IRST

ISTITUTO PER LA RICERCA
SCIENTIFICA E TECNOLOGICA

38050 Povo (Trento), Italy
Tel.: + 39 461 314575 · Fax: + 39 461 314591
e-mail: prdoc@itc.it · url: http://www.itc.it

Verification of a Safety-Critical Railway Interlocking System with Real-time Constraints

Vicky Hartonas-Garmhausen, Edmund Clarke, Sergio Campos, Alessandro Cimatti, Fausto Giunchiglia

February 1998

Technical Report # 9802-01

Verification of a Safety-Critical Railway Interlocking System with Real-time Constraints

Vicky Hartonas-Garmhausen, Edmund Clarke
Carnegie Mellon University, USA
{hartonas,emc}@cs.cmu.edu

Sergio Campos
Federal University of Minas Gerais, Brasil
scampos@dcc.ufmg.br

Alessandro Cimatti, Fausto Giunchiglia[†]
Istituto per la Ricerca Scientifica e Tecnologica (IRST), Italy
[†] DISA, Universita' di Trento, Italy
{cimatti,fausto}@irst.itc.it

February 4, 1998

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 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 potential sources of serious problems, but also makes it significantly less expensive to debug the system.

Keywords: Formal Methods, Symbolic Model Checking, Quantitative Analysis, Real-Time Systems, Railway Control Systems.

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") [12], a complex real-world safety-critical system developed by Ansaldo Transporti for the control of railway stations.

The verification was performed using VERUS [3, 7], 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 [2, 5, 6, 9]. 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.

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 significance of this behaviour is that it was present in a prototype version of the Safety Logic.

The same system has also been verified in SPIN [8]. The present work has provided insight into the ACC design by means of verifying CTL formulas and computing quantitative timing information, which cannot be performed in SPIN. Furthermore, in order to scale up the verification for larger configurations, the use of symbolic methods based on Ordered Binary Decision Diagrams implemented in VERUS is more powerful compared to the explicit state search used in SPIN.

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 Figure 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, for instance, commands of the following type: "Open level crossing 1", or "Set route from track 2 to track 5". Physical device sensors may report, e.g., "Level crossing 1 is open", or "Switch 2 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

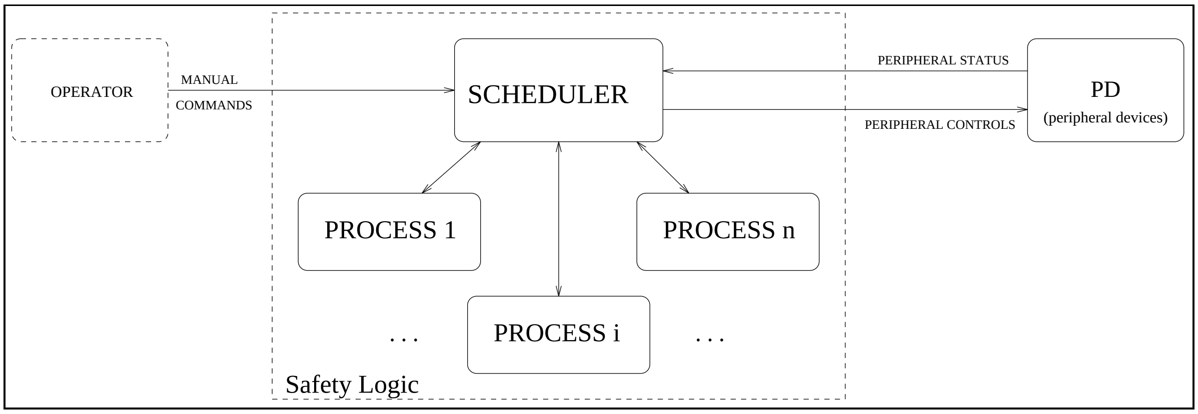

Figure 1: The architecture of the SL and its environment

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.

Processes are designed by means of structured specifications written in a structured semi-natural language. The specifications define the behavior of processes, i.e. how a 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 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. Basic actions may be conditioned to tests. Operations consist of statements

```
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:---
```

Figure 2: The Schema of Operations

which are interpreted sequentially following the schema shown in Figure 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 for this case study, 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. Figure 3 shows a segment of the VERUS program which implements the manual operation in SL corresponding to the manual command `CLOSE_GATE`:

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

Figure 3: An Example of VERUS code

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 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 [3, 4].

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};
```

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 and process `SHUNT` if .

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 [3].

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 [2]. The transition relation is represented by boolean formulas, and implemented by Binary Decision Diagrams (BDD's) [1]. 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.

CTL and RTCTL Symbolic Model Checking

Verus allows the verification of untimed properties expressed as CTL formulas [10] and of timed properties expressed as real-time CTL, RTCTL, formulas [11]. 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 10ms" directly. RTCTL model checking overcomes this restriction by introducing time bounds on all CTL operators. For example, the formula

$$\mathbf{AG}(req \supset \mathbf{AF}[0\ldots 10]ack)$$

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 modeled as a property of the form $\mathbf{AF}(\textit{end of cycle})$. During this analysis a deadlock was found. This turned out to be a behavior where the system actually would deadlock, given that there is no operation which can be executed in the state. 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. Indeed, the `RESTING` state of either process is not defined; the specifications do 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 interaction among various elements of the system. It is unlikely that simulation or other verification methods can generate similar information. A similar behaviour was present in a prototype version of the Safety Logic. The problem was fixed by defining a null operation associated with the resting state, which simply terminates.

Quantitative Algorithms

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

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.

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

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

[5] S. Campos, E. Clarke, W. Marrero, and M. Minea. Verifying the performance of the pci local bus using symbolic techniques. In *International Conference on Computer Design*, 1995.

[6] S. Campos, E. Clarke, W. Marrero, and M. Minea. Verus: a tool for quantitative analysis of finite-state real-time systems. *ACM SIGPLAN Notices*, 30(11), November 1999.

[7] S. Campos, E. Clarke, and M. Minea. The Verus Tool: A quantitative approach to the formal verification of real-time systems. In Orna Grumberg, editor, *Proc. Computer Aided Verification (CAV'97)*, number 1254 in LNCS, Haifa, Israel, June 1997. Springer-Verlag.

[8] A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, and 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. Also IRST-Technical Report 9702-09, Trento, Italy.

[9] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K.McMillan, and L. Ness. Verification of the FUTUREBUS+ cache coherence protocol. In *11th CHDL*, 1993.

[10] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[11] E. Emerson, A. Mok, A. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *Proc. Computer Aided Verification*, LNCS. Springer-Verlag, 1990.

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