SICS/T-90/9006

Specification and Validation of a Simple Overtaking Protocol using LOTOS

by

Patrik Ernberg, Lars-åke Fredlund
and Bengt Jonsson

SICS technical report
T90006
ISSN 1100-3154

Specification and Validation of a Simple Overtaking Protocol using LOTOS*

Patrik Ernberg
Lars-åke Fredlund
Bengt Jonsson†
Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, Sweden

October 18, 1990

Abstract

We present a specification of a simple Overtaking protocol for vehicles using the Formal Description Technique LOTOS. A detailed description of the design process leading to this specification is given. The design process involves early use of simulation and validation tools available for LOTOS. We discuss the applicability of existing tools in the context of this example.

1 Introduction

We present a specification of a simple overtaking protocol for vehicles using the standardized formal description technique LOTOS. Emphasis is placed on describing the actual design process leading to the protocol specification. The purpose of this exercise is to evaluate the applicability of LOTOS for the early design stages in the construction of a communication protocol, and to illustrate an interactive design process involving extensive use of available validation tools.

Previous work performed by SICS and Swedish Telecom Radio within the PROMETHEUS[1] project has mainly made use of specification and verification methods based on labelled transition systems (LTS)[HJOP89a, ELN+89, LNEF90]. Some advantages of LTSs are that they are easily understood and that a substantial theoretical framework to analyze their correctness exists. A number of process algebras such as CCS [Mil89] and CSP [Hoa85] which allow operations to be performed on LTSs have also been developed. Furthermore, tools exist for automatically analyzing LTSs and process algebras [CPS89, Fer89, SV89, VdS89, JKZ88].

*This work has been partially funded by the ESPRIT BRA project CONCUR and the EUREKA-PROMETHEUS project.

†Authors' email: pernberg@sics.se; bengt@sics.se; fred@sics.se

[1]PROMETHEUS is a European project which ultimately aims at improving traffic safety and reducing traffic pollution.

However, experience has shown that labelled transition systems as well as some of the process algebras lack certain structural constructs which are desirable when specifying medium to large sized software systems [BEH89, HBE90, Ern90]. Desirable constructs are, for example, data types, process instantiation, and certain other operators which improve the structure and legibility of a specification.

One FDT which caters for some of the deficiencies of simple process algebras and LTSs is LOTOS [BB89]. LOTOS has been successfully used to specify and validate large protocols [EVD89] and is an accepted ISO standard [ISO87]. This contributes to a wider recognition and use of the language as well as providing a basis for extensive tool development. Tools for validating reasonable size LOTOS specifications are starting to emerge [Eij89, QPF89, Tre89], and the Caesar tool [Gar89] can translate LOTOS specifications into LTSs.

Much work in the literature seems to address the issue of specifying already existing protocols using FDTs, whereas very little work has, to our knowledge, been published on specifying and verifying protocols in the early stages of the design process. Also, many papers describing specifications give the impression that validation is either completely omitted or performed when the protocol has been fully specified.

The design of the Overtaking protocol was inspired from [PRO89] where desirable PROMETHEUS functions such as safe overtaking are informally specified. Previous papers [ELN+89, LNEF90, HJOP89a, HJOP89b] have presented variations of this protocol using communicating LTSs. The functionality of our specification is approximately the same as that presented in earlier papers and it is still far from being a realistic protocol. We are limited by the automatic validation tools, which have difficulty handling protocols larger than the one presented here. However, LOTOS provides us with the structural constructs to scale up a specification, and we are confident that this scaling can be performed on a specification which is correct.

In Section 2 we present the overtaking protocol and specify it formally in LOTOS. We will assume that the reader has basic familiarity with LOTOS and refer to [BB89] for a concise introduction to the language. The validation of the protocol specification is presented in Section 3 and a discussion of our results and conclusions are presented in Sections 4 and 5.

2 Specification of the Overtaking Protocol

In this Section we will first describe the overtaking protocol informally and list some basic assumptions that we made regarding its functionality. We then formalize this informal description in LOTOS.

2.1 Informal Description of the Overtaking Protocol

The overtaking protocol assumes the existence of a queue of cars following each other. To be more specific, we will in the following assume that the queue consists of three vehicles: the Tail vehicle succeeds the Middle vehicle, which in turn succeeds the Head vehicle of the queue (see Figure 1). Each vehicle in the queue can communicate with adjacent vehicles through unique radio channels.

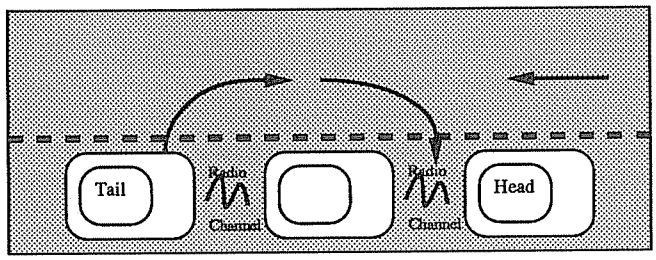

Figure 1: The Overtaking Scenario

When a driver wishes to overtake, the protocol entity in its vehicle will initiate a negotiation with the preceding vehicle, i.e. the adjacent vehicle closer to the head of the queue. Henceforth, the vehicle requesting permission to overtake will be referred to as a *client* vehicle whereas the requested vehicle will be referred to as a *server* vehicle. The *server* vehicle decides if overtaking is possible, in which case a positive response is sent to the *client* vehicle. If overtaking is not possible, the *server* waits for the next overtaking request from the client.

To keep our specification of the overtaking protocol relatively simple, we have made certain assumptions regarding the environment as well as the involved vehicles and communication channels. We will mention a few general assumptions here and point out others as we explain the protocol specification:

- We only consider a fixed and finite number of vehicles in the queue, i.e. we do not allow vehicles to join or leave the queue. This also means that we do not allow the *head* vehicle to overtake or the *tail* vehicle to be overtaken.

- The channels used to negotiate the overtaking are simplex mediums which are presumed to be unreliable, i.e. they may lose messages nondeterministically without warning. The contents of a message cannot be altered by the communication medium. Either a message is received unchanged or it is lost completely.

- At the actual overtaking moment, communication is assumed to be reliable. The intuition behind this is that the vehicles have equipment for sending and receiving very short-range messages safely (not necessarily by radio). At the overtaking moment, the two vehicles involved in the overtaking will be sufficiently close to each other to allow the use of this equipment. The established temporary channel (represented by the overtaking medium) can then be viewed as a perfect point-to-point communication medium between the two vehicles.

- A vehicle may only overtake its directly preceding vehicle. To pass two vehicles, at least two overtaking protocol negotiations must be performed.

- We assume that the environment will remain "friendly" once a vehicle has received a go ahead signal from the preceding vehicle, i.e. no unexpected events, such as the sudden appearance of a large moose on the road, will occur.

- Once a *client* vehicle has engaged in an overtaking negotiation, it will keep on negotiating until it is given a positive acknowledgment from the *server* vehicle, i.e. it is not possible to abort an overtaking.

2.2 Formal Description of the Overtaking Protocol in LOTOS

Our formal specification consists of `Vehicle` processes which communicate with each other through the `Medium` process, and during an overtaking through the `Overtake_Medium` process. We call primitives that are exchanged between vehicle and medium processes *Protocol Data Units* (PDUs). These will all be prefixed with a "`p_ot_`". Primitives that are exchanged between the vehicle and the overtaking service user[2] are referred to as *Service Primitives* and are prefixed with "`ot_`". A `rcv` or a `snd` value is appended to all PDUs to distinguish the sending of a PDU from the reception of a PDU. The entire specification is presented in Appendix A.

2.2.1 Formal Specification of the Vehicle Process

The `Vehicle` process has three ports for communication purposes. Below we give a brief description of each port.

`S` The interface between the overtaking service user and the vehicle.

`M` The interface between the vehicle and the communication medium which is used to send messages to the succeeding and preceding vehicles in the queue.

`Ot` The interface between the vehicle and the overtaking medium, used only at the moment of overtaking.

Each `Vehicle` has knowledge of the communication medium addresses of the succeeding and the preceding vehicles (`B` and `F`), and the overtaking medium address of the preceding vehicle (`Op`). Additionally, a `Vehicle` has information about its own name (`car`), and whether it is at the head, middle, or tail of the vehicle train (`pos`).

The address of the recipient of a PDU is added before the message is sent over the selected medium. For example, `M!F!snd!pdu` sends `pdu` to the preceding vehicle over the `M` port (via the `Medium` process). During an overtaking, the protocol takes care to swap the information of addresses between the two involved vehicles to reflect the new ordering of vehicles in the queue.

The passing of addresses between the processes involved in an overtaking scenario is illustrated in Figure 2. The scenario consists of a road with two-way traffic, with two vehicles, `Volvo` and `Saab`, travelling from left to right. In the other lane, traffic normally flows from right to left. In each `vehicle` process, the name of the vehicle as well as the current values of the `F`, `B`, and `Op` addresses are illustrated. In the `Volvo` vehicle, for example, the `F` address has the value `FT`, the `B` address has the value `BT`, and the `Op` address has the value `T`. The figure starts in the left hand configuration, and tries to illustrate how the values of the `F`, `B` and `Op` addresses change during an overtaking. The

[2] A user of the overtaking service in a given vehicle may either be the physical driver of the vehicle or another process in the vehicle

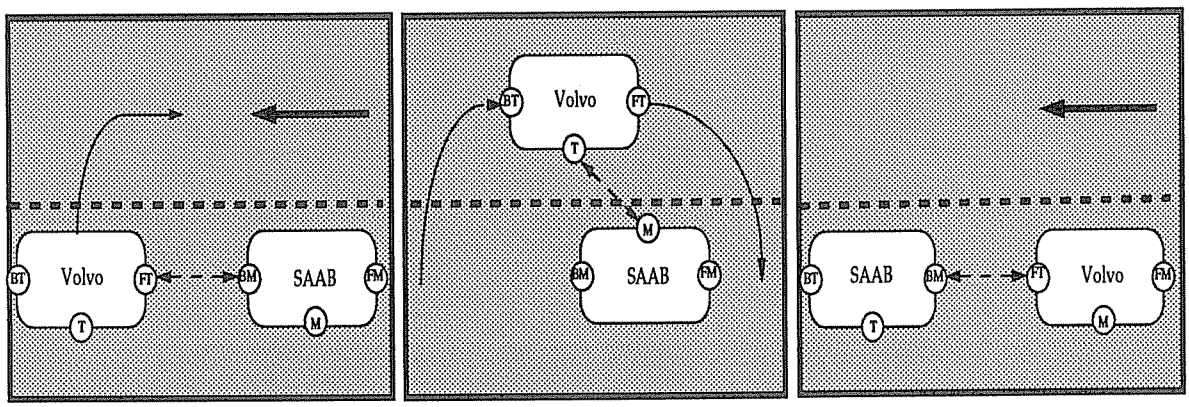

Figure 2: An overtaking situation

connection between a pair of addresses, a communication channel, is implemented by the `Medium` and `Overtake_Medium` processes. In the figures such a connection is illustrated by an arrow between two `vehicles`. The initial situation of the figure is that the `Volvo` vehicle is negotiating an overtaking with the `Saab` vehicle. The overtaking then starts and the two vehicles communicate address values. Finally, when the overtaking has finished, the `Volvo` vehicle is at the head of the queue.

Our specification of a `Vehicle` can be divided into three parts:

1. A description of how an overtaking entity can become either a server or a client in a given overtaking scenario, formalized in the LOTOS process `Vehicle`.

2. A client part, to enable the local service user to overtake safely, specified in the LOTOS process `Client`.

3. A server part, to serve the overtaking request initiated by another overtaking entity, formalized in the LOTOS process `Server`.

```
process Vehicle[S,M,OT](F:M_Port,B:M_Port,Op:Ot_Port,pos:Position,car:CarId):noexit :=
          S!ot_req!car!pos[pos <> Head]; Client[S,M,OT](F,B,Op,pos,car)
        [] M!B!rcv!p_ot_req[pos <> Tail]; Server[S,M,OT](F,B,Op,pos,car)
endproc
```

If the `Vehicle` process first accepts an overtaking request `ot_req` from the service user, which is not possible at the head of the queue (i.e. `[pos <> head]`), the vehicle becomes a client. If however a `p_ot_req` PDU is received from a succeeding vehicle the vehicle becomes a server.

The client process starts by sending a `p_ot_req` to the preceding vehicle announcing its intention to overtake, and awaits a response. A timer may then either time out (`client_timer`), in which case the `Client` process is restarted (which will resend `p_ot_req`), or a positive acknowledgment is received (`p_ot_conf_ok`) from the preceding car, thus initiating the actual overtaking procedure. If a positive acknowledgment arrives too late from the preceding car, i.e. the `client_timer` has already timed out and the client wants to send a new overtaking request `p_ot_req`, the system may deadlock since the medium only supports simplex communication (there is a conflict between the sending

of p_ot_req and the reception of p_ot_conf_ok). This problem is resolved by accepting "old" acknowledgments in the Client, but ignoring them since they are out-of-date.

The client starts the actual overtaking procedure by informing the service user that overtaking is about to begin (ot_begin). The client then sends its own position, its own overtaking medium address, and the communication medium address of its successor to the server via the overtaking medium. When the client receives the servers's position and the addresses to the vehicle preceding the server, the client first informs the service user that the overtaking has been successfully completed (ot_end), and then restarts the Vehicle process with updated addresses and position.

```
process Client [S,M,OT](F:M_Port,B:M_Port,Op:Ot_Port,pos:Position,car:CarId):noexit :=
        hide client_timer in
           M!F!snd!p_ot_req;
                (   (* time out *)
                    client_timer!car; Client[S,M,OT](F,B,Op,pos,car)
                 [] (* or positive acknowledgment *)
                    M!F!rcv!p_ot_conf_ok;
                        S!ot_begin!car!pos;
                        OT!Op!B!Op!pos;
                        OT!Op?s_F:M_Port?s_Op:Ot_Port?s_pos:Position;
                        S!ot_end!car!pos;
                        Vehicle[S,M,OT](s_F,F,s_Op,s_pos,car))

        (* We ignore "old" confirmation messages *)
        [] M!F!rcv!p_ot_conf_ok; Client[S,M,OT](F,B,Op,pos,car)
endproc
```

The server part is activated when a p_ot_req PDU is received from a succeeding vehicle. This will result in an ot_ind service primitive being sent to the overtaking service user. We assume that the overtaking service user has the intelligence to decide if it is reasonable to overtake and appropriate responses are given to the server. If the response is negative, the server will await another p_ot_req from the succeeding vehicle. Otherwise, if the response is positive, this will be forwarded to the succeeding vehicle by means of a p_ot_conf_ok, and the server will be ready for the actual overtaking. In order to avoid deadlocks in the system, the Server_answer and Server_ok processes have been modified to handle the case when the client times out and resends the overtaking request p_ot_req (either due to slow processing in the server, or message loss in the communication medium).

```
process Server[S,M,OT](F:M_Port,B:M_Port,Op:Ot_Port,pos:Position,car:CarId):noexit :=
          S!ot_ind!car!pos;
             (   S!ot_resp_no!car!pos;
                    M!B!rcv!p_ot_req; Server[S,M,OT](F,B,Op,pos,car)
              [] S!ot_resp_ok!car!pos; Server_answer[S,M,OT](F,B,Op,pos,car))
endproc

process Server_answer[S,M,OT](F:M_Port,B:M_Port,Op:Ot_Port,pos:Position,car:CarId):noexit :=
          M!B!snd!p_ot_conf_ok; Server_ok[S,M,OT](F,B,Op,pos,car)
```

```
        [] M!B!rcv!p_ot_req; Server[S,M,OT](F,B,Op,pos,car)
endproc

process Server_ok[S,M,OT](F:M_Port,B:M_Port,Op:Ot_Port,pos:Position,car:CarId):noexit :=
          OT!Op?c_B:M_Port!F?c_Op:Ot_Port!Op?c_pos:Position!pos;
                Vehicle[S,M,OT](B,c_B,c_Op,c_pos,car)
        [] M!B!rcv!p_ot_req; Server[S,M,OT](F,B,Op,pos,car)
endproc
```

The server part of the overtaking synchronization is activated upon the reception of the client's position and the addresses to the client's succeeding vehicle. In the same communication, the server transmits its own position, and the addresses of its preceding vehicle. After this synchronization (over the overtaking medium), the `Vehicle` process is restarted with addresses and position parameters updated to reflect the result of the overtaking.

2.2.2 Formal Specification of the Communication medium

The communication medium used for negotiating overtakings is modelled as two static channels, each channel connecting two ports. Using actual LOTOS ports for this purpose is not possible because LOTOS ports cannot be communicated in a process synchronization. Instead we modelled these abstract ports using addresses (constant values of a LOTOS type), and a global port `M`. Each channel thus connects a fixed address (which can be communicated in a process synchronization) to another fixed address. As previously mentioned, the communication channels are lossy and simplex.

```
process Medium[M]:noexit :=
        M_Channel[M](F_Tail,B_Middle) ||| M_Channel[M](F_Middle,B_Head)
endproc

process M_Channel[M](P1:M_Port,P2:M_Port):noexit :=
        hide medium_loss in
           M!P1!snd?pdu:PDUsort;
                (   M!P2!rcv!pdu;  M_Channel[M](P1,P2)
                 [] medium_loss; M_Channel[M](P1,P2))
        [] M!P2!snd?pdu:PDUsort;
                (   M!P1!rcv!pdu;  M_Channel[M](P1,P2)
                 [] medium_loss; M_Channel[M](P1,P2))
endproc
```

2.2.3 Formal Specification of the Overtaking medium

The overtaking medium is similar to the communication medium, except that the medium is perfect and one-way. Thus only one address per vehicle is needed: if the vehicle is a client in the current overtaking scenario, it communicates only with the preceding vehicle; if it is acting as a server, it communicates only with the succeeding one.

```
process Overtake_Medium[OT]:noexit :=
        Ot_Channel[OT](ot_Tail, ot_Middle)
```

```
                |||
        Ot_Channel[OT](ot_Middle, ot_Head)
endproc


process Ot_Channel[OT](P1:Ot_Port,P2:Ot_Port):noexit :=
        OT!P1?c_B:M_Port?c_Op:Ot_Port?c_pos:Position;
        OT!P2!c_B?s_F:M_Port!c_Op?s_Op:Ot_Port!c_pos?s_pos:Position;
        OT!P1!s_F!s_Op!s_pos;
        Ot_Channel[OT](P1,P2)
endproc
```

3 Validation of the Overtaking Protocol

In this Section, we present the different tools and methods used when validating the Overtaking protocol. It should be emphasized that the process of proving the protocol correct was closely intertwined with the specification work. In order to make the verification methods work well, care had to be taken to write the specification in a verifiable way (this remark is especially valid given the rather severe constraints todays tools place on verifiable specifications).

In order to make efficient validation possible, we restricted our queue to a length of three vehicles. The vehicles are called `Volvo`, `Saab`, and `BMW`, and are initially in this order with `BMW` at the head of the queue. Below we present a skeleton specification of the Overtaking protocol for three vehicles:

```
specification overtaking [S]:noexit

(* ======================================== *)
(* ... Abstract data type definitions ... *)
(* ======================================== *)

(* Behavioural specification *)

behaviour
        hide M, OT in
        (
        Vehicle[S,M,OT](F_Tail, B_Tail, ot_Tail, Tail, Volvo)
                |||
        Vehicle[S,M,OT](F_Middle, B_Middle, ot_Middle, Middle, Saab)
                |||
        Vehicle[S,M,OT](F_Head, B_Head, ot_Head, Head, BMW)
        )
                |[M,OT]|
        (
        Medium[M]
                |||
```

```
        Overtake_Medium[OT]
        )

where

(* ======================================== *)
(*     ... Rest of Specification ...        *)
(* ======================================== *)
```

Parameters prefixed with `F_` and `B_` are communication medium addresses to the preceding and succeeding vehicles, while those prefixed with `ot_` are overtaking medium addresses to the preceding vehicle. Notice that each vehicle also has a name and a current position.

Below, the different tools and methods will be described in the order in which they were applied.

3.1 Simulation

When our specification had become reasonably complete, we simulated it using the Hippo[Tre89] tool. The simulator essentially performs a step by step expansion of the protocol, allowing the user to specify which nondeterministic steps should be taken. We stepped through a number of key scenarios that were central to the workings of the protocol. We checked, for instance, that the tail vehicle could overtake the middle vehicle, and that the new tail vehicle could then overtake the new middle vehicle. Several errors in the protocol were discovered by testing such selected scenarios. The errors were fixed and the simulation repeated until no more errors were found.

3.2 Expansion and Minimization

Once we were fairly sure that the LOTOS specification was correct, we translated it into a labelled transition system using the Caesar[Gar89] tool. This produced a transition graph with 89879 states and 286716 transitions. The transition system was then minimized with respect to branching bisimulation semantics using an implementation (BB) of the algorithm presented in [GV90] to produce a graph with 156 states and 318 transitions. The reason for using branching bisimulation was that BB was the only tool which could efficiently handle such a large state space on a SPARCstation-1 with 16 Megabytes of memory. It also turned out that minimizing the agent produced by BB with respect to observation equivalence using the Aldébaran[Fer89] tool produced no further identifications.

3.3 Projection

As a first preliminary test of correctness we verified that there were no deadlocks in the specification using the Concurrency Workbench[CPS88, CPS89].

It is difficult to provide a global service specification for a specification with approximately 150 states. A more realistic method of validating the protocol is therefore to

choose suitable *projections*. This involves looking at only a subset of all primitives which are visible to the environment. Technically, this can either be done by hiding additional ports explicitly in the LOTOS specification using the `hide` operator, or by renaming unwanted actions to the internal action (`i`) using a tool working on LTSs, and then minimizing the result using Aldébaran. The latter method is often preferred as it can be done on an already minimized automaton and specific actions at a certain port may also be hidden.

The projections were displayed graphically using the Auto and Autograph[SV89, VdS89] tools, resulting in an increased understanding of how and why the protocol works. As an example of a picture created by Autograph, the projection of the actions of the Volvo vehicle for the three vehicle scenario can be found in Figure 3. `St_0` represents the

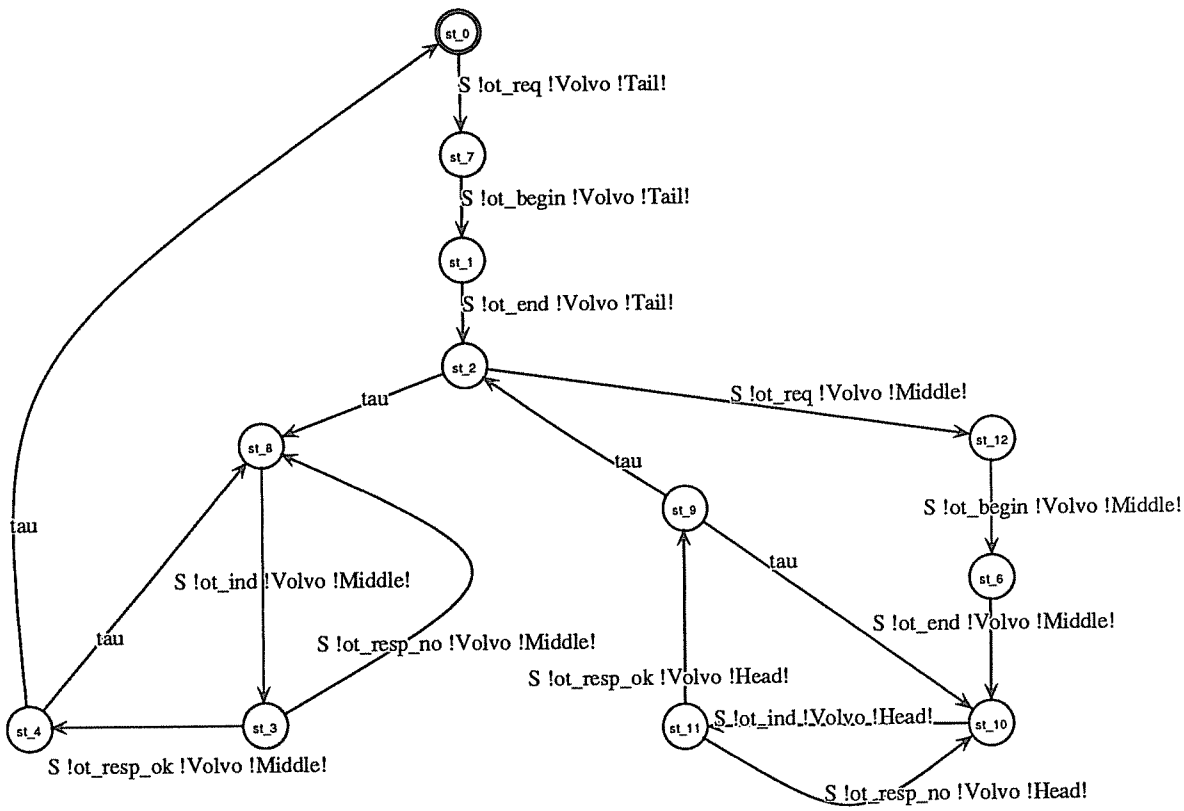

Figure 3: The graphical representation of the actions of the Volvo vehicle.

initial state, with the Volvo at the tail of the queue. `St_2` represents the state when the Volvo has just overtaken to reach the middle position in the queue and `st_10` represents the state where the Volvo has reached the head of the queue. Observe that the `tau` transitions are analogous to the internal `i` action in LOTOS.

3.4 Model Checking

We additionally validate the protocol using model checking. This amounts to proving that a given specification satisfies a set of properties expressed in a modal logic. The advantages of the method are mainly

- Computational efficiency
- Abstract service specification, i.e. certain temporal properties of a protocol can be concisely described using modal logic.

In the following, we will make use of Hennessy-Milner Logic (HML). We will here introduce HML, see [CPS89, CPS88] for in depth descriptions of the logic. More general discussions about temporal logics for communicating systems can be found in [Sti87] and [Sti89b].

3.4.1 Hennessy-Milner Logic

HML is a propositional modal logic with relativized modal operators. The set of HML formulae is defined as follows:

- $\mathtt{tt}$ is a HML formula
- if P and Q are HML formulae then so are $\neg P$, $P \vee Q$, and $< e > P$ where e is an event.

The truth of a proposition in HML is defined relative to a transition system. We use a satisfaction relation, denoted $\models$, to define when a proposition is true in a state. $S \models P$ should be read: "the proposition P is true in the state S". The semantics of the HML formulae is defined as follows:

- $S \models \mathtt{tt}$ for all states S. The proposition $\mathtt{tt}$ is the most primitive proposition and is satisfied by all states.
- $S \models \neg P$ if and only if not $S \models P$.
- $S \models P \vee Q$ if and only if $S \models P$ or $S \models Q$.
- $S \models < e > P$ if and only if there exits a transition labelled with event e from state S to state S' and $S' \models P$.

We also have the derived modal operators $\mathtt{ff} \equiv \neg\mathtt{tt}$, $\mathtt{P} \wedge \mathtt{Q} \equiv \neg(\neg\mathtt{P} \vee \neg\mathtt{Q})$, and $[e]P \equiv \neg < e > \neg P$. We shall also use the two derived operators $[.]P$ and $< . > P$, where "." can be regarded as wild card events. Thus, a state satisfies $[.]P$ if all its derivatives satisfy P, and it satisfies $< . > P$ if it has a derivative that satisfies P.

In order to further increase the expressive power of the language we introduce maximal and minimal fixpoint operators. The maximal fixpoint operator, $\nu X.(P)$, can be interpreted as the infinite conjunction $P_0 \wedge P_1 \wedge, ...,$ where $P_0 = \mathtt{tt}$ and $P_{i+1} = P[P_i/X]$. This operator is useful for expressing invariance properties. The minimal fixed point operator $\mu X.(P)$ is the dual of the maximal one and can be interpreted as the infinite disjunction $P_0 \vee P_1 \vee, ...,$ where $P_0 = \mathtt{ff}$ and $P_{i+1} = P[P_i/X]$. This constructor is useful for expressing eventuality properties.

3.4.2 Validating the Overtaking protocol using HML

The general method of validation involves taking the minimized LTS produced by Caesar and feeding it into the Concurrency Workbench. It is then possible to verify that the LTS satisfies certain HML formulae. Most interesting properties described in HML make use of fixed point operators. Unfortunately, these are, at least to a novice, rather unintuitive. The best way of using them is therefore to define some well-understood properties and code these as macros. Below, we define some macros which have proved useful when specifying desirable behaviours:

$$\begin{aligned} AG\ P &\equiv \nu X.(P \wedge [.]X) \\ EF\ P &\equiv \mu X.(P \vee <.>X) \\ A(P\ U\ Q) &\equiv \mu X.(Q \vee (P \wedge [.]X \wedge <.>\mathtt{tt})) \end{aligned}$$

$AG\ P$ holds in a state s if P holds in every state reachable from s. This macro is used to describe a property which invariantly holds for a labelled transition system. $EF\ P$ holds in a state s if P holds in some future state reachable from s. $EF\ P$ can be used to describe properties which will eventually hold in some future state. $A(P\ U\ Q)$ holds in a state s if Q is guaranteed to hold in some future state and P holds in every state until then. Using the macros defined above, we can go on and specify some interesting properties which the Overtaking protocol should satisfy:

Property 1 *The protocol does not contain any deadlocks.*

In order to specify that no deadlock occurs we merely have to make sure that the initial state can perform an event and that all states reachable from the initial state can perform an event:

$$NoDeadlock = AG\ <.>\mathtt{tt}$$

Property 2 *It is always possible for all vehicles to reach a state where they can initiate an overtaking operation.*

This property ensures that there exist no "sink" states in the protocol from which other states are unreachable. An alternative formulation of the above property could be that there exist no future states where the `S!ot_req` action remains invariantly deadlocked.

$$NoSink = \neg EF\ AG\ [S!ot\_req]\mathtt{ff}$$

Property 3 *Once the tail vehicle has started overtaking, neither the tail vehicle nor the middle vehicle will attempt to start overtaking before the tail vehicle has finished overtaking.*

This property ensures that all overtaking operations performed by the middle vehicle are safe.

$$SafeOver = AG\ [S!ot\_begin!Tail]\ (A(([S!ot\_begin!Tail]\mathtt{ff} \wedge [S!ot\_begin!Middle]\mathtt{ff})\ U\ <S!ot\_end!Tail>\mathtt{tt}))$$

A similar formula could also be written for the case when the middle vehicle engages in an overtaking operation. For the sake of simplicity, both Properties 2 and 3 have been defined using service primitives which do not include any information about vehicle names. Of course, HML formulae which are applied directly to a graph produced by Caesar from the specification in Section 2 would have to take vehicle names into account.

It should also be noted that the list of properties presented here is by no means exhaustive. We have only described a few interesting safety properties to give a flavour for what behaviours can concisely be described in HML. Safety properties are properties that assure that no undesirable events will ever occur. On the other hand, they do not require that anything ever does happen. The agent `nil`, for example, which can not perform any action, satisfies property 3. In fact, it turns out that this protocol does not satisfy certain liveness properties which are expressible in HML. Liveness properties assure that desirable events will eventually happen. A liveness property could for example be that a vehicle requesting an overtake will eventually be allowed to overtake. We will not give any HML formulas describing liveness properties as this would require a more subtle description of the logic which is beyond the scope of this paper (see [Wal89, Sti89a] for examples of liveness properties which can be modelled in HML).

4 Discussion

In this Section we discuss some important issues regarding the specification of the Overtaking protocol in LOTOS. We divide our discussion into three parts. The first part discusses issues related to the LOTOS language. In the second part, we discuss the tool environment and the restrictions that it places on the specification and validation of the Overtaking protocol. Lastly, we consider the actual protocol and possible enhancements which can be made to the specification.

4.1 Comments about LOTOS

We originally planned to specify the Overtaking protocol using only Basic LOTOS (i.e. LOTOS without data types) constructions because the Caesar tool did not fully support LOTOS data types. Furthermore we had previously specified protocols in basic CCS (which lacks value-passing constructions) so we did not believe that the restriction to Basic LOTOS was critical. However, our trial specifications using Basic LOTOS looked ugly. This was mainly due to the fact that all ports used in a process had to be passed to it as actual parameters when it was instantiated. From a specifier's point of view, Basic LOTOS specifications would look much better if the scoping of ports in LOTOS were extended to allow subprocesses to refer to the formal parameters of their process ancestors (lexical scoping).

The following Basic LOTOS expression would then be legal whereas today it isn't:

```
process test[P1,P2]:noexit := sub_test
```

```
where
        process sub_test:noexit := P2; P1 endproc
endproc
```

Due partly to this problem, we decided to write the specification using full LOTOS with data types. In retrospect, this turned out to be a wise decision since:

- It reduced the number of ports and thereby made the specification more readable.
- The value-passing concept made the specification more realistic; in reality most ports can synchronize on more than one value.
- The parameterization of processes (in this case the position and name of the vehicle were parameters in the `vehicle` process) kept the specification concise.

All communication in our protocol has a direction in the sense that there is always a sender and a receiver. LOTOS communication does not distinguish between senders and receivers. We therefore had to append a value to each action indicating if the intention of the action was a send operation or a receive operation. For example, sending to the vehicle in front may be accomplished by the action `M!F!snd!`*pdu*. The communication medium will pass this message along to the vehicle in front (if no message loss occurs) where it can be received using `M!B!rcv?`*pduvar*:*PDUtype*.

There are nevertheless a number of advantages with the LOTOS parallel composition operator. Firstly, the fact that all ports have to be explicitly hidden means that the specifier has a lot of freedom in choosing what actions he wants to observe when debugging his/her specification. This should be compared with the CCS parallel composition operator where all communication between processes is implicitly hidden as an internal τ-event. Secondly, the fact that the parallel composition operator is a broadcast operator means that it is easy to introduce new processes which synchronize on already existing ports. These processes can either be part of the actual specification or they may be "testers" introduced to test certain properties of a specification. Both these advantages seem to make the parallel composition operator in LOTOS more appropriate for interactive design than the corresponding CCS operator.

Another shortcoming of LOTOS is that it is not possible to pass port names between processes. Our specification circumvents this problem by coding ports as abstract data types, and passing these as values between processes. A cleaner specification could probably have been achieved by using a specification formalism such as the π-calculus [MPW89a, MPW89b], where port names can be passed as parameters between processes.

4.2 Comments about the Tool Environment

While specifying and validating our specification we made extensive use of a collection of different tools. The fact that we could move between different tools and analyze our protocol using different methods made work both more interesting and more efficient. Still, this tool interaction could be greatly enhanced if a common format for LTSs could be used by all the tools. In our example, we had to change the transition names produced by Caesar manually when we wanted to analyze the LTS using the Concurrency Workbench. Discussions within the tool development community are under way to decide upon a common format for LTSs which will hopefully make tool interfacing easier.

A related problem was that validation messages produced by the Concurrency Workbench and Auto referred to the LTS generated by Caesar and not to the original LOTOS specification. Validation could be improved if there were some hook from the common format back to the original system description, so that information generated by the tool can be presented to the user at the level he or she understands.

Besides the problem of tool interaction, tools also place restrictions on what can be realistically verified. We were therefore forced to specify our Overtaking protocol in a verifiable way. The most serious restrictions were due to the Caesar verification tool:

- The specification had to be finite-state (we only consider a finite, fixed amount of vehicles), and furthermore, had to satisfy the more restrictive *static control constraints* (see section 2.3 in [Gar89]).
- When writing the specification, care had to be taken to cut down on the size of the state-space which otherwise easily becomes too large for effective analysis. In essence this involves cutting down on the number of parallel and disable operators in the specification. In fact, we did not use any disable operators in our specification.

As specifications become larger, verification becomes more difficult and simulation and testing are often the only possible methods of validation. In contrast to Caesar, Hippo can be conveniently used even for large specifications. Simulation is also an interesting method of validation because of its interactive nature. In our example, simulating the protocol often gave us more understanding of how the protocol worked than using projection or model checking techniques. In our work using the Hippo simulator, we would have liked a feature allowing the generation of random traces of actions from the specification. This form of validation has also proved to be effective in practice (see [Wes86]).

4.3 Comments about the Overtaking Specification

The specification of the Overtaking protocol presented in this paper is a simple and naive one. We have deliberately kept it simple to make it easy for the reader to follow the specification and to allow for verification using tools which can only cope with a very limited state space. Several improvements could be made to the existing protocol:

- The protocol does not allow a driver to abort an overtake operation.
- No attempts have been made to model the rest of the environment in an explicit way. This is of course very unrealistic. A good protocol should take into account vehicles in the opposite lane as well as unexpected events such as obstacles on the road.
- Assuming a fixed queue size is unrealistic. The protocol should cater for attachments to and detachments from the vehicle queue.

Most of the improvements suggested above would not permit the verification techniques presented in Section 3, at least not using the Caesar tool. We would probably have to rely more on validation techniques based on simulation and correctness-preserving transformations (see [Lan90] for an example of a correctness-preserving transformation)

which can be used even for large specifications. However, the structural constructs as well as the abstract data types present in LOTOS should make these protocol enhancements more feasible than if we had used a simple LTS or process algebra.

5 Conclusions and Further Work

We have presented a simple specification of an Overtaking protocol in LOTOS using an interactive design process involving the early application of different validation techniques. The validation was performed using a variety of different tools. We used a simulation tool, Hippo, to step through key scenarios of the protocol and identified a number of errors when doing this. Caesar was then used to compile the LOTOS specification into a labelled transition system (LTS). The resulting LTS was first minimized with respect to branching bisimulation semantics using BB and then with respect to observation equivalence using Aldébaran. Appropriate "projections" were made to further reduce the state space and graphical representations of these projections were produced using the Auto and Autograph tools. Furthermore, model checking using the Concurrency Workbench was applied to verify certain properties of the Overtaking protocol.

Our experience suggests that LOTOS is an appropriate language to use for the early steps in the design of a protocol; the structural constructs available in LOTOS make it possible to produce concise specifications and the parallel composition operator lends itself to efficient interactive validation. The validation techniques, involving a number of different methods and tools, as well as considerable interaction between the designer and specification, also seem applicable in the early design process.

As the specification becomes more complex, verification using the Caesar tool becomes more difficult. Our present specification is close to the limit of what the current version of Caesar can efficiently handle. On the other hand, our specification is quite loosely synchronized leading to a relatively large state space. Other LOTOS specifications which are textually larger but more tightly synchronized can probably be successfully handled in Caesar. Regardless, validation techniques based on simulation tools such as Hippo are useful even for large specifications.

The virtues of LOTOS presented above should simplify steps from an early specification to a more enhanced design. It would be interesting to try to improve the specification in this paper with some of the suggestions presented in Section 4.3 in order to verify this conjecture.

6 Acknowledgements

We would like to thank Joachim Parrow for valuable comments on earlier versions of the paper.

References

[BB89] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P. van Eijk, C. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 77–82. North-Holand, 1989.

[BEH89] L. Beckman, P. Ernberg, and H. Hansson. Utvärdering av en CCS-baserad metod för specifikation och verifiering av protokoll. Result of project performed by SICS in cooperation with ELLEMTEL(in Swedish), 1989.

[CPS88] R. Cleaveland, J. Parrow, and B. Steffen. *The Concurrency Workbench: Operating Instructions*, 1988.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. A semantics-based verification tool for finite-state systems. In *Protocol Specification, Testing, and Verifiation, IX*, 1989.

[Eij89] P. Eijk. Lotos tools based on the cornell synthesizer. In E. Brinksma, G. Scollo, and C. Vissers, editors, *Proceedings of IFIP IX*, 1989. (to be published).

[ELN+89] P. Ernberg, K. Laraqui, A. Nazari, C. Odmalm, B. Pehrson, and M. Svärdh. A PROMETHEUS - PROCOM framework, a specification model for PROMETHEUS functions and communication services. In *PROMETHEUS proceeedings of the 2nd Workshop, Stockholm.* SICS and Swedish Telecom Radio, 1989.

[Ern90] P. Ernberg. CCS as a method of specification and verification: Analysis of a case study. Technical report, SICS, 1990. (draft).

[EVD89] P. van Eijk, C. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS.* North-Holand, 1989.

[Fer89] J-C. Fernandez. Aldébaran: A tool for verification of communicating processes. Technical Report RTC 14, IMAG, Grenoble, 1989.

[Gar89] H. Garavel. *Caesar 3.2 Reference Manual.* IMAG - LGI, Grenoble, France, 1989.

[GV90] J. Groote and F Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. Report CS-R9001, Centre for Mathematics and Computer Science, Amsterdam, 1990.

[HBE90] T. Hovander, L. Beckman, and P. Ernberg. Statusrapport protokollprojektet. Result of project performed by SICS in cooperation with ELLEMTEL (in Swedish), 1990.

[HJOP89a] H. Hansson, B. Jonsson, F. Orava, and B. Pehrson. Guidelines for the specification and verification of services and protocols. In *PROMETHEUS proceedings of the 1st workshop, Wolfsburg.* SICS, 1989.

[HJOP89b] H. Hansson, B. Jonsson, F. Orava, and B. Pehrson. Specification for verification. In *FORTE*, 1989.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[ISO87] ISO Information Processing Systems - Open Systems Interconnection. LOTOS - a formal description technique based on the temporal ordering of observational behaviour. DIS 8807, 1987.

[JKZ88] J.Godskesen, K.Larsen, and M. Zeeberg. *TAV (Tools for Automatic Verification) Users Manual.* Aalborg University Center, Aalborg, Denmark, 1988.

[Lan90] R. Langerak. Decomposition of functionality: a correctness-preserving LOTOS transformation. In L. Logrippo, R. Probert, and H. Ural, editors, *Tenth International Symposium on Protocol Specification, Testing, and Verification*, 1990. (to be published).

[LNEF90] K. Laraqui, A. Nazari, P. Ernberg, and L. Fredlund. Communication systems architecture - a case study. In *PROMETHEUS Proceedings of the 3rd Workshop.* Swedish Telecom Radio and SICS, 1990.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[MPW89a] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part 1. LFCS Report Series ECS-LFCS-89-85, LFCS, Department of Computer Science, Edinburgh, 1989.

[MPW89b] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part 2. LFCS Report Series ECS-LFCS-89-86, LFCS, Department of Computer Science, Edinburgh, 1989.

[PRO89] PROMETHEUS. Functions or how to achieve PROMETHEUS objectives, 1989.

[QPF89] J. Quemada, S. Pavon, and A. Fernandez. State exploration by transformation with LOLA. In *Workshop on Automatic Verification Methods for Finite State Systems*, 1989.

[Sti87] C. Stirling. Modal logics for communicating systems. *Theoretical Computer Science*, (49):311–347, 1987.

[Sti89a] C. Stirling. An introduction to modal and temporal logics for ccs. In *Proceedings of joint UK/Japan Workshop on Concurrency, Oxford.* LNCS, 1989.

[Sti89b] C. Stirling. Temporal logics for CCS. In *Lecture Notes in Computer Science*, volume 354, 1989.

[SV89] R. de Simone and D. Vergamini. *Aboard Auto.* INRIA, Sophia-Antipolis, 1989.

[Tre89] J. Tretmans. Hippo: A lotos simulator. In P. Eijk, C. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 391–396. North-Holland, 1989.

[VdS89] V.Roy and R. de Simone. *An Autograph Primer.* INRIA, Sophia-Antipolis, 1989.

[Wal89] D.J. Walker. Automated analysis of mutual exclusion algorithms using CCS. Technical Report ECS-LFCS-89-91, Universisy of Edinburgh, Dept. of Computer Science, 1989.

[Wes86] C. West. Protocol validation by random state exploration. In *Protocol Specification, Testing and Verification, VI.* North-Holland, 1986.

A The Entire LOTOS Overtaking Specification

This specification can be directly fed into the Hippo simulator. In order for the Caesar tool to be able to compile the specification into an LTS, appropriate C files have to be defined for all the specified datatypes.

```
(* A LOTOS specification of the famous Overtaking protocol *)

specification overtaking [S]:noexit

(* ========================================================== *)
(* Type definitions *)

library BOOLEAN, NaturalNumber endlib

type CarId is sorts CarId

opns
        volvo, saab, bmw :-> CarId
endtype

(* Position type *)
type Position is Boolean, NaturalNumber sorts Position


opns
        Head  :  -> Position
        Middle  :  -> Position
        Tail  :  -> Position
        _<>_  :  Position, Position -> Bool
        ord :  Position -> Nat
eqns
        forall x, y, z  :  Position
        ofsort Nat
                ord(Tail)   = 0;
                ord(Middle) = succ(0);
                ord(Head)   = succ(succ(0));

        ofsort Bool
                x <> y = ord(x) ne ord(y);

endtype

(* Type to specify ditection of PDUs being sent      *)
type SENDdirection is sorts SENDdirection

opns
```

```
        snd, rcv :-> SENDdirection
endtype

(* Service Primitives *)
type SAPsort is sorts SAPsort

opns
        ot_req,
        ot_ind,
        ot_begin,
        ot_end,
        ot_resp_no,
        ot_resp_ok,
        ot_conf_ok :-> SAPsort
endtype

(* Protocol Data Units *)
type PDUSORT is sorts PDUsort

opns
        p_ot_req, p_ot_conf_ok :  -> PDUsort
endtype

(* Port type for communication medium *)
type M_Port is Boolean sorts M_Port

opns
        F_Tail,
        F_Middle,
        F_Head,
        B_Tail,
        B_Middle,
        B_Head :-> M_Port
endtype

(* Port type for overtaking communication medium *)
type Ot_Port is sorts Ot_Port

opns
        ot_Tail,
        ot_Middle,
        ot_Head :-> Ot_Port
endtype


(* ========================================================== *)
```

```
(* Behavioural specification *)

behaviour
        hide M, OT in
        (
        Vehicle[S,M,OT](F_Tail, B_Tail, ot_Tail, Tail, Volvo)
                |||
        Vehicle[S,M,OT](F_Middle, B_Middle, ot_Middle, Middle, Saab)
                |||
        Vehicle[S,M,OT](F_Head, B_Head, ot_Head, Head, BMW)
        )
                |[M,OT]|
        (
        Medium[M]
                |||
        Overtake_Medium[OT]
        )

where


(* ------------------------------------- *)
(* Lossy Point-to-Point medium *)

process Medium[M]:noexit :=
        M_Channel[M](F_Tail,B_Middle) ||| M_Channel[M](F_Middle,B_Head)
endproc

process M_Channel[M](P1:M_Port,P2:M_Port):noexit :=
        hide medium_loss in
           M!P1!snd?pdu:PDUsort;
                (   M!P2!rcv!pdu;  M_Channel[M](P1,P2)
                 [] medium_loss; M_Channel[M](P1,P2))
        [] M!P2!snd?pdu:PDUsort;
                (   M!P1!rcv!pdu;  M_Channel[M](P1,P2)
                 [] medium_loss; M_Channel[M](P1,P2))
endproc


(* ------------------------------------- *)
(* Perfect Overtaking medium *)

process Overtake_Medium[OT]:noexit :=
        Ot_Channel[OT](ot_Tail, ot_Middle)
                |||
        Ot_Channel[OT](ot_Middle, ot_Head)
endproc
```

```
process Ot_Channel[OT](P1:Ot_Port,P2:Ot_Port):noexit :=
        OT!P1?c_B:M_Port?c_Op:Ot_Port?c_pos:Position;
        OT!P2!c_B?s_F:M_Port!c_Op?s_Op:Ot_Port!c_pos?s_pos:Position;
        OT!P1!s_F!s_Op!s_pos;
        Ot_Channel[OT](P1,P2)
endproc

(* ------------------------------------ *)
(* Vehicle *)

process Vehicle [S,M,OT](F:M_Port,B:M_Port,Op:Ot_Port,pos:Position,car:CarId):noexit :=
          S!ot_req!car!pos[pos <> Head]; Client[S,M,OT](F,B,Op,pos,car)
        [] M!B!rcv!p_ot_req[pos <> Tail]; Server[S,M,OT](F,B,Op,pos,car)
endproc


(* Client *)

process Client [S,M,OT](F:M_Port,B:M_Port,Op:Ot_Port,pos:Position,car:CarId):noexit :=
        hide client_timer in
          M!F!snd!p_ot_req;
               ((* time out *)
                   client_timer!car; (* S!ot_conf_wait!car; *)
                       Client[S,M,OT](F,B,Op,pos,car)
                [] (* or positive acknowledgement *)
                   M!F!rcv!p_ot_conf_ok;
                       (* Overtaking starts! *)
                       S!ot_begin!car!pos;
                       OT!Op!B!Op!pos;
                       OT!Op?s_F:M_Port?s_Op:Ot_Port?s_pos:Position;
                       S!ot_end!car!pos;
                       Vehicle[S,M,OT](s_F,F,s_Op,s_pos,car))

        (* We ignore "old" confirmation messages *)
        [] M!F!rcv!p_ot_conf_ok; Client[S,M,OT](F,B,Op,pos,car)
endproc


(* Server *)

process Server[S,M,OT](F:M_Port,B:M_Port,Op:Ot_Port,pos:Position,car:CarId):noexit :=
          S!ot_ind!car!pos;
             (   S!ot_resp_no!car!pos;
                    M!B!rcv!p_ot_req; Server[S,M,OT](F,B,Op,pos,car)
```

```
            [] S!ot_resp_ok!car!pos; Server_answer[S,M,OT](F,B,Op,pos,car))
endproc

process Server_answer[S,M,OT](F:M_Port,B:M_Port,Op:Ot_Port,pos:Position,car:CarId):noexit :=
          M!B!snd!p_ot_conf_ok; Server_ok[S,M,OT](F,B,Op,pos,car)
        [] M!B!rcv!p_ot_req; Server[S,M,OT](F,B,Op,pos,car)
endproc

process Server_ok[S,M,OT](F:M_Port,B:M_Port,Op:Ot_Port,pos:Position,car:CarId):noexit :=
          OT!Op?c_B:M_Port!F?c_Op:Ot_Port!Op?c_pos:Position!pos;
                Vehicle[S,M,OT](B,c_B,c_Op,c_pos,car)
        [] M!B!rcv!p_ot_req; Server[S,M,OT](F,B,Op,pos,car)
endproc

endspec
```