A Formal Specification and Verification of a Safety Critical Railway Control System

S. Gnesi D. Latella G. Lenzini C. Abbaneo
IEI - CNR * CNUCE - CNR † IEI - CNR * ASF - GENOVA ‡

A. Amendola P. Marmo
ASF - NAPOLI § ASF - NAPOLI §

March 7, 2000

Abstract

This paper describes an experiment in formal specification and validation performed in the context of an industrial joint project involving Ansaldobreda Segnalamento Ferroviario (ASF) and CNR Institutes - IEI and CNUCE - of Pisa. Within this project two formal models have been developed, describing different aspects of a wider safety-critical system for the management of medium-large railway networks. Validation of safety and liveness properties has been performed on both models. More specifically safety properties have been checked also in presence of byzantine behavior as well as other kinds of faults embedded in the models themselves. Liveness properties have been more focused on a communication protocol used within the system. Properties have been specified by means of assertions or temporal logical formulae. We used PROMELA as specification language, while the verification was performed using SPIN.

Keywords: safety-critical systems, dependable protocols, formal verifications, model checking.

*Istituto Elaborazione dell’Informazione CNR - Via S. Maria 46, 56123 PISA (Italy) tel:+39 050 593 489/485 - fax:+39 050 554 342, {gnesi, lenzini}@iei.pi.cnr.it
†CNUCE Institute CNR - Via S. Maria 40, 56123 PISA (Italy) tel:+39 050 593 230 - fax:+39 050 904 052, (d.latella@cnuce.cnr.it)
‡Ansaldobreda Segnalamento Ferroviario - Via dei Pescatori 35, 16129 GENOVA (Italy) tel:+39 010 655 2317 - fax:+39 101 655 2444, (cabbaneo@asf.atr.ansaldo.it)
§Ansaldobreda Segnalamento Ferroviario - Via Argine 425, 80147 NAPOLI (Italy) tel:+39 081 243 2621/2982, {amendola, marmo}@atr.ansaldo.it
1 Introduction

The increasing request of safety and better performance in the field of modern railways and metropolitan networks has forced the introduction of sophisticated dependable control software and hardware in the automatic management of railways systems. These systems have a high degree of complexity, and require innovative validation techniques during both their design and developing phases. Traditional techniques, such as testing and simulation, could be insufficient when applied to this kind of systems. Exhaustive testing is usually ineffective because of the high number of running sequences to be analyzed, while the definition of of crucial tests is very hard due to the possibility of subtle behaviors. Simulation can provide useful information only on a limited (often already predicted) sequences and, due to the intrinsic lack of continuity, it is actually impossible to infer any global conclusion on the simulation paths not checked.

Despite of these technical difficulties industries are undoubtedly interested in discovering, in all their dependable applications, as many errors as possible before entering the production phase: during this last stage, in fact, the cost of correction per error increases enormously, while unpredictable errors can nullify months of hard work. In addition, usually governments, institutions and generally customers often require, as a guarantee of quality, accurate documentation about reliability features of a dependable system they are going to acquire. Finally, international standards have been studied to define precise safety and quality certificates (e.g. EN 50128 CENELEC Railways Applications [19], or IEC 65108 [12]). It worth pointing out that these standards strongly suggest Formal Methods (FM) for validation.

In fact, FM are widely recognized as fault avoidance techniques that can increase dependability by removing errors during the specification of requirements and during the design stages of development [3]. FM can be used to study the safety of a system by formally verifying that certain safety properties holds on a model of a system. In addition most FM approaches are well suitable to be mechanized and a great variety of tools for automatic validation are nowadays available.

On the other hand the use of FM introduces additional costs that cannot be ignored in industry management. In theory, the use of FM drastically lowers the developing cost of a system [15]; in practice, industries would carefully evaluate the cost/benefits ratio which can derive from changing a well established developing schedule. The integration of a new formal analysis step, in fact, requires either the intervention of specialized professionals, expert in the theory and tools of FM, or training of internal engineering teams. At the end additional series of validation steps are required to validate the real
system implementation respect to its formal specification, or more in general, the consistency of the implementation of design with its formal model.

In the last decade many industries, like Ansaldobreda Segnalamento Ferroviario (ASF), started pilot projects [8, 14, 17, 2] directed to evaluate the impact of FM on their production costs. As a result, positive experiences [1, 4] have shown how, for railway control systems, it could be possible to formalize significant models and to perform verification using the model checking [6, 20, 5] approaches.

In this paper we describe the results of a real project jointly carried out by ASF and CNR Institutes - IEI and CNUCE - in the context of the Pisa Department Computer Center of Consorzio Pisa Ricerche. The whole project consisted of two distinct parts: (a) designing a formal model of the behavior of a critical control system used in medium-large scale railway stations (b) verifying specific safety properties under the hypothesis of byzantine behavior [13] of one of the system component, and verifying general liveness properties on a dependable communication protocol developed by ASF. In this paper we focus more on the major results of the validation effort, only recalling the main modeling issues. Further details on the latter can be found in [7].

Industrial choices internal to ASF induced us to use PROMELA [10] to specify distributed asynchronous systems and express general correctness requirements. As a verification tool we used the SPIN [11] model checker.

The paper is organized as follows: in Section 2 we briefly and informally describe the system and all its component units; in Section 3 we recall the most important features of PROMELA and SPIN; in Section 5 and Section 7 we explain the PROMELA processes we defined and used as formal models, and the properties verified with some significant results; finally in Section 8 we critically discuss on the whole experience.

2 System Description

The railway system considered in this work was the Computerized Central Apparatus (ACC) [16], a control system - developed by ASF - responsible of the supervision of railway stations. The ACC is an highly programmable centralized control system, which constitutes the core of a node in a distributed architecture specifically designed to manage a large railway network. The ACC can be instantiated onto particular railway realities, and it is devoted to the control of a medium-large railway station, or a line section with small stations, or a complete low traffic line with simple interlocking logic. Its architecture (see Figure 1) consists in two subsystems that independently
perform management and vital functions. In particular:

- the vital section (VS) controls train movements and wayside equipment. It consists of a Safety Nucleus (SN), of input/output Peripheral Control Units (PCUs), and of Control Posts.

- the management section, also called RDT (Recording, Diagnosis and data Transmission), is dedicated to auxiliary functions, such as data recording, diagnostic management and remote control interface.

In our experiment we considered only the VS and in particular we focused mainly on the SN and the PCUs. In particular, the SN has been designed for the execution of safe operations, and for control and safety purposes. It is interposed between the Control Posts, from which human operator can digit commands to be sent to the periphery, and the PCUs that, in turn, execute the commands. These commands are considered critical because they affect critical machinery such as railway signal light, rail points, or crossing levels. The SN achieves its purpose of safely delivering critical commands to the periphery by running monitor tasks on the state of the ACC system. In case of software/hardware faults in some components the SN tries to recover a consistent state or to exclude the faulty component. The SN is based on a triple modular redundant [21] configuration of computers which independently run different versions of the same program.
3 PROMELA and SPIN

PROMELA (Process Meta Language) [10] is a modeling language of general applicability introduced to describe distributed systems, communication protocols and, in general, asynchronous process systems. A PROMELA specification, usually called a model, consists in one or more process templates (called also proctype) and in at least one process instantiation. In a proctype a user defines the behavior of a process as an imperative program in a C-like syntax. The language is extended with nondeterministic control constructs, and with communication primitives, in a CSP [9] style. Processes can communicate via asynchronous message passing through buffered channels or shared memory. Rendezvous is modeled by buffers of length zero. In addition any running process can instantiate further asynchronous processes using proctypes.

SPIN [11] is an efficient formal verification tool for checking the logical consistency of a specification given in PROMELA, and it can generate an optimized on-the-fly verification program from a PROMELA model. Technically SPIN translates each PROMELA process template given as input, into a finite automaton. Conceptually a global automaton of a system behavior is obtained by the interleaving product (referred as the space state) of all the automata of the processes composing the system. In practice, efficient representations of the state space are used. PROMELA has been defined in such a way a model is necessary bound and has only finite state behavior. Then, in theory all the correctness properties become formally decidable. In practice, users needs to cope with limitations set by the state size and by computational resources.

SPIN accepts correctness claims specified either in the syntax of standard Linear Temporal Logic (LTL) [18], or as process invariants (using assertions) expressing safety and liveness properties. It can be used as an efficient on-the-fly verifier to check for deadlock presence, assertions violation, progress cycles, unreachable code, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes. Used as a LTL model checker, SPIN supports all correctness requirements expressible in this logic, either directly in the syntax of next-time free LTL, or indirectly as Büchi Automata\(^1\).

\(^1\)Further information on PROMELA and SPIN can be find at the official URL: http://cm.bell-labs.com/cm/cs/what/spin/
4 Formal Specification and Verification

In the following sections we introduce the general structure of our formalization work and the verification properties checked on it. About the formalization we developed two PROMELA models, we called respectively TMR and TMR-PCUs, each describing different views of the SN-PCUs system\textsuperscript{2}. In particular:

1. the TMR model has been designed to describe in detail the SN. In this model the PCUs behavior is described more abstractly, and all the details regarding on the communication protocol between SN and PCUs have been omitted. The TMR model has been reserved to verify safety properties on the triple modular redundant mechanism of the SN in presence of byzantine behavior of one of its components;

2. the TMR-PCUs model has been designed to describe in detail the SN-PCUs communication protocol, and the PCUs themselves. Aspects of the SN behavior not related to the communication protocol have been left out. The TMR-PCUs model has been reserved to verify liveness properties on the SN-PCUs protocol, and safety properties on the same protocol in presence of specified hardware faults in the communication buses or in some of the PCUs.

5 The TMR model

The TMR model describes in detail the triple modular redundant mechanism of the SN. In Figure 2 we report a scheme of the general architecture of the system. We want to point out: (1) the three identical central module, called A, B and C, implementing the triple modular redundancy; (2) a special module called exclusion logic, devoted to checking the consistency of the three modules, and able to disconnect a module; (3) the interconnections between the modules (three symmetric channels), the modules and the exclusion logic (three symmetric channels), and the modules and the PCUs (a single bus); (4) the PCUs composed by \( n \) control units\textsuperscript{3}. Our PROMELA model reflects quite faithfully this general architecture: we reserved a process for each of the central module, a process for the exclusion logic and a process for each PCUs. We defined three symmetric channels of length zero between the

\textsuperscript{2}Indeed a whole model was first considered, but we successively decided to split it because of serious state space dimension problems.

\textsuperscript{3}In our study we have considered \( n = 2 \)
central modules, between the central modules and the exclusion logic. Finally we defined a bus between the central modules and the PCUs.

To have an idea of the difficulties faced within the formalization work and to understand the properties verified on it, we now briefly describe the algorithm run by a central module and the algorithm run by a peripheral unit. We first give a high level description of these algorithms and successively we report some meaningful part of the corresponding PROMELA code.

5.1 A central module

The behavior of each module consists of repeated sequences of *phases*, as described in the following pseudo-code.

```
loop
    * <synchronization>
    * <data exchange with the other modules>
    <distribute voting>
    * <communication to exclusion logic>

{communication with the PCUs}
    for i = 1 to n do
        if <is my turn> then
            <synchronization>
            * <send command to the ith PCU>
        endif
        * <receive acknowledge from the ith PCUs>
    endfor
endloop
```
During each phase a central module runs local computations or communicates with other components of the system (these latter phases are stressed with an *). In particular, in the synchronization phase each module sends to and receives from the other two modules, with time-out\(^4\), a synchronization message. This phase is used to collect information about the activity state of the other modules, and in particular: (1) if a time-out occurs, it is interpreted by the receiver as a sign of inactivity; (2) if two time-out occur the receiver switches in a safe shut-down. In the data exchange phase each module sends to the other modules a message containing its local state. Symmetrically it receives from the other modules, with time out, information about their local states. In the distribute voting each module performs a majority voting using the information received in the previous phase. In the communication with the exclusion logic, the result of the voting is sent to the exclusion logic which can disconnect a module considered potentially faulty. In the communication with the PCUs, a module communicates with the PCUs. At each loop only two modules are selected to send a command to the PCUs: a tournament distributed procedure assures a cyclic selection of the two modules communicating with the periphery.

In developing the PROMELA code we had to solve an important problem: the simulation of a time-out in the communication. In fact PROMELA does not deal with time. As a general solution we defined a particular EMPTY message, whose presence in a channel must be interpreted, by the receiver, as absence of any message it was waiting for. Then, whenever we had a receive action we introduced additional code finalized to discern, depending on the message content, if a time-out has been expired. The send action changed too: it has been implemented a non deterministic choice between either transmitting the “real” message or transmitting the EMPTY message. In the following we report a synthesis of the PROMELA code implementing the synchronization phase for the module A:

```
/** In the global environment ***/

#define EMPTY 0 // the empty message
#define SYNCH 1 // the synchronization message

d_step{
    /* i = active */
    activeB = 1; // local (for the module A) state of the module B
    activeC = 1; // local (for the module A) state of the module C
}

\(^4\) Most of communications in the ACC are with time-out; moreover because all the channels are supposed to have no memory, a message sent in delay is to be considered lost.
/** Synchronization phase **/ */--- inB,inC: input channels from module B,C */ */--- outB,outC: output channel to module B,C */} d_step{ sentB = 0; /* flag "sent" to module B */ recvB = 0; /* flag "received" from modules B */ sentC = 0; /* flag "sent" to module C */ recvC = 0; /* flag "received" from modules C */ } atomic{ do :: (!sentB) -> if // send the synch message if module B is active */ :: true -> outB(SYNCH && activeB); // send the empty message :: true -> outB(EMPTY); fi; sentB = 1; :: (!recvB && inB? [synB]) -> inB? [synB]; recvB = 1; :: (!sentC) -> if // send the synch message if module C is active */ :: true -> outC(SYNCH && activeC); // send the empty message :: true -> outC(EMPTY); fi; sentC = 1; :: (!recvC && inC? [syn1]) -> inC? [synC]; recvC = 1; :: (sentB && sentC && recvB && recvC) -> if :: synB == SYNCH -> activeB = 1; // if a time-out occurred the module // is considered not active :: else -> activeB = 0; fi; fi

9
:: sync == SYNCH -> activeC = 1;
// if a time-out occurred the module
// is considered not active
:: else -> activeC = 0;
fi;

/* Conditional jump to the code implementing a safe shutdown*/
atomic{
if
/* if the other modules are recognized not active */
:: !activeB && !activeC ->
global_activeA = 0; \ global state of module A
  goto SHUTDOWN
:: else -> skip
fi;
}

The PROMELA code implementing the other phases is similar to the one of synchronization, except for the type of messages involved or for some local computation.

More interesting is the implementation of a byzantine behavior, in order to model a situation in which the failure in one module may cause conflicting information to be sent to the other modules. In this context, byzantine behavior is to be intended as it was in Lamport et al. [13]: a byzantine module runs the same algorithm of a loyal module, but it can arbitrarily fail in executing it, and in particular it may send wrong messages, or send a message delayed respect to a synchronization, or send no message at all. In this interpretation of byzantine behavior, we focused the attention on communication events: all the communication phases (tagged with a * in the pseudo-code) have been realized in a byzantine version, where a communication error in sending a message may be possible. A communication error has been modeled as either a communication of a corrupted message, or as a delayed communication, or as no communication at all. In the following we report the PROMELA code used to implement the synchronization phase of module C, supposed to be affected by byzantine errors:

/*** In the global environment ***/
define EMPTY 0 // the empty message
define SYNCH 1 // the synchronization message

  /* 1 = active */
d_step{
activeA = 1; // local (for the module C) state of the module A
activeB = 1;  // local (for the module C) state of the module B
}

/**
   * Synchronization phase
   **/
   /* - inA, inB: input channels from module A,B */
   /* - outA, outB: output channel to module A,B */
   
   d_step{
     sentA = 0;  /* flag "sent" to module A */
     recvA = 0;  /* flag "received" from modules A */
     sentB = 0;  /* flag "sent" to module B */
     recvB = 0;  /* flag "received" from modules B */
   }

   atomic{
     do
       :: (!sentA) ->
           if
               // send the synch message, if module A is active */
               :: true -> outA(SYNCH && activeA);
               // send the wrong message */
               :: true -> outA(-SYNCH);
               // send the empty message
               :: true -> outA(EMPTY);
               fi;
               sentA = 1;

       :: (!recvA && inA?{synA}) ->
           inA?{synA};
           recvA = 1;

       :: (!sentB) ->
           if
               // send the synch message, if module B is active */
               :: true -> outB(SYNCH && activeB);
               // send the wrong message */
               :: true -> outB(-SYNCH);
               // send the empty message
               :: true -> outB(EMPTY);
               fi;
               sentB = 1;

       :: (!recvB && inB?{syn1}) ->
           inB?{synB};
           recvB = 1;

       :: (sentA && sentB && recvA && recvB) ->
           if
               :: synA == SYNCH -> activeA = 1;
// if a time-out or an error occurred
// the module is considered not active*/
:: else -> activeA = 0;
fi;

fi
:: synB == SYNCH -> activeB = 1;
// if a time-out or an error occurred
// the module is considered not active*/
:: else -> activeB = 0;
fi;

od;
}

/* Eventually safe shutdown */
atomic{
if
/* if the other modules are considered not active */
:: !activeA && !activeB ->
global_activeC = 0; // global state of module C
goto SHUTDOWN
:: else -> skip
fi;
}

5.2 A peripheral control unit

On TMR model the behavior of a peripheral control unit is quite simple: it consists in waiting a commands from two modules, and in returning an acknowledgment back to all the modules. The PROMELA code implementing this simple communication protocol, for one of the peripheral unit (PCU1), is the following:

#define PCU1 <value>

/* loop */
do
::
count == 0; // number of commands received within the current loop

atomic{
do
:: (count < 2) && (bus?[PCU1, sender, cmd]) ->

   // message = (<PCU name>, <module name> <msg>)
   bus?[PCU1, sender, cmd;
   count ++;
   /* send acknowledgment to all the modules */
bus!PCU1, A, ACK;
bis!PCU1, B, ACK;
bis!PCU1, C, ACK;

:: count == 2 -> break
od;
}
/* endloop */

5.3 Formal Verification on TMR

In this section we list some of the most meaningful properties verified on the TMR model and the most meaningful results. Some properties have been formalized as LTL formulae, while the others with PROMELA assertions\(^5\). We used assertions for those properties that could be expressed as an invariant on all the run sequences (i.e., as the modal formula \textit{always} \textit{p}). In the following we assume the module C can show byzantine behavior, while modules A and B are loyal.

\textbf{(TMR1)} after a communication phase it is always true that if two modules do not receive any reply from the third module, this latter module will be eventually disconnected by the exclusion logic;

\[
\Box \ (p1 \rightarrow \Box \ (q1 \rightarrow \leftrightarrow r1))
\]

In the previous formula \textit{p1} stands for “the module A does not receive any message from C”, \textit{q1} stands for “the module B does not receive any message from C” and \textit{r1} “C is disconnected”.

\textbf{(TMR2)} after a communication phase, it is always true that if one module does not receive any reply from the other two modules, it will switch eventually in a safe shut-down state;

\[
\Box \ (p2 \rightarrow \Box \ (q2 \rightarrow \leftrightarrow r2))
\]

In the previous formula \textit{p2} stands for “the module A does not receive any message from C”, \textit{q2} stands for “the module A does not receive any message from B” and \textit{r2} “A jumps to the SHUTDOWN entry label”.

\(^5\)An assertion in PROMELA is a statement including a boolean expression, which is evaluated each time the statement is executed. If the expression evaluates \textit{false} a violation of the correctness requirement is reported.
(TMR3) after a distributing voting phase, it is always true that if two modules, in reciprocal agreement on the global state knowledge, recognize that a third module is not in agreement with them, this latter module will be eventually disconnected by the exclusion logic:

\[
\Box (p_3 \rightarrow \Box (q_3 \rightarrow r_3 \&\& t_3))
\]

In the previous formula \(p_3\) stands for “the module \(A\) and module \(B\) agree on their local states”, \(q_3\) stands for “module \(C\) local states differs from module \(A\) local state”, \(r_3\) “\(C\) is disconnected” and \(t_3\) “\(A\) and \(B\) are active”.

(TMR4) after a communication phase, every module has sent and received a message (eventually the empty message) from the other modules;

\[
\text{assert}(\text{recvB} + \text{recvC} == 2) \&\& (\text{sentB} + \text{sentC} == 2))
\]

The previous assertion has been posed after each communication phase.

(TMR5) if a module is in safe shut-down state then necessarily the other two have caused a time-out in a previous communication phase;

\[
\text{assert}(\text{activeB} + \text{activeC} == 0)
\]

The previous assertion has been posed after the SHUTDOWN entry label.

The verification runs have been performed on different computers (for scheduling needs) by using different optimization options\(^6\) In Table 1 we report on hardware characteristics of our computational resources and Table 2 we report on the results of the verifications.

<table>
<thead>
<tr>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARC St.</td>
<td>SPARC St. 20</td>
<td>AMD K6 200</td>
<td>PC 486 133</td>
</tr>
<tr>
<td>40Mb RAM</td>
<td>64Mb RAM</td>
<td>64Mb RAM</td>
<td>32Mb RAM</td>
</tr>
<tr>
<td>SunOS 4.1.4</td>
<td>SunOS 5.5.1</td>
<td>LINUX RedHat 5.0</td>
<td>LINUX Debian 2.6.27</td>
</tr>
</tbody>
</table>

Table 1: Resources used in the verification work

\(^6\)In particular we used the following compiler options of SPIN: COLLAPSE (CO) to compress the state vector and MA to obtain a minimal automaton encoding.
<table>
<thead>
<tr>
<th>property</th>
<th>PC</th>
<th>state vector</th>
<th>options</th>
<th>RAM (bytes)</th>
<th>depth</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMR1</td>
<td>III</td>
<td>192</td>
<td>CO+MA</td>
<td>22.098</td>
<td>5266</td>
<td>success</td>
</tr>
<tr>
<td>TMR2</td>
<td>IV</td>
<td>192</td>
<td>CO+MA</td>
<td>14.898</td>
<td>5266</td>
<td>success</td>
</tr>
<tr>
<td>TMR3</td>
<td>IV</td>
<td>196</td>
<td>CO+MA</td>
<td>21.029</td>
<td>45273</td>
<td>fail</td>
</tr>
<tr>
<td>TMR4</td>
<td>IV</td>
<td>188</td>
<td>MA</td>
<td>25.170</td>
<td>297515</td>
<td>success</td>
</tr>
<tr>
<td>TMR5</td>
<td>IV</td>
<td>188</td>
<td>CO</td>
<td>9.515</td>
<td>6808</td>
<td>success</td>
</tr>
</tbody>
</table>

Table 2: Verification results on the TMR model

6 Discussion

We briefly discuss the failed properties TMR3. In analyzing the counterexample we noted that the byzantine module C can cause a module B to be disconnected by the exclusion logic. In fact module C causing a time-out in communicating with module B, makes module B to believe that module C is not active. Successively in the distribute voting module B is found in disagreement with A and C and then module B (and not module C) is disconnected by the exclusion logic. This situation depicts a weakness in the exclusion logic mechanism, already found by Ansaldo using traditional simulation techniques.

7 The TMR-PCUs model

The TMR-PCUs describes in detail the SN-PCUs communication protocol, and the PCUs behavior. Target of the protocol is to deliver critical commands also in presence of faults in the communication media. A scheme of TMR-PCUs architecture is reported in Figure 3. We want to stress: (1) the three identical central modules, called module A, B and C, implementing part of the SN; (2) the PCUs composed by $n$ control units\(^7\), each constituted by a configuration of two computers, called A and B; (3) the interconnections among the modules (three symmetric channels), the ones between the modules and the PCUs (two busses).

With the TMR-PCUs model we were interested to verify:

1. liveness properties of SN-PCUs communication protocol in an error-free environment hypothesis. This protocol is implemented as a distributed algorithm designed to assure a cyclic use of the buses and a cyclic selection of the two modules demanded to to send the commands.

\(^7\)In our study we considered $n=2$
2. safety properties of SN-PCUs communication protocol in case of some hardware faults. In particular we were interested in injecting faults in the interconnection buses and in the computers component a peripheral unit.

In our PROMELA code we reserved a process for each central module, and a process for each peripheral unit. We defined a symmetric channel of length zero between each pair of central modules, and two busses between the SN and the PCUs.

We now briefly describe the algorithm run by a central module and the one run by a peripheral unit. The algorithm of the central module is given only in a high level description, while the PROMELA code relative of the algorithm run by a peripheral unit is reported.

7.1 A central module

The behavior of a module can be described as a repeated sequences of phases, as in the following pseudo-code:

```plaintext
loop
  <synthesis>
  {communication with PCUs}
  for i=1 to 2 do
    for j=1 to n do
```
<synchronization>
    <diagnostic>
    <message elaboration>
        if <is my turn> then
            <send message to i-th computer of the j-th PCUs using the selected bus>
        endif
        <receive acknowledge from the i-th computer of the j-th PCUs using the selected bus>
    endfor
    endif
endfor
endloop

In the synthesis phase we have synthesized a possible outcome from phases 1 to 5 of the TMR model: substantially we decide if a module is active, or not active. Before communicating with each computer of each peripheral unit a module tries to infer information about the global state of the system. In particular, in the synchronization phase a module tries to know the other modules activity state, by exchanging a synchronization message. This information is used in the tournament procedure to decide what two modules are selected to send message to the periphery. In the diagnostic phase (which is quite complex in reality), by considering global information collected in a previous loop, a module tries to infer the global state of the PCU computers and of the two buses. Information collected is used to decide which buses to use. In addition, in the message elaboration phase depending on the state of peripheral computers, either the effective peripheral command, or a special diagnostic message is prepared.

7.2 A peripheral unit

In TMR-PCUs the PCUs model is realized in a deeper detail. Its behavior can be synthesized with the following pseudo-code:

    loop
        <decide the state of each of the two busses>
        <decide the state of each of the two computers>

    {communication with the safety nucleus}
    parallel for i=1 to 2 do
        <computer[i] receives a message from bus1 and sends acknowledgments to all the modules>
        <computer[i] receives a message from bus2 and sends acknowledgments to all the modules>
    endfor
endloop
In the **decide the state** phase, a non-deterministic choice is made to
decide on the functional state of the buses and of the computers of the
peripheral unit. In case of state set to “fault” every communication via
the faulty bus or coming from the faulty computer resulted in an expiration
time-out until the end of the loop. In the following we report the **PROMELA**
code used to implement the previous algorithm.

```c
#define DONE recvA1+recvA2+recvB1+recvB2==4
/* loop */
do
::
/* == Initialization of the variables: == */
/* recvA1: counter of messages received by the computer A via bus1 */
/* recvB1: counter of messages received by the computer B via bus1 */
/* recvA2: counter of messages received by the computer A via bus2 */
/* recvB2: counter of messages received by the computer B via bus2 */
/* stateBUS1: the state of bus1 */
/* stateBUS2: the state of bus2 */
/* stateA: the state of computers A */
/* stateB: the state of computers B */
/* decide the state */
d_step
{
  if
    /* fault in the 1st computer */
    :: stateA = 0
    /* 1st computer is ok */
    :: stateA = 1
    /* fault in the 2nd computer */
    :: stateB = 0
    /* 2nd computer is ok */
    :: stateB = 1
    /* fault in the 1st bus */
    :: stateBUS1 = 0
    /* 1st bus is ok */
    :: stateBUS1 = 0
    /* fault in the 2nd bus */
    :: stateBUS2 = 0
    /* 2nd bus is ok */
    :: stateBUS2 = 1
    /* no fault */
    :: else -> skip
  fi
};
RECEIVING:skip;
```
atomic(
    i = 0;
    /* A1 : computer A - BUS 1 */
    /* A2 : computer A - BUS 2 */
    /* B1 : computer B - BUS 1 */
    /* B2 : computer B - BUS 2 */
    do
    :: !DONE & & A1in?[PCU1, senderA1, msg] ->
        A1in?PCU1, senderA1, msg;
        if
            /* if it is a diagnostic message */
            :: msg == HEADER -> skip;
        /* if it is a command message */
        :: else -> msg[i] = msg; i++;
        fi;
    /* acknowledgment to all the module */
    A1out!PCU1,A,(stateA & & stateBUS1);
    A1out!PCU1,B,(stateA & & stateBUS1);
    A1out!PCU1,C,(stateA & & stateBUS1);
    recvA1++;
    :: !DONE & & A2in?[PCU1, senderA2, msg] ->
        A2in?PCU1, senderA2, msg;
        if
            /* if it is a diagnostic message */
            :: msg == HEADER -> skip;
        /* if it is a command message */
        :: else -> msg[i] = msg; i++;
        fi;
    /* acknowledgment to all the module */
    A2out!PCU1,A,(stateB & & stateBUS1);
    A2out!PCU1,B,(stateB & & stateBUS1);
    A2out!PCU1,C,(stateB & & stateBUS1);
    recvA2++;
    :: !DONE & & B1in?[PCU1, senderB1, msg] ->
        B1in?PCU1, senderB1, msg;
        if
            /* if it is a diagnostic message */
            :: msg == HEADER -> skip;
        /* if it is a command message */
        :: else -> msg[i] = msg; i++;
        fi;
    /* acknowledgment to all the module */
    B1out!PCU1,A,(stateA & & stateBUS1);
    B1out!PCU1,B,(stateA & & stateBUS1);
B1out!PCU1,C,(stateA & stateBUS1);
recvBi++;

:: !DONE & & B2in?CDA1, senderB2, msg =>
   B2in?PCU1,senderB2,msg;
if
  /* if it is a diagnostic message */
  :: msg == HEADER -> skip;
  /* if it is a command message */
  :: else -> msg[i] = msg; i++;
fi;

/* acknowledgment to all the module */
B2out!PCU1,A,(stateB & stateBUS2);
B2out!PCU1,B,(stateB & stateBUS2);
B2out!PCU1,C,(stateB & stateBUS2);
recvB2++;

:: DONE -> break;

od)

RECEIVED: skip
/* endloop */

od;

7.3 Formal Verification on TMR-PCUs

In this section we informally list some of the properties verified on the TMR-PCUs model, and the most meaningful results. The properties can be informally described as:

(PCUS1) correctness of the communication protocols, in absence of faults;

We verified these two properties checking for absence of deadlock. With the term correctness we mean a a general correctness of the diagnostic and of the tournament algorithm run by a central module. In this case we slightly modified the PROMELA code of the PCUs in such a way to force a peripheral unit to receive messages according to the right cyclic use of the busses. An incorrect use of it by one of the central module will have caused a deadlock.

The following properties has been verified in presence of faults.

(PCUS2) when two or more modules are active each peripheral unit eventually receives exactly two messages, in a single loop;

([P2) => (( [<> q2 ) & & [] (q2 -> (<> r2 ))

20
In the previous formula \( p2 \) stands for “at least two modules are active”, \( q2 \) stands for “PCU1 is in RECEIVING” and \( r2 \) “PCUS1 is in RECEIVED and it has received exactly two messages”.

(PCUS2') in presence of byzantine errors in one module, when two or more modules are active each peripheral unit eventually receives exactly two messages, in a single loop;

This properties is the same of PCUS2, but was verified with one module running a byzantine synchronization phase.

(PCU3) when two or more modules are active each peripheral unit eventually receives exactly two message via different buses, in a single loop;

\[
(\Box p3) \rightarrow (( \Box \leftrightarrow q3 ) \&\& (\neg q3 \rightarrow (\leftrightarrow r3 \&\& s3 )) )
\]

In the previous formula \( p3 \) stands for “at least two modules are active”, \( q3 \) stands for “PCU1 is in RECEIVING”, \( r3 \) “PCUS1 is in RECEIVED and it has received exactly two messages”, and \( s3 \) “the messages received come from different buffers”.

(PCU4) when two or more modules are active each computer of every peripheral units receives exactly one message, in a single loop.

\[
(\Box p4) \rightarrow (( \Box \leftrightarrow q4 ) \&\& (\neg q4 \rightarrow (\leftrightarrow r4 \&\& s4 )) )
\]

In the previous formula \( p4 \) stands for “at least two modules are active”, \( q4 \) stands for “PCU1 is in RECEIVING”, \( r4 \) “PCUS1 is in RECEIVED and it has received exactly two messages”, and \( s4 \) “each computer has received at most one message”.

In the Table 3 we report some of the most significant results.

<table>
<thead>
<tr>
<th>property</th>
<th>PC</th>
<th>state vector</th>
<th>options</th>
<th>RAM (bytes)</th>
<th>depth search</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IV</td>
<td>352</td>
<td>C0</td>
<td>60.702</td>
<td>44047</td>
<td>success</td>
</tr>
<tr>
<td>2</td>
<td>II</td>
<td>284</td>
<td>CO+MA</td>
<td>23.808</td>
<td>25465</td>
<td>success</td>
</tr>
<tr>
<td>2'</td>
<td>IV</td>
<td>284</td>
<td>CO+MA</td>
<td>33.491</td>
<td>1295</td>
<td>fail</td>
</tr>
<tr>
<td>3</td>
<td>II</td>
<td>284</td>
<td>CO+MA</td>
<td>23.808</td>
<td>25465</td>
<td>success</td>
</tr>
<tr>
<td>4</td>
<td>IV</td>
<td>284</td>
<td>CO+MA</td>
<td>33.553</td>
<td>405178</td>
<td>success</td>
</tr>
</tbody>
</table>

Table 3: Verification results on the TMR-PCUs model
7.4 Discussion

We briefly discuss the result of property PCU2'. We wanted to prove safety properties of the tournament algorithm in the hypothetic situation of a byzantine behavior. In fact the fail was due to the byzantine behavior of a module, and analyzing the counter-example, we noticed that three modules (and not two) send a message to the periphery. It worth to point out that exclusion logic, we have omitted in this model, should has been disconnected the potentially byzantine module before it enters in the communication with the periphery phase. Indeed this is what happens in the reality, as proved by ASF on the real system.

8 Conclusions

The work described in this paper, related to a real and wider project, consisted of the verification effort performed on a formal model of a safety-critical system developed by Ansaldobreda Segnalamento Ferroviario, to manage medium-large scale railway networks. The real system, actually running at one of the main Italian railway stations, has been validated also by Ansaldobreda Segnalamento Ferroviario. During the formal verification we found some interesting erroneous situations some of them due to imprecision in the requirements, while others due to weakness in the system itself. In particular the fail entry in Table 2 pointed out a situation in which a tricky combination of byzantine communications distorted the exclusion logic mechanism; in Table 3 the fail entry refers to a misunderstanding in the requirements of the dependable protocol, successively fixed. We would underline that, by using the results obtained in the formal analysis, all implementation errors were also confirmed by Ansaldobreda by using traditional verification techniques.

Although briefly described in this paper, many formalization problems has been faced during the modeling phase, primarily due to the missing of any concept of time in the PROMELA language, and secondly to an inappropriate (respect to our needs) treatment of the termination of a processes in its run time support. These weaknesses obliged us both to abstract, in our models, any reference to the time (for example in the communication with time out), and to realize a safe-shutdown state as active process that participates in all the active communications. Those modeling choices have had a substantial impact in the formalization effort and, indirectly, in the state dimension of the model realized. To face with this last problem we need to design ad hoc abstraction strategies. All these formalization issues in a companion paper [7]. On the contrary the contribution of this paper relies on
describing the bulk of the verification work, consisting prevalently in defining two different models each depicting different aspects of the control system at a different degree of abstraction.

9 Acknowledgment

This work was partly supported by the CNR/GMD cooperation project DECOR and by Progetto speciale CNR “Strumenti Automatici per la Verifica Formale nel Progetto di Sistemi Software”.

References


