A Formal Specification and Validation of a Critical System in Presence of Byzantine Errors*

S. Gnesi[1], D. Latella[2], G. Lenzini[1],
C. Abbaneo[3], A. Amendola[3], and P. Marmo[3]

[1] Istituto Elaborazione dell'Informazione – CNR
{gnesi,lenzini}@iei.pi.cnr.it
[2] CNUCE – CNR
d.latella@cnuce.cnr.it
[3] Ansaldobreda Segnalamento Ferroviario
{cabbaneo,amendola, marmo}@asf.atr.ansaldo.it

Abstract. This paper describes an experience in formal specification and fault tolerant behavior validation of a railway critical system. The work, performed in the context of a real industrial project, had the following main targets: (a) to validate specific safety properties in the presence of byzantine system components or of some hardware temporary faults; (b) to design a formal model of a critical railway system at a right level of abstraction so that could be possible to verify certain safety properties and at the same time to use the model to simulate the system. For the model specification we used the PROMELA language, while the verification was performed using the SPIN model checker. Safety properties were specified by means of both assertions and temporal logic formulae. To make the problem of validation tractable in the SPIN environment, we used ad hoc abstraction techniques.

Keywords: safety critical systems, formal verifications, fault tolerant behavior, linear temporal logic, model checking.

1 Introduction

In the area of industrial processes, the use of Formal Methods (FM) to check safety critical components is in evident increase. Due to the high integration of information technology in the quite total amount of control systems, safety request is becoming more and more pressing. However some other important factors induce industries to use FM. First of all, the interest in discovering as many errors as possible before entering in the production phase; in fact, during this stage the cost of correction per error increases enormously (see [16] for a good statistical study). Moreover, governments and international institutions require industries to conform to international standards (*e.g.*, EN 50128 CENELEC

* This work was supported in part by the CNR project "Strumenti Automatici per la Verifica Formale nel Progetto di Sistemi Software"

S. Graf and M. Schwartzbach (Eds.): TACAS/ETAPS 2000, LNCS 1785, pp. 535–549, 2000.

Railways Applications [20], or IEC 65108 [13]) where FM are strongly suggested for validation and verification analysis.

In the last decade many industries, like the Ansaldobreda Segnalamento Ferroviario, started pilot projects (e.g., the ones documented in [8,15,18,2]) directed to evaluate the impact of FM on their production costs. Within Ansaldobreda Segnalamento Ferroviario encouraging results [1,3] - using CCS process algebras, with properties expressed in CTL and verified in the JACK environment - have shown how, for railway control systems, could be possible to formalize significant models and to perform verification in the model checking [7,21,4] approaches. Similar studies, using different formalisms (e.g., [9] used VCL* and μCRL to model a station vital processor and propositional logic to specify properties then verified in ASF+SDF), seem to confirm this positive trend. A recent thesis in [6], formally supports how railway systems share important robustness and locality properties, that distinguish them from most hardware systems and make them easily checkable in symbolic model checking and Stålmarck checking.

In this paper we describe the principal results of a real project jointly carried out by Ansaldobreda Segnalamento Ferroviario and CNR Institutes - IEI, CNUCE and CPR - of Pisa. The project consisted in designing a formal model of a critical control system called *Computerized Central Apparatus*, and successively in verifying specific safety properties under the hypothesis of *byzantine* faults. In this context, byzantine is to be intended as it was in Lamport et al. [14], where a byzantine component can arbitrarily fail in running its algorithm. In addition, other fault tolerant properties were verified under a weaker definition of byzantine fault, where a consistent behavior has been required. Industrial choices in Ansaldobreda suggested the use of the PROMELA [11] specification language, and of the SPIN [12] 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 4 we explain the PROMELA specification used as formal model, and how we described, in PROMELA, communication time-out and byzantine behavior; in Section 5 we discuss some abstraction and implementation techniques we used to contain the state explosion problem; in Section 6 we report some significant result of the verification phase, where a subtle and erroneous situation, due to the byzantine behavior of a module, was discovered; finally in Section 7 we conclude with some consideration on the whole experience.

2 System Description

The application we studied is a safety software within Safety Nucleus, which is part of a control system called *Computerized Central Apparatus* (ACC)[1] produced by Ansaldobreda Segnalamento Ferroviario [17]. The ACC is a highly programmable centralized control system for railway stations. It plays a critical

[1] "Apparato Centrale a Calcolatore", in Italian.

role in a wider railway signaling system, which is a very complex distributed architecture designed to manage a large railway network. Each node in the network 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 a simple interlocking logic.

The ACC architecture (see Figure 1) consists in two sub-systems that independently perform *management* and *vital* functions. **Management functions**,

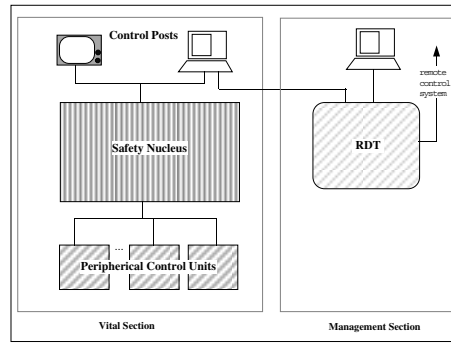

Fig. 1. The Computerized Central Apparatus architecture and its environment

run by a sub-section called Recording, Diagnosis and data Transmission (RDT), consists in auxiliary tasks, such as data recording, diagnostic management and remote control interface. **Vital functions** are reserved to control train movements and wayside equipment, and generally safe-critical procedures. This section of ACC is composed by a *Safety Nucleus* (SN), *Peripheral Control Units* (PCUs) and *Control Posts* (CPs).

Due to the critical characteristic of the vital section of ACC, particular attention has been paid to design fault tolerant mechanisms aimed to avoid that non-predictable (temporary or permanent) faults might compromise the correct operation of the system. To guarantee a trustable level of robustness many components have been replicated and consistency control tests have been inserted into the algorithm defining the behavior of the system. The **Safety Nucleus** is specifically designed for these control and safety purposes. It is interposed between CPs, from which an human operator can digit commands, and the PCUs that, in turn, execute them. Those commands are considered *critical* because their execution takes effect to critical machineries such as railway semaphores, rail points, or crossing levels. The SN has the principal aim to safely deliver the commands to the PCUs in case of faults in some hardware components. It is based on a triple modular redundancy configuration of computers which independently run different versions of the same application program.

Peripheral Control Units are designed to execute critical operation and to directly command physical devices. **Control Posts** are formed by input/output interfaces and by terminal by with an human operator compose a request or a command. Control Posts will not be considered in this study.

3 PROMELA and SPIN

Industrial choices within Ansaldobreda Segnalamento Ferroviario induced us to use PROMELA (Process Meta Language) [11] as specification language and SPIN as model checker environments. The fact that PROMELA is an imperative language with variables, with a C-like syntax makes it quite appreciated in industrial environment: the use of C++ is quite common in industrial development, and then with very low cost local engineers can learn PROMELA syntax and informal semantics, so that they can use it as a formal interchange language in the model refinement step. In addition PROMELA is a language of general applicability introduced to describe distributed systems, communication protocols and, in general, asynchronous process systems and resorted to be quite appropriate for our project.

For similar reasons SPIN has been preferred. SPIN can run on different platforms (Unix, Linux, Windows NT or Windows98), and this makes it possible, for the industries to have a closer control on the verification phase; for example by running some of the most significant test. In addition SPIN performs on-the-fly analysis, and support several state compression strategies, quite useful in dealing with state explosion problems which usually arise in this kind of work.

A PROMELA specification consists in one or more *process templates* (called also *proctype*) and in at least one process instantiation. The language is extended with non-deterministic constructs and with communication primitives, *send* and *receive*, using a weakly recalling Dijkstra's guarded command language notation [5] and Hoare's language CSP [10]. Processes can communicate via rendezvous, or via asynchronous message passing through buffered channels or shared memory. In addition any running process can instantiate further asynchronous processes using process templates.

SPIN [12] is an efficient formal verification tool for checking the logical consistence of a specification given in PROMELA. SPIN translates each PROMELA process template given in input, into a finite automaton. 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. SPIN accepts correctness claims specified either in the syntax of standard Linear Temporal Logic (LTL) [19], or as process invariants (using assertions) expressing base *safety* and *liveness* properties[2].

4 Formalization

In this section we describe the PROMELA model of the vital section of ACC, and the PROMELA models used to formalize time-out expiring and byzantine faults[3]. We used four PROMELA processes for the SN, and a PROMELA process for each

[2] Further information about PROMELA and SPIN can be found at the official SPIN URL `http://plan9.bell-labs.com/netlib/spin/whatispin.html`.

[3] The detailed specification is property of Ansaldobreda Segnalamento Ferroviario. We describe here, with permission, just what is needed to understand this work.

PCUs[4]. In the following with safety nucleus (lowercase) we mean the PROMELA model of the SN and with peripheral units (lowercase) the one of the PCUs.

4.1 The Safety Nucleus Model

A scheme of the safety nucleus processes and of the channels among them is reported in Figure 2. We want to underline:

1. the three identical *central processes*, called *module* A, B, and C, implementing the triple modular redundancy;
2. a special process called *exclusion logic*, devoted to checking the consistency of the three modules, and able to disconnect each of them if necessary;
3. the *interconnections* among the modules, between the modules and the exclusion logic, and between the modules and the PCUs;
4. the PCUs, here represented as a black box, composed by n control units.

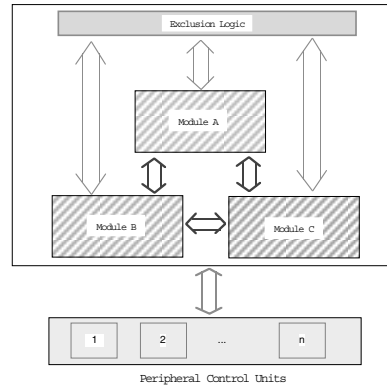

Fig. 2. The PROMELA processes with which we modeled the SN and the relative connections

The **modules** A, B, and C are designed for: (a) collecting global information on the system state, composed by the local states of each modules, by the state of the peripheral units and busses; (b) performing local computation taking care of the information collected and composing commands to be sent to the peripheral units. The three modules can communicate each other via symmetric channels; each module is further connected, via symmetric channels, with the exclusion logic and, via a double bus, with the peripheral units.

The behavior of a module is composed by a repeated sequence of *phases*, formally described with the following pseudo-code[5]:

[4] in the work we considered only two of such devices.
[5] n is the number of peripheral units

```
loop
1. * <synchronization>
2.   <command elaboration>
3. * <data exchange with the other modules>
4.   <distributed voting>
5. * <communication to exclusion logic>
{ communication with the PCUs }
       for i = 1 to n do
       6.1   if <is my turn> the
       6.2 *   <synchronization>
       6.3 *   <send command to the PCUs>
       6.4 *   <receive acknowledge from the PCUs>
       endfor
endloop
```

During each phase a central module runs local computations or communicates with other components of the system (we have pointed out these phases with an ∗). In particular, in the **synchronization** phase each module sends to and receives (with time-out) from every other module a synchronization message. This phase is used to collect information about the activity state of the other modules: a time-out expiring is interpreted as a sign of the non activeness, and the module that caused the time-out will be excluded from any successively communication within the current loop. Because the system is expected to run at least 2 out of 3, if a module detect a time-out from all the other modules, then it commutes in a *safe shutdown state*. In the **command elaboration** phase each module performs local computations, and calculates commands to be sent to the PCUs. In the **data exchange** phase each module sends to and receives (with time-out) from every other module a message containing information about the local state of the other modules. In the **distributed voting** phase, each module checks the consistence of its local information with the one received from the other modules. In the **communication with the exclusion logic**, the result of this test is sent to the exclusion logic which, after having analyzed all the results, can disconnect a module considered potentially faulty. Successively, in the **communication with the PCUs**, a module communicates (with time-out) with the PCUs, following a particular circular protocol. At each loop only two modules are able to communicate their command to the PCUs: a distributed procedure assures a cyclic selection of the modules communicating with the periphery and a cyclic use of the busses, also in case of faults.

4.2 The Peripheral Units Model

A scheme of the *peripheral units* its processes and of the channels connecting them to the safety nucleus is reported in Figure 3. We can identify:

- a process for each unit;
- the interconnection (a double bus) between the units and each of the module of safety nucleus, here represented as a black box.

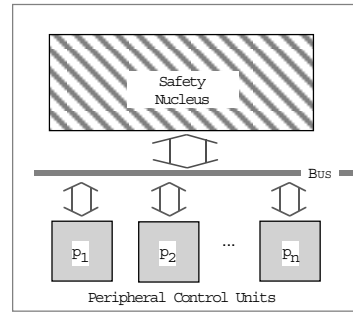

Fig. 3. The PROMELA processes with which we modeled the Peripheral Control Units and their connections with the safety nucleus

In the real system each peripheral unit is composed by two computers in configuration 2 out-of 2, that we modeled by a single process. Its behavior can be summarized with the following pseudo-code:

```
loop
  {communication with the safety nucleus}
parallel for i=1 to 2 do
      <computer[i] receives a command from a module
         and sends acknowledgements in reply>
    endfor
endloop
```

Informally each computer waits for a command, and then returns an acknowledgement back to all the modules.

4.3 Other Formalization Issues

The PROMELA model given so far describes the correct behavior of the system, but to complete the specification phase we needed to formalize also:

- a time-out expiring in the communications;
- a byzantine behavior of a module of the system.
- an arbitrary temporary fault in some system units.

Time-Out Expiring. In the ACC most communications are with time-out. Since PROMELA does not deal with time, we had to abstract from any definition of it. To simulate a communication with time-out 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, an than as a time-out expiring in a receive action[6]. In addition, wherever we had a **send** action we indeed introduced

[6] Formally the *empty* message is defined as follows: supposing the type of a channel was the tuple $(t_1, t_2, \ldots t_k)$, the *empty* message is the tuple $(EMPTY)^k$, where $EMPTY$ is a specific non-null integer constant.

a non deterministic choice between either transmitting the "real" message or transmitting the *empty* message, as in the following PROMELA pseudo-code[7]:

```
/* implementation of a send with time-out  */
define EMPTY <value>
chan c = [0] of <t>; // (synchronous) channel of type <t>
<t> msg;             // message of type <t>

[...]

if
:: true -> c!msg    // send the real message
:: true -> c!EMPTY  // send the empty message
fi;
```

Consequently a **receive** action needs to discern, depending on the content of the message, if a time-out has been expired or if the receiving has been successfully executed. Formally [8]:

```
/* implementation of a receive with time-out  */
c?x ->
     if
     :: (x == EMPTY)  -> <time-out failure>
     :: else          -> <success>
     fi;
```

Modeling a Byzantine Behavior. In order to model a situation in which the failure in one module of the safety nucleus, may cause conflicting information to be sent to the other modules, we need to develop a model of a byzantine behavior. In this context, byzantine behavior is to be intended as it was in Lamport et al. interpretation [14], and precisely:

1. all loyal modules run the same algorithm, and in particular correctly send all messages as specified in the algorithm of Section 4.1 ;
2. 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 have focused the attention on communication events. We have supposed that an arbitrary fault in the procedure will be visible, to the environment, only when the unit tries to communicate.

[7] We remind that in Promela, `if ::`$guard_1$ `-> x ::`$guard_2$ `-> y fi` is a guarded non-deterministic choice between `x` and `y`, and `c!x` is a send operation on the channel `c`, of the variable, or value, `x`.

[8] We remind that, in Promela, `c?x` is a receive operation from the channel `c`, of a value locally memorized into the variable `x`.

A consequence of this assumption is that an arbitrary fault is modeled as a *communication error*, and precisely as either a communication of a corrupted message, or as a delayed communication, or as no communication at all. To generate corrupted messages, we have supposed to have a function $corrupt() : T \longrightarrow T$, for each message type T, used to compromise the contents of a message[9]. Then a byzantine behavior can be modeled as in the following PROMELA code:

```
/* implementation of a send with byzantine failure */
define EMPTY <value>
chan c = [0] of <t>;
<t> msg;

[...]

if
:: true -> c!msg    // send the corrected message
:: true -> c!EMPTY  // send the empty message (i.e., no messages)
:: true -> c!corrupt(msg) // send the corrupted message
fi;
```

Modeling a Temporary Faulty Component. Besides modeling a byzantine behavior of a central module, we were interested in some other arbitrary faults in:

1. one or both busses connecting SN to PCUs;
2. one or both computers of one or both peripheral units.

In this case we were interested in formalizing faults that were persistent for at least one loop. This could be interpreted as a weaker byzantine behavior definition, in which we wanted to model an arbitrary fault in a component of the system, under the assumption that it behaves consistently (within a loop) when interacting with the other components.

This weaker byzantine fault has been implemented as in the following pseudo-code, relative to the PCU formalization:

```
loop
0.1 <decide the state of each of the two busses>
0.2 <decide the state of each of the two computers>

{communication with the safety nucleus}
 parallel for  i=1  to  2  do
```

[9] Possible instantiation for the *corrupt*() are: (a) if the type T is the boolean type, the *not*() function; (b) if the type T is the integer type, supposing that $EMPTY$ is a non-null integer value, the *corrupt*() function can be any integer valued function such that $corrupt(n) = EMPTY$ iff $n = EMPTY$ (to avoid semantic ambiguities from the $EMPTY$ value and a corrupted message).

```
        <computer[i] receives a command from a module
    [*]    and EVENTUALLY sends acknowledgements in reply>
    endfor
endloop
```

With "decide the state", we mean a preliminary setting of the functional state either of the busses or of the computers of the peripheral unit. In case of state "fault" every communication via the faulty bus or coming from the faulty computer, until the end of the loop, results in a time-out .

5 Abstraction and Implementation Strategies

The complexity of the model of ACC, more critic respect to state dimension than the other SN components, forced us to introduce *modularity* techniques to cope with the state explosion problem. We proceeded in the following ways:

1. by fisically separating the implementation of each phase composing the ACC behavior, with the intention to use them as building blocks In other words we planned to develop the phases in separate files, to be included in main file representing the whole ACC model;
2. by implementing each building block representing a *communication* phase, i.e. the ones where the three modules exchange a message, in a *correct* and in a *byzantine* version;
3. by implementing each building block representing a correct or a byzantine communication phase in a *concrete* and in an *abstract* version.

In the byzantine (versus the correct) version, we modified communication primitives as described in Section 8. In this way we: (a) could take under control the state dimension growing of the whole model by inserting a byzantine phase, which introduces more non determinism than a correct phase, at a time; (b) could test the robustness of the system in presence of some particular byzantine phases and not in presence of a widely distributed, quite less realistic, byzantine behavior.

In the concrete (versus the abstract) version, we modeled the communication without fixing, a priori, any ordering of the send/receive events. That is what happens in the real system. On the contrary, in the abstract version we impose a total order on those events. For example, we decided that the module A sends and receives first from B and then from C, that the module B first receives and sends to A and then sends and receives from C, and finally that the module C first receives from A and from B and then sends to A and to B. Note that the correct and the concrete, respect to the byzantine and the abstract implementations, have different impact on the state space. In fact: (a) the correct version has less not determinism, as least in our implementation of byzantine communication error; (b) forcing a total order on the send/receive eliminates all the non determinism in the external communication events. Inserting either the concrete or the abstract version of a particular phase in the whole model of ACC, we could obtain a set models of different abstraction level (see Figure 4).

While planning a modular model, we had tried to maintain an acceptable degree of *scalability*. In this case scalability is referred of abstract versus the concrete implementations and respect to certain properties decided in accordance with Ansaldobreda Segnalamento Ferroviario. Those properties express fundamental, known a priori, invariants on the communication phases among internal modules composing the ACC. Relatively to the local knowledge of each module, these properties can be informally described as:

(P1) before starting a communication phase, at least two out of three modules are active;
(P2) after a communication phase, each module has sent a message to all the other active modules;
(P3) after a communication phase, in receiving from all the other active module, a module has either received a message, or detected a time-out expiring;
(P4) after a communication phase, if a module has detect a time-out in receiving from all the other active modules, it commutes in a safe shutdown state.

Those properties, expressed as assertions on the code, was verified using the tool SPIN, and resulted satisfied on both the concrete and abstract models.

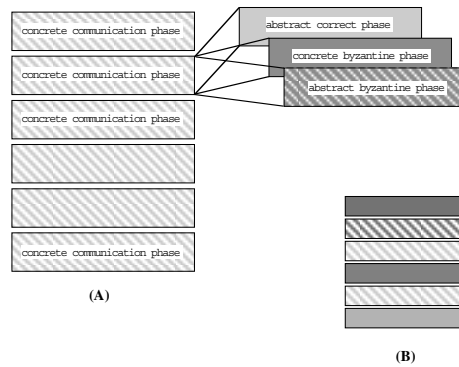

Fig. 4. The framework in which we developed abstract/concrete and byzantine/correct model (A). An example of instantiation of the model (B)

6 Formal Verification

We checked safety properties by varying the number of the byzantine phases inserted in the model. In addition, whenever the state dimension started to become problematic for our computational resources, we preferred the abstract to the concrete implementation of some, or all, the phases. In this way we executed a wide set of verification runs.

In the following we list only some of the more significant properties checked and their formalization as LTL formulae:

(F1) If two modules agree in recognizing something wrong in a third module, that module will be in the future disconnected.

```
[] (p1 -> [] (q1 -> <> r1) )
```

In the previous formula `p1` stands for "the module A recognizes something wrong in C", `q1` stands for "the module B recognizes something wrong in C" and `r1` "C is disconnected".

(F2) When two or more modules are active, then a peripheral units is in a receiving state infinitely often and when it is in this state it effectively receives from two different modules.

```
([] p2) -> ( ([] <> q2) && [] ( q2 -> (<> r2 && t2) ) )
```

In the previous formula `p2` stands for "at least two modules are active", `q2` stands for "the peripheral unit is before the receiving phase" and `r2` "the peripheral unit is after the receiving phase" and `t2` "the senders of the two messages are different".

(F3) When two or more modules are active, if a peripheral units receives messages then it receives exactly two messages.

```
([] p3) -> [] ( q3 -> (r3 && t3) )
```

In the previous formula `p3` stands for "at least two modules are active", `q3` stands for "the peripheral unit is before the receiving phase" and `r3` "the peripheral unit is after the receiving phase (it has received two messages)" and `t3` "it has received exactly two messages".

Interesting results were obtained in testing these properties on a model of the system with different communication phases affected by byzantine faults. The first and third properties resulted to be verified in the model with the byzantine faults in phases 1 to 5, while for the second SPIN reported an interesting counterexample in presence of byzantine faults in phase "communication with the periphery". The counterexample showed how the byzantine module can maliciously induce the other two modules in erroneous deduction on the global state and consequently to wrongly execute the communication protocol with the peripheral control units.

Most verifications, due to the high state space size required the use of both the two optimization strategies native in SPIN: the MA e COLLAPSE methods, which respectively use a minimized version of the Büchi Automata and a compressed representation of the state vector. As an example we report in the following the SPIN output relative to the verification to property (F2):

```
for p.o. reduction to be valid the never claim must be stutter-closed
(never claims generated from LTL formulae are stutter-closed)
pan: acceptance cycle (at depth 2342)
pan: wrote mainltl.c.trail
```

```
(Spin Version 3.2.3 -- 1 August 1998)
Warning: Search not completed
        + Partial Order Reduction
        + Compression
        + Graph Encoding (-DMA=180)

Full statespace search for:
        never-claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid endstates       - (disabled by never-claim)

State-vector 196 byte, depth reached 2345, errors: 1
     990 states, stored
     699 states, matched
    1689 transitions (= stored+matched)
     256 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)
```

7 Conclusions

The project described in this paper consisted in verifying certain safety properties on a model of a safety-critical control system in presence of byzantine behavior of one of its components. The real system has been validated also by Ansaldobreda Segnalamento Ferroviario, and errors we found confirmed the ones discovered with traditional techniques. The importance of developing a formal model, however stood in its great flexibility and in its high expandibility. In fact, during this project itself the model has been enriched, respect to the first requirements, or modified in some its procedures.

On the basis of this project an assessment on the application of the tool we used to support formal specification and verification process has been done. For what concerns the language PROMELA, we already underlined its suitability and expressivity power in describing this type of distributed system. The only disadvantage we found was the missing of any automatic management of termination of processes, that obliged us to model ad hoc time-out expiring as an active communication with heavy repercussion on the state dimension. In fact, we needed to explicitly formalize the shut-down behavior of a module as a module that does not anything but participating in all the communications by sending *EMPTY* messages to cause time-out.

Regarding the tool SPIN the most important fact to be underlined is related to strategies against the state explosion problem. In particular, the use of a minimized automaton encoding technique (MA) combined with the state compression option (COLLAPSE) resorted to be quite useful in helping with out-of memory problems, but at the cost of a very long execution time.

As an example, in Figure 5 we have reported a quite significative representative data, respect to all the other we obtained, concerning a verification run

on a 256 Mbyte RAM Pentium II - Linux Suse 5.3 - for a system model whose complete description required 348 bytes per state; in the figure memory and time resources have been compared using, respectively, the COLLAPSE (for which we had an out-of-memory termination, with the longest depth-first search path contained 15125 transitions from the initial state) and the COLLAPSE + MA options (for which we have successully terminated the verification, with longest depth-first search of 15916).

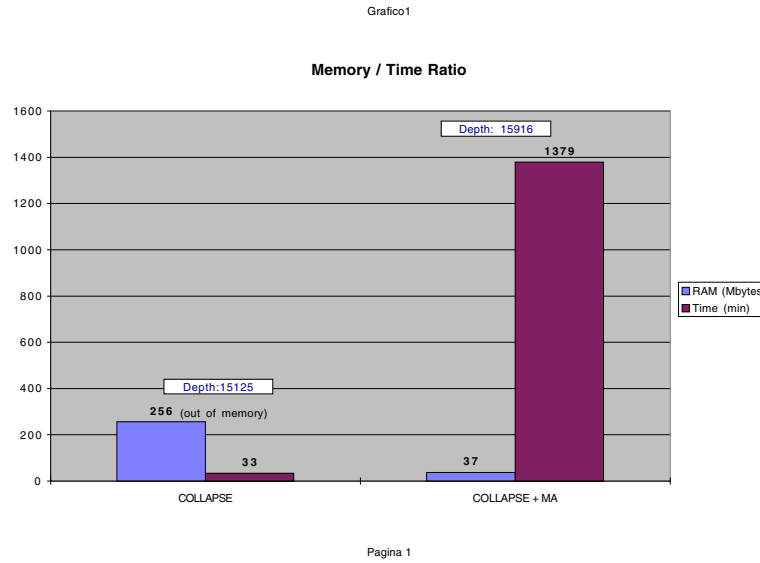

Fig. 5. The Memory/Time ratio using collapse and graph encoding reducing memory techniques

References

1. C. Bernardeschi, A. Fantechi, S. Gnesi, S. Larosa, G. Mongardi, and D. Romano. A Formal Verification Environment for Railway Signaling System Design. *Formal Methods in System Design*, 2(12):139–161, 1998. 536
2. A. Borälv. A Fully Automated Approach for Proving Safety Properties in Interlocking Software Using Automatic Theorem-Proving. In *Proceedings of the 2nd International ERCIM Workshop on Formal Methods Industrial Critical Systems*, 1997. 536
3. A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, and P. Traverso. Formal Verification of a Railway Interlocking System using Model Checking. *Formal Aspect of Computing*, 10(4):361–380, 1998. 536
4. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specification. *ACM Transaction on Programming Languages and Systems*, 8:244–263, 1986. 536

5. E. W. Dijkstra. Guarded Commands, Non-Determinacy and a Calculus for The Derivation of Programs. *ACM SIGPLAN Notices*, 10(6):2–14, June 1975. 538
6. Cindy Eisner. Using Symbolic Model Checking to Verify the Railway Stations of Hoorn-Keersenboogerd and Heerhugowaard. In *Proceedings of CHARME '99*, 1999. 536
7. E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, 1981. Springer-Verlag. 536
8. W. J. Fokkink. Safey Criteria for Hoorn-Keersenboogerd Railway Station. Technical Report Preprint Series 135, Utrecht, 1995. 536
9. J. F. Groote, S. F. M. van Vlijemn, and J. W. C. Koorn. The Safety Guaranteeing System at Station Hoorn-Kersenboogerd in Propositional Logic. In *Proceedings of 10th Annual Conference on Computer Assurance (COMPASS'95)*, pages 57–68, 1995. 536
10. C. A. R. Hoare. *Communicating Sequential Processes*. Prantice-Hall International, 1991. 538
11. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991. 536, 538
12. G. J. Holzmann. The Model Checker SPIN. *IEEE Transaction on Software Engineering*, 5(23):279–295, 1997. 536, 538
13. IEC 61508 IEC. Functional safety of electrical/electronic/programmable electronic safety-related systems. 536
14. L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transaction on Programming Languages and Systems*, 4(3):382–401, 1982. 536, 542
15. P. G. Larsen, J. Fitzgerald, and T. Brookers. Applying Formal Specification in Industry. *IEEE Software*, 13(7):48–56, 1996. 536
16. P. Liggersmeyer, M. Rothfelder, M. Rettelbach, and T. Ackermann. Qualitätssincherung Software-basierter Technischer Systeme - Problembereiche und Lösungsänsatze. *Informatik Spektrum*, 21:249–258, 1998. in German. 535
17. G. Mongardi. *Dependable Computing for Railway Control System*, chapter 3. Springer-Verlag, 1993. 536
18. M. J. Morely. Safety-Level Communication in Railway Interlockings. *Science of Communication*, 29:147–170, 1997. 536
19. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, 1977. IEEE, IEEE Computer Society Press. 538
20. pr EN 50128 CENELEC. Railways Applications: Software for Railway Control and Protection Systems. 536
21. J. P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of 5th International Symposium on Programming*, Lecture Notes in Computer Science, Vol. 137, pages 337–371. SV, Berlin/New York, 1982. 536