Formal Aspects of Computing (1998) 10: 361–380
© 1998 BCS

Formal Aspects of Computing

Formal Verification of a Railway Interlocking System using Model Checking*

A. Cimatti[1], F. Giunchiglia[1], G. Mongardi[2], D. Romano[3], F. Torielli[2] and P. Traverso[1]

[1]Istituto per la Ricerca Scientifica e Tecnologica (IRST), Trento, Italy;
[2]Ansaldo Segnalamento Ferroviario (ASF), Genova, Italy;
[3]Ansaldo Trasporti (ATR), Genova, Italy

Keywords: Formal methods; Model checking; Industrial applications; Safety critical systems; SPIN

Abstract. In this paper we describe an industrial application of formal methods. We have used model checking techniques to model and formally verify a rather complex software, i.e. part of the "safety logic" of a railway interlocking system. The formal model is structured to retain the reusability and scalability properties of the system being modelled. Part of it is defined once for all at a low cost, and re-used. The rest of the model can be mechanically generated from the designers' current specification language. The model checker is "hidden" to the user, it runs as a powerful debugger. Its performances are impressive: exhaustive analysis of quite complex configurations with respect to rather complex properties are run in the order of minutes. The main reason for this achievement is essentially a carefully designed model, which exploits all the behaviour evolution constraints. The re-usability/scalability of the model and the fact that formal verification is automatic and efficient are the key factors which open up the possibility of a real usage by designers at design time. We have thus assessed the possibility of introducing the novel technique in the development cycle with an advantageous costs/benefits relation.

Correspondence and offprint requests to: Alessandro Cimatti, Istituto per la Ricerca Scientifica e Tecnologica, 38050, Povo, Trento, Italy. Tel: + 39 0461 314320; Fax: + 39 0461 314591. E-mail: cimatti@irst.itc.it

* Financial support for the work described in this paper was provided by an ANSALDO - IRST project, Contract Number 158/242140 (NA). The authors thank Ing. Bruno Pietra for his invaluable support.

1. Introduction

The industrial take-up of formal methods in the area of safety critical applications is currently at a crucial stage. From one side, the need for innovative design and validation techniques is compelling. First of all, the complexity of the services required by safety critical software results in an extremely large number of possibly different discontinuous behaviour evolutions, which can be hardly kept under control during design and cannot be exhaustively tested at validation time. Second, international standards [BoS93] highly recommend (and in some case mandate) the use of formal methods for design and validation. Third, companies are sometimes under the pressure of government, regulation institutions, and even customers, which require higher and higher assurances on the computer controlled safety critical systems. For all these reasons, the effective use of innovative techniques like formal methods is a potential key feature for the company's business and for gaining a market against competitors. In the last ten years, indeed, several companies have started to investigate different techniques by assessing the state of the art of support tools and by conducting pilot projects and case studies (see e.g. [GVK95, Mor97, Bor97, BFG98, Ste95, LFB96]) on (simplified versions of) industrial applications. Some of these case studies have given positive results, showing how a model of the system can be formalized and validated by means of mechanized formal validation techniques (like model checking and theorem proving).

From the other side, the industrial take-up of formal methods is strictly conditioned by the costs and benefits of their actual introduction in the company's software development process. While the assessments on tools and methods and the experimental pilot projects have provided useful evaluation of the technology, they are still far from providing an evaluation of the actual introduction of formal methods in the "critical path" of the company's development process. Two of the key issues to address the industrial take-up of formal methods are the following.

1. Each company has its own well-established software engineering practices. These, for instance, might involve the usage of custom design specification languages which are often highly application dependent. Enforcing a complete change of the company's practice is often impossible in practice (especially in terms of cost/benefits), due e.g. to the existing knowhow within the company. This deserves a need for tight integration of the novel technique.
2. The formal model must conform to the system being modelled. This raises a number of issues. First, the application experts should be able to understand the formal model, even if they are not and cannot be experts in formal methods, and to evaluate it w.r.t. the real system (specification). Second, the model and its validation should mimic the actual behaviour of the system, at the right level of abstraction.

In this paper, we describe a joint project between Ansaldo and IRST, where we have started to tackle these two issues. The project focuses on the Ansaldo's interlocking system for medium-large scale railway stations [Mon92]. This system presents several intrinsic sources of complexity, which make it hard to validate with traditional techniques, e.g. simulation. The project aimed at the assessment of the possibility of integrating formal methods as a powerful debugging technique used effectively by designers during the development phase. This has been achieved by taking into account some main characteristics of the interlocking development process. Each interlocking system is constructed through a development practice which is currently well-established and well-tested in Ansaldo. The starting point is a kernel, developed once for all, which is common to all railway stations, interlocking rules and railway requirements (which may e.g. change from nation to nation). Different interlocking systems are then developed through a high-level custom specification language, which allows the designer to specify effectively and separately the particular signaling rules (e.g. the Italian ones) and the particular configuration (e.g. that of the Genoa station). This approach provides significant advantages, both from the development point of view (re-use of the kernel) and from the validation point of view (possibility to validate separately the kernel and the high level specifications).

Our goal was therefore to assess the effective integrability of formal methods without modifying the current development approach, thus reducing costs, and adding more powerful validation features, thus increasing the level of safety of the system. In order to achieve this goal, we have decomposed the construction of the formal model according to the structure of the system, in two steps: the construction of the model of the kernel, and the construction of the model of the signaling rules of the interlocking.

1. In order to model the kernel of the system, we have chosen a proper specification language which is easy to understand by the designers. The model has been checked and validated by the engineers which have constructed the kernel itself. This assures a high confidence in the fact that the model is coherent with the real kernel implementation. This step needs to be done only once for all different interlocking systems by a restricted team of experts of the kernel and of the formalism. The cost of the formalization is therefore not prohibitive.
2. The formal model of the signaling rules can be generated automatically from the designer specifications. This does not force the (possibly many) designers to learn the formal language and to build the formal model for each specification, thus highly reducing the costs of the introduction of the new technique.

During the design phase, the model is generated from the design specifications, and then model checking techniques [Hol91] are applied in order to validate the requirements. From the designer point of view, the model checker is seen as a powerful debugger which is used in a similar way to a simulator, and which provides the significant advantage to test (the model of) the system exhaustively. As a result, the model checker has detected a subtle design error, which has been judged hard to find with traditional techniques.

A further problem we had to face in this application is its high complexity, which might of course result in a decreased performance of the model checking, or at worst in a state explosion problem. Of course there are several reduction techniques to address the state explosion problem. For instance, abstraction techniques have been used in a similar application [BFG98]. Nevertheless, the practical applicability of reduction techniques must take into account the possibility and the cost to introduce them in the development process. In particular, most of these techniques are not fully automatic, and would require the intervention of the designer, with a consequent increase in development time. We have addressed this problem by devising a way to compile automatically the original model into a kind of "optimized" model, which can be model checked much more efficiently. This, far from being a definite solution to the problem of state explosion, is a first step which allows for an efficient model checking of interesting interlocking configurations.

This paper is structured as follows. In next section we review the formal technique and tool. In Section 3 we present an overview of the application. In Section 4 we describe in detail the Safety Logic, the software subsystem analyzed during the project. In Section 5 we discuss the formal model of the Safety Logic, its modularity with respect to the configuration, and its scalability. In Section 6 we discuss how several significant configurations were successfully verified, and we report the results of the analysis. In Section 7 we show how the model can be optimized in order to tackle the state explosion problem and analyze more complex configurations. In Section 8 we discuss related work. In Section 9 we draw some conclusions and present future work.

2. The Verification Technique and Tool

The problem of selecting the right tool for the formal verification of the Safety Logic is by no means trivial. First, representational issues must be tackled. Second, the state explosion problem must be contained. In this section we review the specification language PROMELA, the exhaustive exploration techniques used by the SPIN model checker, and the validation environment.

PROMELA (Process Meta Language) is the input language of the SPIN model checker [Hol91]. It is a language originally designed for the representation of distributed systems and communication protocols, which turns out to be of rather general applicability. A PROMELA program specifies a collection of processes, which can communicate through channels of different kinds.

The body of a process is basically an imperative program, extended with non-deterministic constructs loosely based on Dijkstra's guarded command language notation. A process can read and modify the value of its local variables, or of the variables global to the program. Communication between processes is inspired to Hoare's language CSP [Hoa85], extended with some powerful new constructs. It contains the primitives for the specification of synchronous message passing systems (rendez-vous). It also allows for specifying asynchronous (buffered) message passing via channels, with arbitrary numbers of message parameters. Mixed systems, using both synchronous and asynchronous communications, are also supported. Concurrent entities can be modelled as different PROMELA processes communicating through channels.

A PROMELA program defines a finite state automaton [VaW86]. A state for the automaton keeps track of the value assigned to each variable, the messages contained in queues, and the control points of the processes. At each instant only one process is selected to perform a transition, corresponding to a change in the global state.

SPIN is a widely distributed software package that supports the formal verification of distributed systems modelled in PROMELA. SPIN has been used to trace logical design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols [Hol97]. The tool checks the logical consistency of a specification. It reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes.

SPIN works on-the-fly, which means that it incrementally explores the state space of the analyzed system, and at the same time checks the specifications. SPIN can be used as a full linear time temporal logic (LTL) model checker, but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Many of the latter properties can be expressed, and verified, without the use of LTL.

Correctness properties can be specified as invariants using assertions, i.e. PROMELA instructions which state that a given condition must be true of a state at a given control point of a process. Properties can also be expressed directly in the syntax of next-time free LTL, or indirectly as Büchi Automata (called "never claims").

SPIN supports random, interactive and guided simulation, and both exhaustive and partial proof techniques. It is specifically designed to handle even very large problem sizes. To optimize the verification runs, the tool exploits efficient partial order reduction techniques, and (optionally) OBDD-like storage techniques. SPIN features a graphical interface which allows for a very friendly interaction, and allows to depict the behaviour of the analyzed system by building Message Sequence Charts (MSC) on the fly.

3. The Application: Overview

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 based interlocking system for the control of railway stations, implemented as a vital architecture based on redundancy [Mon92].

The interlocking signaling system is a distributed system. Each node is responsible for the interlocking logic of a set of track units. A node can control a medium-large railway station, a line section with small stations, or a complete low traffic line with simple interlocking logic. The ACC is the core of each node. Its architecture consists of two subsystems that independently perform the vital and the management functions (see Fig. 1).

The vital section controls train movements and trackside equipments. Trackside equipments are connected to Trackside Units (also called Peripheral Devices), e.g. level crossings, track circuits, signals and switches. The human operator of the railway station can monitor and command the system through interfaces in the Control Post. The core of the vital part is the Safety Nucleus connected to Trackside Units and devices in the Control Post. It is designed for the execution of safe operations. It 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 management section (RDT - Recording, Diagnosis and data Transmission) performs auxiliary functions, such as data recording, diagnostic management and remote control interface.

In the project, we have focused on a software subsystem of the Safety Nucleus, called "Safety Logic". It implements the logical functions requested by the 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 com-

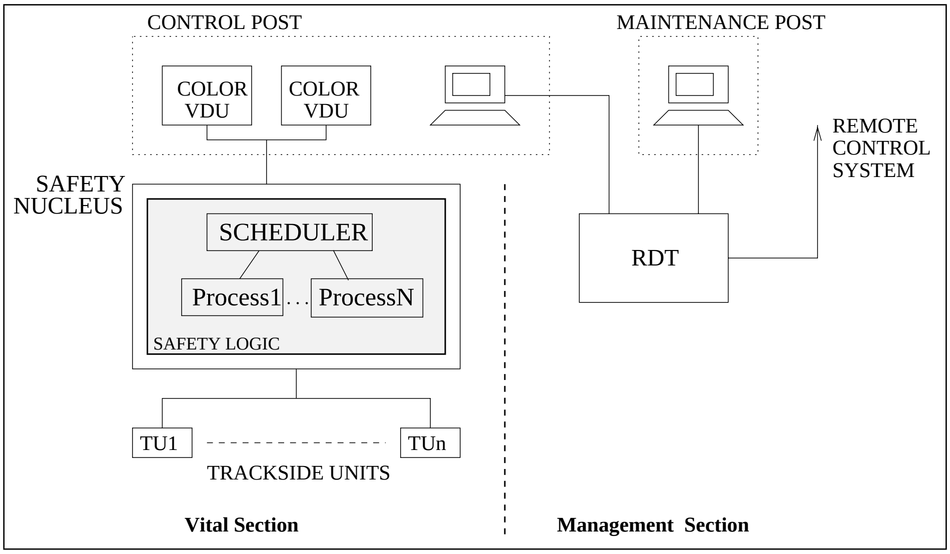

Fig. 1. The application and its environment.

manded 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 preliminarily 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 [Hol91] was selected for its robustness and 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 project. First, it should be possible for the project engineers to manipulate the formal 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 problem to be avoided, at least for significant configurations of the Safety Logic, is the so-called "state explosion problem", i.e. the fact that large amounts of memory can be needed to complete the exhaustive exploration of the model.

4. The Safety Logic

At a high level of abstraction, the Safety Logic (SL) is connected to the Peripheral Devices (PD) of the station and to an 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 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. 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. Each process is associated with a set of (state) variables, defining its configurations, and with certain operations, 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 defining the signature for the Level Crossing (LC) process. (In this paper the specifications have been translated from Italian and slightly edited for sake of readability.) State variables are distinguished in Logical Variables and Control Variables. Logical Variables

```
*** 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
```

Fig. 2. The specification of the signature of the LC process.

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 beginning 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.

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

```
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.
```

Fig. 3. The specification of an operation of the LC process.

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`.

5. A Scalable Model of the Safety Logic

The formalization of the Safety Logic in PROMELA is depicted in Fig. 4. (Boxes are interpreted as PROMELA processes, and arrows are interpreted as PROMELA channels.) The Operator and PD processes model the environment of the Safety Logic. The role of the Operator is to send commands, called manual commands, requesting the execution of certain functionalities by the Safety Logic. A simple model of the external operator performs an infinite loop, selecting randomly a manual command and sending it to the Safety Logic encoded as a message.

The PD process models the behaviour of the peripheral devices controlled by the Safety Logic. The PD performs an infinite loop. Periodically, it sends a message to the Safety Logic which describes the status of the peripheral devices. Then, waits for a message from the Safety Logic describing the controls to the peripheral devices, and simulates the evolution of the model. Models of different granularity can be defined. For instance, a level crossing can be modelled as a

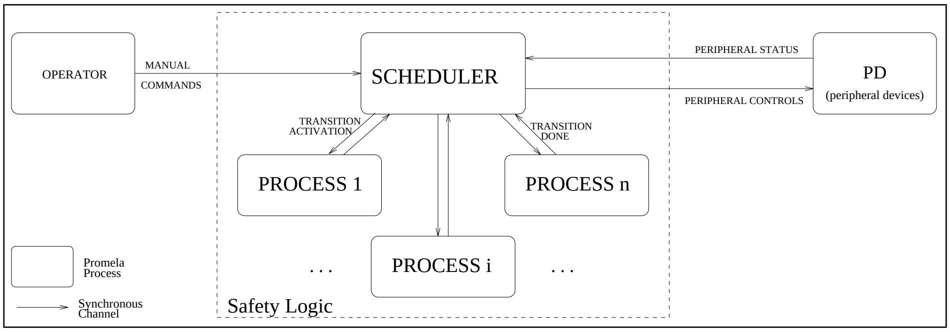

Fig. 4. The PROMELA model of the SL and its environment.

deterministic, fault-free device, always opening in exactly three time units, or as a non-deterministic device, which can complete an opening in between two and ten time units, and can fail to open. One of the models that we used models devices as completely random.

The Safety Logic is modelled as a collection of PROMELA processes. One process models the Scheduler, while each process of the Safety Logic is modelled by a corresponding PROMELA process implementing the functionalities described by the specification.

5.1. The Model of the SL Scheduler

The Scheduler of the Safety Logic is modelled as a PROMELA process repeating a cycle. At the beginning, it reads a message from the operator representing a manual command, and a message from PD representing the current status of the peripheral devices. The control variables of the processes are updated according to the sensed values. Then, according to the read command, processes are activated with a uniform master-slave way mechanism, until certain conditions are reached. Then, the commands for the peripheral devices which have been devised during the cycle are sent as a message to PD.

Given a process configuration, the Scheduler operates on two synchronous channels per process, one for each direction. The activation protocol of processes is given by the simple fragment of PROMELA:

```
to_lc!<activation_message>;
from_lc?lc_done
```

First the Scheduler sends an `<activation_message>` to the process to be activated (in the above case the Level Crossing) on the synchronous channel toward the process (`to_lc`). The `!` represents the writing operation. Then, the Scheduler suspends its execution by reading on the synchronous channel (`from_lc`). Reading from a channel is specified in PROMELA with the operator `?`. This amounts to waiting for the process to return control.

The details of the algorithm implemented by the Scheduler can not be disclosed in this paper. A careful modelling and a strict interaction between IRST and Ansaldo allowed to clarify and understand the relevant features of the SL, and reproduce the actual behaviour of the Scheduler. The final PROMELA model of the

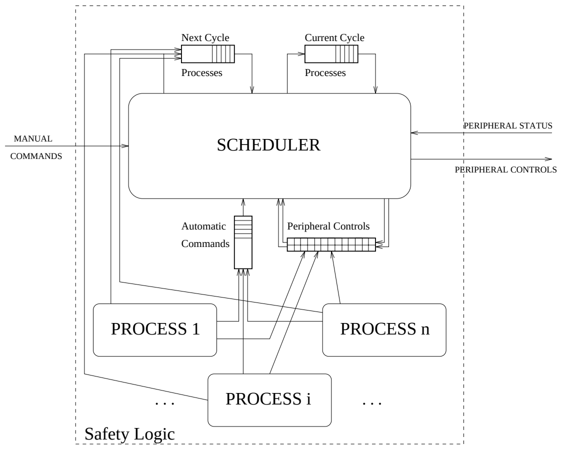

Fig. 5. A detailed view of the PROMELA model of the SL.

Scheduler represents in detail the different computation phases of the cycle of the Safety Logic.

In order to implement its functionalities, the Scheduler handles a complex set of data structures (see Fig. 5), which were modelled as follows. First, the controls to be issued to PD at the end of each cycle are modelled as the record `cur_pd_ctrl`, containing a boolean field for each of the commands which can be issued to PD. This record is global, and when a process issues a control to the peripheral devices (see e.g. the instruction `II.a` in Fig. 3) it actually sets the suitable record field in `cur_pd_ctrl`. The record is then sent as a message to the PD process, and then reset by the Scheduler at the beginning of each cycle. As processes can be suspended from cycle to cycle, the Scheduler needs to store the processes that will be activated at next cycle (`next_cycle_processes`), and the ones which were suspended at the previous cycle and will be resumed at the current cycle (`cur_cycle_processes`). At the end of each cycle, next cycle processes become current cycle processes. Finally, automatic commands, i.e. commands from process to process, are served in a first-in-first-out policy, and are to be served at the cycle they have been issued.

Just like the Safety Logic, the model of the Scheduler is largely independent of the structure and number of processes of the configuration. Given a process configuration, a model of the corresponding Scheduler can be obtained via instantiation from the information provided by the signatures specification of the controlled processes. Therefore, the model of the Scheduler retains the reusability and scalability properties of the Safety Logic.

5.2. The Model of the SL Processes

The PROMELA processes modelling the processes of the Safety Logic all share a fixed structure, with two channels to and from the Scheduler, and implementing the general activation table described in previous section. This structure is instantiated according to the signature of the actual process being modelled. In Fig. 6 we present the instantiation for the case of the LC process. The definition and initialization of the logical variables of the process can be determined according

```
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
}
```

Fig. 6. The PROMELA model for process structure.

to the specification. Then, the process implements a loop (`do - od` instruction) where it first waits for the Scheduler to pass control. Waiting is again modelled by means of handshaking on the synchronous channel `from_sched`. The activation event, which is specified by the Scheduler through a message sent on the channel `from_sched`, 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. The `if - fi` instruction implements guarded commands. For

```
...
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
...
```

Fig. 7. The PROMELA model of the operation specified in Fig. 3.

instance, if the received transition is lc_manual_open, then the jump to the label manual_open can be executed. <operation_body> stands for the set of PROMELA statements which model each operation, discussed below. 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 with the instruction add_next_cycle_processes), 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. Notice that the precise structure of the process (e.g. the number of jumps, the tests) and the information exchanged between modelled SL process (e.g. LC) and the Scheduler can be determined directly from the process signature, and do 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 operations (the <operation_body>'s in Fig. 6). The model of the operations of the process can be obtained directly and mechanically from the operation specifications described in previous section. The fully automatization of this translation, upon which the integration of the model checker as a debugging tool in the process of designing the specifications is based, appears easily feasible. In Fig. 7 we report the PROMELA model for the operation specified in Fig. 3.

For each action specified there is a corresponding PROMELA statement. For instance, the VERIFY instruction I.a corresponds to the test (lc_p_cmb == reverse). Assignments are naturally modelled as PROMELA assignments to the corresponding logical variables. Exceptions are modelled depending on the kind of operations. The exceptions described here correspond to a state operation, and are modelled by setting both variables terminates and continues to false, which amounts to waiting for the conditions to become ready and trying again next cycle.

The correspondence is straightforward. This is due to the fact that PROMELA statements have a semantics which is very close to the semantics of the specifications, which shares a lot with the standard semantics for imperative programming languages. The correspondence between PROMELA and the specification made it possible for the development engineers to understand and validate the PROMELA model of operations.

Notice that a process of the Safety Logic, even with a limited number of operations, may result in a complicated finite state machine. For instance, the Level Crossing process, which has six values for the `lc_prc` state variable, and eleven operations, yields a state machine of 239 control points. (Control points are only one part of the state space of the machine, which includes also the possible values of the logic and control variables.)

5.3. Scaling up to Different Configurations

The general structure of the model described so far has been instantiated to different configurations. As explanatory examples, we report here three of them.

Configuration 1: The first configuration was composed of a physical level crossing, together with two processes in the Safety Logic, namely a Shunting and a Level Crossing.

Configuration 2: We experimented with a more complex configuration, modelling the control of a physical railway track divided in three parts and containing a level crossing. The SL contains four processes. The model 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.

Configuration 3: This is a complex configuration which includes three kinds of processes and seven instances: two Shunting, one Switch, and four Liberation processes.

Configuration 1 was originally proposed by the designers of the Safety Logic, and specified in [CGM96], since they consider it to be of rather complex nature. It involves indeed two processes of different kinds which highly interact. A specification of such configuration is far from being trivial to test.

Configuration 2 is a significant design shift. A new process type is introduced, and the resulting safety logic is rather different from Configuration 1. This has been done on purpose, in order to test the scalability of the model. The interesting fact is that the model scaled up at a very low cost, due to its mechanical correspondence with the original specification of the Safety Logic.

Configuration 3 has been devised mainly in order to test the state explosion problem. Indeed, it adds one new process type (the switch), but involves more instances of existing processes (Shunting, Liberation). To this extent, a remark is in order. Each process in the safety logic is a very complex entity involving a very high number of state variables and state transitions. For instance, level crossings are not modelled in a trivial way as entities which can be either open or closed.

5.4. Bridging the Semantic Gap

The PROMELA model is extremely easy to understand by the SL designers. One reason is that PROMELA has been designed to be easily understood by designers (in this respect it is far easier than other formalisms, e.g. first order logic). Moreover, the understanding of the model of the SL processes is facilitated by the mechanical correspondence between the processes specifications and their PROMELA model. This helps in gaining confidence that the model is faithful to the actual system being designed.

Nevertheless, even for a simple process configuration, the model of the safety logic can be very complicated, though highly structured. Even for a two process configuration, the model results in more than 2000 lines of PROMELA. Therefore, there is a need to reduce as much as possible the semantic gap, i.e. to make sure that the formal definitions do not describe a different system than the one intended. This was done by using SPIN and its graphical interface tool as a guided and random simulator. This allowed to gain confidence in the behaviour of the model with respect to the modelled system. `assert` instructions[1] were added to the model in order to make sure that expected conditions were holding in the model. For instance, in Fig. 7 the statement `assert(0)` in the `else` branch of the `State operation` section explicitly signals an error if the logical variable `lc_prc` has a value which is not compatible with the process signature specification. Other assertions were written to enforce invariants in the Scheduler behaviour. For instance, at the end of the cycle the list of current processes and the list of automatic commands must be empty. Thanks to this technique, several bugs were removed from the model in the initial phase of the simulation. These explicit checks proved particularly useful in order to detect type mismatches. Notice indeed that symbolic data are obtained in PROMELA by expansion into numerical data, and therefore strong type checking is not automatically performed.

6. Formal Verification

SPIN can be used as an exhaustive state space analyzer, in which case it is capable of rigorously proving the validity of user specified correctness requirements. The idea is that the (initial) states of the model are recursively expanded, the resulting states are stored, until states can no longer be expanded (i.e. exploration is completed), or a violation of some form (e.g. a deadlock, a behaviour which does not obey the specified requirements) is detected. SPIN tackles the state explosion problem with different techniques (e.g. state space compression, partial order reduction) to optimize the search (see [Hol91]).

The aim of the project was to assess the possibility of introducing the exhaustive state space analyzer as a powerful debugging tool used by designers during the design phase. The design phase cannot be slowed down. Each delay in this phase can result in very high costs, due to a consequent delay in the delivery

[1] The instruction `assert(cond)` states that the condition `cond` must be true in every state where the instruction is executed, otherwise an error is signaled.

Table 1. Experimental results

	S.M	S.T	A.M	A.T	N.M	N.T
C1	5MB	6 s	10MB	15 s	7MB	27 s
C2	11MB	17 s	30MB	56 s	18MB	1.15 m
C3	77MB	29.15 m	–	–	–	–
C3.c	17MB	22 s	45MB	1.02m	29MB	1.48 m

of the product. In order to study the applicability of the model checker in this phase, we have tested the performance of the exhaustive analysis in the different configurations that we have modelled.

The first step was to model realistic requirements to be validated. Beyond the usual requirements, such that absence of deadlocks, we have modelled a set of application dependent requirements suggested by the SL designers. One of them is "the process Shunting does not signal free way to the train if the Level Crossing is not closed" (Requirement 1). This is an obvious safety requirement, which was 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" (Requirement 2) were represented in a similar way.

The formalization of the following requirement (that we refer to as Requirement 3) 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 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. This criterion was modelled by means of a linear temporal logic formula, in order to express the relation between the previous and future behaviours of the system. Such temporal logic formulae can be analyzed by SPIN by extending the model of the SL with a translation into observing automata. Another relevant issue (Requirement 4) was testing for the termination of the cycle of the SL (a non termination in the model would result in a deadlock in the actual 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, non termination can occur depending on the nature of the command flow among processes (e.g. if two processes keep sending automatic commands to each other). This was expressed requiring that the beginning of cycle of the SL be a point of progress (this property can be automatically analyzed by a particular modality of the exhaustive analysis).

Given this set of requirements, they were all analyzed with SPIN. Analyses were performed on the different configuration models (see for instance Configuration 1–3, described in the previous section). In the following we discuss some experimental results [2].

The performances obtained analyzing exhaustively the configuration with two

[2] All the tests were executed on a SUN-SPARCSTATION 10 with 128MB RAM. Some of the times reported here are better than the ones reported in [CGM98] thanks to the improved performance of the new releases of SPIN.

processes (Configuration 1) were very good (see the first line in table 1). It was possible to complete a full verification of state properties in less than 6 seconds (S.Time), using less than 5MB of memory (S.Memory). Requirement 2 was validated with the algorithm for acceptance cycles in less than 15 seconds (A.T), with 10MB of memory. The search for non-progress cycles, a very expensive form of analysis, was completed in about 27 seconds (N.T), using 7MB of memory. 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 results in Configuration 2 were again very positive (second line in table 1). The resources required to analyze such a complex configuration were rather limited. The complete analysis of Requirement 3 took less than one minute, while the non-progress cycles check (Requirement 4) was completed in less than 2 minutes.

During the analysis of Configuration 2, 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. 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. 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. The problem was fixed by associating by default every resting state with a fictitious operation which simply returns control to the Scheduler.

7. Scaling up by Model Compression

The analysis of the experimental results reported in previous section highlighted that the model could be further reduced. Once the model of the SL has been fully understood, other considerations such as memory occupation can be taken into account. The basic idea is that we are interested in verifying the input/output behaviour of the Safety Logic. Therefore, any observation which can be performed between these states is actually useless, and wastes memory and states. For instance, every time the control is passed from the Scheduler to one process, the corresponding state must be stored. Furthermore, it is sufficient to store one single control point for the whole Safety Logic, while the information relative to the status of the channels must be stored as well.

This idea is shown in Fig. 8. The whole Safety Logic is represented as a single PROMELA process, composed of the body of the different processes. The difference is that control is passed among different code segments via jumps, rather than by message passing along synchronous channels. This simple idea allows us to hide the internal states of the Safety Logic, and results in a striking improvement in the experimental results, in particular when larger configurations of processes are analyzed. We performed experiments with the seven process configuration (Configuration 3), controlling two two-section tracks and a switch (third line of figure 1). This configuration is very complex. For instance, storing one single state vector requires 248 bytes. An exhaustive analysis of the state space, which contains 2.5 million states, was possible only thanks to SPIN compression techniques (the

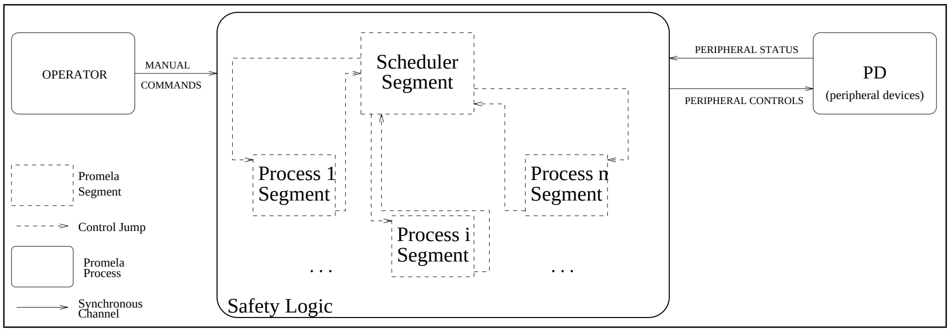

Fig. 8. A compressed PROMELA model of the SL shown in Fig. 4.

actual memory required without compression would be more than 650MB). It was not possible to complete the search for acceptance and non-progess cycles. With the compressed model of the Safety Logic, the size of the state vector is reduced to 108 bytes. This makes the model much more manageable (fourth line of figure 1), and allows for a complete state exploration within 22 seconds. The whole state space is only 130 thousand states. The non-progress cycles analysis took less than 2 minutes.

8. Related Work

There is a vast literature on application of formal methods to railway systems. Many approaches (e.g. [Hoo95, Mor97]) focus on the ability to prove theorems off-line rather than at design time.

The formal verification of the same application discussed in this paper was tackled in an independent project [BFG98]. There are however several differences. First, the model focuses on the modelling of the operations in process algebra, and does not seem to take into account the structure of the Scheduler. This approach seems to generate complex models, and interesting abstraction techniques are used to tackle the problem. Abstraction techniques, however, unless strongly automatized (see e.g. [Kur94]), can be hard to integrate within the development cycle as they may require human intervention of specialized users.

In [HGC98] the Verus [Cam96] tool is used to analyze yet the same system of specifications. Verus allows for the analysis of real-time systems. The input language is a C-like language extended with constructs for specifying timing information. The analysis of temporal and real time properties is based on Binary Decision Diagrams (OBDD) [Bry86]. Although OBDD are widely used for the analysis of synchronous systems, the experience discussed in [HGC98] did not report substantial advantages in terms of scaling up to model complex configurations.

A very interesting class of applications to the verification of railway interlocking systems are the ones based on automated procedures for propositional logic and linear arithmetic [Bor97, GVK95]. These application are based on the remarkable Stålmarck procedure, a patented method for propositional satisfiability [StS90]. The idea is to model the dynamics of the system in propositional logic by imposing a time bound on the temporal interval to be analyzed. The applications validated automatically with this approach appear to be of very high complexity, which is dealt with very efficiently by the automated procedures. The major difference with our work is that the Safety Logic has a highly recursive behaviour, more than the programs in VPI's (see [GVK95]), which seems not to be easy to handle by provers based on the Stålmarck method.

9. Conclusions

In this paper we have described an industrial application of formal methods, where we have used model checking techniques to model and formally verify a rather complex part of an Ansaldo system used to perform railway interlocking functions. We have assessed the possibility of introducing this novel technique in the development cycle of Ansaldo with an advantageous costs/benefits relation. The project has revealed the possibility of integrating the formalization step and the formal verification step with the current development process. The formal model structure retains the reusability and scalability properties of the system being modelled. Part of it is defined once for all at a low cost, and re-used. The rest of the model can be mechanically generated from the designers' current specification language. The model checker is "hidden" to the user, it runs as a powerful debugger. Its performances are impressive: exhaustive analysis of quite complex configurations with respect to rather complex properties are run in order of minutes. This is essentially due to a carefully designed model which exploits all the behaviour evolution constraints. The re-usability/scalability of the model and the fact that formal verification is automatic and efficient are the key factors which open up the possibility of a real usage by designers at design time.

This project has been a successful experience in formal methods applied to the design of safety-critical systems. Although the configurations analyzed in this project are simpler than the ones needed to control a large-size railway station, it is 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 traditional methods such as testing, as very often the exhaustive analysis of a scaled down configuration can reveal problems which are present also in larger configurations.

Furthermore, this experimental project successfully higlighted that formal methods, if applied in the right way, can provide significant advantages. Ansaldo is currently applying formal methods in the development of several safety-critical systems, in order to achieve validation at earlier stages of the development cycle.

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 with different (e.g. symbolic) model checking techniques [CCG98] have shown that there is room for improvement by using different representations of the system, and decomposition and abstraction techniques.

References

[BFG98] Bernardeschi, C., Fantechi, A., Gnesi, S., Larosa, S., Mongardi, G. and Romano, D.: A Formal Verification Environment for Railway Signaling System Design. *Formal Methods in System Design*, 12(2):139–161, March 1998.

[Bor97] Borälv, A.: A Fully Automated Approach for Proving Safety Properties in Interlocking Software Using Automatic Theorem-Proving. In S. Gnesi and D. Latella, editors, *Pro-*

ceedings of the Second International ERCIM Workshop on Formal Methods for Industrial Critical Systems, Pisa, Italy, July 1997.

[Bry86] Bryant, R. E.: Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BoS93] Bowen, J. P. and Stavridou, V.: Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209, July 1993.

[Cam96] Campos, S.: *A Quantitative Approach to the Formal Verification of Real-Time Systems.* PhD thesis, SCS, Carnegie Mellon University, 1996.

[CCG98] Cimatti, A., Clarke, E., Giunchiglia, F. and Roveri, M.: NuSMV: a new Symbolic Model Verifier. In *Proceedings of CAV99*, Trento, Italy, July 1999.

[CGM96] Cimatti, A., Giunchiglia, F., Mongardi, G., Pietra, B., Romano, D., Torielli, F. and Traverso, P.: Formal Validation of an Interlocking System for Large Railway Stations: A Case Study. Confidential IRST Technical Report, 1996.

[CGM98] Cimatti, A., Giunchiglia, F., Mongardi, G., Romano, D., Torielli, F. and Traverso, P.: Model Checking Safety Critical Software with SPIN: an Application to a Railway Interlocking System. In *Proceedings of SAFECOMP'98 – Seventeenth International Conference on Computer Safety, Reliability and Security*, Heidelberg, Germany, 1998. Presented at the Third SPIN Workshop, Twente University, Enschede, The Netherlands, April 1997.

[GVK95] Groote, J. F., van Vlijmen, S. F. M. and Koorn, J. W. C.: The Safety Guaranteeing System at Station Hoorn-Kersenboogerd. In *Proceedings of COMPASS'95*, 1995.

[HGC98] Hartonas-Garmhausen, V., Campos, S., Cimatti, A., Clarke, E. and Giunchiglia, F.: Verification of a Safety-Critical Railway Interlocking System with Real-time Constraints. In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing (FTCS-28)*, pages 458–463, Munich, Germany, June 1998. IEEE Computer Society Press.

[Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes.* Prentice-Hall International, London, 1985.

[Hol91] Holzmann, G. J.: *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[Hol97] Holzmann, G. J.: The Spin Model Checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[Hoo95] Hoover, D. N.: A Mathematical Model for Railway Control Systems. Technical report, Odyssey Research Associates, Ithaca, NY 14850 USA, June 1995.

[Kur94] Kurshan, R. P.: *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[LFB96] Larsen, P. G., Fitzgerald, J. and Brookes, T.: Applying Formal Specification in Industry. *IEEE Software*, 13(7):48–56, May 1996.

[Mon92] Mongardi, G.: 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.

[Mor97] Morley, M. J.: Safety-level Communication in Railway Interlockings. *Science of Computer Programming*, 29:147–170, 1997.

[StS90] Stålmarck, G. and Säflund, M.: Modelling and Verifying Systems and Software in Propositional Logic. *Ifac SAFECOMP'90*, 1990.

[Ste95] Miller, Steven P. and Srivas, Mandayam.: Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, 1995. IEEECS.

[VaW86] Vardi, M. Y. and Wolper, P.: Automata-Theoretic Techniques for Modal Logics of Programs. *Journal of Computer and System Sciences*, 32:183–221, 1986.

Received March 1997
Accepted in revised form July 1998