Correct Design of PROMETHEUS Control Strategies

K. Laraqui, F. Reichert - Telia Research / Sfp - S-136 80 Haninge - Sweden
G. Holzmann - AT&T - 600 Mountain Av. - Murray Hill, NJ 07974 - USA

Abstract

A system for PROMETHEUS functions will consist of multiple, parallel control processes which are distributed over several local processors as well as possibly several cars. The complexity of interactions is out of human abilities for proving correctness properties like absence of deadlocks, infinite loops and reception of unspecified messages. Correctness proof of control strategies by automated tools is a main research area in computer science.

In this paper, a case study for cooperative overtaking is undertaken to demonstrate such an automated correctness proof for control strategies. The purpose of this article is not to show how a control strategy for overtaking might look like but to rise awareness of the danger of realization of untested strategies. It is obvious that failures of the PROMTHEUS system will cause the loss of safety and the belief of consumers in the system. In addition each detected error would cause a huge call back and replace action of the system suppliers.

CONTENTS

1. Introduction

PROMETHEUS implements control strategies with extreme high reliability requirements. In the design process automated tools are able to ensure the desired quality in terms of correctness by validating the strategies against predefined criteria.

The case study in this paper shows the description of a problem for cooperative overtaking, the design of a control strategy as well as the definition of some correctness criteria, and finally the automated validation by an tool called SPIN [Holz 91] realized by G. Holzmann at AT&T Labs.

1.1. Motivation

PROMETHEUS / DRIVE is not only concerned with R&D on technical developments but also with the identification of the best choice of systems from an economic and

technical point of view, and the best strategy for their implementation [DRIVE 91]. [Tri* 91] discusses the dangers of premature applications from both a political as well as a technical point of view.

Structural decomposition of a complex system is needed in order to manage the specification of behavior and to facilitate analysis [Ern* 90a, Ern* 90b]. Not surprisingly, many failures of critical systems result from our inability to foresee a unlikely set of contributory circumstances. The likelihood of such an error increases with system complexity [Clu* 91, RACE 89].

Unambiguity of specifications is not enough since, for example, software could become totally reliable, always performing the same, possibly incorrect function [Clu* 91]! Contradictions within a specification must be revealed by analysis and an extensive review of the specification. This yields the need for early validation of correctness and completeness in systems design [Clu* 91, Hills 91]. Verification should hence be considered in the specification phase when designing both control strategies as well as communication systems.

Arguments for the use of formal methods in communication aspects of PROMETHEUS/DRIVE have already been presented in [Bea* 91, BuOn 91, Ern* 89, Hüb* 90, LNEF 90] and other reports. For an account of the latest discussions on formal methods within PROMETHEUS/PRO-COM refer to [Lara 91].

The following case study of the overtaking protocol is, as in the case of [Ern* 91], a simple and naive one. As in [Ern* 91] we have deliberately kept it simple to make it easy for the reader to follow the specification and verification. However, the tool of this study is considerably more powerful than the ones used in [Ern* 91]. Also, our approach is somewhat different in its modelling as we will see below.

2. A Case Study

This section will describe the design of a cooperative overtaking problem in a top-down approach. According to [RACE 89], a top-down approach facilitates the design of good reusable objects, or system components, while simple objects point towards a bottom-up approach.

First an overview of the system is given. The services (benefits) are outlined and followed by a more detailed description of its components. The end of this section defines some test criteria for the validation of the system.

2.1. The Sample Problem

Overtaking is a critical manover for every driver. Depending on many physical and psychological influences a driver decides whether he executes an overtake or not. Nevertheless even an experienced driver (like we all are...) sometimes takes the wrong decision. Results are common efforts of the overtaker, the overtaken and the opposite traffic to avoid an accident.

To prevent a driver from entering such dangerous situations future cars will be

equipped with assistance systems to estimate the outcome of manovers. The decision for advising the driver to suspend a manover will be based on a local check of sensors and on a global check by a additional data exchange with other cars or roadside devices.

The system studied here has the task to warn (and only to warn) the driver if it detected a critical outcome of a manover. Positive results are not indicated in order not to take some of the driver's responsibility or reduce his attention.

The driver invokes the process by setting a blinker. The system interprets this as a wish to overtake and first computes the local situation. After a positive decision the upcoming manover is indicated via data exchange to other cars in the environment. The cars in the environment answer the indication by a positive or negative advise. Negative local decisions or negative advises are displayed to the driver e.g. by flashing a danger symbol or emitting a danger tone. The driver will overtake after a certain (short) time when he was not warned.

The RTI application specification structure of SECFO [LBMO 91] defines applications in terms of the four basic building blocks *processes*, *data bases*, *sources/sinks* and *inputs/outputs*. The overtaking "algorithm" presented in this paper could be regarded as an outline of a function within the area "Vehicle Control" (See also scenario 3 in [Cepe 91]).

Still, we are aware of conclusions from other projects, for example that the feedback which is made available to drivers is not necessarily sufficient to sustain skill [Groe 91]. As is pointed out in [GoBe 91, Wied 91], the impacts on the driver of additional sources of information is still largely unknown.

2.2. The System

Our model system consists of two drivers and their cars (fig.1). Additional cars can be included. In this study we restrict the scenario to two cars which is also the first step in real validations to keep complexity and computation low in the beginning.

The driver signals an overtaking by the blinker (on/off). In reaction the car warns the driver by a display if a manover is critical. The driver is a allowed to cancel and to restart his manover at any time.

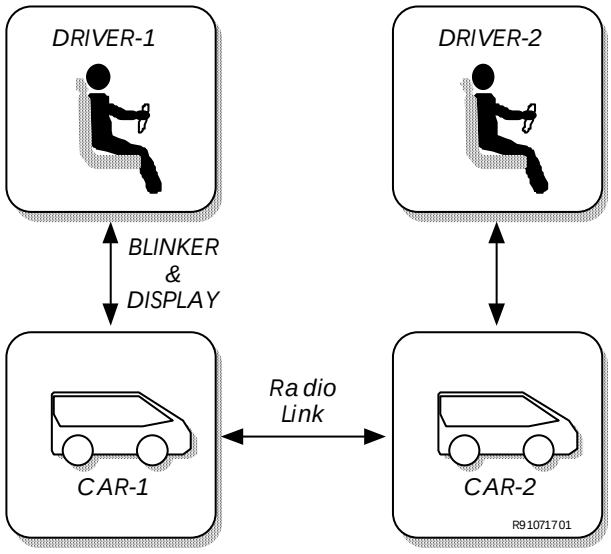

fig.1 The system under test

A car shall be divided into two modules: the man-machine-interface (mmi) and the intelligent copilot. The mmi is responsible for the interaction with the driver. It reads the blinker and forwards on/off information to the copilot. A negative answer of the copilot must displayed as a danger warning to the driver.

The copilot receives the start of an overtaking by the mmi. It checks the situation first locally. In case of a positive decision the copilot starts to notify the other copilot of the intended manover. The other copilot then confirms or disagrees by a message. The copilots exchange indications, negative and positive agreements by means of radio communication. The radio environment will loose messages at random instances. The communication system is not modelled in further details as our interest concentrates on the cooperative overtaking strategy.

Functional characteristics of a generic intelligent driver support can be found in [MiMc 91]. For more details on levels of automation, i.e. informative / warning / assisted / automatic when assisting the driver, see [BLMM 91].

2.3. The DRIVER

The behavior of a driver during his driving life has four distinguishable stages (=states). They are the following (fig.2):

State 1 - IDLE:	He is just driving and listening perhaps to some music.
State 2 - WAIT:	He has set the blinker and awaits the display for danger.
State 3 - OVERTAKE:	He waited enough and overtakes.
State 4 - STOP:	The display flashed and he suspends the manover.

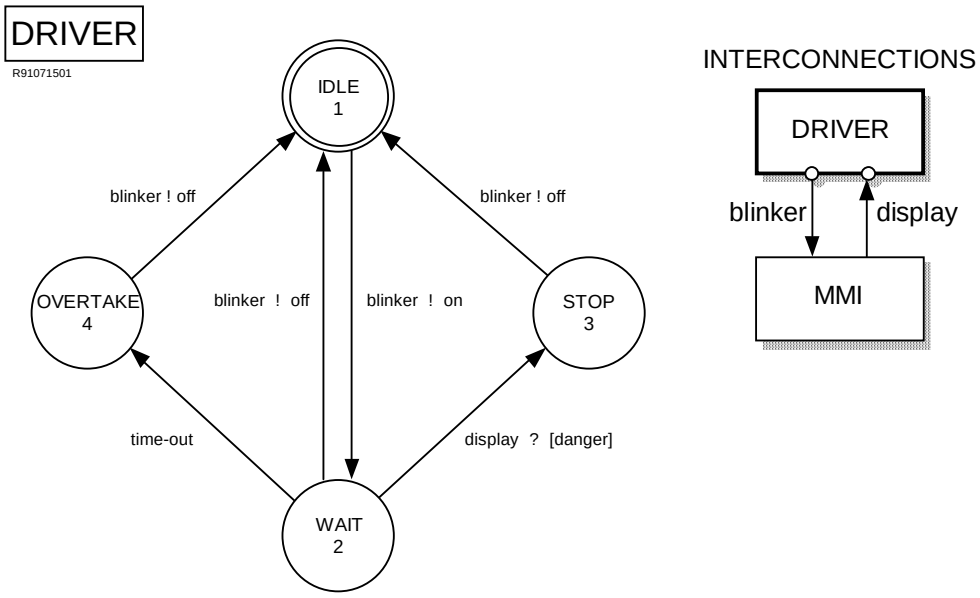

fig.2 the driver states

In our study we assume that he cannot switch from one state to any other without obeying some basic rules. He will never overtake without setting the blinker, that means he never goes directly from state IDLE -> OVERTAKE. This basic rules are indicated by the arcs between the states.

The drivers output - indicated by a "!" - to the mmi is always via the blinker. If he wishes to overtake he must first set the blinker to "on", the formal writing for this is "**blinker!on**" in the so called "*driver state diagram*".

The drivers input - indicated by a "?" - from the mmi comes always through the display. While waiting for the warning he might read a danger warning from the display, that is written as "**display?danger**". (Remark: [danger] means that the mmi must delete the danger-display not the driver)

The driver state diagram can now be interpreted. If he wishes to overtake he must set the blinker and than wait for a danger display. If he has waited a minimum time he starts to overtake. If he gets a warning he stops his manover. The last possibility is that he set the blinker but was aware of a danger afterwards and therefore ignores the mmi and goes back to IDLE immediately.

2.4. The MMI

The mmi has the difficult task to synchronize the driver's activities with the copilot system. As the driver can reset the blinker at any time, outdated decisions might come in after he started a new manover. Therefore the mmi attaches transaction numbers "n" to all requests.

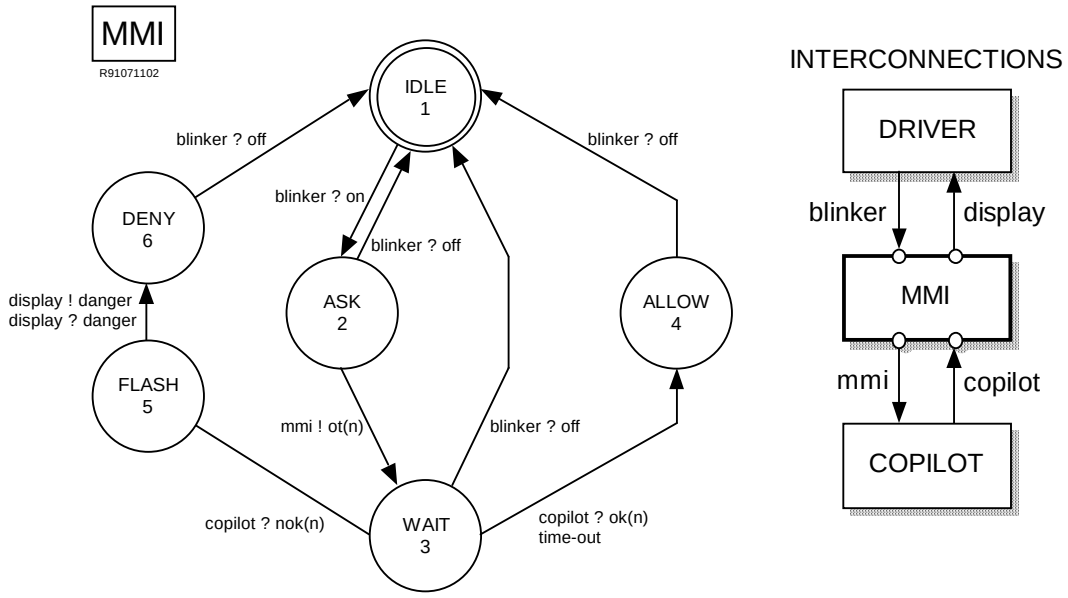

fig.3 the man-machine interface states

After receiving a request *(blinker?on)* the mmi forwards it to the copilot via its mmi connection *(mmi!ot(n)*. Then the mmi waits for the reaction of the copilot.

After a positive reaction *(copilot?yes(n))* or if the copilot fails to answer in time, which should not happen but might happen in an multi-tasking environment, the mmi allows overtaking. Before returning to the idle state the mmi must wait for the end of the overtaking. This is indicated by the driver's reset of the blinker *(blinker?off)*.

A negative copilot response forces the mmi to display the danger by *display!danger*. It then removes the warning again and waits for the driver to cancel the manover.

A driver can cancel his manover at any time even while the mmi is waiting for copilot reactions. This indicated by the direct transition from the wait to the idle state.

2.5. The COPILOT

The copilot (fig.4) interacts with the local mmi and cooperates with other copilots in the environment. It computes the local situation but bases his final decision on the external consults.

Copilots consult each other by an indication message *(out!ind(x))*. This indication includes again a transaction number x to sort out old messages. The answer can be positive *(in?yes(x))* or negative *(in?no(x))*. A negative answer results from simultaneous overtakes or due to other reasons.

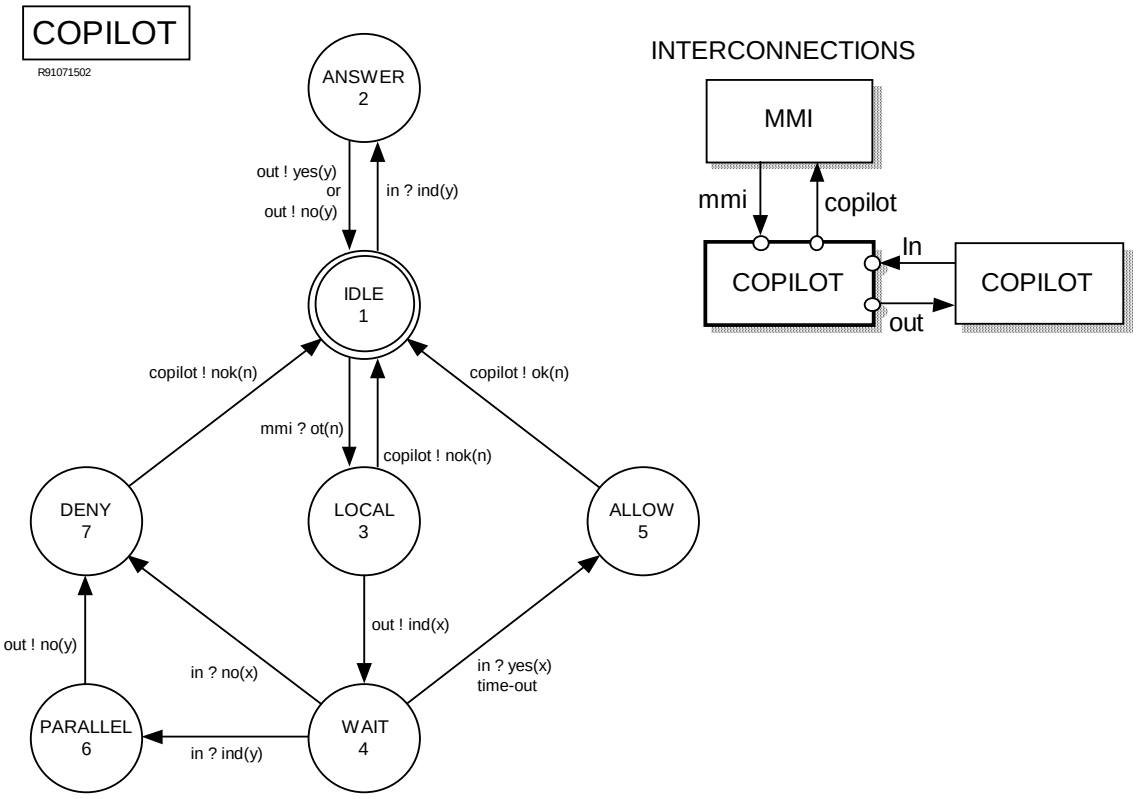

fig.4 the copilot states and interconnections

These other reasons are not modelled in our example. They are possible and of importance for the inclusion in an algorithm. This demonstrates that parts of the final strategy can be developed also not all details are available.

Requests of the mmi are evaluated locally and answered immediately if a problem was detected *(copilot!nok(n))*. After a positive evaluation other copilots are consulted. A positive reaction or time-outs lead to allowance of the overtaking *(copilot!ok(n))*. Time-outs occur when other copilots answer to late or due to loss of messages on the radio channel.

A negative answer, or even worse the indication of a simultaneous manover, both cause a negative decision (copilot!nok(n)).

2.6. The Criteria

A correctness proof is always accompanied by a specification of system properties. They describe the desired strategy service and behavior. Criteria can be classified into two categories. Global strategy properties, independent of the strategies contents and specific properties, dependent on the designer's intentions.

Global strategy properties

Is the strategy specified sufficiently? A sequence of events could cause the reception of unexpected messages. The copilot in the IDLE state should not be confused by delayed answers like *in?no(y)* of other copilots.

Does the strategy reach all system states? Is the strategy overspecified? After specification of the system an automated tools will try to find "dead code". This dead code could consist of useless and never reached strategy parts, but also it could be

very important code. If a copilot sends indications but never gets reactions from other copilots, then the designer should worry about his specifications. . .

Does the strategy deadlock the system? A designer might specify that a copilot must wait for the reaction of other copilots. But as messages can be lost the copilot will wait forever. Even worse, the copilot is unable to continue his local work.

Are there infinite loops ("non-progress cycles") ? The responses of other copilots could be so important to a strategy designer that he repeats indications after a time-out of answers. The strategy can be locked in this cycle forever if the designer forgets to include something like an "emergency exit" after enough trials.

Specific properties

After checking the strategy carefully against absence of the above negative properties some more specific properties can be validated.

Can the mmi be confused by a driver playing around with the blinker? A "confused" mmi might not be able to warn the driver although there is a danger.

Is the driver always warned if a danger has occured? This property must be checked in two ways. A danger can be detected by the local copilot or through the advise of an external copilot.

Will the strategy prevent two driver's from overtaking simultaneously? The sample strategy could be designed to avoid parallel overtaking of two cars in order to guarantee more safety.

Is it possible that a danger warning is never removed after the driver stopped overtaking?

More criteria like these can be formulated easily.

2.7. The Impossible

Some properties are important for the designer but are impossible for an automated validation tool to validate. Performance characteristics like violation of real-time requirements or system throughput are not validatable by current tools. They would add an additional level of sophistication to the systems. Besides all possible interactions now information on these parameters would have to be collected in parallel to state the absence of violations.

Some of the results are achievable by simulation which indicate the desired behavior in combination with a statistical expectation. Simulators such as the Simulator for Cooperative Automotive Network (SCAN) [KePi 91] are currently used for real-time simulation of control strategies.

3. Validation Process and Results

3.1. An Complexity Estimation
3.2. Translation into PROMELA
3.3. Validation by SPIN
3.4. The SPIN Analyzer
3.5. The SPIN Simulator
3.6. Interpretation of Results

In this section the validation process and its requirements will be discussed. Based on the described state diagrams a complexity estimation shows that the validation must be undertaken by automated tools. The model is translated from graphical representation into a validation language called PROMELA. The source code of our

model is then compiled by the PROMELA to build an model analyzer. The results of the analysis are interpreted for different cases of correctness properties.

3.1. An Complexity Estimation

During the discussion of our model we used graphic representations of our strategy solution. A process (the driver, the mmi, the copilot) was at a certain time in a certain state. An external event (input) or internal event (output) caused the process to move to another state. The so-called finite state machine (FSM) described the behavior of our overtaking system. Between the processes we established channels to exchange messages. On these channels only one message at a time was allowed.

A first estimate of the potential system complexity will be derived from the number of system states and possible messages types.

The system consists of six processes:

• two drivers à 4 states • two mmis à 6 states • two copilots à 7 states

Between these the following connections ("channels") exist:

• blinker channel	à 3 types	(on, off, no message)
• display channel	à 2 types	(danger, no message)
• mmi channel	à n+1 types	(ot(n), no message; here n=2)
• copilot channel	à 2n+1 types	(ok(n), nok(n), no message; here n=2)
• out channel	à 3x+1 types	(ind(x), yes(x), no(x), no message; here n=2)
• in channel	à 3x+1 types	(ind(x), yes(x), no(x), no message; here n=2)

All processes can be in one of 4x4x6x6x7x7 = 28224 combined states. The channels can be in one of 3x3x2x2x3x3x5x5x7x7x7x7 = 19448100 states. Our system is at a random instant in one of 28224 x 19448100 = 548.903.174.400 states ≈ 10E+12 states. Not all possible states of the "state space" but e.g. a subset of 10E+8 are reached.

Tools using advanced validation methods are able to check 10E+4 states/sec. A validation of our system will therefore cause a minimum of computation time of about 3 hours. Additional time is needed to validate global properties like absence of "non-progress cycles" and specific properties like "no parallel overtaking".

3.2. Translation into PROMELA

Before our system can be validated we must transform the graphical and textual representation into statements of a specification language. This specifications will be used by a compiler to build an analyzer(fig. 5).

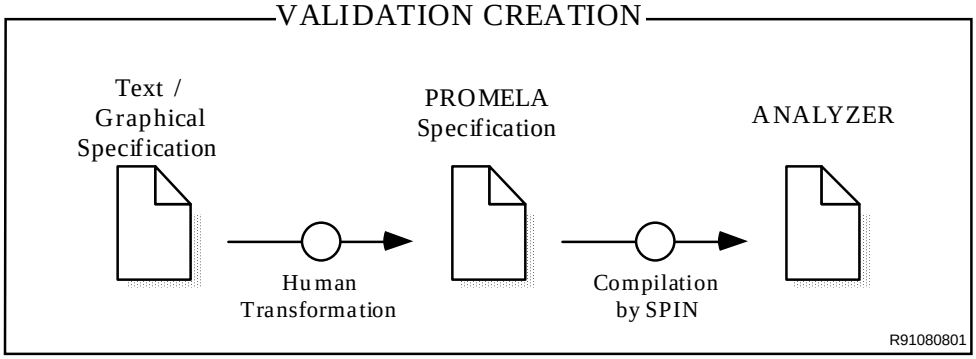

fig.5 Building the Analyzer

The language used here is called PROMELA [Holz 91]. It is close to C but has some

extensions to provide formalization of processes, communication channels and correctness criteria. The specification are compiled into an automated analyzer program by the SPIN [Holz 91] compiler.

The realization of our model into PROMELA is straight forward. First, the required channels and the possible message types are declared. Second the generic processes for the driver, the mmi and the copilot are defined. Last, the complete scenario is created by running two sets of these processes for the two vehicles. (For more details see section 4).

3.3. Validation by SPIN

SPIN is a tool for analyzing the logical consistency of concurrent systems. Given a model in PROMELA, SPIN can generate an analyzer that performs a fast exhaustive validation of the system state space, or it can perform random simulations of the system's execution (fig.6).

A standard validation procedure by SPIN consists of three steps:

- Compilation of PROMELA specification into an analyzer by SPIN
- Automated validation by analyzer which will produce an error trail
- Simulation of the error trail to generate a global system dump for debugging

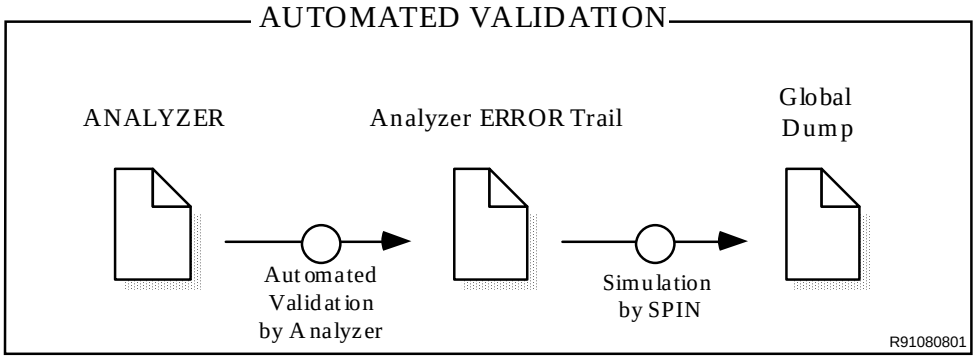

fig.6 Automated Validation by SPIN

3.4. The SPIN Analyzer

SPIN offers the choice to generate a full system state space search (each state representated e.g. by 100 bytes) or a partial search with very high speed (each state needs only 1-2 bits). The second option must be selected if the memory of the host system is not large enough to hold the system states, or when high speed is desired.

With the second option an error can be missed - theoretically. The states are mapped by a hash function from e.g. 100 bytes down to a bit position which indicates that a state has been visited. If the hash function mappes two states onto the same bit, the minimal probability exists that an error path will not be followed up to its end. Nevertheless it is highly realistic to assume that the same error can be reached through multiple paths. Therefore it would be also detected. In addition a "bit state" search is able to handle a state space which is approx. 10.000 times larger.

The resulting analyzer will tract the system for all interactions, process and channel states. If an error occures a compressed trail is dumped by the analyzer. This trail can be used as an input for the SPIN simulator to watch the history of an error.

3.5. The SPIN Simulator

The SPIN tool offers the possibility to perform a random or error-trail-guided simulation of a PROMELA specification. Options of the dump are process states, local variables, receive and send messages events. With the desired dump a step by step path is given with all necessary information to understand why an error occured.

Some examples for possible errors are given in the following example.

3.4. Interpretation of Results

For a more detailed understanding of the SPIN validator some scenarios with sample errors are explained. At this point it must be emphased again that the correctness of a specification dependends on the defined correctness properties by the designer.

To say it more directly: SPIN knows nothing about driving a car, but it can check whether the specification fell into a predefined trap build by its user.

Normal termination without errors

The following sample dump (tab.1) of a validation checked the system for invalid endstates (deadlocks), unreached states (dead code) and in addition for non-progress cycles (useless repetition of states). No user criteria (assertions) were included.

```
Thu Aug  8 13:53:00 MET DST 1991

bit statespace search for:
        assertion violations and non-progress loops
vector 128 byte, depth reached 9999, non-progress loops: 0
11253511 states, stored
      10 states, linked
58405074 states, matched           total: 69658595
hash factor: 1.490843 (best coverage if >100)
(max size 2^24 states, stackframes: 0/11317)

Thu Aug  8 15:25:18 MET DST 1991
```

Tab.1 Normal SPIN analyzer termination

```
Thu Aug  8 13:53:00 MET DST 1991
```

This line was included by a self written batch file to check the required runtime

```
b i t   s t a t e s p a c e   s e a r c h
f o r :     a s s e r t i o n
v i o l a t i o n s     a n d
n o n - p r o g r e s s     l o o p s
```

The systems checks user assertions and non-progress loops. The search for loops requires a second bit for every state.

```
vector 128 byte, depth reached 9999, non-progress loops: 0
```

The description of a temporary system state needs 128 bytes, that includes process states, channel states, variable states.
A path of system states is limited to a max. depth of 9999 states in a line.
When checking the system for non-progress loops none were found.

```
11253511 states, stored
```

During the validation the system detected 11.253.511 different states which were added to the system state space.

```
10 states, linked
```

Linked states are states encountered within atomic sequences. As atomic sequences of states are not interruptable by other processes they are treated in common and not stored separately.

```
58405074 states, matched            total: 69658595
```

During its search through the state space the analyzer recognized 58.405.074 states it already had visited a state. This is caused by loops or joining system pathes. This states are not stored again.

The total visited states during this validation run summed up to a total of 69.658.595 states. This illustrates the necessity for automated tools to validate control strategies.

```
hash factor: 1.490843 (best coverage if >100)
(max size 2^24 states, stackframes: 0/11317)
```

The hash factor describes the relation between required and offered memory. Offered by the user were 2^24 positions for storing of states. Due to the search for non-progress loops a state consumes two bits yielding in 2 x 2^24 bits = 33.554.432 bits ≈ 4 Mbyte. The search stored 11253511 à 2 bits requiring 22.507.022 bits ≈ 2,7 Mbyte. The quotient of these figures which is 1.490843 demonstrates wheter there was sufficient memory and how it was used.

A hash function is dependend on the available space for mapping the 128 bytes state vectors down to bit position. The higher the usage of the offered memory the higher is the probability that two states were mapped onto the same bit. Therefore high numbers for the hash factor (>100) are desirable.

```
Thu Aug  8 15:25:18 MET DST 1991
```

This line was included by a self written batch file to check the required runtime. It is not the pure CPU time as the system (a SUN SPARC station with 15 MIPS) was also loaded with other tasks.

During the run the analyzer got about 90% of the CPU time. The run took ≈ 90 minutes brutto, which results to a rate of ≈14.000 checked states every second. Current validation system are only able to check about 100 states a second. If the validation would have been possible with a current system or if SPIN performs a full state space search representing each state by 128 bytes it would have taken about 153 days and minimum 1.3 Gbytes of state memory to get the result!

Example of invalid end state

To show the detection of a deadlock the copilot's state :::::: was changed. There the copilot waits for an agreement of another copilot. If the time-out transition from state 4 "WAIT" to state 5 "ALLOW" is removed then the copilot will wait forever when a message is lost. The system is deadlocked and the local driver is not supported anymore by his overtaking system.

The validation by the SPIN analyzer stopped the validation after the detection of the endstate. The output (tab.2) is interpreted as follows:

```
bit statespace search for:
     assertion violations and invalid endstates
```

This time no progress loops are searched but the detection of endstate is wanted

```
unreached in proctype the_copilot:
     line 191 (state 58)
     reached: 57 of 58 states
```

The SPIN tool decomposed the copilot process into 58 states. That only 57 of 58 states are reached is correct because after generation of a copilot process it is never terminated or deleted. Reaching the 58th state would be the termination of the copilot.

```
Thu Aug  8 16:18:39 MET DST 1991
bit statespace search for:
        assertion violations and invalid endstates
vector 124 byte, depth reached 99, errors: 0
 1370449 states, stored
       5 states, linked
 5849995 states, matched           total:  7220449
hash factor: 3.060530 (best coverage if >100)
(max size 2^22 states, stackframes: 0/16)

unreached in proctype _init:
        reached all 8 states
unreached in proctype the_copilot:
        line 191 (state 58)
        reached: 57 of 58 states
unreached in proctype the_mmi:
        line 115 (state 42)
        reached: 41 of 42 states
unreached in proctype the_driver:
        line 52 (state 23)
        reached: 22 of 23 states
Thu Aug  8 16:26:28 MET DST 1991
```

tab.2 Detection of an invalid endstate (deadlock)

HERE MORE TEXT AN THE FINAL DUMP WILL BE INCLUDED.

4. Implementation Details

Some additional information on the transformation into PROMELA are given in this chapter. They correspond to the diagrams given in section 2 and the listing provided by section 6.

4.2. The System

The system was coded very close to the state diagrams given in figures 2 trough 4. All states are represented in the listing by labels. Processes were written in a generic way to enable multiple reuse.

The dynamic creation of a processes is done at the end by the "init{}" process which creates all other process by "run process_name(parameters)".

4.3. The DRIVER

One main feature of a validation tool is that all possible developments of the system states must be examined. Random choices are replaced by outlining all alternatives.

While the driver waits for the mmi reply on his overtaking wish, three things can happen:

- He gets a warning and stops to overtake
- He starts to overtake (he waited long enough, he ignores a warning)
- He cancels the manover (whether he was warned or not)

All three alternatives are listed in the State's Do-loop (tab.3), the SPIN analyzer will follow each of the three possibilites in seperate system state paths. A time-out reaction can happen at any time.

```
S2:     /*--- WAIT ---*/
        do
        :: display?[danger]    -> goto S3       /* No overtake          */
        :: skip                -> goto S4       /* timeout: overtake    */
        :: skip -> blinker!off -> goto S1       /* drv changed mind     */
        od;
```

tab.3 DRIVER wait state for MMI reaction

4.4. The MMI

The mmi interacts with an asynchronously operating driver and a copilot process which might not answer in time or too late. Therefore the reception of message not expected in a state is possible. A late answer from the copilot while the mmi is idle is just discarded (tab.4).

```
S1:     /*--- IDLE ---*/
        do
        :: blinker?on  -> goto S2       /* New Request          */
        :: copilot?any(m) -> goto S1    /* Delayed answer ?     */
        od;
```

tab.4 MMI discards unexpcted messages

The display of a warning is done by two consecutive statements in the mmi process. (tab.5). As they are not protected by an atomic sequence (see init{}) the analyzer checks all possibilities which occure by events between these two statements e.g. the driver cancels.

4.5. The COPILOT

The copilot advices other copilots whether a manover is ok or not. In addition the total loss of the answer message on the radio channel is included (tab.5).

```
S2:     /*--- ANSWER ---*/
        do
        :: skip -> out!yes(y) -> goto S1        /* Random        */
        :: skip -> out!no(y)  -> goto S1        /* Random        */
        :: skip               -> goto S1        /* Loss of mesg  */
        od;
```

tab.5 MMI discards unexpcted messages

The selection algorithm of the answer is simulated by an alternate choice of yes or no. The decision finding process of these answers is not topic of this paper and therefore not regarded. Loss of an answer is implemented by the third alternative in which the copilot just does not answer.

4.6. User Criteria

The listing contains no user criteria. They are possible by inclusion of assert statements and temporal claims [Holz 91]. An example is the check that it would be never possible that two drivers overtake in parallel. This assertion will be violated due to two reasons: First, the driver is allowed to ignore the warning or he might get the warning too late, second one of the answers among copilots was lost.

5. Summary and Conclusions

A new vaildation tool called SPIN was applied to the design of PROMETHEUS control strategies. It was shown that the transformation from graphical and textual specification into a specification language (here: PROMELA) is straight forward for normal experienced programmers. The results of a sample validation run demonstrated that the designed model for an overtaking manover had no deadlocks, unreached states or useless cycles.

Available tools for system validation must be improved by the use of bit representations of the state space in order to gain speed (100 times) and credibility ($\approx$1000-10000 times larger state space).

6. Listing

This is the PROMELA source listing of the discussed overtaking strategy. Researchers interested in the source can request it from reichert@sics.se.

```
/* File: overtake.src */
/* Date: 910809       */


/*-------- Cooperative Overtaking ------------*/

#define N 1


/*-------- Required Channels ------------*/

chan c_blinker[2] = [N] of { byte };
chan c_display[2] = [N] of { byte };
chan c_mmi[2]     = [N] of { byte, byte };
chan c_copilot[2] = [N] of { byte, byte };
chan c_com[2]     = [N] of { byte, byte };

mtype = {on,off,danger,ot,ok,nok,ind,yes,no}


/*--- THE DRIVER -------------------------------*/
/*                                              */
/* He talks to the mmi via the blinker          */
/*                                              */
/* The blinker can only be toggled, therefore   */
/* only alternating !on !off sequence are send  */
/*                                              */
/* MMI interaction                              */
/* ---------------                              */
/* he can activate a blinker:      !on          */
/* or can deactivate the blinker:  !off         */
/* The MMI signals danger by:      ?danger      */
/*                                              */
/*----------------------------------------------*/

proctype the_driver(byte id; chan blinker; chan display)
{
S1:     /*--- IDLE ---*/
        do
        :: blinker!on  ->  progress: goto S2    /* try to overtake      */
        od;


S2:     /*--- WAIT ---*/
        do
        :: display?[danger]    -> goto S3       /* No overtake          */
        :: skip                -> goto S4       /* timeout: overtake    */
        :: skip -> blinker!off -> goto S1       /* drv changed mind     */
        od;

S3:     /*--- STOP ---*/
        skip;                                   /* driver cancels       */
        blinker!off -> goto S1;                 /* reset blinker        */

S4:     /*--- OVERTAKING ---*/
        skip;                                   /* driver overtakes     */
        blinker!off -> goto S1                  /* reset blinker        */
}



/*--- THE MMI ----------------------------------*/
/*                                              */
/* The MMI (man machine interface)              */
/*                                              */
```

```
/* Reads requests from the driver and forwards  */
/* a overtake indication to the copilot.        */
/* Displays the result of copilot computation   */
/* with a "DANGER" light/buzzer etc...          */
/*                                              */
/* DRIVER Interaction:                          */
/* Reads request by the blinker: ?on / ?off     */
/* Displays warnings by display: !danger        */
/*                                              */
/* COPILOT interaction:                         */
/* Sends a indication + transaction no: !ot(n)  */
/* reads response with transaction no:  ?nok(n) */
/*                                      ?ok(n)  */
/*----------------------------------------------*/


proctype the_mmi(byte id; chan display, blinker , mmi, copilot)
{    byte n,m,any;
S1:     /*--- IDLE ---*/
        do
        :: blinker?on  -> goto S2       /* New Request          */
        :: copilot?any(m) -> goto S1    /* Delayed answer ?     */
        od;

S2:     /*--- ASK ---*/
        do
        :: blinker?off -> goto S1       /* driver cancels       */
        :: mmi!ot( (n+1)%2 );           /* New Transaction      */
           n = (n+1)%2 -> goto S3       /* WAIT for copilot     */
        od;

S3:     /*--- WAIT ---*/
        do
        :: blinker?off -> goto S1       /* driver cancels       */
        :: copilot?ok(m) ->
                if
                :: (m==n) -> goto S4    /* no danger seen       */
                :: (m!=n) -> goto S5    /* invalid no           */
                fi
        :: copilot?nok(m) ->
                if
                :: (m==n) -> goto S5    /* problem detected     */
                :: (m!=n) -> goto S5    /* invalid no           */
                fi
        :: skip  -> goto S4             /* time out             */
        od;

S4:     /*--- ALLOW ---*/
        skip;                           /* No warning           */
        blinker?off -> goto S1;         /* Overtaking ended     */

S5:     /*--- FLASH ---*/
        display!danger;                 /* Buzz, beep etc..     */
        display?danger;                 /* End of beeping etc   */

S6:     /*--- DENY ---*/
        blinker?off;                    /* wait for manover end */
        goto S1
}
```

```
/*--- THE COPILOT -------------------------------*/
/*                                               */
/* The copilot computes the local situation      */
/* If there is                                   */
/* a problem  -> overtaking denied               */
/* no problem -> asks other copilots             */
/*               -> others see no problem        */
/*               -> others deny overtaking       */
/*               -> parallel request from other  */
/*                                               */
/* MMI Interaction                               */
/* ---------------                               */
/* Gets informed by mmi:             ?ot(n)      */
/* Indicates no problem by:          !ok(n)      */
/* Indicates problem with:           !nok(n)     */
/*                                               */
/* COPILOT interaction                           */
/* -------------------                           */
/* sends an indication:  !ind(x)                 */
/* receives answers or simulatuously indication  */
/*                       ?ind(y) ?yes(x) ?no(x)  */
/* sends answers after indications:              */
/*                        !yes(y) !no(y)         */
/*                                               */
/*-----------------------------------------------*/


proctype the_copilot(byte id; chan out, in, copilot, mmi)
{       byte x,y,n;                               /* Transaction no */
S1:     /*--- IDLE ---*/
        do
        :: in?ind(y)    -> goto S2              /* Neighbor request     */
        :: mmi?ot(n)    -> goto S3              /* Driver request       */
        :: in?no(y)     -> goto S1              /* too old to use       */
        :: in?yes(y)    -> goto S1              /* too old to use       */
        od;

S2:     /*--- ANSWER ---*/
        do
        :: skip -> out!yes(y) -> goto S1        /* Random        */
        :: skip -> out!no(y)  -> goto S1        /* Random        */
        :: skip               -> goto S1        /* Loss of mesg  */
        od;

S3:     /*--- LOCAL ---*/
        do
        :: skip -> copilot!nok(n) -> goto S1    /* Local problem  */
        :: skip -> x=(x+1)%2;                   /* no loc problem */
                   out!ind(x)  -> goto S4       /* inform others  */
        od;

S4:     /*--- WAIT ---*/
        do
        :: in?yes(y) ->
                if
                :: (x==y) -> goto S5            /* others allow */
                :: (x!=y) -> goto S7            /* invalid no   */
                fi
        :: in?no(y)  ->                         /* others deny  */
                if
                :: (x==y) -> goto S7            /* others deny  */
                :: (x!=y) -> goto S7            /* invalid no   */
                fi
        :: in?ind(y) -> goto S6                 /* parallel ind */
        :: skip      -> goto S5                 /* time out     */
        od;

S5:     /*--- ALLOW ---*/
        copilot!ok(n) -> goto S1;               /* allow to mmi */

S6:     /*--- PARALLEL ---*/
        out!no(y)  -> goto S7;                  /* stop others  */

S7:     /*--- DENY ---*/
        copilot!nok(n) -> goto S1               /* no overtaking*/
}


/*--- Start the World of Drivers, Controllers & Oracles -----*/
init {
atomic{
        run the_driver( 0, c_blinker[0], c_display[0]);
        run the_mmi(    0, c_display[0], c_blinker[0], c_mmi[0], c_copilot[0]);
        run the_copilot(0, c_com[0],     c_com[1],     c_copilot[0], c_mmi[0]);

        run the_driver( 1, c_blinker[1], c_display[1]);
        run the_mmi(    1, c_display[1], c_blinker[1], c_mmi[1], c_copilot[1]);
        run the_copilot(1, c_com[1],     c_com[0],     c_copilot[1], c_mmi[1])
}}
```

7. Literature

[AuCo 91] •••K. Aulin, A. Corfdir, Implementation Aspects Concerning Legislation and Standardization, DRIVE V1067, Advanced Telematics in Road Transport, Proceedings of the DRIVE Conference, Brussels, Volume I, February 4-6, 1991, Elsevier Science Publishers B.V., P.O. BOX 211, 1000 AE, Amsterdam, The Netherlands

[Bea* 91] P. Beadle, M. Chanine, We. Kremer, G. Votsis, F. Lucazeau, A. Kemeny, Wo. Kremer, Vehicle Intercommunication: Architecture and Perspectives for Implementation, DRIVE V1063, Advanced Telematics in Road Transport, Proceedings of the DRIVE Conference, Brussels, Volume I, February 4-6, 1991, Elsevier Science Publishers B.V., P.O. BOX 211, 1000 AE, Amsterdam, The Netherlands

[BLMM 91] F. Broqua, G. Lerner, V. Mauro, E. Morello, Cooperative Driving: Basic Concepts and a First Assessment of "Intelligent Cruise Control" Strategies, DRIVE , Advanced Telematics in Road Transport, Proceedings of the DRIVE Conference, Brussels, Volume II, February 4-6, 1991, Elsevier Science Publishers B.V., P.O. BOX 211, 1000 AE, Amsterdam, The Netherlands

[BrAy 91] •••J. Bright, N. Ayland, Evaluating Real-time Responses to In-vehicle Driver Information Systems, DRIVE V1025, Advanced Telematics in Road Transport, Proceedings of the DRIVE Conference, Brussels, Volume I, February 4-6, 1991, Elsevier Science Publishers B.V., P.O. BOX 211, 1000 AE, Amsterdam, The Netherlands

[BuOn 91] S. Bueno, D. Ongaro, Vehicle/Roadside Communication for Route Guidance, DRIVE V 1011, Advanced Telematics in Road Transport, Proceedings of the DRIVE Conference, Brussels, Volume I, February 4-6, 1991, Elsevier Science Publishers B.V., P.O. BOX 211, 1000 AE, Amsterdam, The Netherlands

[Cepe 91] M. Cepeda, New Generation of Vehicle Scheduling and Control Systems, DRIVE , Advanced Telematics in Road Transport, Proceedings of the DRIVE Conference, Brussels, Volume II, February 4-6, 1991, Elsevier Science Publishers B.V., P.O. BOX 211, 1000 AE, Amsterdam, The Netherlands

[Clu* 91] D. Clutterbuck, W. Asmuth, T. Buckley, P. Jesty, J. Sonntag, R. Elink Schuurman, J. Giezen, Drive Safely, DRIVE , Advanced Telematics in Road Transport, Proceedings of the DRIVE Conference, Brussels, Volume II, February 4-6, 1991, Elsevier Science Publishers B.V., P.O. BOX 211, 1000 AE, Amsterdam, The Netherlands

[DRIVE 91] Commission of the European Communities, Directorate General XIII, Telecommunications, Information Industries and Innovation, Directorate F, RACE Programme and Development of Advanced Telematics Services, R+D in Advanced Road Transport Telematics in Europe, DRIVE 91, 200, Rue de la Loi, B-1049 Brussels, Belgium

[Ern* 89] P. Ernberg, K. Laraqui, A. Nazari, C. Odmalm, B. Pehrson, M. Svaerdh, A PROMETHEUS - PROCOM Framework, PROMETHEUS Proceedings of the 2nd Workshop, Volume I, Stockholm, Sweden, October 30-31, 1989, PROMETHEUS Office c/o Daimler-Benz AG, Postfach 60 02 02, D-7000 Stuttgart 60, Germany

[Ern* 90a] P. Ernberg, L. Fredlund, H. Hansson, B. Jonsson, F. Orava, B. Pehrson, Guidelines for Specification and Verification, SICS (Swedish Institute of Computer Science, PO Box 1263, S-164 28 Kista, Sweden) report PROMETHEUS/DRIVE

[Ern* 90b] P. Ernberg, L. Fredlund, B. Jonsson, K. Laraqui, A. Nazari, Specification Using LOTOS, Technical note to DRIVE/SECFO prepared by SICS (see [Ernberg 90a]) and the Swedish Telecom (K. Laraqui, A. Nazari, Swedish Telecom, S-136 80 Haninge, Sweden, Tel: +46 8 707 51 67, Fax: +46 8 707 46 84)

[Ern* 91] P. Ernberg, L. Fredlund, B. Jonsson, Specification and Validation of a Simple Overtaking Protocol using LOTOS, SICS (see [Ernberg 90a]), January 1991

[GoBe 91] H. Godthelp, F. op de Beek, Driving with GIDS: Behavioral Interaction with the GIDS Architecture, DRIVE 1041, Advanced Telematics in Road Transport, Proceedings of the DRIVE Conference, Brussels, Volume I, February 4-6, 1991, Elsevier Science Publishers B.V., P.O. BOX 211, 1000 AE, Amsterdam, The Netherlands

[Groe 91] J. A. Groeger, Supporting Training Drivers and the Prospects for Later Learning, DRIVE V1041, Advanced Telematics in Road Transport, Proceedings of the DRIVE Conference, Brussels, Volume I, February 4-6, 1991, Elsevier Science Publishers B.V., P.O. BOX 211,

1000 AE, Amsterdam, The Netherlands

[Holz 91] Gerard J. Holzmann, Design and Validation of Computer Protocols,Prentice Hall, 1991

[Hüb* 90] D. Hübner, O. Andrisano, F. Davoli, P. Godlewski, D. Hübner, W. Kremer, K. Laraqui, A. Nazari, C.-H. Rokitansky, S. Tabbane, PROMETHEUS Communication Systems Architecture - International PRO-COM Proposal for a Cooperative Control 1st Step Demnostrator, PROMETHEUS Office c/o Daimler-Benz AG, Postfach 60 02 02, D-7000 Stuttgart 60, Germany

[Kell 91] •••D. Kelly, Information Flow and Information Centres in a IRTE, DRIVE V1068, Advanced Telematics in Road Transport, Proceedings of the DRIVE Conference, Brussels, Volume II, February 4-6, 1991, Elsevier Science Publishers B.V., P.O. BOX 211, 1000 AE, Amsterdam, The Netherlands

[KePi 91] A. Kemeny, J.M. Piroird, A Simulator for Cooperative Driving, DRIVE V1048, Advanced Telematics in Road Transport, Proceedings of the DRIVE Conference, Brussels, Volume II, February 4-6, 1991, Elsevier Science Publishers B.V., P.O. BOX 211, 1000 AE, Amsterdam, The Netherlands

[Lara 91] K. Laraqui (editor), Proceedings of the PRO-COM meeting, PROMETHEUS/PRO-COM Stockholm, Sweden, May 29-30, 1991, PROMETHEUS Office c/o Daimler-Benz AG, Postfach 60 02 02, D-7000 Stuttgart 60, Germany

[LBMO 91] G. Lerner, G. Beccaria, V. Mauro, J. Olszewski, Functional Approach Towards RTI Applications and Impact Assessment, DRIVE V1057 (SECFO), Advanced Telematics in Road Transport, Proceedings of the DRIVE Conference, Brussels, Volume I, February 4-6, 1991, Elsevier Science Publishers B.V., P.O. BOX 211, 1000 AE, Amsterdam, The Netherlands

[LNEF 90] K. Laraqui, A. Nazari, P. Ernberg, L. Fredlund, Communication Systems Architecture - A Case Study, PROMETHEUS Proceedings of the 3nd Workshop, Turin, Italy, April 26-27, 1990, PROMETHEUS Office c/o Daimler-Benz AG, Postfach 60 02 02, D-7000 Stuttgart 60, Germany

[MiMc 91] J. A. Michon, H. McLoughlin, The Intelligence of GIDS, DRIVE V1041, Advanced Telematics in Road Transport, Proceedings of the DRIVE Conference, Brussels, Volume I, February 4-6, 1991, Elsevier Science Publishers B.V., P.O. BOX 211, 1000 AE, Amsterdam, The Netherlands

[RACE 89] RACE Project R1068, RACE Open Service Architecture, ROSA, 1st Deliverable, Object-oriented Techniques for ROSA, 68/BTR/425/DS/B/001/b1, British Telecom Research Laboratories, Martleham Heath, Ipswich, Suffolk IP5 7RE, Great Britain

[Tri* 91] U. Triulzi, H. Swahn, G. Agyrakos, M. Giaoutzi, K. Petrakis, A socio-political Approach to RTI Implementation Process - The SIRIUS Experience, DRIVE V1065, Advanced Telematics in Road Transport, Proceedings of the DRIVE Conference, Brussels, Volume I, February 4-6, 1991, Elsevier Science Publishers B.V., P.O. BOX 211, 1000 AE, Amsterdam, The Netherlands

[Wied 91] •••R. Wiedemann, Modelling of RTI-Elements on Multi-Lane Roads, DRIVE V1025, Advanced Telematics in Road Transport, Proceedings of the DRIVE Conference, Brussels, Volume II, February 4-6, 1991, Elsevier Science Publishers B.V., P.O. BOX 211, 1000 AE, Amsterdam, The Netherlands