Model Checking Safety Critical Software with SPIN: an Application to a Railway Interlocking System.

Alessandro Cimatti[1], Fausto Giunchiglia[1], Giorgio Mongardi[2], Dario Romano[3], Fernando Torielli[2], Paolo Traverso[1]

[1]Istituto per la Ricerca Scientifica e Tecnologica (IRST)
38050 - Povo, Trento, Italy
{cimatti,fausto,leaf}@irst.itc.it

[2]Ansaldo Segnalamento Ferroviario (ASF)
Via dei Pescatori 35, 16129, Genova, Italy
gmon@gex25.atr.ansaldo.it

[3]Ansaldo Trasporti (ATR)
Via dei Pescatori 35, 16129, Genova, Italy
romano@cr.ansaldo.it

Abstract

This paper reports on an experience in formal verification using SPIN. The analyzed system is the Safety Logic of an interlocking system for the control of railway stations developed by Ansaldo. The Safety Logic is a process-based software architecture, which can be configured to implement different functions and control stations of different topology.

In this paper we describe how a PROMELA model has been devised, which retains the configurability features of this architecture. Furthermore, we discuss the verification with SPIN of significant process configurations.

1 Introduction

This paper describes a joint project between Ansaldo and IRST. The goal of the project was the evaluation of the possibility to integrate formal methods technology within the development cycle of a safety critical application. Particularly relevant was the assessment of formal methods techniques as an advanced debugging tool for the design.

The project focuses on a complex real-world safety critical application developed by Ansaldo, called ACC ("Apparato Centrale a Calcolatore"), a highly programmable and scalable computer interlocking system for the control of railway stations, implemented as a vital architecture based on redundancy [2]. The system is composed of a central nucleus connected to peripheral posts for the control of physical devices (e.g. level crossings, track circuits, signals and switches). The nucleus of the system is based on three independent computers, connected in parallel to create a "2-out-of-3" majority logic. Each of these sections runs (independently developed versions of) the same application program. When one of the sections disagrees, it is automatically excluded by vital hardware. The peripheral posts are also based on a redundancy architecture, with a "2-out-of-2" configuration of processors.

The "Safety Logic" of the ACC implements the logical functions requested by an external operator (e.g. preparing a path for moving a train from track to track). The distinguishing feature of the Safety Logic is that it is highly programmable and scalable. First, it is possible to program the modalities under which the commanded logical functions are performed. Furthermore, it is possible to program different configurations of physical devices, i.e. control for different stations. This is achieved by means of a logical architecture composed of a Scheduler controlling the activation of application-dependent processes. The Safety Logic is designed by specifying the processes controlled by the scheduler, which are then converted into executable code.

Specifying processes for this architecture is not a trivial task, due to two intrinsic sources of complexity. The first is the size of the controlled physical plants. Railway stations can contain a high number of physical devices, and processes of many different kinds can be required, to take into account the relations and interconnections among physical devices. The second source of complexity is nondeterminism, although the software is completely deterministic, and the possible external events (e.g. task requests, response and even faults of peripheral devices) have been exhaustively classified. The system can not know if and when external events will happen. For instance, tasks can be requested at any time. Furthermore, the peripheral devices will typically react to controls with (unpredictable) delays, and may even manifest (classified forms of) faulty behaviours.

Currently, the specification is validated by means of traditional techniques, such as simulation. The project aimed at the assessment of the possibility to integrate formal methods as a powerful debugging technique for the development cycle of the Safety Logic. Different formal methods techniques and tools, including theorem provers, CASE tools based on formal methods, and model checkers, have been preliminarly evaluated with respect to the particular features of the problem. Model Checking was preferred to other techniques, being completely automatic and therefore easier to integrate within the development cycle. Among different model checkers, SPIN was selected for its quality as a software product, the adequacy of its input language PROMELA for the specification of the Safety Logic, and the graphical interface which greatly eases the interaction with the user.

The structure of the Safety Logic imposed several precise requirements on the solution. First, it should be possible for the project engineers to manipulate the formal

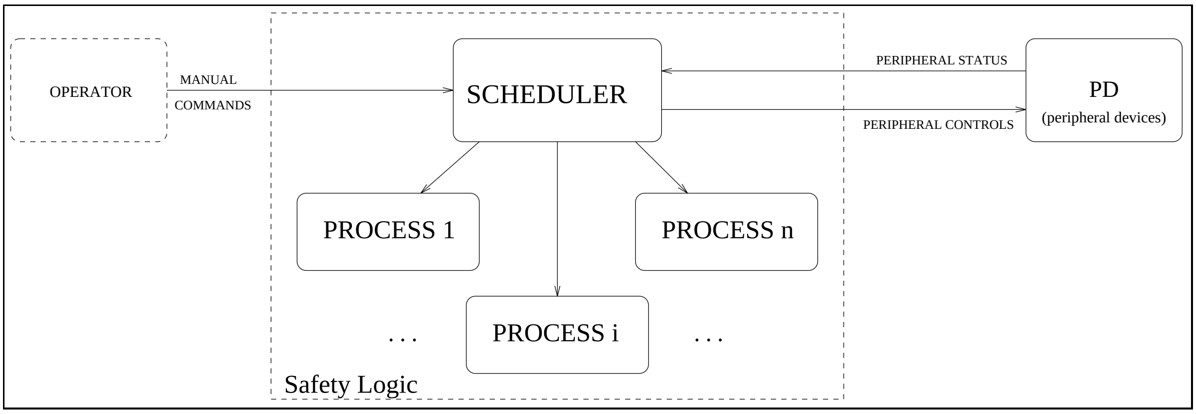

Figure 1: The architecture of the SL and its environment

model, and understand its relations with the Safety Logic. Second, it should be possible to obtain a large part of the model in a mechanical (and therefore automatizable) way from the specification. Finally, the model should retain the property of the Safety Logic that different configurations share the same Scheduler. From the computational side, the obvious problem to be avoided is the state explosion problem, at least for significant configurations.

In the rest of this paper we show how these goals have been achieved using SPIN. In section 2 an overview of the application is presented. In section 3 we discuss the PROMELA model of the Safety Logic, its modularity with respect to the configuration, and its scalability. In section 4 we discuss how several significant configurations were successfully verified, and we report the results of the analysis. In section 5 we draw some conclusions and present future work.

2 The Safety Logic

A high level picture of the Safety Logic of the ACC, together with its environment, is reported in figure 1. The Safety Logic (SL) is connected to the Peripheral Devices (PD) of the station and to the external operator (OP). The SL can be thought of as a deterministic reactive controller embedded in a nondeterministic environment. The SL repeats a cycle which consists of reading its inputs and determining the corresponding outputs according to its internal state. The SL takes as input the commands issued by OP (also called *Manual Commands*) and the status of PD. Manual Commands specify the tasks to be performed by the SL. Some examples are “Set route from track 2 to track 5”, and “Open level crossing 3”. The status of the peripheral devices (also called PD Status) represents information as conveyed from connected sensors. Examples might be “Position of Switch 12 is normal” and “Level crossing 3 is open”. The output of the SL are *Peripheral Controls*, i.e. the controls (also called PD Controls) issued to PD. Examples of PD controls are “Move Switch 12 to normal position”, and “Close level crossing 3”.

The architecture of the SL is based on a general Scheduler controlling a number

```
*** LOGICAL STATE VARIABLES ***
PROCESS-STATE: WAITING-FOR-TIMER, REQUESTED-CLOSING, REQUESTED-OPENING,
                 ACTIVATED-OPENING, COMPLETED-OPENING, RESTING (I.V.)
COMMAND-STATE: AUTOMATIC (I.V.), MANUAL, RECLOSED
MANUAL-OPENING-STATE: TRUE, FALSE (I.V.)

*** CONTROL STATE VARIABLES ***
P-COMBINATOR-STATE:  NORMAL, REVERSE, UNDEFINED (I.V.)
A-COMBINATOR-STATE: RESTING, WORKING, UNDEFINED (I.V.)
CONTROL-POSITION-STATE: CLOSED, OPEN, UNDEFINED (I.V.)

*** PD CONTROLS ***
P-COMBINATOR-NORMAL, P-COMBINATOR-REVERSE, A-COMBINATOR-WORKING

*** MANUAL COMMANDS ***
CLOSE, OPEN, RESTORE-AUTOMATIC-MODE

*** AUTOMATIC COMMANDS ***
CLOSE, OPEN (from process SHUNT)

*** ACTIVATION TABLE ***
* MANUAL COMMAND       -> MANUAL OPERATION
CLOSE                  -> LC-MAN-OP-CLOSE
OPEN                   -> LC-MAN-OP-OPEN
RESTORE-AUTOMATIC-MODE -> LC-MAN-OP-RESTORE-AUTOMATIC-MODE

* STATE           -> STATE OPERATION
WAITING-FOR-TIMER -> LC-STATE-OP-ACTIVATE-CLOSING
REQUESTED-CLOSING -> LC-STATE-OP-CHECK-CLOSING
REQUESTED-OPENING -> LC-STATE-OP-ACTIVATE-OPENING
ACTIVATED-OPENING -> LC-STATE-OP-COMPLETE-OPENING
COMPLETED-OPENING -> LC-STATE-OP-FINISH-OPENING

* AUTOMATIC COMMAND -> AUTOMATIC OPERATION
CLOSE               -> LC-AUTO-OP-REQUEST-CLOSING
OPEN                -> LC-AUTO-OP-REQUEST-OPENING
```

Figure 2: The Specification for the signature of Level Crossing Process

of Processes. Intuitively, the Scheduler can be thought of as an operating system's scheduler, i.e. a general program controlling the activation, suspension and termination of Processes according to their execution status and activities. (Further details on the behaviour of the Scheduler are proprietary information and can not be disclosed in this paper.) The processes controlled by the Scheduler can have different functionalities. For example, a process can be devoted to the control of a PD (e.g. a level crossing, a switch), or be responsible for a logical function (e.g. shunting, that is setting a route through the station). Processes are (often) organized in a hierarchical way, so that a "routing" process can control the activities of a "device" process. Therefore, Processes are able to issue commands, called *Automatic Commands*, to each other. During its activation, a process can issue PD controls and automatic commands, and can terminate its computation with different modalities.

Processes implement the functionalities to be guaranteed by SL. Intuitively, a process can be thought of as a procedure with a persistent state. It is associated with a set of (state) variables, defining its configurations, and with certain operations,

```
3.3.4 LC-STATE-OP-COMPLETE-OPENING
(Activated when PROCESS-STATE has value ACTIVATED-OPENING).
I - VERIFY:
a.      P-COMBINATOR-STATE with value REVERSE;
b.      MANUAL-OPENING-STATE with value FALSE.
II - SEND:
a.      to PD the control A-COMBINATOR-WORKING;
if:
1. A-COMBINATOR-STATE has value other than WORKING.
III - ASSIGN:
a. to PROCESS-STATE the value
COMPLETED-OPENING.
IV - AFTER THE OPERATION THE PROCESS
a.      does not terminate;
b.      does not continue.

EXCEPTIONS
[a]
WAIT
ACTIONS : ---

[b]
ERROR
ACTIONS:
I - ASSIGN:
a. to MANUAL-OPENING-STATE the value FALSE.
```

Figure 3: Specification of an Operation for the LC process

defining its behaviour.

Processes are designed by means of structured specifications written in a structured semi-natural language. The specifications define the signature and the behaviour of processes. Figure 2 shows the specification[1] defining the signature for the Level Crossing (LC) process. State variables are distinguished in Logical Variables and Control Variables. Logical Variables represent the status of the process computation. Each process is associated with a special variable called `PROCESS-STATE`, which, in this case, can assume the six specified values. The value `RESTING` is tagged as initial (`I.V.`), and is assigned to the variable at the system startup. The other variables are specified similarly. A process can modify the values of its logical variables during the execution of the operations associated with it. Control Variables represent the status of the peripheral devices of interest to the process (as perceived by the sensors). The values of control variables can not be modified by the process. They are set at the begin of the cycle of the SL by the sensing operation, and do not change until the next cycle. Then, the controls that the process can send to PD are specified. An Activation Table is specified which associates an operation to each event determining the activation of the process. For instance, the LC process will execute the operation `LC-STATE-OP-COMPLETE-OPENING` when its execution is resumed and the value of its state variable `PROCESS-STATE` is `ACTIVATED-OPENING`.

[1]In this paper they have been translated from italian and slightly edited for sake of readability.

Each operation referenced in the activation table is fully defined by a specification. Operations are described in a semi-natural language, and follow a fixed pattern. Figure 3 shows the specification for the operation `LC-STATE-OP-COMPLETE-OPENING`. Operations are collections of basic actions to be performed by the process. Basic actions include testing the value of variables, assigning values to logical (state) variables, sending PD controls to peripherals and automatic commands to other processes. These actions can be conditioned to tests.

Statements in operations are interpreted sequentially. The `VERIFY` tests are executed first. If one of the tests is not satisfied (e.g. test `b.`), then the corresponding `EXCEPTION` action (e.g. exception `[b]`) is executed and the execution of the operation ends. If the preliminary tests are satisfied, then commands may be issued during the `SEND` part, and then variables may be set during the `ASSIGN` part. After the execution of an operation, a process can act under different modalities according to what specified in part `IV`.

3 A scalable model of the Safety Logic

We describe now the PROMELA model of the SL described in previous section. This PROMELA model has the very same structure presented in figure 1, with boxes interpreted as SPIN processes, and arrows interpreted as SPIN channels.

Each process of the Safety Logic is modeled as a SPIN process with a fixed structure, taking two channels in input, and implementing the general activation table described in previous section. This structure is presented in figure 4 for the case of the LC process. The definition and initialization of logical variables can be determined according to the specification (control variables are set by the Scheduler upon the receipt of sensed data). Then, a loop begins where the process waits (on the synchronous channel `from_sched`) for the Scheduler to pass control. The activation event is stored in the variable `transition`, which is then tested to devise the operation to be executed. The activation table is implemented as a set of conditioned jumps. `<operation_body>` stands for the set of PROMELA statements which model each operation. After each operation there is a jump to the `test_end_transition` location, where the boolean variables `terminates` and `continues` are tested to check whether control can be returned to the Scheduler (for instance, requesting a further activation at next cycle), or another operation must be executed. `lc_return` returns the control to the Scheduler by sending a `lc_done` message on the synchronous channel `to_sched`.

In the model we make a substantial use of the `mtype` functionality of SPIN, which allows for encapsulation of numerical values in symbolic data. The information exchanged between LC and the Scheduler can be determined simply by the process signature, and does not depend on the actual functionality of the operations.

Given this general structure, the PROMELA model for a process can be completed by filling the slots corresponding to the local variables and the operations. The model of the operations of the process can be obtained directly and mechanically from the specifications described in previous section, and might be fully automatized in case of

```
proctype lc (chan from_sched, to_sched)
{
/* Initialization */
lc_prc = resting; lc_cmd = automatic; lc_man_open = false;

do
:: from_sched?transition ->  /* Read the activation event */
   if
   :: (transition == lc_manual_open) -> goto manual_open
   ...

   :: (transition == lc_state) -> goto state

   :: (transition == lc_auto_close) -> goto auto_close
   ...
   fi;

   /* Manual Operations */
   manual_open:  <operation_body> goto test_end_transition
   manual_close: <operation_body> goto test_end_transition
   restore_automatic_mode: <operation_body> goto test_end_transition

   /* Automatic Operations */
   auto_close: <operation_body> goto test_end_transition
   auto_open:  <operation_body> goto test_end_transition

   /* State Operations */
   state:
   if :: (lc_prc == waiting_for_timer) ->
         <operation_body> goto test_end_transition
      :: (lc_prc == requested_closing) ->
         <operation_body> goto test_end_transition
      :: ...
      :: else -> assert(0)
   fi;

   /* Check termination of transition */
   test_end_transition:
   if
   :: (terminates) -> goto lc_return
   :: (!terminates && !continues) -> add_next_cycle_processes(lc_name);
                                     goto lc_return
   :: (!terminates && continues) -> goto state
   fi;

   /* Return control to Scheduler */
   lc_return: to_sched!lc_done
od
}
```

Figure 4: The PROMELA model for Process Structure

```
...
:: (lc_prc == activated_opening) ->
   if
   :: (lc_p_cmb == reverse) ->           /* Verify a. */
      if
      :: (lc_man_open == false) ->       /* Verify b. */
         if
         :: (lc_a_cmb != working) ->     /* Send PD controls */
            cur_pd_ctrl.a_cmb_working = true
         :: else -> skip
         fi;
         lc_prc = completed_opening;     /* Assign */
         terminates = false;             /* After the operation */
         continues = false
         :: else ->                      /* Exception b. */
         lc_man_open = false;
         terminates = false;
         continues = false
      fi
   :: else ->                            /* Exception a. */
      terminates = false;
      continues = false
   fi
...
```

Figure 5: The PROMELA model of the operation specified in figure 3

integration of the model checker as a debugging tool in the process of designing the specifications. In figure 5 we report the PROMELA model for the operation specified in figure 3 (i.e. `<operation_body>` in figure 4). It is easy to see that for each action specified, there is a corresponding PROMELA construct. Exceptions are modeled depending on the kind of operations. An exception during a state operation is modeled as "Does not terminate and does not continue", which usually amounts to waiting for the conditions to become ready and trying again next cycle. An exception during a manual or automatic operation is interpreted as "Terminates", as the process should reject a command if the preconditions for executing it are not satisfied.

Modeling the Scheduler required a strict integration between IRST and Ansaldo, in order to clarify and understand the relevant features of the SL. Although the details can not be disclosed in this paper, the final PROMELA model of the Scheduler represents in detail the different computation phases of the cycle of the Safety Logic. Furthermore, just like the Safety Logic, the model of the Scheduler is largely independent of the structure and number of processes of the configuration. Therefore, it retains the reusability and scalability properties of the Safety Logic.

4 Experimental Results

In this section we discuss the exhaustive analysis with SPIN of models of the SL discussed in previous section. It is worth pointing out that the first configuration to be validated, as specified in [1], was composed of a physical level crossing, together

```
(Spin Version 2.9.1 -- 16 September 1996)
        + Partial Order Reduction

Full statespace search for:
        never-claim             - (not selected)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid endstates       +

State-vector 136 byte, depth reached 3447, errors: 0
   25088 states, stored
   13044 states, matched
   38132 transitions (= stored+matched)
       4 atomic steps
hash conflicts: 2062 (resolved)
(max size 2^19 states)

4.96276e+06     memory usage (bytes)


real        9.5
user        8.1
sys         0.5
```

Figure 6: Results of the analysis of the case study

with two processes in the Safety Logic, namely a Shunting and a Level Crossing. The performance obtained analyzing this configuration were very good. It was possible to complete a full verification of state properties in less than ten seconds on a SUN-SPARCSTATION 10 (see the SPIN output reported in figure 6). Rather than being a sign of simplicity of the problem, we believe that this shows that the model was carefully designed to exploit all the constraints. The case study was considered to be of rather complex nature, as previous attempts to the verification of such a configuration had lead to the state explosion problem.

In the following we discuss the results of the analysis of a more complex configuration, modeling the control of a physical railway track divided in three parts and containing a level crossing. The SL contains four processes, and was obtained by extending the model for the first configuration (Shunting-Level Crossing) by adding the model of two additional processes of different type (Liberation). Intuitively, the Shunting process is in charge of preparing the conditions for the train to pass, for instance book the parts of railway track and command the level crossings to close. When the conditions are ready, it signals the free way to the train. When the train has passed, the Liberation processes are activated to free the track components which had been booked and open the level crossing.

Several properties were imposed on the model. The first requirement is "the process Shunt does not signal free way to the train if the Level Crossing is not closed". This is an obvious safety requirement, and can be represented as an assertion relating the controls to be delivered to PD and the status of peripheral devices. Other requirements such as "the SL never issues contradictory PD controls during a cycle" were

```
warning: for p.o. reduction to be valid the never claim must be stutter-closed
(never claims generated from LTL formulae are stutter-closed)
(Spin Version 2.9.1 -- 16 September 1996)
        + Partial Order Reduction

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 192 byte, depth reached 7815, errors: 0
  188905 states, stored
  268894 states, matched
  457799 transitions (= stored+matched)
       6 atomic steps
hash conflicts: 38257 (resolved)
(max size 2^19 states)

2.89913e+07     memory usage (bytes)


real     1:30.7
user     1:17.1
sys         3.8
```

Figure 7: Results of the analysis of the four-processes configuration

represented in a similar way.

The formalization of the following requirement was more complex: "if the free way was signaled to the train, and the Level Crossing was closed with a manual command, then the manual command `RESTORE-AUTOMATIC-MODE` must leave the Level Crossing closed". This criterion is a liveness property. It relates a generic state in which the premises must hold, a following state in which the manual command `RESTORE-AUTOMATIC-MODE` is issued, and then a sequence of future states where no further manual commands are issued to the SL. Modeling this criterion required the use of the SPIN translator of LTL formulas into never claims.

Another relevant issue was testing for the termination of the cycle of the SL. Non termination can in principle occur in two different situations. First, a process can have a non terminating transition, depending on the termination value returned by the operations. Second, the automatic phase may not terminate, depending on the nature of the command flow among processes (e.g. if two processes keep to send automatic commands to each other). This was modeled by requiring the beginning of cycle of the SL to be a point of progress.

These properties, together with a number of assertions to make sure that the data flow was correctly modeled, were all analyzed with SPIN. Analyses were performed checking state properties, acceptance cycles and non-progress cycles. Again, the resource required to analyze such a complex configuration were rather limited. The complete analysis of the liveness property discussed above took less than two min-

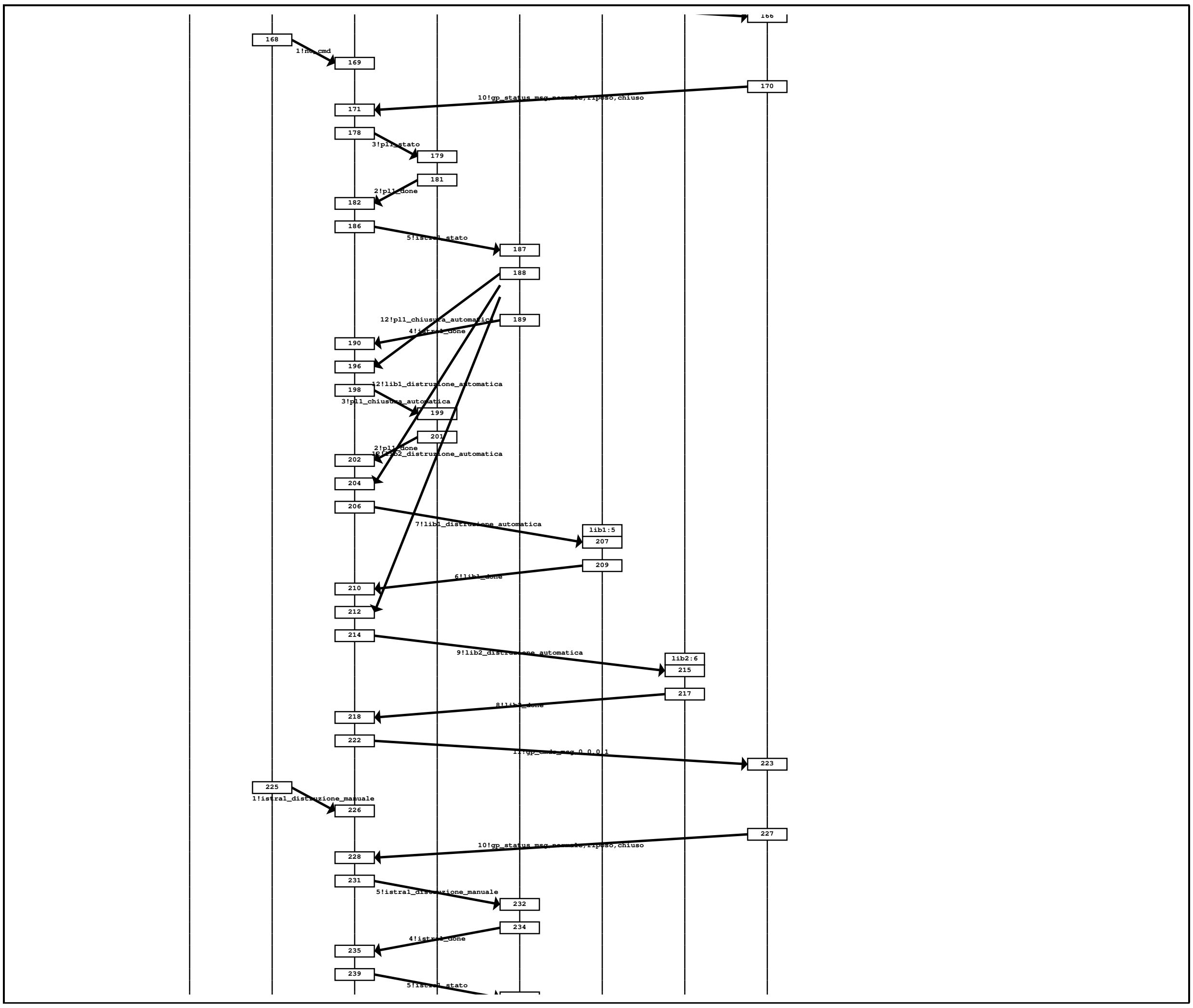

Figure 8: Trace of the activation of the resting liberation.

utes (as reported in figure 7), while the non-progress cycles check required less than 6 minutes.

Some remarks about the experimental results. Of all the reduction mechanisms provided by SPIN, `d_step` was the most effective to reduce the storage of irrelevant states. Without it, it would not have been possible to complete the above analysis. Notice also that the optimization of the model was never a major concern, given the requirement of scalability for the model. A number of redundancies in the used representation have been identified which seem to allow for optimization, and for the verification of even more complex configurations.

During the analysis, SPIN was able to detect an anomalous behaviour. One of the processes was activated while being in a resting state, i.e. a state having no associated operations in the activation table. The significance of this behaviour is that it was present in a prototype version of the Safety Logic. This was solved by associating by default every resting state with a fictitious operation which simply returns control to the Scheduler. Pinning out this behaviour was very valuable to highlight the power of exhaustive verification. Such a behaviour is extremely hard to point out via testing.

After the anomalous behaviour was found, the SPIN mechanism for finding the shortest counterexample was very useful to reduce the length of the trace. The "shortest" trace we could come up with is generated by a particular sequence of four manual commands issued during seven cycles of the Safety Logic, and amounts to several hundred steps of simulation. In figure 8 part of the trace shown by the SPIN interface is reported, covering just a little more than one cycle of the SL. The vertical traces represent (from left to right), OP, Sched, LC, Shunt, the two Liberations and PD.

5 Conclusions and Future Work

This project has been a successful experience in formal methods applied to the design of safety-critical systems. The configurations analyzed in this project are much simpler than the ones needed to control a railway station. However, it seems possible to deal with configurations of significant size and complexity, which can help the designer to gain confidence in the system being specified. This is a great advantage with respect to testing, as very often the exhaustive analysis of a scaled down configuration can reveal problems which are present also in larger configurations.

The features of SPIN which mostly contributed to the success of the project are its quality as a software product, the adequacy of PROMELA for the representation of the system, and the graphical interface which greatly eases the interaction with the user.

Future activity will try to scale up the size of configurations which can be exhaustively and automatically analyzed. A brute-force, direct approach to the verification appears to be infeasible. However, preliminary experiments have shown that there is room for improvement by using different representations of the system, and decomposition and abstraction techniques.

References

[1] A. Cimatti, F. Giunchiglia, G. Mongardi, B. Pietra, D. Romano, F. Torielli, and P. Traverso. Formal validation of an interlocking system for large railway stations: A case study. Confidential IRST Technical Report, Trento, Italy, 1996.

[2] G. Mongardi. Dependable Computing for Railway Control Systems. In *Proceedings of the Working Conference on Dependable Computing for Critical Applications*, pages 255–273. IFIP Working Group, 1992.