ELSEVIER

Science of Computer Programming 36 (2000) 53–64

Science of Computer Programming

www.elsevier.nl/locate/scico

Verification of a safety-critical railway interlocking system with real-time constraints ☆

Vicky Hartonas-Garmhausen[a,*], Sergio Campos[b], Alessandro Cimatti[c], Edmund Clarke[d], Fausto Giunchiglia[c,e]

[a] *Carnegie Mellon University, Pittsburgh, PA 15213, USA*
[b] *Federal University of Minas Gerais, Brazil*
[c] *Istituto per la Ricerca Scientifica e Tecnologica (IRST), Trento, Italy*
[d] *Carnegie Mellon University, Pittsburgh, PA 15213, USA*
[e] *DISA, University of Trento, Trento, Italy*

Abstract

Ensuring the correctness of computer systems used in life-critical applications is very difficult. The most commonly used verification methods, simulation and testing, are not exhaustive and can miss errors. This work describes an alternative verification technique based on symbolic model checking that can automatically and exhaustively search the state space of the system and verify if properties are satisfied or not. The method also provides useful quantitative timing information about the behavior of the system. We have applied this technique using the Verus tool to a complex safety-critical system designed to control medium and large-size railway stations. We have identified some anomalous behaviors in the model with serious potential consequences in the actual implementation. The fact that errors can be identified before a safety-critical system is deployed in the field not only eliminates sources of very serious problems, but also makes it significantly less expensive to debug the system. © 2000 Published by Elsevier Science B.V. All rights reserved.

Keywords: Safety-critical systems; Railway systems; Formal verification; Symbolic model checking; Quantitative analysis

☆ This research is sponsored by the Semiconductor Research Corporation (SRC) under Contract No. 97-DJ-294, the National Science Foundation (NSF) under Grant No. CCR-9505472, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-96-C-0071. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, NSF, DARPA, or the United States Government.

* Corresponding author.

E-mail addresses: hartonas@cs.cmu.edu (V. Hartonas-Garmhausen), scampos@dcc.umfg.br (S. Campos), cimatti@irst.itc.it (A. Cimatti), emc@cs.cmu.edu (E. Clarke), fausto@irst.itc.it (F. Giunchiglia)

0167-6423/00/$ - see front matter © 2000 Published by Elsevier Science B.V. All rights reserved.
PII: S0167-6423(99)00016-7

1. Introduction

Ensuring the correctness of computer systems is a complex task of paramount importance, especially when such systems control and monitor life-critical operations. The verification of industrial computer systems is particularly difficult due to their size and complexity. The most frequently used methods, simulation and testing, are not exhaustive and can miss important errors. While the use of both methods can increase the reliability of the application, they cannot fulfill the verification needs of modern complex safety-critical systems. Formal methods are an additional methodology to tackle this problem. Formal verification tools allow an exhaustive search to be automatically performed on the state space of the system, avoiding the shortcomings of both simulation and testing.

In this paper we describe a case study in formal verification based on a real industrial system. We verified the safety logic of ACC ("Apparato Centrale a Calcolatore") [26], a complex real-world safety-critical system developed by Ansaldo Transporti for the control of medium and large-size railway stations.

The verification was performed using Verus [7, 13], a formal verification tool, which combines symbolic model checking and quantitative timing analysis. Verus allows for the formalization of systems in an imperative language with a syntax similar to C. This language includes special constructs for the straightforward expression of timing properties, simplifying the description of real-time and safety-critical systems. The language is compiled into state-transition graphs, to which powerful symbolic model checking and quantitative algorithms can be applied. Symbolic model checking algorithms search the state space exhaustively to determine whether the model satisfies its specifications. The method has proven to be very successful in finding design errors in several industrial systems and protocols [6, 10, 12, 18]. Moreover, Verus extends the power of model checking by allowing the determination of quantitative timing information such as response time to events, schedulability of a set of tasks, and performance measures.

The model of a Verus program is a finite state-transition graph, where each state is one possible assignment of values to all variables in the model. Verus uses a discrete notion of time. Each transition in the graph corresponds to one time unit. The simplicity of this representation makes it amenable to a symbolic implementation using binary decision diagrams. This representation is very efficient, as attested by the complex systems verified. One example has 15 concurrent processes and counterexamples that have thousands of states produced in seconds. Perhaps more indicative of the usefulness of the method than the size of the counterexamples are the types of systems verified. We have applied this method to the verification of several real systems, such as an aircraft controller [9], a robotics controller [11] and a distributed heterogeneous real-time system [14]. In all cases, the examples verified are either actual existing systems or use components and protocols employed in current industrial products.

We use a discrete notion of time. In recent years, there has been considerable research on algorithms that use continuous time [1, 2, 21, 22]. Most of these techniques use a transition relation with a finite set of real-valued clocks and constraints on times when

transitions may occur. It can be argued that such algorithms lead to more accurate results than discrete time algorithms. However, an uncountable infinite state space is required to handle continuous time, because the time component in the states can take arbitrary real values. Most verification procedures based on this paradigm depend on constructing a finite quotient space called a region graph out of the infinite state space. Unfortunately, the region graph construction is very expensive in practice and current implementations of the algorithms can only handle quotient spaces with at most a few thousand states. This makes it impossible to verify large complex systems using continuous time tools.

In this work a formal model of the interlocking system has been produced in the input language of Verus. A set of qualitative (e.g. safety, liveness) and quantitative (e.g. response times) properties have been automatically analyzed. Despite the complexity of the system (the model has about 10^{27} states) the analysis has been performed within minutes. A subtle and anomalous behavior leading to a deadlock of the system has been discovered by Verus. The anomalous behavior was pinpointed by an automatically generated counterexample trace, showing precisely the behavior leading to the violating state. The same behavior had blocked the entire operation during a field test of an earlier design of the system.

Related work.

A large part of literature on the application of formal methods to railway systems (e.g. [24, 27]) addresses the proof of requirements with theorem proving tools. These tools are often difficult to apply at design time because they require an intense and specialized user-intervention.

The formal verification of the application discussed in this paper has also been tackled in an independent project by Bernardeschi et al. [3]. There are however significant differences: the model focuses on the modeling of the operations in process algebra and does not take into account the structure of the Scheduler. This approach generates complex models and abstraction techniques, unless strongly automatized (see [25]), are hard to apply within the development cycle as they often require human intervention of specialized users.

The same system of specifications has been analyzed [15, 16] using the SPIN [23] tool. SPIN is mainly designed for the analysis of asynchronous systems and protocols. The input language is an imperative style programming language, extended with constructs for specifying nondeterministic computations and communication. The analysis of temporal and real-time properties is based on explicit state space search. For this particular application, the compression techniques implemented in SPIN allow the user to limit the amount of memory required for the search, usually the main bottleneck in explicit-state verification, and thus scale up the analysis to complex interlocking configurations.

An interesting class of applications in the verification of railway interlocking systems is the one based on automated procedures for propositional logic and linear arithmetic [4, 20]. These applications are based on the Stålmarck procedure, a patented method for propositional satisfiability [28]. 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 the work presented here is that the safety logic has a highly recursive behavior, more than the programs in VPIs [20], which seems difficult to handle by provers based on the Stålmarck method.

Outline. This paper is structured as follows. In Section 2 we present the Interlocking system. In Section 3 we describe the formal model and in Section 4 we describe the formal verification of the safety logic. In Section 5 we draw some conclusions.

2. The application

We focused on a complex real-world safety critical application developed by Ansaldo Transporti, 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. 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.

Two intrinsic sources of complexity make the verification of this system an important work. The first is the large size of the controlled physical plants. Large railway stations may include as many as 2000 physical devices. The second source of complexity is due to 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 does not know when the next resource will be requested, or when a peripheral device may fail. Furthermore, the system is subject to timing constraints, as it is important to ensure bounds to response time.

The "safety logic" of the ACC is a software subsystem that implements the logical functions requested by the external operator. A high-level picture of the Safety Logic of the ACC, together with its environment, is shown in Fig. 1. The Safety Logic (SL), which is connected to the peripheral devices of the station and to an external operator, can be thought of as a deterministic reactive controller embedded in a nondeterministic environment. The inputs to the system include manual commands from the external operator and sensor readings from the peripheral devices. The external operator may issue the following commands: "Open level crossing 1", or "Set route from track 2 to track 5". Physical device sensors may report "Level crossing 1 is open", or "Switch 2

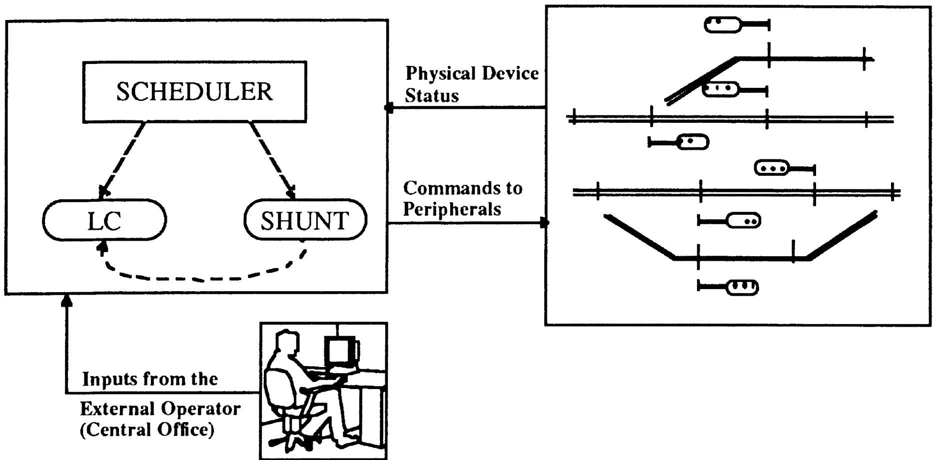

Fig. 1. The ACC safety logic.

is in normal position”. The outputs of the Safety Logic are the controls of the physical devices, such as “ Move switch 12 to normal position”, “Close level crossing 3”.

This is achieved by means of a logical architecture based on a Scheduler controlling the activation of application-dependent processes. The Scheduler is a cyclic program, which activates, suspends, and terminates processes according to the process execution status. One Scheduler is used across all configurations.

The specifications define the behavior of processes, i.e. how the process communicates with other processes, accesses and modifies variables, reacts to exceptions, and so on. Each process is associated with a set of state variables, which depend on the particular physical configuration. State variables are distinguished into logical variables, which represent the status of the process computation, and control variables, which represent the status of the peripheral devices as they are read by the sensors. A process can modify the value of a logical variable during the execution of the corresponding operations, but it can not modify the value of a control variable. The control variables are set at the beginning of each cycle and do not change until the next cycle. Processes may issue physical device commands and automatic commands, which are commands from one process to another. Processes are often organized in a hierarchical fashion: a process, which sets routes, may control a lower process, which controls a physical device. The Safety Logic performs a single-thread computation, i.e. at most one process is active at any one instant. Processes are activated in a master-slave way: the Scheduler passes control to the process and suspends its own execution until the process returns control. Global variables keep track of the status of the computation and control the execution of the processes.

The system behavior is defined by operations, i.e. collections of basic actions to be performed by the process. Operations include testing the value of variables, assigning values to logical variables, sending commands to the peripheral devices, and sending

automatic commands to other processes. An activation table associates an operation to each event determining the activation of the process. Operations are characterized by the event that causes their activation: manual operations correspond to manual commands, state operations to process states, and automatic operations to automatic commands. The actions above are conditioned to tests. Operations consist of statements, which are interpreted sequentially following the schema shown in Fig. 2. The `VERIFY` tests are executed first. If one of the tests fails, the corresponding `EXCEPTION` action is executed, and the operation ends. If the preliminary tests are satisfied, commands are issued during the `SEND` part and variables are set during the `ASSIGN` part.

The Scheduler executes operations of different types (e.g. manual, automatic, state) in different phases of the cycle. The specifications also determine what the process should do after the execution of an operation. A process can terminate the activation and go in a resting state, continue the current activity by executing another operation, and suspend its execution to the next cycle. In the last case, the Scheduler will reactivate the process at next cycle. Two queues are used by the Scheduler to store the processes to be reactivated at current and next cycle.

3. A formal model of the safety logic

We verified a two process configuration of the Safety Logic. The system, which controls the safe operation of a level crossing, is composed of the Scheduler and two processes, `LC` and `SHUNT`, with over 17 operations and 18 different configurations of the physical level crossing. The specifications, which were defined in a confidential technical report during a technology transfer project involving IRST and Ansaldo, were considered to be of significant complexity due to the intrinsic complexity of the actual Scheduler software design, which we modeled, and the large state space of the system. In this section we describe how the SL together with its environment were modeled in Verus. In the next section we show how the properties and timing requirements were modeled and verified in Verus avoiding a state explosion.

A formal model of the actual SL was produced in the Verus language. The imperative C-like language provided by Verus made it straightforward to express the Safety Logic of the ACC. The Verus model preserves the cyclic structure of the SL, which repeatedly acquires inputs from the external environments, evaluates the logic, and activates processes. The main loop of the SL is implemented by means of a never-ending `while` construct. Fig. 3 shows a segment of the Verus program that implements the manual operation in SL corresponding to the manual command `CLOSE_GATE`.

Language constructs have been kept simple in order to make the compilation into a state-transition graph as efficient as possible. Simple constructs allow the precise expression of the desired features, since complex constructs sometimes force unnecessary details into the specification. Time passes only on `wait` statements. This feature allows a more accurate control of time, generates smaller models, since contiguous statements are collapsed into one transition, and eliminates the possibility of implicit delays

```
OPERATION
I - VERIFY
a. "VARIABLE1" with "VALUE2"
b. "VARIABLE23" with value other than "VALUE1"
c....
II - SEND
a. the PD command "COMMAND1"
b. to "PROCESS2" the automatic command "COMMAND2"
III - ASSIGN
a. to "VARIABLE5" the value "VALUE3"
only if:
1. "VARIABLE2" has the value "VALUE4"
b. to...
IV -AFTER THE OPERATION OF THE PROCESS
a. terminates
only if:
1. "VARIABLE3" has value other than "VALUE5"
b. does not terminate;
only if:
1. "VARIABLE3" has the value "VALUE5"
c. continues
only if:
1. "VARIABLE3" has the value "VALUE5"
EXCEPTIONS:
[a]
LOST COMMAND
ACTIONS:---
[b]
WAITING
ACTIONS:---
[c]
WAITING
ACTIONS:---
```

Fig. 2. The schema of operations.

influencing verification results. Smaller representations can then be generated, which is critical to the efficiency of the verification and permits larger examples to be handled. Details about the Verus language can be found in [7, 8].

Verus supports nondeterminism, which allows partial specifications to be described. For example, we have used this feature, which is implemented with the `select` statement, to assign the manual command at the start of each cycle:

```
MAN_cmd = select{ NO_CMD, CLOSE_GATE, OPEN_GATE,
        RESTORE_AUTO, ACTIVATE_SHUNT, KILL_SHUNT};
```

```
...
if (MAN_cmd == CLOSE_GATE) {
if (CMD_state != MANUAL){
CMD_state = MANUAL;
LC_state = REQUESTED-CLOSING;
}
}
wait(1);
...
```

Fig. 3. Example of verus code.

The activation of processes is defined by means of queues and the priority mechanism. Verus allows an elegant formalization of queues based on dynamic scheduling. We created the request variables `req1` and `req2` (for the current-cycle-processes) and `nreq1` and `nreq2` (for the next-cycle-processes) to be integers with values corresponding to the priority level at which each process is requesting execution. The Scheduler chooses the process with the highest requested priority. At the beginning of each cycle we copy the list of next-cycle-processes to the current-cycle-processes and re-initialize the next-cycle-process list to the empty list:

```
req1 = nreq1;   nreq1 = 0;

req2 = nreq2;   nreq2 = 0;
```

The request variables are updated depending on the type of transition (manual, state, or automatic) and whether the process terminates or continues its execution. For example in the manual phase, if process `SHUNT` does not terminate and does not continue, we schedule its execution for the next cycle with priority higher than the priority of process `LC`:

```
if (nreq2 == 0 && req2 == 0) nreq2 = nreq1 + 1;
```

The Scheduler checks whether there are state processes to run in the current cycle

```
while(req1 != 0 || req2 != 0)
```

and then activates process `LC` if `req1` $\geqslant$ `req2` and process `SHUNT` if `req2` $\geqslant$ `req1`.

The Verus exception handling mechanism provides a general way to signal that a certain condition is not verified. This feature has been exploited to gain confidence in the model by expressing a number of simple properties. For instance, we checked the valid range of variables and the reachability of certain control points.

Process instantiation in Verus follows a synchronous model. All processes execute in lock step, with one step in any process corresponding to one step in the other processes. Asynchronous behavior can be modeled by using stuttering, which introduces nondeterministic transitions and effectively models the desired behavior. This technique is described in detail in [7].

4. Formal verification of the safety logic

The ACC SL has been verified using algorithms derived from symbolic model checking because these algorithms are amenable to efficient implementations using symbolic techniques [6]. The transition relation is represented by boolean formulas, and implemented by binary decision diagrams [5]. This usually results in a much smaller representation for the transition relation, allowing the verification of models several orders of magnitude larger than those verified using traditional implementations. We have used the Verus verification tool, which implements these techniques.

Verus allows the verification of untimed properties expressed as CTL formulas [16] and timed properties expressed as real-time CTL, RTCTL, formulas [19]. CTL formulas allow the verification of properties such "*p* will eventually occur", or "*p*" will never be asserted". However, it is not possible to express bounded properties such as "*p* will occur in less than 10 ms" directly. RTCTL model checking overcomes this restriction by introducing time bounds on all CTL operators. For example, the formula $\mathbf{AG}(req \rightarrow \mathbf{AF}_{0..10}\ ack)$ specifies that requests will always be acknowledged in 10 time units or less. We represented the following safety properties as invariants in CTL:

1. The process `SHUNT` does not issue `CLEAR-SIGNAL` if the Level Crossing is not closed.
2. If the low signal was issued `CLEAR-SIGNAL`, the loss of the closed control of the Level Crossing implies the stop of issue of `CLEAR-SIGNAL`.
3. The Safety Logic never issues contradictory commands during a cycle or the same command twice in a cycle.

These properties, which were verified within seconds, are true. Next, we checked for the absence of deadlocks and the termination of cycles. We had to ensure that the recursion, which may happen when a process finishes executing an operation and continues with another operation during the same cycle, must terminate. This requirement was modelled as a property of the form AF(`end of cycle`). During this analysis a deadlock was found. Verus produced a counterexample, pointing to the loop which happens when the `RESTING` state of a process is activated while that process is running in the state phase. The specifications did not include state operations corresponding to the `RESTING` state. The loop occurred after a long execution sequence of events. The counterexample is 58-steps deep pointing to complex interactions among various elements of the system. It is unlikely that simulation or other verification methods can generate similar information. A similar problem, which blocked the operation of the system, was reported during a field test on an early version of the Safety Logic. The problem was fixed by defining a null operation associated with the resting state, which simply terminates.

Verus implements algorithms that determine the minimum and maximum length of all paths leading from a set of starting states to a set of final states. It also has algorithms that calculate the minimum and the maximum number of times a specified condition can hold on a path from a set of starting states to a set of final states. For example, by choosing as starting states those in which a process requests execution,

and as final states those in which the process completes execution we can compute the response time for that process. If we specify as third condition for the same intervals the execution of lower-priority processes we can compute the amount of priority inversion time that can affect the process.

Several types of information can be produced by this method. Response time to events is computed by making the set of starting states correspond to the event, and the set of final states correspond to the response:

```
MIN(init_cycle, end_of_cycle);

MAX(init_cycle, end_of_cycle);
```

Schedulability analysis can be done by computing the response time of each process in the system and comparing it to the process deadline. A performance analysis was carried out by exploiting the timing primitives and the quantitative algorithms of Verus. We modeled performance of operations by generating different models of the SL. We first used a unit weight for each operation, and then we specified different weights taking into account the number of basic actions (e.g. tests, assignments). The quantitative analysis provided detailed insight into the system behavior, which can be used to optimize the performance and improve the reliability of the system.

5. Conclusions

In this paper we have described the verification of a safety critical railway interlocking system called ACC. The system has been modeled and analyzed with the Verus tool, which uses efficient symbolic algorithms for the verification of complex systems. Verus uses a language especially designed to allow a natural representation of real-time software systems. It also combines symbolic model checking and quantitative analysis techniques to determine system correctness and to provide useful information about its behavior.

The ACC is a very complex system designed to control medium to large railway stations. Because of this it has been necessary to take advantage of several features of Verus in order to complete the verification. The use of a procedural ‘C-like’ language has made it possible to model the system in a straightforward manner. The efficiency of BDD-based algorithms has been vital to verify a model that has more than 10^{27} states. Two other features have been very important in understanding how the model behaves: quantitative analysis has determined the performance of the system and the counterexamples generated have assisted in understanding what the results mean.

This case study shows how formal methods can be used to ensure the correctness of safety-critical systems of industrial complexity. It also shows that the Verus tool is a viable alternative in the verification of such systems. Based on these results we believe that not only can formal methods be used in current industrial systems but also that they can provide results that cannot be obtained by other means.

References

[1] R. Alur, C. Courcourbetis, D. Dill, Model-checking for real-time systems, Proceedings of the fifth Symposium on Logics in Computer Science, 1990, pp. 414–425.

[2] R. Alur, D. Dill, Automata for modeling real-time systems, Lecture Notes in Computer Science, 17th ICALP, Springer, Berlin, 1990.

[3] C. Bernardeschi, A. Fantechi, S. Gnesi, S. Larosa, G. Mongardi, D. Romano, A formal verification environment for railway signaling system design, Formal Methods System Des. 12(2) (1998) 139–161.

[4] Borälv, A fully automated approach for proving safety properties in interlocking software using automatic theorem-proving, in: S. Gnesi, D. Latella (Eds.), Proceedings of the Second International ERCIM Workshop on Formal Methods for Industrial Critical Systems, Pisa, Italy, July 1997.

[5] R.E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Trans. Comput. C-35(8) (1986).

[6] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, Symbolic model checking: 10^{20} States and Beyond, Inform. Comput. 98 (2) (1992).

[7] S. Campos, A quantitative approach to the formal verification of real-time systems, Ph.D. Thesis, SCS, Carnegie Mellon University, 1996.

[8] S. Campos, E. Clarke, The Verus language: representing time efficiently with BDDs, in Fourth AMAST Workshop on Real-Time Systems, Concurrent and Distributed Software, 1997.

[9] S.V. Campos, E.M. Clarke, W. Marrero, M. Minea, H. Hiraishi, Computing quantitative characteristics of finite-state real-time systems, in: IEEE Real-Time Systems Symposium, 1994.

[10] S. Campos, E. Clarke, W. Marrero, M. Minea, Verifying the performance of the PCI local bus using symbolic techniques, in International Conference on Computer Design, 1995.

[11] S.V. Campos, E.M. Clarke, W. Marrero, M. Minea, Timing analysis of industrial real-time systems, in Workshop on Industrial-strength Formal specification Techniques, 1995.

[12] S. Campos, E. Clarke, W. Marrero, M. Minea, Verus: a tool for quantitative analysis of finite-state real-time systems, in ACM SIGPLAN 30 (11) (1995).

[13] S. Campos, E. Clarke, M. Minea, The Verus tool: a quantitative approach to the formal verification of real-time systems, in Conference on Computer Aided Verification, 1997.

[14] S.V. Campos, O. Grumberg, Selective quantitative analysis and interval model checking: verifying different facets of a system, Comput. Aided Verif. (1996).

[15] A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, P. Traverso, Model checking safety critical software with SPIN: an application to a railway interlocking system, in Proceedings of the Third SPIN Workshop, Twente University, Enschede, The Netherlands, April 1997.

[16] A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, P. Traverso, Formal verification of a railway interlocking system using model checking, Formal Aspects Comput. 10 (1998) 361–380.

[17] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM TOPLAS 8(2) (1986) 244–263.

[18] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, L. Ness, Verification of the Futurebus+ cache coherence protocol, in Proceedings of the 11th CHDL, 1993.

[19] E. Emerson, A. Mok, A. Sistla, J. Srinivasan, Quantitative temporal reasoning, in: Lecture Notes in Computer Science, Computer Aided Verification, Springer, Berlin, 1990.

[20] J.F. Groote, S.F.M. van Vlijmen, J.W.C. Koorn, The safety guaranteeing system at station Hoorn-Kersenboogerd, in Proceedings of the COMPASS'95, 1995.

[21] T. Henzinger, X. Nicollin, J. Sifakis, S. Yovine, Symbolic model checking for real-time systems, in Proceedings of the 7th Symposium on Logic in Computer Science, 1992.

[22] T. Henzinger, P. Ho, H. Wong-Toi, HyTech: the next generation, in IEEE Real-Time Systems Symposium, 1995.

[23] G.J. Holzmann, Design and Validation of Computer Protocols, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[24] D.N. Hoover, A mathematical model for railway control systems, Technical Report, Odyssey Research Associates, Ithaca, NY 14850 USA, June 1995.

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

[26] G. Mongardi, Dependable computing for railway control systems, in Proceedings of the Working Conference on Dependable Computing for Critical Applications, IFIP Working Group, 1992, pp. 255–273.
[27] M.J. Morley, Safety-level communication in railway interlockings, Sci. Comput. Programm. 29 (1997) 147–170.
[28] G. Stålmarck, M. Sä, Modelling and verifying systems and software in propositional logic, in Ifac SAFECOMP'90, 1990.