Formal Specification and Validation of a Vital Communication Protocol*

A. Cimatti[1], P.L. Pieraccini[2], R. Sebastiani[1], P. Traverso[1], and A. Villafiorita[1]

[1] ITC-IRST, Via Sommarive 18, 38055 Povo, Trento, Italy
{cimatti,leaf,rseba,adolfo}@irst.itc.it
[2] Ansaldo Segnalamento Ferroviario, Via dei Pescatori 35, Genova, Italy
pieraccini@ansaldo.it

Abstract. Formal methods have a great potential of application as powerful specification and early debugging methods in the development of industrial systems. In certain application fields, formal methods are even becoming part of standards. However, the application of formal methods in the development of industrial products is by no means trivial. Indeed, formal methods can be costly, slow down the process of development, and require changes on the development cycle, and training. This paper describes a project developed by Ansaldo Segnalamento Ferroviario with the collaboration of IRST. Formal methods have been successfully applied to the development of an industrial communication protocol for distributed, safety critical systems. The project used a formal language to specify the protocol, and model checking techniques to validate the model.

1 Introduction

Formal methods have a great potential of application as powerful specification and early debugging methods in the development of industrial systems [2]. In certain application fields, formal methods are even becoming part of standards [1, 4]. However, the application of formal methods in the development of industrial products is by no means trivial. Indeed, formal methods can be costly, slow down the process of development, and require changes on the development cycle, and training.

This paper describes a project developed by Ansaldo Segnalamento Ferroviario (ASF) with the collaboration of IRST, where formal methods have been successfully applied to the development of an industrial communication protocol, called Safety Layer. The Safety Layer is used to present to safety-critical applicative software point-to-point, dependable channels, implemented over a double field bus. The design of such a protocol is a very complex task. A previous implementation, developed without the assistance of formal methods, required an expensive activity of debugging on the field, and was difficult to maintain and extend. On the other hand, the protocol was chosen (for its good qualities in safety and dependability) as the communication basis of several computer-based, distributed safety-critical products under development in ASF. Formal

* The work described in this paper was founded under contracts 1607/249958(NA) and 4508/324731(GE).

methods were thus applied to develop a high-quality product. The goal of the project described in this paper was to produce and validate a detailed design specification of the Safety Layer, which could be used as a basis for newly developed, well documented and highly maintainable software. Besides the technical difficulties, the project was subject to strict timing constraints. Furthermore, it was necessary to make the specification easily understandable also by software developers which were not expert in formal methods. Finally, the specification had to be as simple as possible to allow for a clean implementation.

The project started from informal requirements of previous implementations, from informal descriptions and from the knowledge internal to the company. During the project, the protocol was completely redesigned and formally specified and validated. The activity was carried out with ObjectGEODE [10], a tool for the formal development of concurrent systems. The SDL [6] graphical specification language was used to provide an operational specification of the protocol. The model checker of ObjectGEODE was heavily used to interactively simulate and exhaustively analyze the protocol. Several anomalous behaviors and incompletenesses in intermediate versions of the specifications were detected by simulation, and conveniently displayed in form of (automatically generated) Message Sequence Charts (MSC's) [7].

The final document specifying the Safety Layer [3] combines the formal model with informal explanations and annotations. It is currently being used as the basis for the implementation, and will be used as the basis for the interface specification of a stand-alone product. The executable formal model will be available for the analysis of possible modifications, and for the generation of the test cases.

This paper is structured as follows. In section 2 we describe the Safety Layer protocol. In section 3 we outline the project requirements, phases, and methodology. In section 4 we describe the specification of the Safety Layer, highlighting the technical details and the interaction of informal and formal specification. In section 5 we discuss the process of formal validation. Finally, in section 6 we discuss the impact of the project and we draw some conclusions.

2 Informal description of the Safety Layer

The Safety Layer is a communication protocol intended to provide reliable communication for distributed safety critical systems, e.g. Automatic Train Control systems, track-to-train communication systems, interlocking systems. Typically, such systems are based on several computing stations connected via field bus.

The Safety Layer provides applicative programs (running on different units) with point to point communication channels. Figure 1 depicts applicative programs A, B and C, running on different units, connected by point-to-point bidirectional channels AB and BC. Such channels are protected against data corruption and out-of-order reception by means of CRC (Cyclic Redundancy Checksum) protection, authentication mechanisms, sequence numbering and final shutdown in presence of repeated errors. The interface to the application

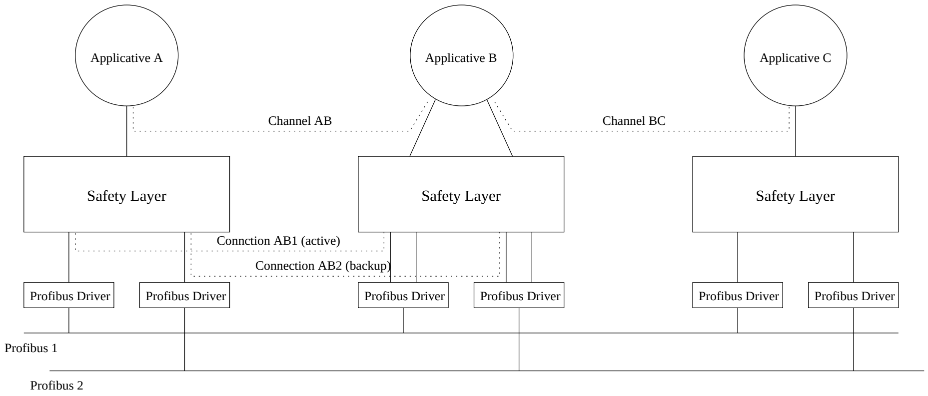

Fig. 1. The Safety Layer

program is extremely simple, i.e. only send and receive primitives are available. The Safety Layer is configured off-line, i.e. the applicative-level channels are fixed at design time.

Besides guaranteeing the *safety* of communication channels to the applicative programs, the role of the Safety Layer is to enhance *dependability*, i.e. to make the channels available for the applicative programs as much as possible. Therefore, each unit running the Safety Layer is connected to two independent field buses. The ProFiBus [8] drivers provide the Safety Layer with connection-oriented communication primitives. The role of the Safety Layer is to hide this redundancy from the application program, which only can see a reliable communication channel. The Safety Layer tries to maintain two connections, one per ProFiBus, for each point-to-point channel to the application level. In Figure 1, the channel AB between applicatives A and B is realized through the two connections AB1 and AB2. In the case of nominal behavior of the protocol, one connection is active, through which data can be transmitted, while the other is in standby. (Notationally, the former is called the *active* connection, while the latter is called the *backup* connection.) Even when no data is transmitted, a minimum rate of transmission is guaranteed on both connections, by means of special control telegrams, called connection monitoring. This mechanism is used to reduce the time needed to detect problems, such as disturbance on the bus, or malfunctioning of hardware components. Such problems may be revealed by messages with corrupted CRC or out-of-order sequence numbers, or by timeouts.

When a problem arises on the backup bus, the backup connection is reset, and the reconnection procedure is retried. The active connection is not affected. The protocol can thus work properly in a degraded mode, when only one connection is available, but this is completely hidden to the applicative program.

If a problem arises on the active connection, and the backup connection is working properly, then the *switch-over* procedure is undertaken. The switch-over is a distinguishing mechanism of the protocol. When a switch-over occurs, the backup connection becomes active, and will be used to transmit applicative data, while the (previously) active connection is reset and restarted in standby.

3 The project

The Safety Layer is a very complicated protocol. A previous implementation had been built with traditional software development methods, for a particular redunded architecture. This activity had been troublesome, and had resulted in high costs in development and maintainability.

3.1 Requirements

The Safety Layer protocol, presented in previous section, is the basis for several computer-based, distributed safety-critical products under development in ASF. The main requirement of the project was to provide a detailed, operational specification of the Safety Layer, describing the state machines realizing the protocol. This specification is a direct input to the implementation of the protocol, and the state variables specified in the model are to be implemented exactly as specified. This direct link between specification and implementation was required to enhance the maintainability of the protocol, and its reusability in subsequent applications. Furthermore, the specification should consider as legal the behaviors generated by the previous implementation of the Safety Layer. Finally, it was required that the time for the specification of the Safety Layer should not be too long, in order not to delay the implementation and thus impact on the projects where the Safety Layer had to be used.

3.2 Phases

The project was carried out in 10 months, along the following stages. First, an analysis of the protocol requirements was carried out. This task was particularly heavy, due to the large incompleteness in the documentation of the first implementation, and its dependence on the hardware architecture. As a result of this phase, a distinction was carried out between the aspects which should be formally analyzed, and the ones which were clear enough to be informally specified. Then, a preliminary model was developed, where the PROFIBUS was assumed not to corrupt but only lose data, i.e. transmission errors were not taken into account. In this stage, a crucial design choice was taken to structure the Safety Layer by introducing an intermediate level. This choice allowed to partition the complexity of the design problem, and resulted in a much clearer and manageable design. In the final stage, the protocol specification was extended to take into account the details of the transmission error. This phase required a substantial complication of (some of) the state machines. However, the modification to be applied to the model generated in the previous phase were uniform, and thus simple to realize.

3.3 Methodology

The methodology applied in this project was heavily based on formal methods. The ObjectGEODE tool was thoroughly applied in the project. ObjectGEODE is a commercial tool for the development of real-time distributed systems, based on formal methods. It allows to specify the system to be analyzed by means of SDL. A translator from StateCharts to SDL allows to obtain skeletons for SDL programs from high-level description of state machines. Requirements can be expressed by means of MSC's, or via observer automata expressed in a programming language. An explicit state model checker is directly connected to the SDL, and allows to exhaustively analyze finite state systems.

In this project, StateCharts were used to provide high level, pictorial representation of the finite state machines realizing the protocol. SDL was used to provide a precise, executable specification of the machines. The ObjectGEODE model checker was applied to reduce the development time, by pointing out a large number of problems in intermediate versions of the design. MSC were generated via simulation to provide an easy-to-understand description of behaviors.

4 Formal specification of the Safety Layer

The design specifications were produced in form of an SDL model. An SDL model of a system is described by a hierarchy of diagrams. In the top diagram (*interconnection diagram*) the system is decomposed into building *blocks*; each block is connected to other blocks and to the environment by *channels*; channels and blocks are interfaced by *gates*; each SDL channel is labeled by a couple of *message lists*, which describe the sets of messages which can be sent in the two directions. In a lower level of interconnection diagrams, each block is decomposed into a set of *processes*, which are interconnected by *routes*. In the further lower level of the hierarchy, each process is exploded into a set of *transition diagrams*, which represents (the SDL code of) the state machine of the process. This code can be executed by the ObjectGEODE simulator.

4.1 Safety Layer = CM + 2 · SL

The Safety Layer allows in principle to handle several channels to different units. For instance, channels AB and BC are handled by the central Safety Layer block in Figure 1. However, since there are no interactions between different channels on the same Safety Layer, it is possible to consider only the problem of specifying the management of a single channel. This means to "fill" a box with a port to the applicative program, and two ports to the ProFiBus drivers. The internal structure of such box is presented in Figure 2, in the form of SDL interconnection diagram.

The system is decomposed into three blocks: an upper block "Connection Manager" (CM) `CM_A` and two lower identical blocks "Safety Layer" (SL) `A1` and `A2`. (For compatibility with the notation used in other Asf documents, the expression "Safety Layer" has been used to denote both the whole system and the lower blocks; to avoid ambiguities, from now on we will denote

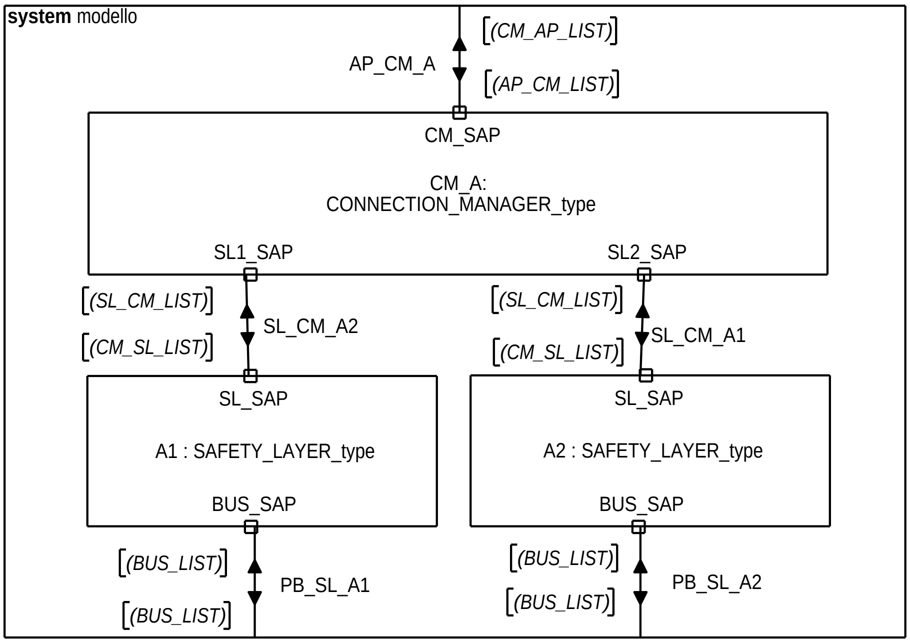

Fig. 2. The SDL diagram of the model of the Safety Layer.

the latter ones simply by "SL".) `CM_A` communicates with the application by means of channel `AP_CM_A` through gate `CM_SAP`, exchanging in the two directions messages belonging to `CM_AP_LIST` and `CM_AP_LIST` respectively. `CM_A` also communicates with the two SL's `A1`, `A2` by means of channels `SL_CM_A1`, `SL_CM_A1` through gates `SL1_SAP`, `SL1_SAP`, exchanging messages belonging to `CM_AP_LIST` and `CM_AP_LIST`. `A1` (resp. `A2`) communicates with its own ProFiBus driver by means of channel `PB_SL_A1` (resp. `PB_SL_A2`) through gate `BUS_SAP`, exchanging messages belonging to `BUS_LIST`.

The idea is thus to hierarchically decompose the protocol in three components. The rationale behind the decomposition is that each SL machine is able to handle a single connection, and is not even aware of the existence of the other SL machine. The CM is not aware of the details of the status of the SL machines. On the other hand, the CM has a clear view of the redundancy of the channel, and enforces control on the SL machines according to the events which are signaled by them.

In this way, it was possible to specify independently the SL and CM machines. This allowed to partition the specification problem, and make it possible to contain the number of states in each machine. The model checker was invaluable in the task of pointing out the bad interactions between them. Furthermore, the

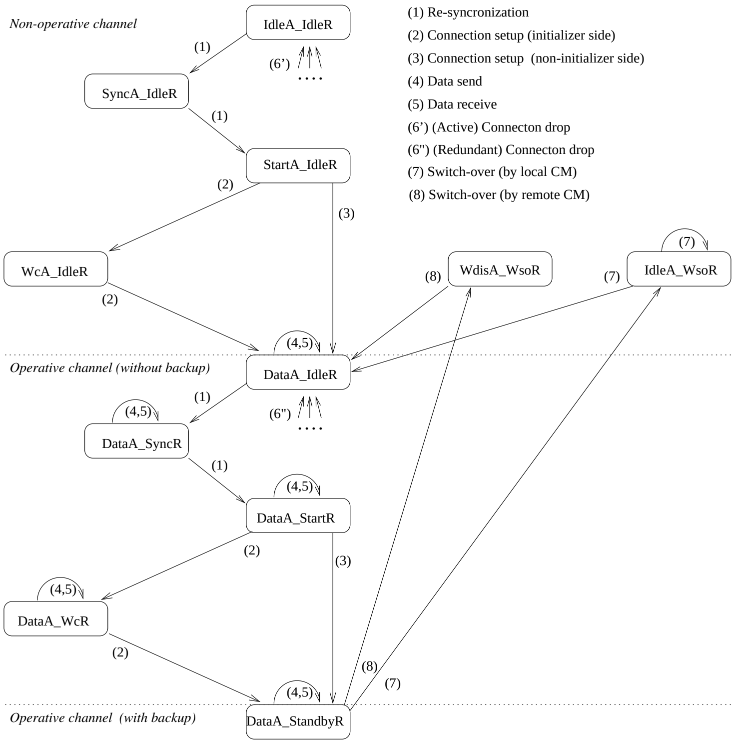

Fig. 3. A simplified view of the CM machine.

blocks A1 and A2 are implemented as two instances of the same process type. This means that the resulting system is conceptually simpler, the code can be heavily factored, and thus it is easier to maintain.

Finally, the SL and CM machines are asynchronously composed in the final system. This is a precise design choice of the model. Once the protocol is validated without constraints on the order of execution of its component machines, it is possible to choose, in the implementation phase, any possible algorithm of activation of the SL and CM machines. The model has been designed to reduce as much as possible the unnecessary constraints on the implementation level.

4.2 Overview of the CM machine

The executable SDL model of the machines was integrated with a set of high level, abstract StateCharts, which were used as a graphical roadmap for the specification. Figure 3 represents (a simplified view of) the CM machine. The name of each state is in the form `<s1>A_<s2>R`, where `<s1>` represents the state of the active connection and `<s2>` represents the state of the backup one. A couple of pointers keep track of which is the active SL and which is the backup one. The states are grouped into three distinct macro-states, each representing an operating mode of the channel.

Non-operative. It involves the states `IdleA_IdleR`, `SyncA_IdleR` (synchronization of the active connection), `StartA_IdleR`, `WcA_IdleR` (setup of the active connection), `IdleA_WsoR` and `WdisA_WsoR` (switch-over). No connection is currently set; the two stations cannot exchange data; the CM tries to setup the active connection.

Operative (without backup). It involves the states `DataA_IdleR`, `DataA_SyncR` (synchronization of the backup connection), `DataA_StartR` and `dataA_WcR` (setup of the backup connection). Only the active connection is set; the CM tries to set up also the backup connection. The two stations can exchange data along the active connection; if no data telegram is to be sent, the active connection is monitored by sending connection monitoring (`CM`) telegrams. If the active connection is dropped, the channel becomes non-operative, as there is no backup connection.

Operative (with backup). It involves only the state `DataA_StandbyR`. Both the active and backup connection are set. The two stations can exchange data along the active connection; if no data are to be sent, the active connection is monitored by sending connection monitoring (`CM`) telegrams; the backup connection is also periodically monitored by sending `CM` telegrams. If the active connection is dropped, the backup connection becomes active (switch-over) and the channel keeps operative (without backup).

The transitions are grouped into eight main functionalities:

(1) Re-synchronization of the active (resp. backup) connection. From the initial state `IdleA_IdleR` (resp. `DataA_IdleR`), before starting the setup of the active (resp. backup) connection, the CM enters the state `SyncA_IdleR` (resp. `DataA_SyncR`) where it waits for a fixed amount of time. During this period all the commands coming from both connections (resp. from the backup connection) are discharged. This removes all the information related to the previous connection and leaves enough time to the remote station to detect the disconnection and to drop the connection too. After the re-synchronization period, the CM enters the state `StartA_IdleR` (resp. `DataA_StartR`).

(2,3) Connection Setup These transitions allow the CM to set up new connections. The modalities depend on the CM (whether it is initializer or not), and on the connection (whether it is to be active or backup).

(4,5) Data send and receive. The CM is in one of the operative channel states (`DataA_XxxxR`) When it receives a data slot to send from the application, it sends a data command to the active SL and waits. When the CM receives the data confirm command, it provides a confirmation to the application. When it receives a data indication command from the active SL, it passes to the application the data slot received.

(6) Drop of one connection. The CM receives a disconnection indication command from one of two SL's. If the command comes from the active SL (Transitions (6')) and the CM is not in the state `DataA_StandbyR`, then the CM drops the active connection (if any) and enters the initial state `IdleA_IdleR`. If

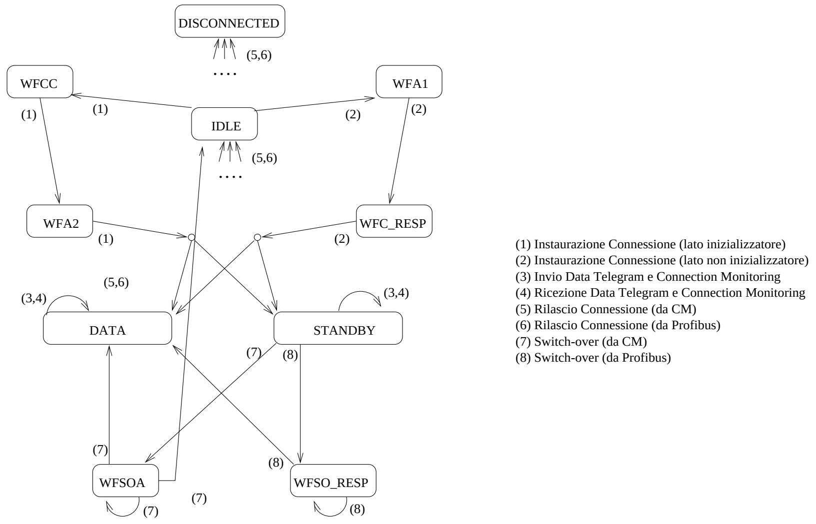

Fig. 4. A simplified view of the SL machine.

the command comes from the backup SL (Transitions (6")) and the CM is in one of the operative channel states, then the CM drops the backup connection (if any) and enters the state `DataA_IdleR`; if the CM is not in one of the operative channel states, the command is ignored.

(7,8) Switch-over. The CM is in `DataA_StandbyR` (channel operative with backup). If the CM receives a disconnect indication command from the active SL (Transitions 7), then it sends to the backup SL a switch-over request command and waits in the state `IdleA_WsoR`. When the backup SL replies with a switch-over confirm command, the CM swaps the roles of the two SL's —so that the former backup connection becomes active— and enters the state `DataA_IdleR`. In the state `IdleA_WsoR` the CM may also receive from the backup SL a switch-over indication command; this happens whenever also the other station detects the disconnection and initiates the switch-over (double switch-over). Then the CM replies with a switch-over response command, remaining in the state `IdleA_WsoR`. The CM can also receive a switch-over indication from the backup SL, in which case activates the suitable switch-over procedure (Transitions 8).

4.3 Overview of the SL Machine

Figure 4 represents (a simplified view of) each SL machine. The transitions are grouped into eight main functionalities:

(1) Connection setup (initializer side). The SL is in the state `IDLE` (non connected). When it receives from its CM a connection request command —indicating the role of the connection: active or backup— it sends via ProFiBus a telegram `CR` (connection request) and waits for a reply in the state `WFCC`. When it receives back the telegram CC (connection confirm), it sends via ProFiBus a telegram A1 (authentication) and waits for a reply in the state WFA2. When it receives back the telegram A2 (authentication acknowledge) it performs an authentication test: if everything is correct, the connection is setup, and the SL enters the state DATA (active) or STANDBY (backup) depending to the role assigned.

(2) Connection setup (non-initializer side). The SL is in the state IDLE (non connected). When it receives via ProFiBus a telegram CR, it replies with a telegram CC and waits in the state WFA1. When it receives via ProFiBus a telegram A1, it replies with a telegram CC and waits in the state WFA1. When it receives back the telegram A1 it performs an authentication test: if everything is correct, it informs its own CM by a connection indication command and waits in the state WFC_RESP. When it receives back a connection response confirm — indicating the role of the connection: active or backup— it sends via ProFiBus the telegram A2. The connection is setup, and the SL enters the state DATA (active) or STANDBY (backup) depending to the role assigned.

(3) Data and Connection Monitoring send. In case of active connection, the SL handles each send data request command received from its CM by sending a data telegram (DT) via ProFiBus containing the data received; the acknowledge received from the ProFiBus is passed to the CM by a data confirm command; when there is no data to be sent within a certain time slot, the SL sends a connection monitoring telegram (CM). In case of backup connection, the SL sends CM telegrams at fixed intervals.

(4) Data and Connection Monitoring receive. In case of active connection, the SL handles each data telegram DT received via ProFiBus by sending to the CM a data send confirm command containing the data received; moreover, the SL monitors the receipt of at least one DT or CM every time slot. In case of backup connection, the SL just monitors the receipt of at least one CM every time slot. In both cases, if the connection monitoring test fails, the connection is dropped.

(5) Connection drop (from CM) The SL may receive from its CM a disconnect request command. If so, the SL drops the connection. Depending on some information contained in the command and on the current state, the SL can enter either the state IDLE (temporary disconnection: another connection can be setup) or the state DISCONNECTED (permanent disconnection: no new connection is possible); it can either send via ProFiBus a disconnect telegram (DI) or not; it can either acknowledge the disconnection by a disconnection indication command or not.

(6) Connection drop (from ProFiBus) The SL may receive via ProFiBus a disconnect telegram DI. If so, the SL drops the connection. Depending on some information contained in the telegram, the SL can enter either the states IDLE or DISCONNECTED; it can either inform the CM by a disconnection indication command or not.

(7) Switch-over (from CM). The SL is in the state STANDBY (backup connection). If it receives from the CM a switch-over request command then it sends via ProFiBus a switch-over indication telegram (SOI) and waits for a reply in the state `WFSOA`. When it receives back a switch-over acknowledgment telegram (`SOA`), it informs the CM with a switch-over confirm command and enters the state `DATA`: the new active connection is set. In the state `WFSOA` the SL may also receive via ProFiBus a `SOI` telegram (double switch-over). If so, the SL simply replies with a `SOA` telegram, remaining in the state `WFSOA`.

(8) Switch-over (from ProFiBus). The SL is in the state `STANDBY` (backup connection). If it receives via ProFiBus a switch-over indication telegram (`SOI`), then it informs its CM with a switch-over indication command and waits for a reply in the state `WFSO_RESP`. When it receives back a switch-over response command, it sends via ProFiBus a switch-over acknowledgment telegram (`SOA`) and enters the state `DATA`. In the state `WFSO_RESP` the SL handles the situations of double switch-over; if it receives from the CM a switch-over request command, then it sends via ProFiBus a switch-over indication telegram (`SOI`) and remains in `WFSO_RESP`; if it receives back a switch-over acknowledgment telegram (`SOA`), it informs the CM with a switch-over confirm command, and remains in `WFSO_RESP`; the same happens if the SL is already in the state `DATA`.

The telegrams `CR`, `CC`, `A1` and `SOI` are acknowledged by the receipt of the telegrams `CC`, `A1`, `A2` and `SOA` respectively: if one telegram is not acknowledged within a certain time slot, the connection is dropped.

In order to send telegram to the partner station, each SL uses the ProFiBus service "Send Data with Acknowledgment" [8], which provides a positive (resp. negative) acknowledgment whenever the telegram sent is successfully delivered (resp. is not delivered) to the remote SL. Thus, each time a SL sends a telegram via ProFiBus, it waits for the acknowledgment in a proper intermediate state: if the acknowledgment is positive, the remaining part of the transition is processed, otherwise the connection is dropped. Notice that the intermediate states can not be interleaved: when the SL is in a intermediate state, any event incoming is temporarily saved and its processing is postponed to the next state. This corresponds to consider the "send with acknowledge" phase as an atomic operation. (Notationally, the positive and negative acknowledgment are denoted by `pb_ack` and `pb_nak` respectively; the intermediate states are called *acknowledgment states*, and are denoted by the suffix "`_ACK`"; the states represented in Figure 4 are called instead the *main states*.)

4.4 An example of transition diagram

To provide an example of an SDL process diagram, in Figure 5 we report the SDL description of the functionality (8) of the SL "Switch-over from ProFiBus". (Notationally, as it is standard practice in protocol theory [9], the service primitives of the protocols —called "commands" here— have been classified into *request* (`.req`), *indication* (`.ind`), *response* (`.resp`) and *confirm* (`.cnf`) *commands*. The SDL notation is rather intuitive and will be clear in the description below.) By "the SL receives correctly a telegram" we will mean that the SL receives a telegram, it tests both the CRC and the sequence number of the telegram and the result is positive (condition `Crc#seq_ok`); by "the SL sends correctly a telegram" we will mean that the SL sends the telegram and waits in a proper

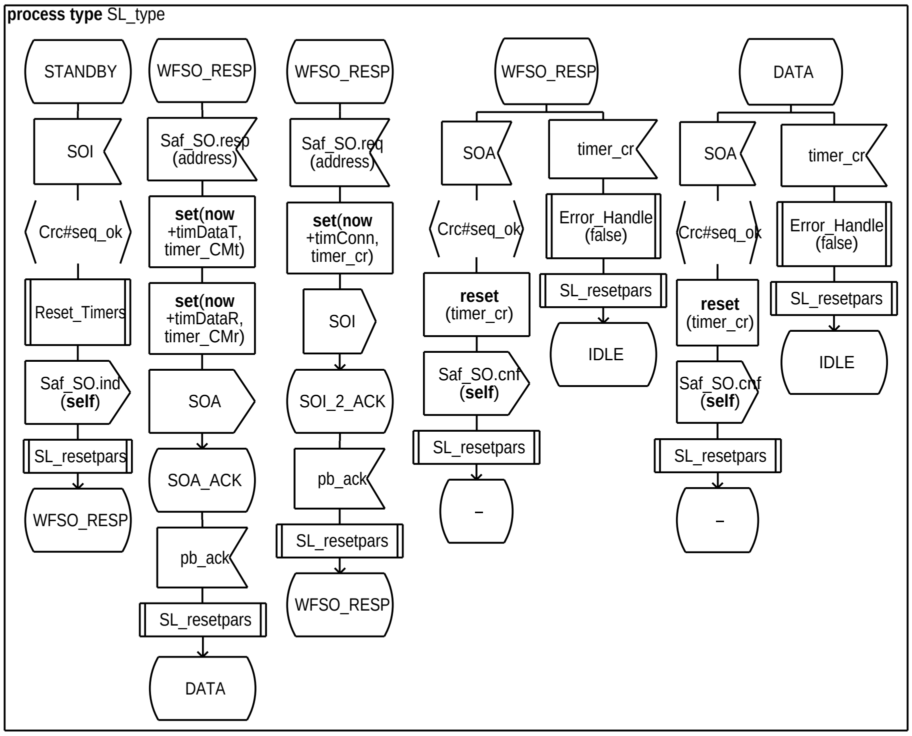

Fig. 5. An example of the SDL specification of SL transitions: (8) Switch-over from ProFiBus.

acknowledgment state "_ACK" until it receives a `pb_ack`. (The unsuccessful send and receipt of telegrams are described in other diagrams.) The procedure call `SL_resetpars` at the end of the transitions resets the values of the fields of input telegrams and commands; it will be ignored in the description. A state labeled with "–" means "the previous state". The timers `timer_CMt`, `timer_CMr` and `timer_cr` check respectively the connection monitoring —transmission and reception— and the acknowledgment of the telegrams `CR`, `CC`, `A1` and `SOI`, as described above.

The transitions represented in Figure 5 can be described as follows:

1. The SL is in the state `STANDBY` (backup connection) and receives correctly via ProFiBus a switch-over indication telegram `SOI`. Thus it resets all timers (procedure call `Reset_Timers`), it sends to the CM a switch-over indication command `Saf_SO.ind` and waits for a reply in the state `WFSO_RESP`.
2. When the CM replies with a switch-over response command `Saf_SO.resp`, the SL sets the timers `timer_CMt` and `timer_CMr` to some proper values, sends successfully via ProFiBus a switch-over acknowledgment telegram `SOA` and

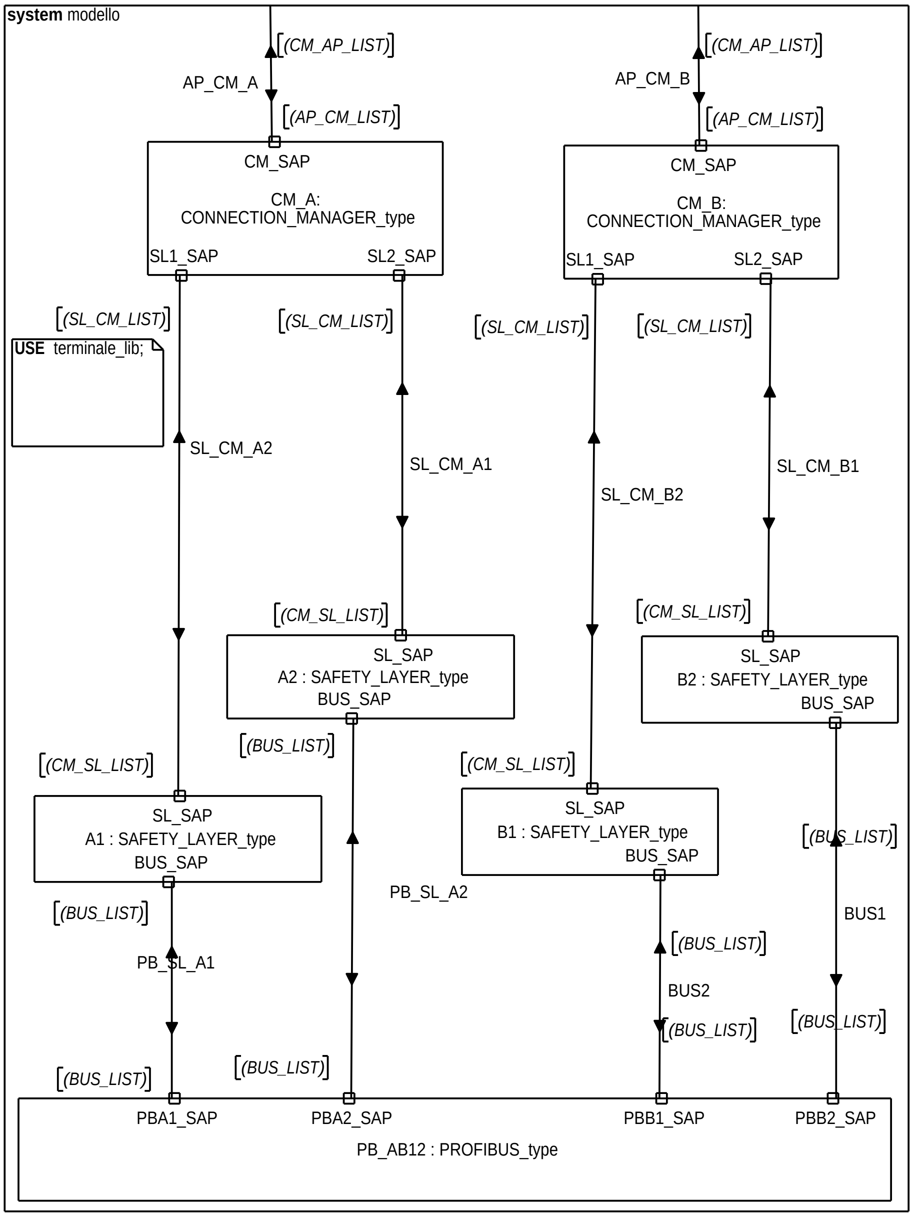

Fig. 6. The SDL interconnection diagram of the simulated model.

enters the state `DATA`. (The new active connection is now set, so that the SL can send and receive data and connection monitoring telegrams.)

3. It may be the case that, in the state WFSO_RESP, the SL receives from the CM a switch-over request command Saf_SO.resp (double switch-over). If so, the SL sets timer_cr to a proper value, it sends successfully the switch-over indication telegram SOI waiting in WFSO_RESP both the switch-over response command from the CM and the telegram SOA from the remote SL.
4. The SL can receive via ProFiBus a telegram SOA in the state WFSO_RESP. (This happens in case of double switch-over, if the telegram SOA is received before the command Saf_SO.resp.) If so, it simply sends to the CM a switch-over indication command Saf_SO.ind and remains in the same state. If timer_cr triggers before the reception of the telegram SOA, the SL considers it as an error, invokes the procedure Error_handle and drops the connection, entering the state IDLE.
5. The SL can receive via ProFiBus a telegram SOA also in the state DATA. (This happens in case of double switch-over, if the telegram SOA is received after the command Saf_SO.resp.) If so, the SL behaves exactly as in the previous situation.

5 Formal Validation

During the project, interactive and exhaustive simulations were performed on the model configured as depicted in Figure 6. (In this model each block is built by one single process; thus from now on we will use the words "block" and "process" as synonyms.) The model represents a point-to-point channel AB between two partner stations A and B. The station A consists on a CM block CM_A and two SL blocks A1 and A2; the station B consists on a CM block CM_B and two SL blocks B1 and B2. (The names of the gates and of the message lists are analogous to the ones in Figure 2.) A1 and B1 (resp. A2 and B2) exchange telegrams through the connection AB1 (resp. AB2), which they setup and monitor. A1 and A2 exchange telegrams with B1 and B2 respectively by means of a block PB_AB12 representing the ProFiBus layer. PB_AB12 is a non-deterministic process: when one of the SL process sends a telegram, PB_AB12 can either send the telegram to the partner SL process and return pb_ack to the source SL, or return pb_nak to the source SL without sending the telegram to the partner SL.

The development and validation of the model has been a cyclic process: at each cycle the model was simulated by running the simulator/model checker of ObjectGEODE in both interactive and exhaustive mode; the errors or anomalous behaviors revealed were traced by means of MSC's and analyzed; then a new version of the model was produced which included the modifications suggested by the analysis. All simulations have been performed on a SUN SPARC10 workstation with 128MB RAM.

5.1 Interactive simulation

The first step of every simulation cycle was to run the simulator in interactive mode, in order to verify by hand the nominal behavior of the system and some typical scenarios. For instance, we have simulated interactively standard

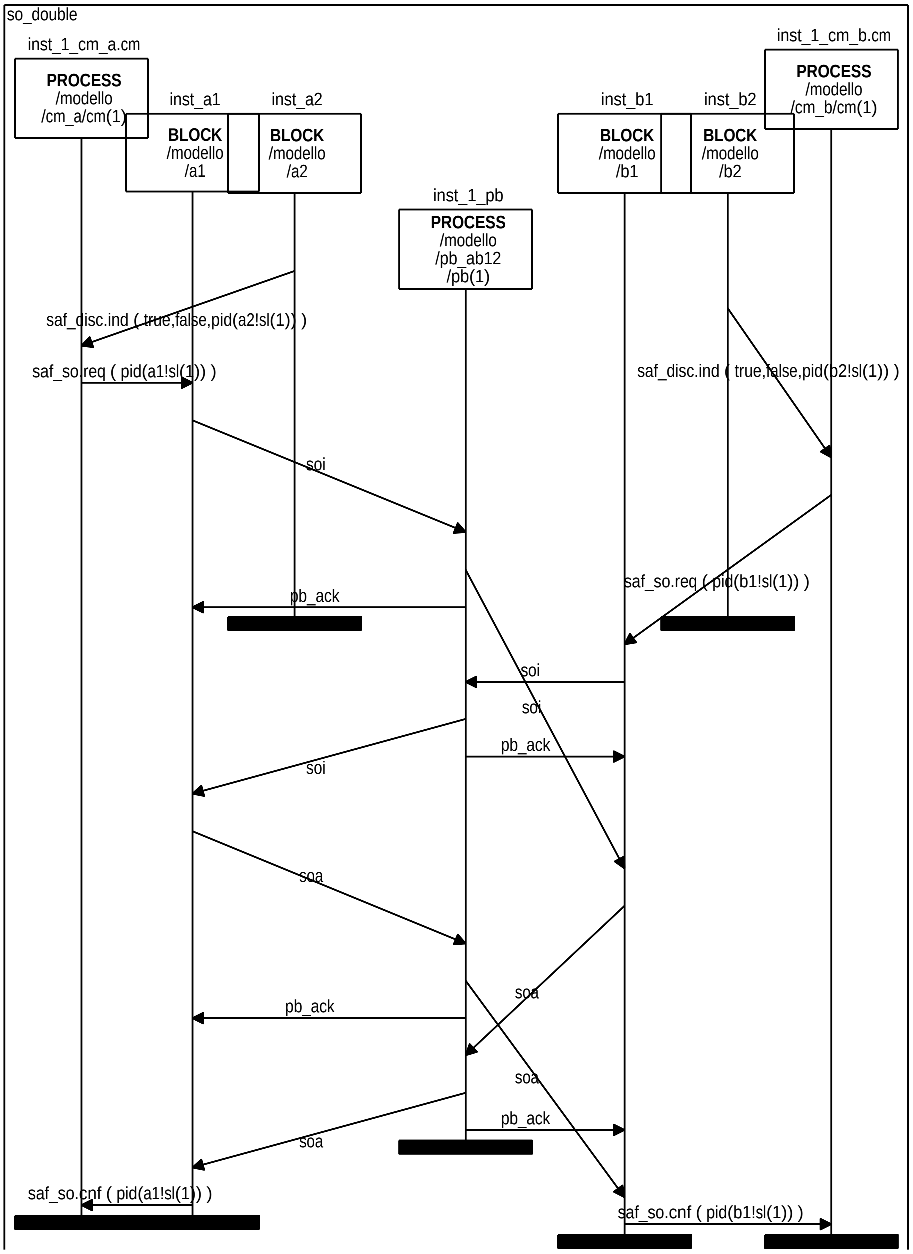

Fig. 7. The MSC of the double switch-over.

situations like setup of both active and backup connection, data and connection monitoring exchange, drop of active and/or backup connection, single and double switch-over, plus some scenarios of typical error handling (CRC errors, ProFiBus errors, channel delays, etc.). The corresponding MSC's have been produced and analyzed.

As an example, the MSC so_double of Figure 7 describes the execution of a scenario of double switch-over. The MSC notation is rather intuitive: the vertical lines represent, from left to right, the seven blocks/processes `CM_A`, `A1`, `A2`, `PB_AB12`, `B1`, `B2` and `CM_B`; the arrows represent the messages exchanged. The MSC has been obtained by simulating interactively the final version of the model. At the starting point, the channel is operative with backup, `AB2` is the active connection and `AB1` the backup one, so that `CM_A`, `CM_B` are in `DataA_StandbyR`, `A2`, `B2` are in `DATA` and `A1`, `B1` are in `STANDBY`. (In the following description all telegrams are sent with success, so that we omit to mention the `pb_ack` messages; we also omit the description of the values of the fields of the commands and telegrams because they are not relevant in this context.) Each active SL `A2` and `B2` reveals a problem on the active connection (e.g., a failed connection monitoring). Then it informs its own CM with a disconnection indication command `Saf_DISC.ind`. Each CM starts the switch-over phase by sending a switch-over request command `Saf_SO.req` to its own backup SL. The latter sends via ProFiBus a switch-over indication telegram `SOI` and waits for the switch-over acknowledge telegram `SOA`. Instead it receives first the `SOI` telegram from the other SL (double switch-over) and simply replies with a `SOA`. When it finally receives the `SOA`, it informs its own CM by a switch-over confirm command `saf_SO.cnf`. Then each CM swaps the roles of its SL's. The connection `AB1` is now the active one, and the CM can start the setup of the backup connection `AB2`.

5.2 Model checking

The second step of every simulation cycle was to run the simulator in exhaustive mode (model checking). This was a very effective form of *early debugging* for the system, as the model checker found automatically a large number of errors or unexpected behaviors. Furthermore, most of these problems were of such a complicate nature to be nearly impossible for a human analyst to conceive.

During this phase, we have pervasively used most of the optional features of the simulator/model checker of ObjectGEODE, in particular:

– different choices of search strategy. For instance, depth-first search allows for detecting strongly connected cycles, while breadth-first search allows for generating error scenarios of minimal length;
– state compression, to reduce memory occupation;
– *filtering*, to hinder the execution of some selected transitions, thus allowing for a selective analysis of the various functionalities of the model.

An example of early debugging In an intermediate version of the SDL model, the last couple of transitions of the diagram in Figure 5 —input of `SOA`

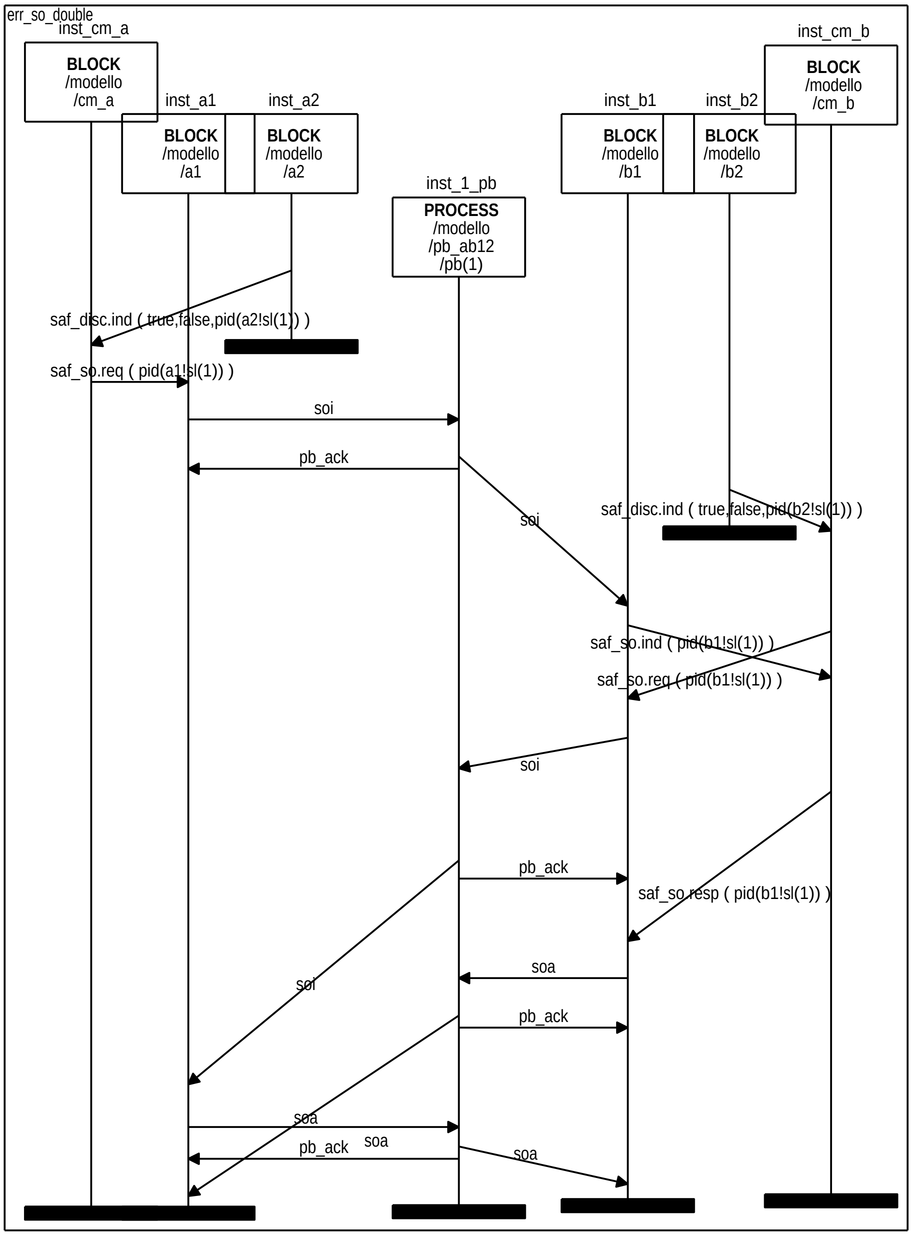

Fig. 8. The MSC of an error occurred during a double switch-over (revealed in an intermediate version of the model).

and timer_cr in the state DATA— was not there, and timer_cr was not used in WFSO_RESP. (Intuitively, this was due to the assumption —which revealed incorrect— that no switch-over telegram could ever be received by an active SL.) During an exhaustive simulation, the model checker stopped for a "false input" error: the SL B1 had received an unexpected input of a SOA telegram while it was in the state DATA.

The MSC derived from the error scenario generated is reported in the MSC of Figure 8. (The starting conditions and the assumptions are the same as for the MSC so_double of Figure 7.) The active SL A2 reveals a problem on the active connection and informs CM_A with a disconnection indication command Saf_DISC.ind. CM_A sends a switch-over request command Saf_SO.req to its own backup SL A1, which sends via ProFiBus a switch-over indication telegram SOI to the remote SL B1. B1 informs CM_B by a switch-over indication command Saf_SO.ind, and waits for the switch-over response command Saf_SO.resp in the state WFSO_RESP (first transition of Figure 5). In the meanwhile also the SL B2 reveals a problem on the active connection and informs CM_B with a disconnection indication command Saf_DISC.ind; CM_B reacts by sending a switch-over request command Saf_SO.req to B1. B1 receives the Saf_SO.req and sends via ProFiBus a telegram SOI to A1 (third transition of Figure 5). CM_B replies to the command Saf_SO.ind with a switch-over response command Saf_SO.resp. B1 receives it, sends via ProFiBus a telegram SOA and enters the state DATA (second transition of Figure 5). A1 replies to the SOI with a telegram SOA. Unfortunately —as the last couple of transitions of Figure 5 is missing— B1 is not prepared to receive a telegram SOA in the state DATA. This causes an error.

Notice that the error happens because the Saf_SO.ind and the Saf_SO.req "cross" between B1 and CM_B, so that B1 receives the Saf_SO.req in between the exchange Saf_SO.ind/Saf_SO.resp. In an implemented version of the system, it is reasonable to assume that the communications between CM and SL are much faster than those via ProFiBus, so that the probability of a similar crossing is very low. Therefore this error would have been extremely hard to detect during the testing phase of the implemented system.

Final validation

We focus now on three very intensive exhaustive simulations we performed of the final version of the model. No transition has been filtered, so that all functionalities are analyzed. Using the state compression allowed by the model checker, we could impose a bound of $3 \cdot 10^6$ reachable states. [1] In the first two simulations the model checker was run with a breadth-first search strategy, starting from the situations "channel non operative" and "channel operative with backup" respectively; in the third simulation the model checker was run with a depth-first search strategy. Each simulation required several hours of CPU time.

[1] Empirically, $3 \cdot 10^6$ was the biggest number of states we could handle with 128M RAM.

The three simulations shared the following results. [2]

```
verify stopped by states limit
Number of exceptions: 0
Number of deadlocks: 0
Number of stop conditions: 0
Transitions coverage rate: 75.00 (94 transitions not covered)
States coverage rate: 100.00 (0 states not covered)
(...)
```

None of the three simulations has been able to cover the whole state space (that is, the number of reachable states is greater than $3 \cdot 10^6$). They have covered respectively 9, 10 and 13 million transitions. However, it is worth remarking that, when an error was detected in intermediate versions of the model, this was always done within up to a few thousands states analized. Since the final runs signaled no exception (dynamic error), deadlock, or stop condition (unexpected inputs, role incongruences of the SL's), the degree of confidence on the correctness of the design was considered to be rather high.

All simulations have covered the 100% of the control states and about 75% of the transitions. Furthermore, all the uncovered transitions of the CM's had been introduced to fill the matrix event/state, and were not expected to be covered:

```
from_wdisa_wsor_input_all: 0
from_idlea_wsor_input_all: 0
from_idlea_idler_synca_idler_starta_idler_wca_idler_wdisa_wsor_
idlea_wsor_input_cm_put.req: 0
from_dataa_startr_dataa_wcr_input_all: 0
from_starta_idler_wca_idler_input_all: 0
from_dataa_syncr_input_all: 0
from_dataa_idler_input_all: 0
from_idlea_idler_input_all: 0
(...)
```

(The SL's filtered the unexpected telegrams and thus they did not send anomalous commands to their CM.)

The explosion of the state space was to be expected: industrial systems tend to explode. Although several tricks were used to contain the state explosion — like, e.g., filtering and state compression— it was impossible to complete the exploration because the model is actually infinite state in principle. This is due to the fact that no rigid synchronization is imposed on the blocks of the system, and that the length of the queues is not limited. Another fancier model checker (e.g., SPIN [5]) could be applied with profit. In particular, the partial order reduction appear to be particularly promising in case of asynchronous composition of processes in the same site.

[2] The simulator's outputs have been slightly reformatted in order to fit into the page frame.

6 Conclusions

In this paper we presented an application of formal methods to the design of a complex communication protocol for distributed, safety critical systems. During the project, the protocol was formally specified as composition of several finite state machines. Model checking techniques were used to automatically analyze the specifications, and pinpoint several subtle errors and unexpected behaviours in early stages of the design.

It is well known that formal methods are not a panacea, and may have drawbacks if not applied in the right way. The quantification of the costs and benefits of formal methods is not easy. For this particular project, however, it is fair to say that the application of formal methods was effective. The specification of the protocol has been judged to be of high quality, it is independent of the architecture, and imposes very weak constraints on the particular combination of the SL and CM machines, thus allowing for different architectural choices at implementation time. An informal specification of the protocol would have been hard, and its manual validation would have been nearly impossible without the support of model checking. The pay-off from the application of formal methods is also in the further availability of an executable model. This can be used for mechanically assisted test design, and provides a basis for an easy and early validation of future modifications. The impact of the use of formal methods in this project is very significant. The formal specifications are currently used to implement two different versions of the safety layer, running on two different hardware platforms.

References

1. J. Bowen. Formal Methods in Safety-Critical Standards. Oxford University Computing Laboratory Technical Report, 1995.
2. J. Bowen. The Industrial Take-Up of Formal Methods. Oxford University Computing Laboratory Technical Report, 1995.
3. A. Cimatti, R. Sebastiani, and P. Traverso. Specifica formale dei protocolli Safety Layer e Connection Manager (Formal specification of the Safety Layer and Connection Manager protocols). In italian. ITC-IRST deliverable 9808-02, project Safety Critical Applications III - SCAPIII, January 1999.
4. European Commitee for Electrotechnical Standardization. European Standard - Railway Applications: Software for Railways Control and Protection Systems. EN 50128, 1995.
5. G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.
6. ITU-T. *CCITT specification and description language (SDL)*, March 1993. ITU-T Recommendation Z.100.
7. ITU-T. *Message Sequence Chart (MSC)*, October 1996. ITU-T Recommendation Z.120.
8. Profibus Nutzerorganization. *Profibus Standard*, July 1996. DIN 19 245.
9. A. Tanenbaum. *Computer Networks.* Prentice Hall, 1989.
10. VERILOG. *ObjectGEODE Documentation.* Available at *www.verilogusa.com.*