hand side) from which any other states cannot be reached. Execution can end up in this state before any messages are received. In the figure we see that both actions *crsync* and *srsync* must be executed at least once to end up in this state. On the other hand, if the system executes some “receive message” actions between the resynchronisation actions, then the state is avoided. This gives us a hint that only when resynchronisation actions are executed in a short time, the protocol may livelock.

The cause of the livelock can be concluded with knowledge of the behaviour of the processes. If both stations are in the resynchronisation mode, that is sending only *SABM* frames and waiting for a *UA* frame to recover, then the system can never recover. Neither station can receive a *UA* frame since they are not sent. (It is not the purpose that the explanations of bugs have to be figured out in this way. Our intention is to build a tool that, given a state of an abstracted LTS, would produce a non-abstracted description of an execution that leads to it.)

In the implementation of the protocol this problem did not appear because also *UA* and *DISC* frames were accepted when resynchronisation was started. However, the resynchronisation mechanism was later removed from the specification of the protocol, so the second version of the model was not examined in more detail. According to the new specification, if a station did not get a flag byte in the correct place, it simply read and ignored bytes until it received one. If some data was lost because of the loss of frame cycle, the sender eventually resent it because of not getting acknowledgement in time.



**Fig. 6.** An execution which causes duplication.

#### 4.4 The third version—CRC errors

With the third version of the model we want to verify the behaviour of the protocol when the frame handling layer can find CRC errors in frames. The first version of the model is again modified because the second version of the model became obsolete. When a station receives a frame in which the checksum is not correct, it sends immediately a reject frame with  $Nr$  counter value.

The channel process can, in addition to losing and passing the frame, synchronise itself with the receiving station by executing a  $ccrc$  or an  $scrc$  action, depending on whether the receiver is the client or the server. Similarly, when a process waits for a frame from the channel, it can execute also the  $ccrc$  or the  $scrc$  action. If the process executes the action, then it executes next a  $cs!REJ!0$  (in case of the client process) or a  $ss!REJ!Nr$  (if the process is the server) action. The reason why the client always sends the number zero in a reject frame is that in the unidirectional model the value of the client's  $Nr$  counter is always zero.

The result of the first composition with message source 2 is similar to LTS  $L_1$  in Figure 4 except that after executing action  $rmsg!M1$  there is an  $rmsg!M1$ -loop. The protocol is thus able to duplicate messages forever. By leaving some actions visible and by using a smaller message source we are able to track down the executions that cause the duplication. One of such executions is presented in Figure 6.

When Instrumentointi Oy was informed about this error scenario, they tried it with the real implementation. The error emerged. The error was there, although the implementation had already been tested without detecting the error.

To correct the error, the model was changed so that when a station found a CRC error, it immediately sent an  $RR$  frame instead of a reject frame. With the correction the model became reliable. Instrumentointi Oy corrected the implementation accordingly.

#### 4.5 The last two versions—bidirectional information transfer

In the fourth version the previous model is modified so that information can be transferred in both directions: from the client to the server and vice versa. The only difference between the behaviour of client and server processes is the connecting sequence. When a connection is established, both stations check if there are send requests. If there are, they send an *I* frame and wait for the acknowledgement. If there are not, they listen to the channel until a frame is received or it is time to check possible send requests again. This version of the model does not reveal any error scenarios. The data transmission seems to be reliable in both directions.

The protocol is designed to handle a situation when the  $N_s$  and  $N_r$  counters of a station differ from the counters of the other station. *REJ* frames are designed to be used to agree upon the correct counter values. The fifth version is made to observe how the protocol recovers from confused counter numbers.

The fourth version of the model is extended so that the counters of the server can suddenly get any values in their range, which is from zero to three, exactly once during the execution. Because of the symmetry of the protocol after establishing connection, the results are believed to be similar if the counters of the client were disturbed instead.

Destroying the counter contents affects reliability of data transfer in both directions. From server to client messages can be lost and from client to server messages can be both lost and duplicated. The duplication appears, for example, in the following execution: the server receives a correctly numbered *I* frame and sends an acknowledgement which is then broken in the channel. The server's counter contents are destroyed. The client resends the *I* frame because of a timeout. The frame is now incorrectly numbered to the server and it responds with a reject frame to synchronise the counter of the client. The next *I* frame sent by the client has a new frame number and is handled as a new frame although it is a retransmission.

The model is changed so that when a station receives a frame with incorrect numbers it sends a *DISC* frame and closes the connection. This change prevents the execution that was described above but losses and duplication can still take place. For example, the server receives a message and sends an acknowledgement which is broken in the channel. The  $N_r$  counter of the server is changed so that it gets a value that it had before receiving a message. The client resends the *I* frame and the server receives the same message again without noticing that it is a duplicate.

Therefore, if the counters are allowed to get confused, the protocol cannot be reliable. Fortunately, according to the model, the counters can never be confused by any error that occurs in the protocol. *REJ* frames are never sent unless the counters are externally set to random values.

## 5 The **L** model

In the **H** model the server accepts only connections in the PC mode. In reality, even if the client wants to connect in the PC mode, the flag byte of the *SABM* frame can accidentally change to  $0D_{\text{hex}}$  forcing the server to go to the terminal mode. Therefore, the interplay of the different connection modes is analysed with the **L** model.

Two versions of the **L** model are built and analysed. With the first version we want to check if the frame handler is able to build frames from the bytes received from the channel in the case that the bytes are not changed, but an arbitrary number of previous bytes can be changed or lost.

For this purpose, we build a unidirectional model in which a message source gives frames to a frame dispenser that forms corresponding sequences of bytes and sends them to a channel. In the opposite site a frame constructor receives bytes and is notified if bytes are lost or changed in the channel. Of course, information is not available in the real system before a checksum is calculated. Therefore, the information does not affect the behaviour before received bytes form a frame. At that point, if errors have happened in the channel during the construction of the frame, a CRC error is announced. Otherwise, the frame is received. A CRC error is also announced when all bytes of a frame are not received before the first byte of the next frame arrives.

The results of this model show that every frame in which bytes have not been changed or lost in the channel is successfully received. This means that any sequence of errors that has happened in any previous frames does not affect building a new frame, if the bytes of the new frame are received without errors. The frame handling layer is thus fault-tolerant.

Also a phenomenon that has not been paid attention to in the **H** model is found: sending one frame may cause a receiver to find a number of CRC errors instead of only one. Many CRC errors may force the receiver to try to send many *RR* frames although only one erroneous frame is actually being received. However, according to the specification, receiving *RR* frames do not cause any action except a resending of an unacknowledged *I* frame in some cases (see Section 4.4). This gives a good reason to believe that sending multiple *RR* frames while receiving one frame has no effect on the reliability of the protocol.

In the second version of the **L** model, which is also the final step of this formal analysis, we extend the previous model with the three modes of the server: the PC mode, terminal mode and synchronisation mode. We want to examine whether changing between these modes has an effect on creating connections and receiving frames. The previous model is changed so that the receiver may change its mode by the received bytes and timer ticks.

The previous verification results are not fundamentally changed. When the channel is allowed to change bytes, the server is able to work in the terminal mode even if the client sends frames. The consequences of this depend on how

the terminal mode output is handled. When the channel is not able to break bytes, all frames are successfully received in spite of the mode of the server and earlier errors.

## 6 Conclusions

In this paper we have gone through a verification project. The protocol, which was given to formal analysis, had been preliminarily designed before the project started, and it had been partially implemented and tested before the verification started.

Instead of an ambitious goal of proving the correctness of the whole protocol, we used verification tools for searching errors, tracking down their causes and validating solutions that might fix the errors. Visual verification turned out to be an easy-to-use and yet powerful tool for this approach. All results were read from the graphs produced by the visualisation tool.

The main emphasis was on the flow control layer, which is the most complicated layer of the protocol. Also the frame handling layer was analysed down to byte level, mostly to ensure that the low-level mode selection mechanism of the protocol does not interfere with the flow control.

State explosion causes problems also in our approach. This makes abstraction very important. We used, for example, a channel process which contains at most one frame at a time. This abstraction prevents us from detecting errors that appear only if protocol peers send frames precisely at the same time. On the other hand, we could use the computer resources thus saved to extend the model to areas where errors are more likely, such as bidirectional transmission of payload.

In addition to abstraction, a number of other state space reduction methods were used. The parallel composition tool took advantage of *stubborn sets* [16] (see [3, 11] for similar methods). Due to data independence only two messages sufficed to detect errors. LTSs were constructed compositionally. We also tried composing the whole state space at one step without stubborn sets: the state space of the third version of the model had approximately 60 000 states, but the state space of the fourth version was too large to be constructed.

The project had a remarkable effect on the design of the protocol. Firstly, to create a formal model of the protocol, the specification has to be sound and complete. Before and during writing the model some ambiguities and incomplete details were found and corrected. Secondly, an important error, which had not been found by testing, was found in the protocol. We were able to track down the behaviour which caused it. We suggested a fix and validated it with the verification tools. Instrumentointi Oy implemented the fix in the protocol. Finally, Instrumentointi Oy confirms that with the information that was produced by the verification, they have obtained deeper understanding of the behaviour of the protocol. They have used the information to generate test cases for the implementation.

## References

1. Bolognesi, T. & Brinksma, E.: "Introduction to the ISO Specification Language LOTOS". *Computer Networks and ISDN Systems* 14 (1987), pp. 25–59.
2. Brookes, S. D. & Roscoe, A. W.: "An Improved Failures Model for Communicating Sequential Processes". *Proc. NSF-SERC Seminar on Concurrency*, Lecture Notes in Computer Science 197, Springer-Verlag 1985, pp. 281–305.
3. Godefroid, P.: *Partial-Order Methods for the Verification of Concurrent Systems, An Approach to the State-Explosion Problem*. Lecture Notes in Computer Science, Springer-Verlag 1996, 143 p. (Earlier version: Ph.D. Thesis, University of Liège, 1994.)
4. Hoare, C. A. R.: *Communicating Sequential Processes*. Prentice-Hall 1985, 256 p.
5. International Standards Organisation: *Information processing systems — Open Systems Interconnection—LOTOS—A formal description technique based on the temporal ordering of observational behaviour*. Ref. No. ISO 8807, ISO, Geneva, 1989.
6. International Standards Organisation: *Data communication networks — High-level data link control (HDLC) procedures – Frame structure*. Ref. No. ISO 3309, ISO, Geneva, 1993.
7. International Standards Organisation: *Data communication networks — High-level data link control (HDLC) procedures – Elements of procedures*. Ref. No. ISO 4335, ISO, Geneva, 1993.
8. Kokkarinen, I.: *Reduction of Parallel Labelled Transition Systems with Stubborn Sets*. M. Sc. (Eng.) Thesis (in Finnish), Tampere University of Technology, Finland, 1995, 49 p.
9. Madelaine, E. & Vergamini, D.: "AUTO: A Verification Tool for Distributed Systems Using Reduction of Finite Automata Networks". *Proc. Formal Description Techniques II (FORTE '89)*, North-Holland 1990, pp. 61–66.
10. Manna, Z. & Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems, Volume I: Specification*. Springer-Verlag 1992, 427 p.
11. Peled D.: "Partial Order Reduction: Linear and Branching Temporal Logics and Process Algebras". *Proceedings of POMIV'96, Workshop on Partial Order Methods in Verification*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science Vol. 29, American Mathematical Society 1997, pp. 233–257.
12. Roscoe, A. W.: *The Theory and Practice of Concurrency*. Prentice-Hall 1998, 565 p.
13. Savola, R.: *A State Space Generation Tool for LOTOS Specifications*. VTT Publications 241, Technical Research Centre of Finland (VTT), Espoo, Finland 1995, 107 p.
14. Tanenbaum, A. S.: *Computer Networks (3rd ed.)*. Prentice-Hall 1996, 848 p.
15. Turner, K. J. (ed.): *Using Formal Description Techniques, An Introduction to Estelle, Lotos and SDL*. John Wiley & Sons 1993, 431 p.
16. Valmari, A.: "Stubborn Set Methods for Process Algebras". *Proceedings of POMIV'96, Workshop on Partial Order Methods in Verification*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science Vol. 29, American Mathematical Society 1997, pp. 213–231.
17. Valmari, A., Kemppainen, J., Clegg, M. & Levanto, M.: "Putting Advanced Reachability Analysis Techniques Together: the 'ARA' Tool". *Proc. Formal Methods Europe '93: Industrial-Strength Formal Methods*, Lecture Notes in Computer Science 670, Springer-Verlag 1993, pp. 597–616.
18. Valmari, A. & Setälä, M.: "Visual Verification of Safety and Liveness". *Proc. Formal Methods Europe '96: Industrial Benefit and Advances in Formal Methods*, Lecture Notes in Computer Science 1051, Springer-Verlag 1996, pp. 228–247.
19. Valmari, A. & Tienari, M.: "An Improved Failures Equivalence for Finite-State Systems with a Reduction Algorithm". *Proc. Protocol Specification, Testing and Verification XI*, North-Holland 1991, pp. 3–18.
20. Valmari, A. & Tienari, M.: "Compositional Failure-Based Semantic Models for Basic LOTOS". *Formal Aspects of Computing* (1995) 7: 440–468.
21. Wolper, P.: "Expressing Interesting Properties of Programs in Propositional Temporal Logic". *Proc. 13th ACM Symposium on Principles of Programming Languages*, 1986, pp. 184–193.