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

Antti Kervinen[1], Antti Valmari[1], and Risto Järnström[2]

[1] Tampere University of Technology, Software Systems Laboratory
PO Box 553, FIN-33101 Tampere, FINLAND
{ask, ava}@cs.tut.fi
[2] Instrumentointi Oy
Sarankulmankatu 20, FIN-33900 Tampere, FINLAND
risto.jarnstrom@insta.fi

Abstract. In this paper the use of a set of verification tools, developed at the Software Systems Laboratory, in a real-life protocol design project is described. First, the CFFD theory, in which the tool set is based on, is taken a quick look at, and the methods used in the analysis are discussed. Then the behaviour of the protocol is presented to the extent needed to follow the discussion on the modelling phases, and on the errors that were found. Each modelling phase, the analysis results from it, and their effects to the design of the protocol are discussed. Abstracted global behaviours of all the models of the protocol were examined visually with the help of a visualisation tool. Visualisation gave answers to verification questions like "How does the protocol behave from this point of view?" instead of the ordinary "Does the protocol behave like this?". This was a great advantage in finding the causes of erroneous behaviour.

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*. The protocol works on two physical channels, one in each direction. 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 have been and are being developed at the Software Systems Laboratory. The Laboratory took the challenge. This article describes the verification project, the methods used in it, and its effect to the design and reliability of the protocol.

An overview of the verification project is given in Section 2. Verification theory, methods and tools used in this study are described in Section 3. Because the common use of the word "verification" has drifted away from its most formal meaning, the section contains also a subsection on the current philosophy of verification. Section 4 discusses the protocol to the extent necessary for understanding the later discussion on error scenarios that were found during the verification. Two models of the protocol were written: the **H** *model* that concerns with the operation of the protocol at a high level, and the **L** *model* that concentrates on the details at the level of individual bytes. The verification of these models is described in Sections 5 and 6, respectively. Our conclusions are presented in Section 7.

2 The Project

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 in the project, while simultaneously continuing his studies. His work was supervised by Professor A. Valmari, who is the head of the Verification Algorithm Research Group in the Laboratory.

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 already 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 will be described in Section 4.

The verification runs were performed mainly during April, May and June. In June Kervinen worked full-time in 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 in the operation of the protocol design was found. This error will be described in Section 5.4. 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, so attention was turned to the byte-level operation of the protocol for a couple of weeks. The **L** model was written, verified and debugged until it seemed reliable. By the end of June Kervinen had written an end report in Finnish. Sections 4 to 6 of this article are based on that report.

3 Methods and Tools

3.1 What "verification" actually means

The phrase "formal verification" means proving formally that a formally given model of a system satisfies a formally given requirements document in some precise sense. Formal verification cannot be applied to real systems, but only to their formal models. In the case of a protocol this means that formal verification cannot be applied to a *real implementation* that runs on silicon chips and copper wires. It can (at least in principle), however, be applied to a *description of the design* of the protocol in some description language, such as the standardised languages SDL [4, 20] and Lotos [6, 2, 20]. Of course, even if the correctness of an implementation cannot be formally established, it would be valuable to know whether the design is correct.

There is a similar gap also at the other end: successful formal verification establishes that the design satisfies the formally stated requirements, but the requirements may be wrong. It is difficult or even impossible to ensure that they encode precisely and completely the informal and perhaps vague notion of "correct behaviour" that the designers or users of the protocol have in their minds.

A more detailed discussion on the setting of a formal verification problem, and of the meanings of "analysis", "validation", "verification" and "error detection" is presented in Section 3 of [21].

A third, more technical problem with formal verification is that it is computationally very demanding. Proving properties of protocols with potentially unbounded memory (such as unbounded fifo queues) is generally an undecidable problem, and remains **PSPACE**-complete even when restricted to networks of finite state machines with queues of capacity one or with synchronous communication. All known practical algorithms for solving **PSPACE**-complete problems have the property that the consumption of both time and memory tends to grow exponentially in the size of the problem. This is true although the very definition of "**PSPACE**-complete" implies that every **PSPACE**-complete problem can be solved in less than exponential memory. Namely, all the known algorithms that do that are woefully slow compared to the best known practical algorithms. (These issues are discussed in Section 5 of [21].)

Because of these reasons, many verification researchers have long ago given up the far too ambitious goal of formal verification of full-scale formal models of real-life systems. Instead, they use verification techniques as an alternative validation method that complements, not replaces, ordinary testing. With them it is possible to find errors that occur too rarely to be found with simulation or ordinary testing, but that may very well occur with bad consequences when thousands or millions of copies of the system are used thousands of days. The attitude, then, is not to try to fully prove correctness, but to find and fix errors until either full correctness proof has been obtained, or the capacity of the available tools has been exceeded without more errors having been found.

Of course, calling this type of activity "verification" can be considered as misuse of the word, but this is the way many researchers use the word, and the authors of this article do not have the power to change that.

3.2 State spaces

In this case study the *ARA* (Advanced Reachability Analysis) tool [24, 18] developed originally at the Technical Research Centre of Finland and later maintained at Tampere University of Technology was used together with new tools developed at Tampere University of Technology [12]. These tools are based on the construction, analysis, visualisation and comparison of a representative abstraction of the *state space* of the system under analysis.

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. The construction, analysis and comparison of state spaces is probably the most popular approach used in verification tools during the most recent 20 years. Depending on the details of the approach and of the author, it is called exhaustive analysis, state space generation, model checking, etc.

State spaces of all but the smallest or most trivial systems are huge, much bigger than what can be stored in even the largest computer of today or tomorrow. This phenomenon is called *state explosion*. It is how the high computational complexity of verification manifests itself in state space methods. Researchers have developed tens of ingenious methods for alleviating the state explosion problem. The most important of them have been surveyed in [21].[1] To be able to reduce the size of the state space or its representation, these methods throw some information away, or at least make it so difficult to access that it cannot be used in practice.

3.3 Labelled transition systems

An automated verification tool needs formalisms for representing not only the system under verification, but also the properties that are to be checked of the system. The choice of these formalisms involves a trade-off: The more expressive they are, the fewer state space reduction methods apply, and the more often verification tools are exhausted.

ARA and its accompanying tools rely on *LTS*s (labelled transition systems) as an internal representation of system components. An LTS can be used to model the behaviour of a system as a whole, an individual component process, a fifo queue, etc. 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 "τ". It is often the case that the behaviour of the object is not represented in full detail by the

[1] The survey lacks at least one important recent method: finite-automata-based representations of sets of variable and/or queue value combinations [1].

LTS, but actions that are not directly relevant for the verification questions at hand are abstracted away. Such actions are called *invisible*, while actions which are shown explicitly in the LTS are *visible*. Invisible actions may have indirect consequences regarding the executability of visible actions. These consequences can be compactly represented with the aid of τ-edges.

LTSs can be connected together in parallel and in other ways, 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 LTSs can be quite easily used to model also asynchronous communication. This happens by representing the communication medium as an LTS that first communicates synchronously with the sending agent and then with the receiving agent.

Somewhat simplifying, the basic idea of ARA is to represent the system under verification as a composition of one or more LTSs, and to compute a joint LTS that represents the behaviour of the system as a whole. This joint LTS represents the state space of the system. Certain state space reduction methods are used to keep it small, most importantly *compositional LTS construction*, whose principle will be explained soon.

LTSs are an internal representation of the system in ARA. The user represents the system in a variant of the Lotos language [6, 2, 20].

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 τ. The result is an LTS that contains many τ-edges. Such an LTS can usually be *reduced* a lot. LTS reduction means the construction of an LTS that is equivalent to the input LTS in some well-defined sense, but smaller.

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 [14].

3.4 CFFD-equivalence

For the compositional method to provide correct verification results, it is crucial that the sense in which the reduced LTS is equivalent to the original LTS has certain properties. Namely, it must preserve enough relevant information, and it must be what is called *congruence* in mathematics. Furthermore, to obtain good reduction results, it is advantageous if the equivalence does not preserve lots of unnecessary information.

The equivalence that is used in ARA is called *CFFD-equivalence* [27, 28]. “CFFD” abbreviates “chaos-free failures divergences”. It resembles a lot the main equivalence notion in the famous CSP theory [3, 5, 17], but some of its properties have been tuned to better support automatic verification. The main difference is that livelocks (that is infinite sequences of τ-transitions) do not necessarily lead to so-called *chaos*, as they do in CSP.

CFFD-equivalence preserves precisely the following information of the behaviour of the system:

- All sequences (both finite and infinite) of visible actions that can arise from the executions of the system. (They are called *traces* and *infinite traces*, respectively.)
- All sequences of visible actions that arise from those executions of the system that end up with a *livelock* (*divergence traces*). Here, a livelock is an infinite sequence of invisible actions.
- All *stable failures*. Their purpose is to make it possible to derive the sequences of visible actions that arise from those executions that end up in a deadlock state. Stable failures contain more information than just that, because the ability to deadlock depends not only on the LTS, but also on the readiness of its environment to execute any of the visible actions that the LTS is willing to execute. This is a consequence of the fact that communication is synchronous. A stable failure is a pair that consists of a trace and a set of visible actions such that the LTS may execute the trace and then be in a state where the only thing it may do is to communicate via actions that are not in the set.
- Whether or not the system can initially perform an invisible action (*initial stability*). This bit of information is preserved for a technical reason, namely to ensure the congruence property in the presence of some LTS composition operators other than parallel composition.

These properties are sufficient for answering many kinds of verification questions. They suffice for analysing whatever can be specified with the well-know "next-state"-free *linear temporal logic* ($\mathrm{LTL}_{-\mathsf{X}}$) of Manna and Pnueli [15], a logic that is used very widely in verification. They can also detect deadlocks and always distinguish them from livelocks, which $\mathrm{LTL}_{-\mathsf{X}}$ cannot always do. What is more, it was shown in [10] that among all equivalences that preserve $\mathrm{LTL}_{-\mathsf{X}}$ and deadlock properties and that can be used in the compositional LTS construction method, CFFD-equivalence preserves the smallest amount of information, and thus gives the best reduction results.

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 only ten or so 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 properties in $\mathrm{LTL}_{-\mathsf{X}}$ or other logics; even users who do not know any temporal logic or similar formalism can use the approach; and the pictures occasionally show out 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 [26]. 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 essentually equally reliable results because of the "second gap" discussed in Section 3.1.

Much more information on CFFD-equivalence and its use in compositional verification can be found in [22]. CFFD-equivalence and the related tools have been used in a number of case studies. In most of them the examples came from the academia or from the literature, and the case study was conducted to develop verification methodology [9, 16, 23, 25, 26]. There have, however, been case studies where the initiative came from a non-university public research organisation [11] or from the research unit of a big company [13]. In the case of this paper, the example system came from an ordinary development unit of an industrial company.

4 The Protocol

As was already mentioned in the introduction, 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.

The physical channel which the protocol uses is an asynchronous bidirectional serial connection, where data is transmitted in the 8N1-format (eight data bits, no parity and one stop bit) in both directions. This makes the protocol byte-oriented, in contrast to the bit-oriented HDLC protocol [7, 8, 19], which means that the smallest unit of information that can be transmitted in the channel is a byte.

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 $\mathsf{0D}_{\mathsf{hex}}$ byte or a $\mathsf{7E}_{\mathsf{hex}}$ byte to select the terminal or the PC mode, respectively. The $\mathsf{7E}_{\mathsf{hex}}$ byte is at the same time the *flag* byte — that is, the byte that indicates start of a frame — of the first frame that the client sends. The server never makes an initiative to start communication or to select the mode, since the client is often a device that may operate in one mode only and the server can handle both modes.

4.1 The terminal mode

The terminal mode basically gives the server and the client direct access to the underlying channels. In the terminal mode, whenever the server or the client gives the protocol a "legal" byte to be transmitted, the protocol simply sends it to the opposite site through the channel. Certain 8-bit combinations are considered "illegal" and not transmitted. The protocol uses some of them for control purposes, such as $\mathsf{7E}_{\mathsf{hex}}$ for signalling change to the PC mode. If the protocol entity, while in the terminal mode, receives a legal byte from the channel, it just delivers it to the station. Both directions (from client to server and vice versa) operate independently of each other.

The terminal mode is no more reliable than the underlying channels. 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 weird 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 was designed to investigate this interaction.

4.2 The PC mode

The PC mode of the protocol is based on the well-known *HDLC* (High Level Data Link Control) protocol [7, 8, 19]. For simplicity and efficiency, the mode does not implement all of the functionality of the HDLC. The PC mode can be thought of as consisting of four layers, as is shown in Figure 1.

In the terminal mode the unit of transmitted data was a *byte*. When transmitting information, nothing but data bytes went to the channel. Instead, in the PC mode the unit is a *message*. Byte sequences in the channel form so-called *frames* which can include bytes of messages.

At the sender side the *message handling* divides messages to pieces short enough to be sealed into frames. The *flow control* is the most complicated layer. It establishes, maintains and breaks off connection between protocol peers and recovers (or at least tries to recover) from channel errors with a mechanism that is based on acknowledgements, timeouts and sequence numbers. The flow control packs pieces of messages into frames. Finally, the *frame handling* sends frames, byte by byte, to the channel and adds two-byte checksums. In order to ensure

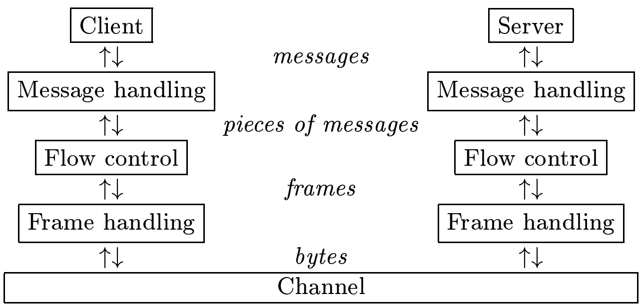

Fig. 1. PC mode layers

that the receiver is always able to distinguish a start of a frame, it adds a special byte after each $7E_{hex}$ byte that does not start a frame. This technique is known as *byte stuffing*.

Naturally, at the receiver side the order is reversed. The frame handling reconstructs frames out of bytes received from the channel: it removes byte-stuffing and checks with the checksum that the bytes were not corrupted. The flow control acknowledges received frames and provides information in correctly numbered frames to the message handing, which finally gives whole messages to the receiver.

A frame that contains a message or a piece of a message is called an I frame (information frame). Each I frame contains also the information that either the message continues in the next I frame or is ended with this I frame. With this information the message handling is able to rebuild messages from the pieces received from frames.

The flow controls of both stations have two three-bit counters: S and R (send and receive) which are set to zero when a PC mode connection is established. Each I frame includes a sequence number (the value of the S counter) and the number of the I frame that the sender expects to receive next (the value of the R counter). Stations acknowledge received I frames with RR or REJ frames (receive ready or reject) which include the number of the I frame that the sender of the acknowledgement expects to receive next. If the sender of the I frame received an RR frame with the number that was the successor of the sequence number of the I frame, the frame was acknowledged. If the sender received a REJ frame, it fixed its S counter value to the one given in the frame and resent the unacknowledged I frame.

RR frames are also sent to avoid disconnecting because of timeout. If a certain time has passed since the last frame was sent, an RR frame is sent. If a station receives a frame that has a CRC error, it is not handled and a REJ frame is sent to inform the other station about the corruption of the frame.

The last three frame types are unnumbered. An $SABM$ frame (set asynchronous balanced mode) is sent by the client to inform the server that it wants to create a connection in the PC mode. The $SABM$ causes the server to set its S and R counter to zero and to reply with a UA (unnumbered acknowledgement) frame. When the client receives the UA frame, it regards the connection to be established and can start sending and receiving other frames.

When a station decides to break off the connection, it sends a $DISC$ (disconnect) frame and leaves the PC mode for the synchronisation mode. If a station receives a $DISC$ frame, it sends a $DISC$ frame and returns to the synchronisation mode. A station disconnects if it did not receive any frame a in certain time.

5 The H model

The protocol was modelled in top-down order — the first model concerned thus the highest layers of the protocol and was called the **H** model. With the model,

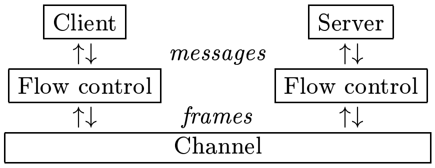

Fig. 2. The structure of the H model

verification of the protocol was focused on the flow control. The structure of the **H** model is presented in Figure 2.

Message handling (see Figure 1) was omitted from the **H** model since it is not essential for the reliability of the protocol. In the **H** model every message fits in one frame. If such messages are received in the same order as they were sent without losing or duplicating any of them, the protocol is reliable and building a message handler is a straightforward task.

All the functionality of the flow control was modelled. To keep the size of the state space as small as possible, the three-bit counters (S and R) were limited to two-bits. This simplification preserves all alternatives with received frames: a correctly numbered frame, a duplication of a frame, an incorrectly numbered frame and a reserved frame number zero that is used only after creating a connection.

Everything below the frame control was abstracted to the model of a channel. When a frame was put into the channel, it could be either passed to the receiver or lost, or an error in the CRC check could occur. This model of the channel assumed that in spite of any errors that could occur in the byte level channel, the frame handler cannot become so confused that it recognises a frame where there is no frame. To make sure that this assumption was correct, the **L** model was used (see Section 6).

Because of the design of the physical connection, the order of frames could not change in the channel.

The **H** model was built in small steps, starting from a very simple model. This made it easier to be aware of the size of the state space and to make sure that the model had no errors. When one version of the model was successfully built and analysed, more details were added to the next version. After a discussion of message sources, the versions of the **H** model are represented with the results of the analysis.

5.1 Message sources

Two different message sources were used to reveal the behaviour of the model in different environments. A message source gives input to the protocol. In this case, the input is composed of two different messages called m_0 and m_1. One message source at a time was put in parallel with the model of the protocol. The reliability of the protocol was verified visually from the resulting system by examining the output of the protocol.

The message source 1 (Figure 3a) sends at first an arbitrary number of messages m_0 and then an arbitrary number of messages m_1. With the message source it was possible to see whether or not the protocol was able to change the order of messages. If the message source in parallel with the **H** model had had a possible output where m_1 appeared before m_0, there would have been an error in the protocol. Furthermore, if the protocol had been able to change the order of messages, then the tools would have shown at least one instance of m_1 appearing before m_0. This is because the protocol is *data independent*, that is, it does not look at the contents of the data it handles. 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 independency in verification was investigated in [29].

The message source 2 (Figure 3b) sent at first an arbitrary number of messages m_0, then once message m_1, and finally an arbitrary number of messages m_0 before it stopped. If the protocol had been able to to duplicate messages, it would have been 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 their delivery — or even input — forever would have been detected. Livelock would have been a sign of such a problem, as would also have been the protocol terminating without outputting an m_1. Furthermore, these suffice. Namely, because of data independency, we may assume that the first message that is lost is the only m_1. If the loss occurs in an execution where the data source tries to send only a finite number of messages, then either a livelock or termination without an m_1 is detected. If the loss occurs in an execution where the data source tries to send an infinite number of messages, either a livelock results, or also the output is infinite. In the latter case, there is a point in time in the execution in which the m_1 has been sent, and an m_0 has been output in its place. Because of the assumption that the protocol cannot swap or duplicate messages, this m_0 must have been input after the m_1. The state space contains also another execution which is identical to the described one until this point, and then stops the data source by executing a τ-transition. Because the protocol cannot swap messages, the m_1 cannot be ouputted even in this execution. The execution thus ends up with a livelock or deadlock without outputting m_1.

The presence of the τ-transition in the message source was used in this reasoning, so it is important. Without the τ-transitions, an error where the protocol delivers a message only after it has input the next message would not have been detected, because the data sources would always have been ready to send a new message.

5.2 The first version — one direction transmission

In the first version of the **H** model only the client was able to send messages and only the server was able to acknowledge them.

At first the model created a connection between stations: the client sent *SABM* frames until it received a *UA* frame. The server ignored all other received

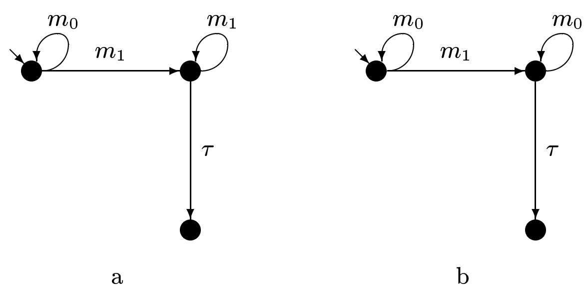

Fig. 3. The message sources used in verification with the H model

frames except *SABM*, to which it responded with a *UA* frame and regarded the connection established.

The client, after receiving the *UA* frame, started transmitting *I* frames. If it did not get an acknowledgement before timeout, it resent the frame. When the connection was created, the server started waiting for frames. If it did not get any, it sent an *RR* frame to preserve the connection. The server was able to acknowledge correctly numbered original and re-sent *I* frames, and to fix the numbering by sending *REJ* frames, if it had received an incorrectly numbered frame.

The channel either passed frames or lost them. Checksum errors were not modelled in this version.

The version of the model seemed to ensure reliable data transfer — the order of messages did not change and the messages were neither lost nor duplicated.

5.3 The second version — resynchronisation

The first version was extended by adding to the model a resynchronisation mechanism. Resynchronisation was initiated after losing a frame cycle, i.e. after not finding the flag byte where it was expected to be. When a station lost frame cycle, it started transmitting *SABM* frames and waiting for a *UA* frame to recover.

The channel was changed so that in addition to passing and losing a frame it was able to force the receiver to lose frame cycle.

This change allowed the model to livelock, which means that the protocol was able to execute forever without doing anything useful. The cause of the livelock turned out to be the situation where both stations lost the frame cycle. Then both stations wanted to receive a *UA* frame but sent only *SABM* frames which made the appearance of *UA* frames impossible.

In the implementation of the protocol this problem did not appear because also *UA* and *DISC* frames were accepted when automatic 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. The idea was that if some data were lost because of the loss of frame cycle, the sender will eventually resend it because of not getting acknowledgement in time.

5.4 The third version — CRC errors

This version of the model was based on the first version which was extended with CRC errors. A CRC error meant a situation where a frame was received but the value of the frame check sequence was not equal to the CRC result.

When a frame was put into the channel, it was passed through, lost, or the receiver was told that there was a frame with a CRC error. The receiver was not told what type of frame caused the error because it did not affect its behaviour. When a station got the CRC error it sent a *REJ* frame with the value of the R counter, which is the same as the number of the *I* frame that was expected to be received next.

The model did neither lose nor change the order of messages but it was able to duplicate them. Duplication occurred, for example, after the following execution: the server received a correctly numbered *I* frame and sent an acknowledgement which disappeared or caused a CRC error to the client. The next frame that the server received (retransmission or a *REJ* frame) had a CRC error and the server sent a *REJ* frame with the new R counter number. The client reset its S counter because of the *REJ* frame and resent the unacknowledged message in an *I* frame with the new frame number. This made the server to believe that it had received a new message.

When Instrumentointi Oy was informed about this error scenario, it 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 sent immediately an *RR* frame. The *RR* frame could either acknowledge a message or inform the sender to resend an *I* frame (in the case that an *I* frame caused the error). Now that the resending does not wait for the timeout, recovering from CRC errors is faster.

With the correction the model became reliable. Instrumentointi Oy corrected the implementation accordingly.

5.5 The fourth version — bidirectional information transfer

The third version of the model was extended to be bidirectional: the client was able to send acknowledgements and also the server could send information frames. The only difference between stations was the way they established connection.

Two message sources were put in parallel with the system so that both the client and the server could send information at the same time. Output of both stations was examined. The model stayed reliable.

5.6 The fifth version — counter values

The protocol was designed to handle a situation when the S and R counters of a station differ from the counters of the other station. *REJ* frames were designed to be used to agree upon the correct counter values. To model this, the fourth version of the model was extended so that exactly once during the execution, the counters of the server could suddenly get just any values. Because of the symmetry of the protocol after establishing connection, the results would have been similar, had the counters of the client been disturbed instead.

Destroying the counter contents did not have any effect on message transmission from the server to the client — this direction stayed reliable. However, messages that were sent from the client to the server could duplicate. The duplication appeared, for example, in the following execution: the server received a correctly numbered *I* frame and sent an acknowledgement which was then broken in the channel. The server's counter contents were destroyed. The client resent the *I* frame because of a timeout. The frame was now incorrectly numbered and the server sent a *REJ* frame to synchronise the counter of the client. The next *I* frame that the client sent had new numbers and the same message but the server handled it as a new one.

The model was changed so that when a station received a frame with incorrect numbers it sent a *DISC* frame and closed the connection. This change prevented the execution that was described above but duplication could still take place. It was caused by the following execution: the server received a message and sent an acknowledgement which was broken in the channel. The R counter of the server changed so that it got a value that it had had before receiving a message. The client resent the *I* frame and the server received the same message again without noticing that it was a duplicate.

Therefore, if the counters were allowed to get confused, the protocol could not be reliable. Fortunately, according to the model, the counters could never get confused by any error that occurred in the protocol. *REJ* frames were never sent unless the counters were externally set to random values.

6 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 could accidentally change to $\mathsf{0D}_{\mathsf{hex}}$, forcing the server to go to the terminal mode. Therefore, the interplay of the different connection modes was analysed with the **L** model.

Two versions of the **L** model were built and analysed. With the first version we wanted to check if the frame handler was able to build frames from bytes received from the channel in the case that the bytes were not changed, but an arbitrary number of previous bytes could have been changed or lost.

To model this, we built a model where frames were sent byte by byte, possibly with byte-stuffing bytes, into the channel. The *frame dissolver* process encoded

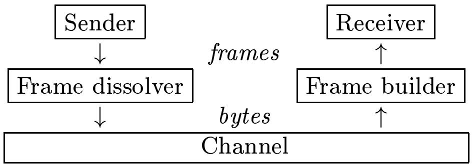

Fig. 4. The structure of **L** model

frames received from the message source as byte sequences. In addition to passing a byte correctly to the receiver, the channel was able to lose the byte or change it into any other byte. However, no protocol can work if every byte is changed to just any byte in the channel, so losing and changing bytes in the channel had to be restricted. The message source controlled a *breaker* process, which either allowed or forbad the channel to break (lose or change) bytes.

In the second version of the **L** model, which was also the final step of this formal analysis, we modelled the three modes of the server: the PC mode, the terminal mode and the synchronisation mode. We wanted to examine whether changing between these modes had an effect on creating connections and receiving frames.

6.1 The message source and the frame dissolver

In the **L** model the message source gave the frame dissolver frames to be put into the channel. At first the message sender sent an arbitrary number of *SABM* frames, whose bytes the channel was able to break. Then breaking of the bytes was prevented and one *UA* frame was sent. The *UA* frame was followed by an arbitrary number of *DISC* frames. The intention was to check whether the *UA* frame would arrive safely, without loss or duplication. If yes, then, because of data-independency, an arbitrary frame would arrive safely provided that the channel has stopped breaking bytes. The only significance of the types of the frames in this model was to make it possible to tell them apart.

The frame dissolver encoded the frames for sending to the channel using six bytes: bFlag, bBS, bSABM, bUA, bDISC and bOther. A frame began with a bFlag byte. It was followed by a control byte that was a bSABM, bUA or bDISC depending on the frame. Finally two CRC check bytes were put to the channel. A CRC check byte could be any of the six bytes in use. If it was a bFlag, a bBS was inserted after it to model byte stuffing. The length of the frame was thus from four to six bytes.

6.2 The breaker

The behaviour of the channel (Figure 5c) was controlled by the message source (Figure 5a) and the breaker process (Figure 5b).

At first the message sender forced the breaker process to enable breaking (action *eb*). The breaker could synchronise with the channel with a break action

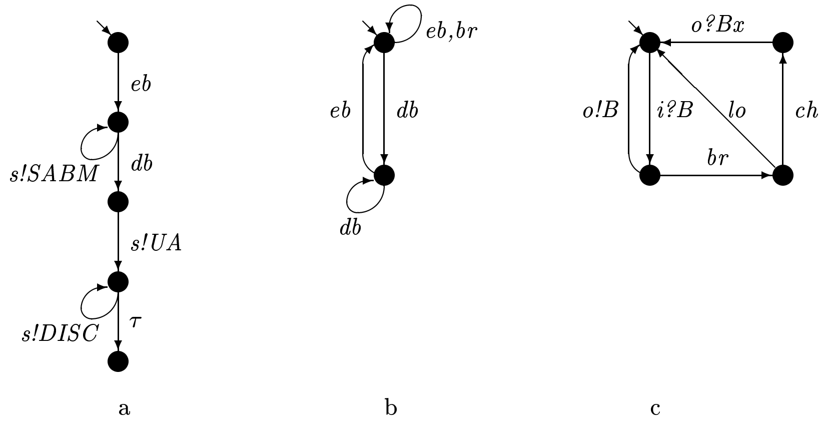

Fig. 5. Controlling errors in the channel

(br) after a byte was input to the channel ($i?B$). After executing the break the channel could either lose (lo) or change (ch) the byte.

When an arbitrary number of *SABM* frames had been sent, the message sender disabled breaking by executing db. The breaker could no more synchronise with the channel with the br action, and thus the channel had to output all bytes without losing or changing any ($o!B$).

6.3 The frame builder and the CRC checking

The frame builder waited for a flag byte from the channel, ignoring all other bytes. When it received one, it started building a frame.

Implementing a real CRC check mechanism in a verification model like this would have been an overkill. Furthermore, a real CRC check has a positive, although small, probability of an error going unnoticed. A real protocol may make an error in such a situation, but this is accepted because it occurs so rarely, and because it is impossible to build absolutely perfect systems. We do not want those errors to be present in the verification model, however, because they might make important errors more difficult to distinguish.

Because of these reasons, the CRC checking was modelled as follows. Whenever the channel broke a byte, it told the frame builder that it did so. The frame builder remembered this information until it had received either the expected number of bytes, or a new flag byte. Then it announced the arrival of a frame with a CRC error. An arrival of a correct frame was announced, if all of its bytes arrived without the channel telling about breaking any of them.

With this construction, the frame builder never mistook a corrupted frame as a correct one. It was, however, possible to interpret a correct frame as an incorrect one, because one option of breaking a byte was changing it to itself.

This situation corresponds to an error in the checksum without any error in other parts of the frame.

6.4 Results from frame building

In the **L** model every frame in which bytes had not been changed or lost in the channel was successfully received. This means that any errors that had happened in any previous frames had no effect on building a new frame, if the bytes of the new frame were received without errors.

Also a phenomenon that had not been paid attention to in the **H** model was found: sending one frame could cause a receiver to find a number of CRC errors instead of only one. This happened, for example, in the case that a byte-stuffing byte after a $\mathsf{7E}_{\mathsf{hex}}$ byte in the body of a frame was changed or lost. In the **H** model many CRC errors would have forced a receiver to send more than one RR frame even if a sender sent only one frame.

The **L** model was not able to diverge or deadlock. So the lower level of the protocol seemed to work correctly, and the assumptions made in the **H** model were reasonable (possibly with the exception of the absence of multiple RR frames).

6.5 Terminal, PC and Synchronisation Modes

The **L** model was extended to cope with the terminal, PC and synchronisation modes. This required that time passing had to be added to the model, because the server changed mode from PC to synchronisation if a timeout occurred.

Time was modelled by adding a *time* action to the model. Executing the time action meant that enough time had passed without receiving any frames, so that the protocol should disconnect, i.e. change the mode from PC to synchronisation. The message source controlled the time action so that it could be executed only between frame sendings. When a frame was given to the frame dissolver, the time action could not be executed until all its bytes were sent through the channel. This modelled the fact that the disconnect timeout time is long compared to the time needed to send a frame.

A new initial state was added to the model of the receiver to correspond to the synchronisation mode. If a $\mathsf{0D}_{\mathsf{hex}}$ byte was received from the channel, the receiver started working in the terminal mode for which another state was also added. From these two states a received flag byte took the server into the first state of the PC mode. Also the bytes that were broken before entering the PC mode were taken into account in the CRC check. From the PC mode, the time event returned the receiver to the initial state.

6.6 Final Results

When the channel was allowed to change bytes, the server was able to work in the terminal mode even if the client was in the PC mode. Consequences of this depend on how the terminal mode output is handled. Other than that, the previous verification results were not fundamentally changed. When the channel was not able to break bytes, all frames were successfully received in spite of the current mode of the server and earlier errors.

With this result we could be certain that the connection could be (re-)established if no errors occurred in the channel. Thus in the beginning of the execution of the **H** model, when there was no connection, it was reasonable to assume that the modes of the server would not cause any trouble in receiving frames.

7 Conclusion

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 was partially implemented and tested during the project.

The design had some errors which earlier testing of the specification and the implementation had not revealed. With verification tools all the behaviour of the entire system, up to the accuracy of the model, could be analysed to find errors that had escaped testing.

The protocol was somewhat complicated: it had two operation modes, frame numbering, resending, acknowledging, rejecting, byte stuffing and CRC checking. The behaviour of every version of the models of the protocol could, however, be examined visually from the results provided by the verification tools. Visual representations, that is, directed graphs with labelled edges, of the behaviour of the system captured both safety (in Section 5.4) and liveness (in Section 5.3) errors. After an error was found, visualisation was used to find an execution that leads to the erroneous behaviour. Visualisation revealed also a behaviour that was not expected and thus could have remained unnoticed in verification against a fixed requirement specification (in Section 6.4).

The project had a remarkable effect on the design of the protocol. Firstly, to create a formal model of the protocol, the specification needed to be exact. Before modelling was started some additions and changes were made to the specification. Some details were added and ambiguities corrected also during the modelling. Secondly, whenever the analysis revealed an error in the model, the implementation was tested with executions that lead to the error. When the error was found in the implementation, too, changes that were verified with the model were adapted to the implementation and the specification.

Acknowledgements

The work of Kervinen was funded by the Academy of Finland, project UVER.

References

1. Boigelot, B.: *Symbolic Methods for Exploring Infinite State Spaces.* PhD Thesis, volume 189, Collection des Publications de la Faculté des Sciences Appliquées de l'Université de Liège, 1999, 300 p.

2. Bolognesi, T. & Brinksma, E.: "Introduction to the ISO Specification Language LOTOS". *Computer Networks and ISDN Systems* 14 (1987), pp. 25–59.
3. 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.
4. Comité Consultatif International de Télégraphe et Téléphone: *Specification and Description Language.* CCITT Z.100, Geneva 1988.
5. Hoare, C. A. R.: *Communicating Sequential Processes.* Prentice-Hall 1985, 256 p.
6. 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.
7. International Standards Organisation: *Data communication networks — High-level data link control (HDLC) procedures – Frame structure.* Ref. No. ISO 3309, ISO, Geneva, 1993.
8. International Standards Organisation: *Data communication networks — High-level data link control (HDLC) procedures – Elements of procedures.* Ref. No. ISO 4335, ISO, Geneva, 1993.
9. Kaivola, R.: "Using Compositional Preorders in the Verification of Sliding Window Protocol". *Proc. Computer Aided Verification 1997*, Lecture Notes in Computer Science 1254, Springer-Verlag 1997, pp. 48–59.
10. Kaivola, R. & Valmari, A.: "The Weakest Compositional Semantic Equivalence Preserving Nexttime-less Linear Temporal Logic". *Proc. CONCUR '92, Third International Conference on Concurrency Theory*, Lecture Notes in Computer Science 630, Springer-Verlag 1992, pp. 207–221.
11. Karsisto, K. & Valmari, A.: "Verification-Driven Development of a Collision-Avoidance Protocol for the Ethernet". *Proc. Formal Techniques in Real-Time and Fault Tolerant Systems*, Lecture Notes in Computer Science 1135, Springer-Verlag 1996, pp. 228–245.
12. 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.
13. Leppänen, S. & Luukkainen, M.: "Compositional Verification of a Third Generation Mobile Communication Protocol". *Proc. International Workshop on Distributed System Validation and Verification*, IEEE, 2000.
14. 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.
15. Manna, Z. & Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems, Volume I: Specification.* Springer-Verlag 1992, 427 p.
16. Puhakka, A. & Valmari, A.: "Verification of Self-Synchronizing Alternating Bit Protocols with ARA". *Proc. Fifth Symposium on Programming Languages and Software Tools*, University of Helsinki, Department of Computer Science, Report C-1997-37, pp. 167–178.
17. Roscoe, A. W.: *The Theory and Practice of Concurrency.* Prentice-Hall 1998, 565 p.
18. Savola, R.: *A State Space Generation Tool for LOTOS Specifications.* VTT Publications 241, Technical Research Centre of Finland (VTT), Espoo, Finland 1995, 107 p.
19. Tanenbaum, A. S.: *Computer Networks (3rd ed.).* Prentice-Hall 1996, 848 p.
20. Turner, K. J. (ed.): *Using Formal Description Techniques, An Introduction to Estelle, Lotos and SDL.* John Wiley & Sons 1993, 431 p.

21. Valmari, A.: "The State Explosion Problem". *Lectures on Petri Nets I: Basic Models*, Lecture Notes in Computer Science 1491, Springer-Verlag 1998, pp. 429–528.
22. Valmari, A.: "Composition and Abstraction". *Modelling and Verification of Parallel Processes, Proceedings of the Summer School*, Nantes, France June 2000, pp. 3–32.
23. Valmari, A., Karsisto, K. & Setälä, M.: "Visualisation of Reduced Abstracted Behaviour as a Design Tool". *Proc. PDP'96, the Fourth Euromicro Workshop on Parallel and Distributed Processing*, IEEE Computer Society Press, pp. 187–194.
24. 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.
25. Valmari, A. & Kokkarinen, I.: "Unbounded Verification Results by Finite-State Compositional Techniques: 10^{any} States and Beyond". *Proc. 1998 International Conference on Application of Concurrency to System Design*, IEEE Computer Society 1998, pp. 75–85.
26. 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.
27. 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.
28. Valmari, A. & Tienari, M.: "Compositional Failure-Based Semantic Models for Basic LOTOS". *Formal Aspects of Computing* (1995) 7: 440–468.
29. Wolper, P.: "Expressing Interesting Properties of Programs in Propositional Temporal Logic". *Proc. 13th ACM Symposium on Principles of Programming Languages*, 1986, pp. 184–193.