Validation of Bosch' Mobile Communication Network Architecture with Spin

Theo C. Ruys and Rom Langerak
Faculty of Computer Science, University of Twente.
P.O. Box 217, 7500 AE Enschede, The Netherlands.
{ruys,langerak}@cs.utwente.nl

March 14, 1997

Abstract

This paper discusses validation projects carried out for the Mobile Communication Division of Robert Bosch GmbH. We verified parts of their Mobile Communication Network (MCNet), a communication system which is to be used in *infotainment* systems of future cars. The protocols of the MCNet have been modelled in Promela and validated with Spin. Apart from the validation results, this paper discusses some observations and recommendations of the use of Promela and Spin.

Keywords protocol, validation, verification, model checking, literate programming, Spin, Promela

1 Introduction

This paper discusses the validation of parts of the Mobile Communication Network (MCNet), which has been developed at the Mobile Communication Division of Robert Bosch GmbH. MCNet is a communication system which is to be used in *infotainment systems* [14] of future cars. Infotainment systems encompass *entertainment* systems like conventional car radios equipped with CD changers and *driver information systems* like navigation units, (mobile) telephones, etc.

New technologies like RDS Plus (Radio Data System), TIM (Traffic Information Memory), DAB (Digital Audio Broadcasting), etc. put high requirements on infotainment systems, which will develop into complex, integrated systems. Communication between the various components of such systems will become increasingly important.

As an example, suppose one is driving in such a modern car. Assume that while listening to a CD the RDS system receives information about a traffic jam on the route that was laid out by the navigation unit. The CD should be immediately interrupted to transmit the radio announcement and furthermore, the navigation unit should find an alternative route. But what should happen if at the same time the mobile phone rings? Should the radio announcement be interrupted? And if so, should the announcement be saved? It should be clear that the communication network underlying the complete infotainment system should be highly reliable.

In the last two years, our group[1] has been involved in the validation of the MCNet Architecture. We have modelled parts of successive versions of MCNet in Promela [7] and we have used Spin[2] to simulate and verify our models.

[1] The Formal Methods and Tools group of the Department of Computer Science of the University of Twente.

[2] In this paper use the term Spin when discussing the simulation and validation of Promela models. In most cases, however, we used the graphical interface Xspin, which resides on top of Spin, to manipulate the Promela models.

With Spin we were able to pinpoint several caveats in earlier versions of the MCNet (i.e. [17] and [18]). Bosch adopted most of our recommendations to correct and improve the reliability of the MCNet protocols. We did not find any serious errors in the last version of MCNet [19] that we validated.

The structure of the rest of this paper is as follows. In section 2, an introduction to the MCNet Architecture is given. Our validation efforts are summarized in section 3. In section 4, some experiences with Promela and Spin are discussed. The paper is concluded in section 5, where some conclusions and recommendations are presented.

2 The MCNet Architecture

The Mobile Communication Network (MCNet)[3] is developed at the Mobile Communication Division of the Bosch Blaupunkt Group, a subdivision of Robert Bosch GmbH (abbreviated to Bosch). Bosch is a German company, which is well known as a supplier of components and subsystems to the automative industry. Other areas of business of Bosch include communication technology, consumer goods and capital goods.

[19] specifies the goal of MCNet: "The aim of MCNet is to specify a reliable system capable of transferring driver information and control data. It was not designed for exchanging audio or video data." MCNet is proposed as a company-wide standarization within Bosch. Later, the MCNet specification will be offered to interested third parties to reach a more widespread usage [19]. For this reason, much work has been carried out within the OSEK[4] group. The main result of this collaboration is that the MCNet's Transport protocol is in conformance with the upcoming OSEK standard.

The MCNet will be built on top of a Controller Area Network (CAN) [3], which is a serial data communications bus system especially suited for networking 'intelligent' I/O devices. Features of the CAN include a multi-master protocol, real-time capabilities, error correction and high-noise immunity. The CAN serial bus system, originally developed by Bosch to reduce the need for massive wiring harnesses in automobiles, is increasingly being used in industrial automation.

CAN is an international standard and is documented in ISO-11898 [3] (for high-speed applications) and in ISO-11519 [1, 2] (for lower-speed applications). Gentle introductions to CAN can be found on the Internet, for example [8] and [21].

2.1 Layered Architecture

Because of similar demands, the MCNet resembles the layered architecture of ISO/IEC's standard on Open System Interconnection (OSI) [4]. Using the standard procedures and protocols of the OSI architecture as a basis for MCNet has the following benefits [19]:

separation of concerns: applications and communications are cleanly separated from each other;

reusability: not only abstract concepts, but also (existing) software and hardware may be reused;

interoperability: individual layers can be modified or replaced in keeping with changing requirements (as long as internal interfaces need not be modified);

testability: the development, manufacturing and diagnosis of the network is more easily tested.

Figure 1 (from [19]) shows the correspondence between MCNet and the OSI Layers.

[3]The MCNet was formerly called Mobile Communication Architecture (MCA).

[4]OSEK is an abbreviation for the German "Offene Systeme und deren Schnittstellen für di Elektronik im Kraftfahrzeug" (Open systems and the corresponding interfaces for automotive electronics) and is the international standardization group of car manufacturers and suppliers. It aims at an industry standard for an open-ended architecture for distributed control units in vehicles. The parties within OSEK are: Opel, BMW, Daimler-Benz, Mercedes-Benz, Robert Bosch, Siemens, Volkswagen and the University of Karlsruhe.

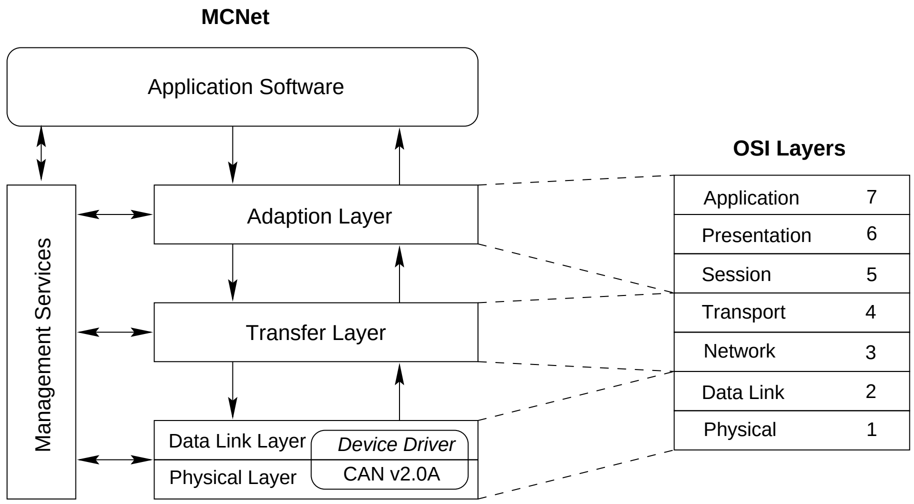

Figure 1: Correspondence between MCNET and the OSI layers.

The **Physical Layer** is responsible for sending and receiving bit streams. Currently, for reasons of cost reduction and extended error-failure management, only a low speed (a bit rate below 125 Kbit/s) is applied.

The **Data Link Layer** is responsible for sending and receiving CAN telegrams. The Data Link services used in MCNET are a subset of those defined in [2] for CAN low speed.

The **Transfer Layer** corresponds to the Transport Layer of the OSI (layer 4) and is responsible for the segmentation of messages of any length. The MCNET Transfer Layer conforms to the upcoming standard of OSEK v2.0. The Transfer Layer offers three data services to the Adaption Layer [19]:

- *Long Data Service:* connection-oriented acknowledged data transfer service for transportation of 'long messages' (more than 7 bytes of application data);
- *Expedited Data Service:* expedited connection-oriented acknowledged data transfer (7 bytes or less of application data);
- *Broadcast Data Service:* unacknowledged broadcast data transfer service.

The Transfer Layer also supports services to test the connection between protocol entities.

In the current version of MCNET [19], the Transfer Layer is specified as a sliding-window protocol where a protocol entity can send a block of Transfer Protocol Data Units (TPDUs) before awaiting the acknowledgement of the block. Previous versions of MCNET [17, 18] only supported a 'send and wait (for the acknowledgement)' protocol.

The **Adaption Layer**, the highest layer of MCNET, serves as the interface between the communication software and the application software.

The **Network Management** is not a part of the OSI reference model, but instead is commonly used in the area of fieldbus systems. It can be viewed as an additional 'vertical' layer with control interfaces to each of the 'horizontal' communication layers (see Figure 1). The management services encompass both network and (local) station management.

2.2 Defining documents

The consecutive versions of the MCNET protocol are defined and well documented in [17], [18] and [19]. For each protocol layer, the documents discuss the following aspects:

Service The service definitions offered by the particular protocol are presented: the service primitives together with their parameters are listed.

PDU If applicable, the Protocol Data Unit (PDU) structure of the protocol is defined.

Protocol The protocol itself is formally defined through *state transition tables*.

Time sequence diagrams Some of the protocols are illustrated with time sequence diagrams describing particular scenarios.

Although the service primitives supported by a particular protocol are well defined, the *desired behaviour* of the protocol is unclear from the service primitives alone; the (formal) relations between the primitives is missing. For example, in the Transfer Layer, it is unclear what should happen if a connection request is issued at a service access point of a Transfer protocol entity. Should this (always) result in a connection indication at the other side?

Relations between the service primitives can be deduced from the state transition tables. Such relations, however, are constraints on this particular design of the protocol. They do not define the 'desired behaviour' of the protocol as a black box.

Summarizing, the design of the MCNet protocols is well defined and well documented, but the service description of the protocols is missing.

3 Validation

In our validation efforts, we have concentrated our attention on the Transfer Layer of MCNet. We were mostly interested in the *connection setup* phase, the *data transfer* phase and the *connection termination* phase of this Transfer protocol. No distinction was made between the Long Data Service and the Expedited Data Service, because the protocol behaves exactly the same for both services. We did not incorporate the 'Testing Services' and the 'Broadcast Data Transfer Services' into our validation. In our last two validations, we also looked closely at the Network Management Protocol of the MCNet.

In the modelling phase, one of our goals was to come up with Promela models of the protocols, which were as close to the original 'code' of the state transition tables as possible. Although this may have resulted in less efficient Promela constructs, we found two advantages:

- We got more confidence in the Promela model of the particular protocol.
- People familiar with the state transition table (namely the engineers at Bosch) were able to read and understand the Promela models.

The first two validations were carried out by Langerak [12, 13], while the last validation was carried out by Ruys [20]. To get two independent views of the protocols, Ruys did not use the modelling results of the first two validations. The reason for this was that the authors wanted to find differences and similarities in the modelling approaches. But more importantly, the authors hoped to find caveats in the protocols, which had not been found in the first two validation efforts.

3.1 Problems encountered

As we said above, although the *protocol* layers of the MCNet are extensively documented in [17], [18] and [19], the intended, desired behaviour (i.e. the service) of the various layers is missing. This had the following consequences for our work:

- Because the 'rationale' behind the protocols was missing, the modelling of the protocols (especially the Network Management Protocol) was time consuming. Deriving the intended service by studying transition tables became reverse engineering.

- If a protocol doesn't contain any obvious errors (e.g. deadlocks), the protocol should be validated against its service description. If such a service description is missing, it is difficult to come up with properties to be checked with Spin.

Another 'documentation problem' was caused by the successive, different versions of the MCNet standard. It proved to be error-prone and time consuming to compare consecutive versions to find the differences.

We also encountered an obvious problem when modelling the Network Management Protocol (NMP) in Promela. The NMP has connections to all horizontal layers (see Figure 1). To model the protocol, we had to include all relevant parts of all these layers.

3.2 Results

In [12], Langerak validated the Transfer Layer of version 0.4 of MCNet. With the help of Spin, he found several caveats in the protocol. For example, he found that:

- when two protocol entities wanted to setup a connection at the same time, this could lead to a deadlock; and
- a single acknowledgement TPDU was used for both the connection acknowledgement and the data acknowledgement. This could also lead to errors.

Bosch adopted the recommendations of [12] and in version 0.7 of MCNet, these errors were corrected. This new version of the protocol, however, contained some minor new errors, which again were discovered with Spin [13]:

- With respect to version 0.4, some - on first sight - redundant transitions had been removed from the protocol. Spin showed that these transitions were really needed, though.
- The protocol contained unspecified receipts that had to be dealt with in an implicit way.

Langerak [13] also modelled the Network Management Protocol (NMP) of MCNet version 0.7. As said before, the main problem with modelling the NMP is that without a service description it is very difficult to deduce the (desired) behaviour of the protocol. Apart from checking for deadlocks, no significant properties were checked.

In our last validation effort, we looked closely at version 0.8 of the OSEK Transport Protocol [5]. In future versions of MCNet, the OSEK Transport Protocol will replace the Transfer Layer in MCNet.

Unlike the 'send-and-wait' protocols of MCNet version 0.4 and MCNet version 0.7, OSEK's Transport Protocol includes a (sliding-window) block transfer mode. We therefore focussed our attention on properties specific to the block transfer mode of this protocol. Again, we did find some errors with Spin. For example:

- The protocol specification of [5] was incomplete. For example, it was not clear how the block size of a connection is negotiated between protocol entities. Therefore it was not certain that the entities of a connection would all use the same block size.
- The protocol description hints on a specific way to implement (and thus, model) 'full-duplex' communication. When adopting this particular scheme in Promela, however, we encountered several problems. These problems are discussed in detail in section 4.3.

In [20] we also modelled the NMP protocol of MCNet version 1.0. Again, it proved too difficult to really validate this protocol as long as the desired behaviour of the protocol is not completely specified.

As we said earlier, to get different views on the protocols, we did not use the modelling results of the first two validations in our last effort. It was remarkable to see that the independent models have much in common. The new, independent look at the protocols did not discover any new errors in the protocols.

Summarizing, with the help of Spin, we were able to find several caveats in the MCNet protocols.

4 Experiences with Spin and Promela

This section discusses some experiences with Spin and some modelling decisions in greater depth.

4.1 Validation trajectory

Our validation effort with Promela and Spin can be summarized as "with every step, more carefully dosed misery". To illustrate this, we present the validation trajectory we used, in the following pseudo algorithm.

```
⟨Modelling and validation in Promela⟩≡
  ⟨Start with abstract model⟩
  ⟨Simulate and verify⟩
  while not ⟨convinced by model⟩
  do
    if
    :: ⟨Add details to model⟩
    :: ⟨Introduce errors into the environment⟩
    fi
    ⟨Simulate and verify⟩
  od
```

In ⟨*Start with abstract model*⟩ we started with a very abstract Promela model of the protocol. In Promela, one also has to specify the *environment* of the protocol, which consists of the user of the protocol (i.e. the upper layer service) and the underlying (lower layer) service of the protocol (e.g. physical lines). In the beginning, we modelled a correctly behaving environment.

We have borrowed Promela's `if` statement to choose between ⟨*Add details* ...⟩ and ⟨*Introduce errors* ...⟩. In ⟨*Add details to model*⟩ more details are added to the model of the protocol. This may introduce errors, which will be discovered in the next ⟨*Simulate and verify*⟩ step. This sometimes meant "one step forward, two steps back".

In ⟨*Introduce errors into the environment*⟩, the carefully dosed 'misery' is introduced. For the protocol to be robust, it should deal correctly with errors in the environment. For example, errors in the underlying service will result in messages that may be lost in transit. Erroneous ordering of (user) service primitives, on the other hand, may lead to unspecified receipts in the protocol, and hence deadlock.

The algorithm that we used to simulate and verify is shown below.

```
⟨Simulate and verify⟩≡
SIM:  while not ⟨convinced by simulation⟩
      do
        ⟨Simulate the model with Spin⟩
        if ⟨errors found⟩ then
          ⟨Correct errors in the model⟩
          goto SIM
        fi
      od
      while not ⟨convinced by verification⟩
      do
        ⟨Verify the model with Spin⟩
        if ⟨errors found⟩ then
          ⟨Correct errors in the model⟩
          goto SIM
        fi
      od
```

Most of the hard work of ⟨*Simulate and verify*⟩ is done by Spin. When errors are found, the errors should be corrected, and the simulation and verification step should be started again.

When correcting errors (or adding details to the model) in Promela, one sometimes has to depart slightly from the model and thus possibly introduce (new) errors. As an illustration of this phenomenon, see section 4.3.

The predicates ⟨*convinced by* ...⟩ are determined by the knowledge, experience and intuition of the validation engineer (i.e. the user of Spin) and depend on:

- the number of *details* in the current model;
- the number and kind of *properties* checked so far;
- all previous *simulation and verification runs* for this particular model;
- the models of the protocol, which have been validated in *earlier steps*.

4.2 Modelling timeouts

The MCNet (Block) Transfer Protocol uses several timers. First of all, when setting up a connection, a timer is used to wait for the connection acknowledgement TPDU. Similarly, a sending entity uses a timer to wait for a data acknowledgement TPDU. Furthermore, in the block transfer version of the protocol, when sending consecutive data TPDUs, the sender waits a certain amount of time between the TPDUs to give the receiver time to deal with the incoming TPDUs.

It should be noted that only the sending party of a data connection uses a timer. This means that when the sending party decides to terminate the connection (because it didn't receive an acknowledgement for a long time), the receiving party will not notice the disconnection and keeps waiting for data TPDUs. Although this may seem to be a problem for the Transfer Protocol, in the MCNet Architecture, this is solved by the NMP protocol, which checks that connections are still alive.

In our model of the (Block) Transfer Protocol, a deamon process has been introduced that may steal TPDUs from the underlying physical channel:[5]

7 ⟨*proctypes* 7⟩≡ (11.1)

```
proctype Deamon()
{
  TPDU tpdu ;
  do
  ::  pchan[0]?tpdu
  ::  pchan[1]?tpdu
  od
}
```

where channel `pchan[i]` models the *incoming* line at protocol entity `i`.

Promela provides the `timeout` statement to model system-wide timeouts: if no other statement in the global system is executable, the `timeout` statement becomes executable. So in our model of the Transfer Protocol this would be:

[5]Please ignore the ⟨*proctypes*⟩ definition; this notation will be explained in section 4.4.

```
#define AckTimerExpired       timeout
```

The global `timeout` statement has two disadvantages:

- The `timeout` statement cannot be used to check for erroneous deadlocks in the system; in situations which should be recognized as deadlocks, the `timeout` statement will still be executable.
- Another problem is the *global* nature of `timeout`. There are situations where a ‘local’ timeout is demanded which becomes executable when only this local process cannot proceed anymore. In this case, other processes in the system would not need to be stopped.

On the other extreme of global timeouts, one may want to model *premature timeouts*. As Holzmann [7] suggests, premature timeouts can be modelled by `skip`, which is always executable:

```
#define AckTimerExpired       skip
```

Limiting the premature timeouts

Normally, the general timeout schemes of Promela suffice. In our model, however, we wanted to model actual timeouts and premature timeouts at the same time. Although `skip` will catch both timeouts, it was too coarse for use: we wanted to limit the number of premature timeouts. So we used the following scheme:

```
#define AckTimerExpired       premature_timeout || timeout
```

where `premature_timeout` is a boolean variable that may be `true` if a premature timeout is still possible and `false`, otherwise. Each time the protocol starts waiting for an acknowledgement, this variable `premature_timeout` variable is set:

```
#define set_premature_timeout                            \
  if                                                     \
  ::  nr_premature_timeouts <= MAX_PREMATURE_TIMEOUTS    \
        ->  if                                           \
            ::  premature_timeout = TRUE ;               \
                nr_premature_timeout++                   \
            ::  premature_timeout = FALSE                \
            fi                                           \
  ::  else  ->  premature_timeout = FALSE                \
  fi
```

Notifying the waiting party

If we want to check for deadlocks and assume that premature timeouts are *not* possible, we can also adopt a different timeout mechanism.

When a message is lost in transit, the entity that waits for the acknowledgement will eventually timeout. Instead of letting the entity wait (for its timer to expire), the waiting party is notified that the message has been lost.

A separate timeout channel `timeout_ch` is introduced which is used to notify a waiting party:

```
chan timeout_ch             = [1] of {bit} ;
#define AckTimerExpired       timeout_ch?1
```

In order to make this work, a *polite* `Deamon` process has to be used which notifies the waiting party via the `timeout_ch` channel. When stealing a TPDU message, the `Deamon` process will examine the TPDU message to decide which entity is waiting for an acknowledgement:

```
proctype Deamon()
{
  TPDU tpdu ;
  do
  ::  pchan[0]?tpdu -> NOTIFY(1,0)
  ::  pchan[1]?tpdu -> NOTIFY(0,1)
  od
}
```

where the macro `NOTIFY(i,j)` takes care of the notification of the waiting entity when a message is lost in the line from `i` to `j`.

A note on this last scheme of modelling timeouts is appropriate here. Earlier, we warned against too much of a deviation from the (original) model. In fact, we would agree with the reader who says that this 'timeout channel hack' is a good example of too much deviation. On the other hand, the *polite* `Deamon` provides more control over the timeouts. This control proved quite valuable in finding a stubborn error in one of our models.

4.3 Modelling the full duplex connection

In the Transfer Layer protocol, after the connection setup phase, the protocol entities proceed to the so-called `CONNECTED` state. Transfer of data is full duplex: each entity can send and receive data at the same time. For simplification reasons, in the description of the protocol [5] and [19], the `CONNECTED` state is viewed as the *parent* state of different receiver and sender substates: "the `CONNECTED` state can be modelled as being composed of four independently working state machines (i.e. sending and receiving machines for both the expedited data transfer and long data transfer)". Initially, this hint put us on the wrong track. For a long time we modelled the `CONNECTED` state by starting a separate `Sender` and `Receiver` process for each entity:

```
CONNECTED:
  ...
  atomic {
    run Sender(i, dest) ;
    run Receiver(i, dest) ;
  }
```

Using separate `Sender` and `Receiver` processes in each protocol entity seems a natural and elegant way to model the full duplex data connection, but it turns out to have several disadvantages. For example, the 'parent' process uses variables that are used by both the Sender and Receiver. Therefore, in Promela these variables had to be defined globally. Furthermore, instead of having just a single `proctype` for each protocol entity, three `proctype`s are needed for each entity. This adds to the complexity of the Promela model.

But the most serious disadvantage is the following. During normal operation, the three processes (i.e. the parent process and `Sender` and `Receiver` processes) can work independently. But in the case of a *reset* or *termination* of the connection, all three processes have to be synchronized. This requires non-trivial synchronization schemes between the three processes which are *not* present in the original description of the protocol [5]. For example, we initially modelled part of the `Receiver` process as follows:

```
do
::  Receive(DT) ->      /* normal operation */
od
unless (disconnect_now || reset_now) ;

if
::  disconnect_now ->  /* handle disconnect */
::  reset_now ->       /* handle reset      */
fi
```

where the variables disconnect_now and reset_now are variables that are set by the Sender process and parent process, when the need for a termination or reset of the connection arises. Using this scheme, serious deviations from the original model had to be introduced. As a result, errors were found which where introduced by the 'new' synchronization scheme. But more importantly, errors in the original description may have been camouflaged by the new synchronization scheme.

To solve these problems, we used a single proctype (instead of three) to define the behaviour of a protocol entity. We used two state variables to hold the 'local' state of the sending and receiving 'subprocess': s_state and r_state. We illustrate this below:

```
do
...
::  (r_state == RECEIVING) && Receive(DT)      ->  /* process DT        */
::  (s_state == WAIT_AK)   && Receive(AK)      ->  /* process AK        */
::  (s_state == WAIT_AK)   && AckTimerExpired ->  /* deal with timeout */
...
od ;
```

Handcoding the substates of the sender and receiver is not very elegant. Furthermore, the resulting Promela model 'looks' different compared with the protocol description as defined in [5] and [19]. In this case, however, it turned out to be a better and more reliable model than using the separate Sender and Receiver processes.

4.4 Literate modelling

Literate programming is the act of writing computer programs primarily as documents to be read by human beings, and only secondarily as instructions to be executed by computers [16]. In general, literate programs combine source code and documentation in a single file. Literate programming *tools* then parse the file to produce either readable documentation or compilable source code. The term "literate programming" was coined by D.E. Knuth, when he described WEB [9, 11] the tool he created to develop the TEX typesetting software. Recent book-length examples of literate programming (using C) include Fraser and Hanson's book on lcc (using noweb) [6] and Knuth's book on "The Stanford Graphbase" (using CWEB) [10].

Parallel programs quickly become complex and difficult to understand. Therefore, a parallel program usually needs (extensive) documentation to explain the constructions used in the program. This observation is also valid for specifications written in Promela.

A natural solution is to use the normal /* .. */ comments in Promela specifications. However, the complexity of the models sometimes require lengthy comments, which obscure the actual Promela code.

From our first experiences with Promela we learned that Promela code with some minor comments is not enough for someone (unfamiliar with the protocol) to understand the model and the reasons behind certain modelling decisions. Moreover, a 'final' Promela model only presents the *last* abstraction of the system. The documentation of the validation of a system, however, should also contain the complete set of Promela specifications and Spin results that have lead to this 'final' Promela model.

In our most recent validation effort, Ruys [20] used the literate programming tool `noweb` to tackle this 'documentation problem'. Using this tool, we strove to get a single document that:

- explains the original protocol;
- not only contains the 'final' Promela model, but also contains important parts of "earlier" models;
- explains the modelling decisions and abstractions that lead to this 'final' Promela model;
- contains important results of validation runs with Spin;
- contains historical notes on the validation trajectory.

In earlier validation efforts, the information above was spread over many Promela files, several output files of Spin, and entries in a log book.

`noweb` [15, 16] developed by Norman Ramsey is a literate programming tool like Knuth's `WEB`, only simpler. Unlike `WEB`, `noweb` is independent of the programming language to be literated. For example, `noweb` is being used with Pascal, C, C++, Modula-2, Icon, ML, Promela and others.

A `noweb` file contains program source code interleaved with documentation. A `noweb` file is a sequence of chunks, which may contain code or documentation. A documentation chunk starts with the symbol '`@`' followed by a space or newline. Documentation chunks have no name. Code chunks begin with:

⟨*chunk name*⟩≡

on a line by itself. Documentation chunks contain text fragments in LaTeX, plain TeX or HTML. In our work, we used LaTeX.

In this section, we present a very small part of a typical Promela model. The top level code chunk ⟨*⟩ of a model, could be defined as follows:

11.1 ⟨* 11.1⟩≡
⟨*prelude* 11.2⟩
⟨*proctypes* 7⟩
⟨*init* (never defined)⟩

where the ⟨*prelude*⟩ contains the constants, the globals, etc. of the Promela model, ⟨*proctypes*⟩ contains all Promela `proctypes` and ⟨*init*⟩ contains the Promela `init` process.

In the left margin, `noweb` identifies the code chunk with a tag: `page.n`, where `n` indicates the `n`-th chunk on page `page`. This tag is also added as a suffix to the code chunks used in a definition of a chunk. The tag for ⟨*init*⟩ is (never defined), because it is not defined in this paper.
The ⟨*prelude*⟩ could be defined as follows:

11.2 ⟨*prelude* 11.2⟩≡ (11.1)
⟨*constants* 11.5⟩
⟨*channels* 12.1⟩
⟨*macros* 11.3⟩

In the right margin of the definition of a chunk, between `(..)` braces, the tag of the code chunk is listed, which uses this particular code chunk. All code chunks with the same name will be collected by `noweb` and put in the place where the chunk is used. To conclude this example, we define some more chunks.

11.3 ⟨*macros* 11.3⟩≡ (11.2) 11.4 ▷

```
#define Send(m)                        pchan[dest]!m
```

11.4 ⟨*macros* 11.3⟩+≡ (11.2) ◁ 11.3 12.2 ▷

```
#define Receive_DT(sn,ar,eom,nb)       pchan[i]?DT(sn,ar,eom,nb)
```

11.5 ⟨*constants* 11.5⟩≡ (11.2)

```
#define N     2
#define CHL   2
```

12.1 ⟨*channels* 12.1⟩≡ (11.2)

```
chan pchan[N] = [CHL] of {tpdu} ;
```

12.2 ⟨*macros* 11.3⟩+≡ (11.2) ◁11.4

```
#define AckTimerExpired          premature_timeout || timeout
#define DelayTimerExpired        timeout
```

Because we have defined three ⟨*macros*⟩ chunks, noweb now provides some more information in the right margin; it indicates the previous chunk (◁) and the next chunk (▷) in the list of ⟨*macros*⟩ chunks.

Results of noweb

Compared with our first validation Promela models, the documentation on the last Promela models has grown substantially. The 'literate model' is now in a state that someone unfamiliar with the particular MCNet protocol should be able to understand the working of the protocol from the literate model alone. Important aspects of the protocol are now described separately, without taking into account the complete global Promela model. Only the code chunks needed to explain a particular aspect have to be defined; noweb will put the code chunks in the right place. Furthermore, various erroneous modelling decisions are also incorporated into the literate model to give insight in the modelling decisions. Unfortunately, we did not succeed in incorporating output results from Spin (e.g. message sequence charts) into our literate model.

To describe a complex concurrent system, the combination noweb and Promela turned out to be a good choice. The formal definition of the system is specified in Promela (and checked with Spin), while the system itself and its design considerations are described in LaTeX. On the negative side, there is the problem of how to keep the noweb file up to date. When analyzing and modelling the system, one is tempted to change the Promela file directly, instead of making the changes in the noweb file, which needs another compilation step to get the new Promela file. Furthermore, one would like to include verification results, message sequence charts in the document to illustrate certain scenarios.

5 Conclusions

The combination Promela/Spin was successful in finding errors in the MCNet protocols. In a first verification effort on preliminary versions of the MCNet (obvious) errors were found. Our recommendations were adopted by Bosch to construct more reliable protocol descriptions. In the latest validation effort, the protocols proved more reliable and errors were more difficult to find. The lack of formal service descriptions of the protocols became an important issue: what properties have to be verified?

Modelling and validation of the design should be started during the initial design phase. The fact that the validation team was not directly involved in the design of the system, had several drawbacks:

- Although the documentation on the MCNet protocol is quite extensive, it was not sufficient to understand easily what the protocol was supposed to be doing. This was especially true for the Network Management Protocol. This meant that the modelling phase took quite some time.
- Furthermore, as the design of MCNet proceeded, several documents were written. It was also time consuming (and errorprone) to compare subsequent descriptions of the protocol.

The logging and documenting of the modelling, simulation and validation activities in Spin is very important during the validation trajectory. As long as there is no support for version- and scenario control in Spin, one has to adopt a rigorous engineering discipline. The easiest way is to log all activities and validation results in a logbook. A more stimulating approach is to use a literate programming tool like noweb.

Acknowledgements We would like to thank the Bosch team: Dietmar Elke, Uwe Zurmühl and Hans Pasch, for giving us the opportunity to work on this real-world application and for their endless patience. We would like to thank Ed Brinksma for managing the validation projects in Twente.

References

[1] ISO-DIS 11519-1. Road vehicles – Low speed serial communication, part 1: General and definitions. Technical Report 11519-1, International Organization for Standarization (ISO), June 1994.

[2] ISO-DIS 11519-2. Road vehicles – Low speed serial communication, part 2: Low speed Controller Area Network (CAN). Technical Report 11519-2, International Organization for Standarization (ISO), June 1994.

[3] ISO-DIS 11898. Road vehicles – Interchange of digital information – Controller Area Network (CAN) for high speed communications. Technical Report 11898, International Organization for Standarization (ISO), November 1993.

[4] ISO/IEC 7498-1. Information Processing Systems – Open Systems Interconnection – Basic Reference Model. International Standard 7498-1, International Organization for Standarization (ISO), Geneva, 1994.

[5] Blaupunkt Bosch Gruppe, Robert Bosch GmbH. *MCNet OSEK Transport Protocol (including Block Transfer)*, July 1996. K7/EFK2, `MCN_TP08.DOC` (internal document).

[6] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

[7] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[8] CAN in Automation (CiA). CAN: A Serial Bus System – Not Just for Vehicles. Internet: `http://www.can-cia.de/introduction.html`, February 1997.

[9] Donald E. Knuth. *Literate Programming*. Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information (CSLI), Stanford University, California, 1992.

[10] Donald E. Knuth. *The Stanford Graphbase: A Platform for Combinatorial Computing*. ACM Press, New York, 1993.

[11] Donald E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 1994.

[12] Rom Langerak. Validation of MCA Transfer Layer Protocol. Technical report, CTIT/University of Twente, Department of Computer Science, Enschede, The Netherlands, June 1995. Validation carried out for Robert Bosch GmbH (confidential report).

[13] Rom Langerak. Validation of MCA (v0.7) Transfer Layer and Network Management Protocols. Technical report, CTIT/University of Twente, Department of Computer Science, Enschede, The Netherlands, April 1996. Validation carried out for Robert Bosch GmbH (confidential report).

[14] H.-L. Pasch and D. Rode. Mobile Communication Architecture – Highlights. Technical report, Blaupunkt Bosch Gruppe, Robert Bosch GmbH, November 1995. K7/EFK2 Pasch, K7/EAM3 Rode, `MCA_HLEN.DOC` (internal document).

[15] Norman Ramsey. Literate Programming Simplified. *IEEE Software*, 11(5):97–105, September 1994.

[16] Norman Ramsey. Noweb. Internet: `http://www.cs.virginia.edu/~nr/noweb/`, February 1997.

[17] Robert Bosch GmbH, Mobile Communication Division. *Mobile Communication Architecture*, version 0.4 edition, February 1995.

[18] Robert Bosch GmbH, Mobile Communication Division. *Mobile Communication Architecture*, version 0.7 edition, December 1995.

[19] Robert Bosch GmbH, Mobile Communication Division. *Mobile Communication Architecture MCNet Layer Definitions, Services and Protocols*, version 1.0 edition, August 1996.

[20] Theo C. Ruys. Validation of the (Block) Transfer Protocol and the Network Management Protocol of MCNet v1.1. Technical report, CTIT/University of Twente, Department of Computer Science, Enschede, The Netherlands, April 1997. Validation carried out for Robert Bosch GmbH (confidential report).

[21] Michael J. Schofield. An Introduction to CAN - The Serial Data Communications Bus. Internet: `http://www.omegas.co.uk/CAN`, February 1997.