Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 130 (2005) 323–343
www.elsevier.com/locate/entcs

Automatic Verification of Safety Rules for a Subway Control Software

Nelson Guimarães Ferreira and Paulo Sérgio Muniz Silva[1,2]

Departamento de Engenharia de Computação e Sistemas Digitais
Escola Politécnica da Universidade de São Paulo
Av. Prof. Luciano Gualberto, trav. 3, n.158 05508-900 São Paulo SP Brazil

Abstract

This paper proposes the introduction of an automatic verification phase for a subway control software development process in which bounded model checking (BMC) and induction proof would be used to anticipate error discovery and increase the quality of the final product. We report the tests we developed for some safety rules of two actual sections of a subway track and the results we achieved. We conclude that the technique seems feasible for the problem domain, but the issue requires extensive research to allow an exact understanding of which requirements the use of the BMC meets, and actual benefits this approach might bring to the project.

Keywords: Bounded model checking, safety requirements model checking, subway control software model checking.

1 Introduction

Computer-controlled systems had a boost in the last few years, as much as the problems that may arise due to software errors. The problem is even more serious when we consider that the systems to be controlled are becoming more and more complex, whereas the delivery times are the same as before, if not shorter. While software correctness may be a very important issue for non-safety-related systems, it is critical for systems controlling devices that can put human lives as well as expensive equipment and installation at risk.

[1] Email: nelson.ferreira@poli.usp.br
[2] Email: paulo.muniz@poli.usp.br

1571-0661 © 2005 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2005.03.017

This paper describes an effort to assess the feasibility of model checking the software controlling a section of an actual subway system. A subway track may be large enough to make it necessary to divide it into many sections and to assign to each section a controller running its own specific (safety-related) software. A section normally encompasses a small number of stations, which means that the software controlling it may be relatively complex. In order to mitigate the risks of delivering an implementation with errors, the company developing the controller defines a verification and validation (V&V) process with inspection and test activities in well-defined phases.

We believe that there are some solid reasons for implementing an automatic verification process for that kind of development. Firstly, it is a well-known fact that fixing a software error becomes more and more expensive as the project progresses. The use of model checking immediately after the software is ready for release to the inspection phase may anticipate error discovery, thereby generating savings. Secondly, it may also contribute to an improvement in the quality of the final product, because an automatic verification may uncover subtle errors that would not be uncovered otherwise. Thirdly, the verification activities presented here may represent reduced man-hours when compared to other activities in the development process. Finally, most of the work done for the first software release may be reused during the development of the next releases.

There is some research on the automatic verification of railway control software. Since, in general, the controlled railway and its associated software can be easily and automatically reduced to a Finite State Machine (FSM), it is natural that model checking and other techniques exploiting that kind of formalism have been used to tackle the problem. Examples in the literature are [8], [7], [6], [9], [11], and [12].

Our approach is close to [8], [7] and [6]. The first common point between all the works is the problem domain: railway control software written in languages that are close to each other. [8] describes the verification of some safety rules of a relatively simple Dutch station. The tool that was used is the Stålmarck theorem prover. The second work uses a Binary Decision Diagram (BDD) based model checker to verify both the same station that [8] verified as well as a more complex one. Both works are related to ours because, although they used different techniques than us, they intended to verify safety rules for railway control software using an exhaustive state-space exploration. Another shared point is that the techniques they used allow the verification of rules which may exhibit very general patterns.

[6] uses bounded model checking (BMC) to verify some safety invariants and proposes using it at the end of the implementation phase of the develop-

ment process. It is also concerned with implementing the verification phase with both low impact on the process and no requirement for personnel training. Although we intend to verify rules that exhibit more general patterns than [6], our work is close to it for two main reasons. First, and in spite of some different points of view, both works are concerned with the development process and the impact caused by an automatic verification phase. Second, we use the same kind of technique (SAT-based bounded model checking).

Hence, we think we are contributing to [8] and [7] by discussing the use of a verification phase in the development process and to [6] by proposing a more general way of verifying the safety rules, as well as by proposing some different ways of executing the activities associated with the automatic verification phase. As a further contribution, this paper intends to draw the attention to the influence of the software structure in the format of the specification to be verified.

The remainder of this paper is structured as follows: section 2 describes the system to be verified and discusses the benefits of our proposal that includes an automatic verification activity. Section 3 introduces the formalisms associated with BMC and explains how we implemented the use of induction proofs. Section 4 describes the experiments that we developed and shows their results. Section 5 discusses some issues and compares our results and proposals with others. Finally, section 6 concludes the paper and shows some future steps.

2 System description

The software to be verified implements the control of a track section that is partially represented in Figure 1. The figure shows two tracks labeled 1 and 2 that can be merged by two points labeled X99-1 and X99-2. The tracks are divided into about 40 track segments called track-circuits, each one labeled either as 1Lnn or 2Lnn, depending on the track in which they are located. Figure 1 shows only ten of the available track-circuits.

Except for 1L06 and 2L06, two transmitters delimit each track-circuit. Each transmitter is able to send a speed-code to a train occupying one of the track-circuits delimited by it. The speed-code informs the train of the maximum speed it is allowed to move on that track-circuit and it is transmitted by outputting a modulated electrical current throughout the tracks. There are speed-codes corresponding to 10, 35, 50, 60, and 68 km/h. Not all the transmitters are able to transmit all the codes. By construction, it is not possible that the signal transmitted by one transmitter reaches a track-circuit not delimited by it.

Each train running on the track is equipped with a front antenna. The

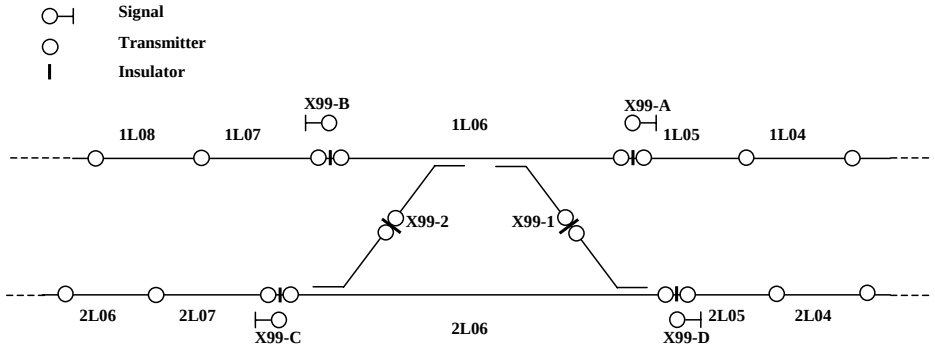

Fig. 1. Section to be verified

antenna captures the current sent by the transmitter located in front of the train, but is not able to receive the signal sent from the transmitter located behind. That is so because the train axes short-circuit the tracks, and as a result the electrical current transmitted from behind the train does not reach the antenna. The train is also equipped with an internal vital controller, ensuring that it never moves at a speed higher than the one corresponding to the speed-code, except in the transitions from a higher to a lower speed-code. In those cases, the train is forced to follow a velocity profile in order to reach the lower velocity before it moves away for a fixed maximum distance.

Each track-circuit is also capable of detecting whether a train occupies it or not. The hardware associated with this capability is not shown in Figure 1.

Track-circuits 1L06 and 2L06 are delimited by electrical insulators. The speed-codes thereon are transmitted by the transmitters installed at the ends of each track-circuit, like in any other track-circuit. The difference here is that the transmitters responsible for the speed-code of each of these track-circuits are interconnected, so in practice the same signal is generated by all the transmitters regardless of the direction of the train movement.

Figure 1 also shows two points between 1L06 and 2L06 which are responsible for making the trains shift from track 1 to track 2 and vice versa. Each point is able to be in normal (train does not shift tracks) or reverse (train shift tracks) position. The equipment has digital outputs, so that the controller is able to detect their position. There is also an output from the controller to each point. The output is intended to lock the point, so it does not move accidentally.

Finally, Figure 1 shows four signals labeled as X99-A/D. Each one of these signals can be either turned off or on. In the former case the train must stop. When it is in the on state, the signal is green and allows the train to go (provided the speed-code also allows it to do so).

2.1 The control software

The control software runs on Triple Modular Redundant (TMR) equipment implementing all the necessary digital inputs and outputs for each device or track-circuit being controlled. Each input or output is represented by its own variable. The software is implemented as a list of about 480 assignments involving digital inputs, digital outputs and internal state Boolean variables. There are about 300 digital inputs. All the assignments are in the format $V_i = f(v_0, v_1, \ldots, v_n)$, where V_i is either a state or an output variable and $v_0, v_1, \ldots, v_n$ are variables of any of the three kinds. The function f accepts only three binary operators, namely NOT (represented by a "."), AND ("*"), and OR ("+"). There may be some few tens of operands at the right-hand side of each assignment. Parentheses may be used in order to change the default operator priority. It is not allowed to define more than one assignment for a certain output or state variable. As an implementation rule, names of variables have a prefix which indicates the equipment or track-circuit associated with it, as well as a suffix denoting its meaning.

The equipment controlling the railway has an executive software responsible for managing the internal tasks the equipment must perform. The executive loop reads all the inputs and assigns their values to the corresponding variables, allows the execution of the assignment list, and finally sends the values of the calculated outputs to the hardware interfacing the physical devices. The only difference between digital inputs, digital outputs, and state variables is that the former are variables whose values are assigned by the executive software only. The other variables, on the contrary, are always assigned a new value in every execution cycle of the assignment list. Their values are preserved between two consecutive cycles. From the assignment list point of view, there is no difference between digital outputs and state variables. In this paper we are interested in the execution of the assignment list. The time interval between any two consecutive cycles is not necessarily fixed. Usually, it is less than 100 ms.

Table 1 shows all the naming convention for variables we are interested in this paper. In the table, tc, $tc1$,and $tc2$ denote track-circuit names, sc denotes the speed of the corresponding speed-code, p denotes a point name, and sgn denotes a signal name.

2.2 The proposed development process

The current development process is shown in Figure 2. It begins with the release of a specification that states the safety rules in natural language and gives examples of how the assignments would typically be implemented. The

Table 1
Variables of interest for safety verification

Variable group	Input - Output	Meaning
pNWCK	I	Indicates that the point is in normal (1) or not in normal (0) position. Example: X99-1NWCK.
pRWCK	I	Indicates that the point is in reverse (1) or not in reverse (0) position. Example: X99-1RWCK.
tcTP	I	Indicates if the track-circuit is occupied (0) or unoccupied (1). Example: 1L05TP.
pLS	O	Locks (0) or unlocks (1) the point p. Example: X99-1LS.
sgn-G	O	Turns the signal ON (1) or OFF (0). Example: X99-AG.
tc1-tc2-sc	O	Forces the transmission of corresponding speed-code in the transmitter located between tc1 and tc2. Example: 1L05-1L04-35

software implementation is based on the specification and on the drawings of each section. The drawings convey information about the organization of tracks into track-circuits, points, and signals, as well as information about track speeds. After the implementation phase, an independent expert engineer inspects the software. The last phase is platform testing, performed by personnel not directly related to the implementation phase. Figure 2 shows the natural flow of the project, as well as the feedbacks it suffers when errors are discovered in later phases.

Although very reliable implementations can be (and are) delivered with that process, we think that the project could benefit from the introduction of an automatic verification phase. The benefits we think that may be achieved are:

- **Earlier detection of errors**. It is well known that the earlier an error is detected, the lower its impact on the schedule and costs of the project.

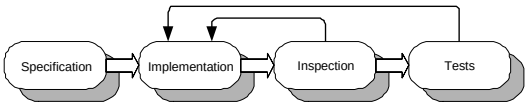

Fig. 2. The current development process

The insertion of an automatic verification activity immediately after the completion of the implementation phase may uncover errors before the code is sent to the expensive inspection and test phases, lowering the costs and delivery time. [9] references some studies concluding that error discovery shift from inspection to testing as projects become more complex. Although we have no equivalent data for the kind of project under study we believe that it is also the case, since it is reasonable to believe that a human verifier can effectively realize inspections in the small, but very hardly he/she will be able to do the same in the large.

- **Improvement in the overall quality**. The implementation may have some very subtle errors that cannot be uncovered by either tests or inspection, but can be detected by an exhaustive automatic analysis.
- **Much of the work done in the first release can be reused for subsequent releases.** Typically, second and later releases neither significantly change the safety rules nor the internal structure of the software. In such cases, most if not all the model used for the verification of the first release can be reused.

We propose a change in the current development process, depicted in Figure 3. We included an automatic verification phase immediately after the implementation phase, using model checking as the verification technique. The reason for that is because model checking fits well into the problem, as we will see in section 4. Another reason is that good model checkers are easily available nowadays.

We cannot guarantee that the automatic verification phase uncovers all the implementation errors. Our objective is to anticipate their finding.

3 Formal models for system verification

In this section we introduce the temporal logic used in the formal verification process. We also introduce model checking very briefly and explain the

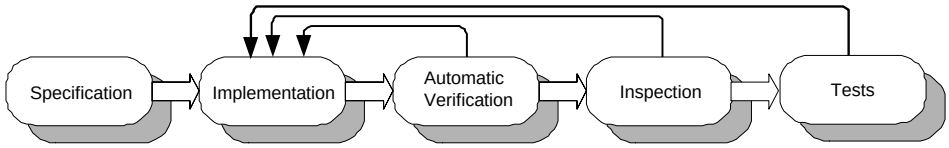

Fig. 3. The proposed development process

technique that we intend to use to verify the properties of the system under study.

3.1 LTL Temporal logic

A Linear Temporal Logic (LTL) formula uses atomic propositions, Boolean operators and temporal operators to define the properties that must hold for the system under study. Although it is possible to define many LTL operators, the most used are: $\mathbf{F}f$ (formula f will eventually be valid), $\mathbf{G}f$ (formula f is globally valid, which means that it is valid now and will be valid in any future state), $f_1\mathbf{U}f_2$ (f_1 is valid until f_2 eventually becomes valid), $f_1\mathbf{W}f_2$ (f_1 is valid until f_2 becomes valid), and $\mathbf{X}f$ (f will be valid on the next state).

Mathematically, the syntax of an LTL formula given a set AP of atomic propositions is defined according to the following rules [5]:

- An atomic proposition $p \in AP$ is an LTL formula.
- if f_1 and f_2 are LTL formulas, then $\neg f_1$ and $f_1 \wedge f_2$ are LTL formulas.
- if f_1 and f_2 are LTL formulas, then $\mathbf{F}f_1$, $\mathbf{G}f_1$, $f_1\mathbf{U}f_2$, $f_1\mathbf{W}f_2$, and $\mathbf{X}f_1$ are LTL formulas.

The semantics of LTL is based on the concept of a Finite State Machine (FSM) path. A path is an infinite sequence of states $\pi = (s_0, s_1, \ldots)$, where s_0 is the first state of the path and is not necessarily the initial state of the FSM. Firstly, we will need to define the following notation for a FSM M:

- π^i is the suffix of π which has s_i as the first state.
- $M, \pi \models f$ means that f holds at s_0.

Now we can define the semantics of an LTL formula according the following rules:

- $M, \pi \models f$ iff f is true on state s_0.
- $M, \pi \models \neg f$ iff f does not hold at s_0.
- $M, \pi \models f_1 \wedge f_2$ iff $(M, \pi \models f_1)$ and $(M, \pi \models f_2)$.
- $M, \pi \models \mathbf{X}f$ iff $M, \pi^1 \models f$.

- $M, \pi \models f_1 \mathbf{U} f_2$ iff there is a $k \geq 0$ such that $M, \pi^k \models f_2$ and $M, \pi^i \models f_1$ for any $0 \leq i < k$.

As an abbreviation, we may write $M \models f$ whenever it is implicit that the first state of the path is one of the initial states of the model. Obviously, other common Boolean operators like $\vee$, $\rightarrow$ and $\longleftrightarrow$ can be defined from $\neg$ and $\wedge$. The operators **F, G,** and **W** may be defined respectively as $\mathbf{F} f \equiv (true \mathbf{U} f)$, $\mathbf{G} f \equiv \neg \mathbf{F} \neg f$, and $f_1 \mathbf{W} f_2 \equiv (f_1 \mathbf{U} f_2) \vee \mathbf{G} f_1$.

3.2 Model checking

Model checking is a set of techniques used to verify some temporal logic specification of a system modeled into an FSM by exhaustively analyzing all its states and transitions. We usually have a set AP of atomic propositions, an FSM $M = (S, S_0, R, L)$ and a specification f stated in some temporal logic, where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is the total transition relation (i.e., any state has at least one successor) and $L : S \rightarrow 2^{AP}$ is a function that labels each state with the atomic propositions that are true in that state [5]. The model checker tries to verify if $M \models f$ for all the paths starting from the initial states of the model and, if it does not, it usually issues a counter-example which can be used to debug the system.

The most common challenge when using model checking is to deal with the state explosion problem created by the combination of all possible values of the system variables. Sometimes it is necessary to abstract some system behavior in order to enable the verification of a given specification with a simplified model. This approach has some drawbacks, namely that it may demand a manual interference in the process and cause some errors in the verification. If possible, we would like to avoid any kind of abstraction when trying to verify the control software we are interested in.

There are many model checking techniques which are intended to be used in different kinds of applications. They can be divided into two main groups: the group of techniques using an explicit representation for the states of the FSM and the group using a symbolic representation for them. In the first group, each state is individually analyzed and stored into the computer main memory, while the second group uses propositional formulas describing sets of states as well as transition relations. The explicit representation methods are usually used when the system can be modeled by an FSM limited in its size by some few million states. That limit is usually given by the amount of main memory available on the computers in use nowadays. Since we are dealing with systems with a number of states much larger than that limit, explicit representation techniques would not be our first choice.

Typically, symbolic model checkers use a technique by which they represent sets of states complying with some propositional formula, instead of working with individual states. One of the most used methods is based on fixpoints calculation. In its basic form, a model checker based on this technique starts with a set representing either all or none of the states of the model, depending on the kind of specification under analysis. That set is iteratively updated by the application of the transition relation until a fixpoint is eventually reached. The model checker then verifies if all the initial states belong to that fixpoint. If that is the case, the specification holds. Most of the model checkers using this kind of technique employ a BDD [3] library for creating Boolean formulas representing the state sets and the transition relations. This technique works fine for some models with few hundreds Boolean variables, but fails when the models are larger than that. The main reason for that is the significant amount of memory and CPU time needed to deal with the state sets and transition relations when the model becomes larger. Even when BDD-based model checking works, it is sometimes necessary to carefully set up the model checker framework, because some parameters like variable orderings and the algorithm used for calculating the fixpoints may be critical to the success of the method. The limitations and difficulties for medium to large-sized models do not make BDD-based model checkers ideal candidates to the systems that we want to verify if we do not consider using abstraction.

The limitations of the BDD solution and the progress that was achieved in recent years with the invention of faster algorithms for the satisfiability (SAT) of propositional formulas problem created the conditions for the introduction of bounded model checking [2]. In BMC, both the transition relations and the set of states are still represented by propositional formulas. Initially, the model checker builds a formula that is the conjunction of the initial states of the system and the negation of the specification under analysis. The formula is then analyzed by a SAT solver which tries to find a valuation that makes it true. If there is such a valuation, it is used as a counter-example for the specification. If there is no such valuation, nothing can be said about the truth of the specification and the model checker starts a new cycle. In the new cycle, the formula to be analyzed is: a disjunction of the initial states, the transition relation, and the negation of the specification. The model checker tries to find a counter-example for the specification, taking into account the initial states as well as their successors. Again, the formula is sent to the SAT solver. The process continues until either a counter-example is found, the maximum configured number of iterations k is reached or the formula to be analyzed becomes large enough to make its analysis infeasible.

Compared to the classical unbounded model checking, BMC has some

disadvantages and advantages. The main disadvantage is that it is not able, in general, to say something about the truth of the required specification in case it either holds or the counter-example is large enough to be handled by the SAT solver. Thus, BMC is primarily an error-finder tool.

The advantages of BMC over unbounded model checking are manyfold. First, the user does not need to set up the environment manually. Usually, the default parameters will suffice. Second, the model checker is able to deal with thousands of variables instead of hundreds and, in doing so, it uses much less memory and spends much less computer time. Third, the generated counter-example is guaranteed to be minimum, simplifying the analysis of the problem. The first and second features will be of great value for us in our verification process.

3.3 Robust systems and induction proofs

A model is defined in [7] as robust with respect to a specification f if f holds for all possible states of the model, even if they are not reachable from the initial states. Mathematically, an FSM model $M = (S, S_0, R, L)$ is robust with respect to f if, given $M' = (S, S, R, L)$, $M \models f \rightarrow M' \models f$.

A robust model with respect to the specification f has an interesting feature, namely that if $f = f(p_0, p_1, \ldots, \mathbf{X}p_0, \mathbf{X}p_1, \ldots, \mathbf{X}^k p_0, \mathbf{X}^k p_1, \ldots)$ holds for all finite sequence of $k+1$ states starting from the initial states s_0, then $\mathbf{G}f$ also holds. In other words, we can say that $M' \models f \rightarrow M \models \mathbf{G}f$. In fact, any finite state sequence for which f holds ends at some final state s_k which belongs, itself, to the set of the initial states of the modified model M'. Therefore, the state s_k is the initial state of a new finite sequence for which f also holds, and inductively $M \models \mathbf{G}f$. We define the depth of a formula as the maximum number k of nested $\mathbf{X}$ operators.

For example, in the subway system under consideration we can expect that signal X99-A must be turned off whenever point X99-1 is in reverse. This rule could be specified as $\mathbf{G}(\text{X99-1RWCK} \rightarrow \mathbf{F}(\neg\text{X99-AG } \mathbf{W} \neg\text{X99-1RWCK}))$. The sentence says that it is always true ($\mathbf{G}$) that if X99-1 is in reverse then signal X99-A will be eventually ($\mathbf{F}$) turned off and stay at that state until ($\mathbf{W}$) X99-1 is not reversed. This sentence format has two drawbacks. The first one is that we have no information about the number of cycles the software takes to respond to the new condition. It is a sensible issue because if we keep the LTL specification as stated above, we are theoretically accepting long response times that could turn the implementation unable to quickly respond to an emergency situation, making it unsafe. Moreover, in order to use the inductive proof scheme we need to have a bound k on the number of software cycles and the sentence, as stated above, does not carry any information about

that.

We worked this problem out by analyzing the source code for the completeness of the proof. Rather than trying to use a sentence like the one above, we start looking for a specification that could be stated as (X99-1RWCK $\rightarrow \mathbf{X}^k\neg$X99-AG), which means that the signal X99-AG is turned off k cycles after point X99-1 is in reverse. If the sentence can be proved true for a given k and a generic initial condition, the safety rule we are interesting in would be verified if k is to be considered small enough. In fact, it would be proven that if the point is in reverse at any state s_i, then the signal is off at any successor state s_{i+k}. It also means that the signal will stay off while the point keeps its reverse state because if the point is still in reverse at s_{i+1} then the signal will be off at s_{i+k+1} and so on. In our example, we concluded after a quick source code analysis that the specification to be proved is (X99-1RWCK $\rightarrow \mathbf{XX}\neg$X99-AG), and that a bound $k = 2$ would be enough to prove it.

4 Verification and experiments

In this section we describe the tools we used, the safety rules we wanted to verify, their translations into LTL, and the results we obtained from two experiments.

4.1 Tools

We chose to use NuSMV [4] as the bounded model checker. It is a re-implementation that extends the Symbolic Model Verifier (SMV) [10], developed at the Carnegie Mellon University[3]. NuSMV accepts as input a file which describes the model of the system to be verified. The input language is basically the same language as defined by SMV (the "SMV language"). Therefore, we needed a program to automatically translate the original source code into the SMV language. The translator we implemented creates a single SMV input file declaring all the variables and the complete set of assignments defined by the original source code. There is a one to one correspondence between the assignments in the original source code and the SMV input language.

The translator is responsible for the following tasks:

- The *assignment*, and the operators *and*, *or* and *not* are represented by different symbols in both languages, so the original representation has to be

[3] We used the NuSMV version 2.1.2, running under C*ygwin* and *Windows XP*. The version implements both bounded and unbounded BDD-based model checking and is freely available at http://nusmv.irst.itc.it/.

converted according to the SMV language rules.

- The SMV language does not accept variables with a digit as the first character, so the translator always includes an underscore (_) at the beginning of every variable.
- The SMV operator *next* is attached to any variable at the left-hand side of an assignment. In doing so, we specify that the next state value of the variable being assigned will be the result of the formula at the right-hand side.
- The translator has to decide how it translates any variable *var* at the right-hand side of an assignment. It can translate it just as "*var*" or it can translate it as "next (*var*)". The former case is used when, starting the translation from the first assignment, *var* has not already been used at the left-hand side of an assignment. The later case is used otherwise.

The last task preserves the semantics of the original source code during its translation into the SMV language. In the original source code, the assignments are executed sequentially from top to bottom. The SMV language has a different semantics, since it considers that every assignment is executed in parallel. Figure 4 shows an example of the translation from one language into the other.

```
Original program:
    A = .B * D * (C + A);
    B = .A * D;
SMV translation:
    next(_A) := !_B & _D & (_C | _A);
    next(_B) := !next(_A) & _D;
```

4.2 Verifying the safety rules

We chose to check safety rules associated with the observable behavior of the track. The software has a large set of variables which are used internally and must comply to its internal rules. The violation of these rules may not necessarily bring the system to an unsafe state, because the internal variables do not directly change the current state of the track.

In order to use the model checking technique, we defined four classes of safety rules expected to hold. The set of rules does not completely describe all the safety-related requirements of the system, but helps the feasibility analysis of the use of model checking for that kind of system.

We classified the rules into four groups. We also recognized that there are rules describing the way the system must respond to some input condition, and others that must hold independently of the state of the inputs and can be considered propositional invariants of the system. All the rules can be written

in the following specification formats:

- $\mathbf{G}(<inputcondition> \rightarrow \mathbf{F}(<safestate>\mathbf{W}\neg<inputcondition>))$, for input-dependent rules. It means that it is always true that if the input condition happens, then a safe state will eventually be reached and the system will stay at that safe state until the input condition is reset. Strictly speaking, those rules may also be considered invariant properties of the system, because any $\mathbf{G}f$ property can be considered to be so. However, in this paper, we will reserve the word invariant to the propositional formulas that must hold without any explicit dependency on the inputs.
- $\mathbf{G}(<invariant>)$, for rules that must always hold and do not depend explicitly on any input.

The four groups used to classify the rules are:

1. A transmitter does not transmit any kind of speed-code when both track-circuits associated with it are occupied. As an example, we must expect that the transmitter between 1L05 and 1L04 does not transmit any speed-code if both 1L05 and 1L04 are occupied. This rule has the following condition/safe state pair:

Condition: $\neg$1L05TP $\wedge \neg$ 1L04TP

Safe state: $\neg$1L05-1L04-35 $\wedge \neg$ 1L05-1L04-50 $\wedge \neg$ 1L05-1L04-68

The transmitter between 1L04 and 1L05 is able to transmit the speed-codes corresponding to 35, 50, and 68 km/h only. The variables associated with other speeds are not defined in the program. Therefore they do not belong to the transmitter safe state definition. Similar condition/safe state pairs apply to the other transmitters.

2. Interlocking between points and signals. We devised the following rules involving signals and points:

2.1. Both points are locked whenever any signal is turned on. An on-signal enables the train passage through its points. Thus, for safety reasons, it is necessary that both points be locked in order to avoid any accidental point movement that may cause train derailment. This rule is input-independent and may be stated by the following invariant:

Invariant: X99-AG $\vee$ X99-BG $\vee$ X99-CG $\vee$ X99-DG $\rightarrow \neg$X99-1LS $\wedge \neg$X99-2LS

2.2. It is not allowed to have more than one on signal if at least one of the points is in reverse. The condition/safe state pair for that is:

Condition: X99-1RWCK $\vee$ X99-2RWCK

Safe state: $\neg$((X99-AG $\wedge$ X99-CG) $\vee$ (X99-AG $\wedge$ X99-DG) $\vee$ (X99-BG $\wedge$ X99-CG) $\vee$ (X99-BG $\wedge$ X99-DG))

2.3. All the signals must be turned off if both points are in reverse. That means that even the passage from C to D or vice-versa is not allowed. The

condition/safe state pair is:

Condition: X99-1RWCK $\wedge$ X99-2RWCK

Safe state: $\neg$(X99-AG $\vee$ X99-BG $\vee$ X99-CG $\vee$ X99-DG)

2.4. Signals A and C must be turned off whenever point 1 is in reverse. By the same token, signals B and D must also be turned off whenever point 2 is in reverse. The condition/safe state pairs of both rules are:

Condition 1: X99-1RWCK

Safe state 1: $\neg$(X99-AG $\vee$ X99-CG)

Condition 2: X99-2RWCK

Safe state 2: $\neg$(X99-BG $\vee$ X99-DG)

2.5. Both points must be in normal position whenever we have one on signal at each track. The condition/safe state pair is:

Condition: $\neg$(X99-1NWCK $\wedge$ X99-2NWCK)

Safe state: $\neg$((X99-AG $\wedge$ X99-CG) $\vee$ (X99-AG $\wedge$ X99-DG) $\vee$ (X99-BG $\wedge$ X99-CG) $\vee$ (X99-BG $\wedge$ X99-DG))

3. Opposite signals must never be on at the same time. We defined two invariants for that rule, which are listed below.

Invariant 1: $\neg$(X99-AG $\wedge$ X99-BG)

Invariant 2: $\neg$(X99-CG $\wedge$ X99-DG)

4. The speed-code, transmitted when a train is approaching an off–signal, should always correspond to low speed. As an example, we must expect that the speed-code is transmitted to the train that is approaching signal A must not correspond to either 50 or 68 km/h. The condition/safe pair for the example is given below. There are equivalent rules for the other signals.

Invariant: $\neg$X99-AG $\longrightarrow$ $\neg$1L05-1L06-50 $\wedge$ $\neg$1L05-1L06-68

4.3 Creating the LTL specifications

In order to prove the rules using the inductive procedure described above, we have to create LTL specifications for those rules. The specifications may be provable true after a finite number of cycles of the software being analyzed. We have already given an example in which we found that the software needs two cycles to turn signal X99-A off after point X99-1 is in reverse. Obviously, we could expect that at least one cycle would be necessary, because the software needs to be executed at least once before it can respond to any input change. However, we do not have any means to know the necessary number of cycles *a priori* without recurring to analysis. It could well be the case that the software needed three or four cycles to respond.

In practice, analyzing the software is a matter of observing the sequence of

assignments of the variables used by the rule to be verified and of the intermediary variables that may affect them. After the analysis, the LTL formula is written and checked. If the model checker indicates that the specification does not hold, it may be either because the LTL specification was incorrectly stated or because the software has some error. In both cases we have to re-analyze the program with the help of the model checker counter-example.

Another way of creating an LTL specification is assuming that the software is correct and starting from a tentative specification. If the model checker states that the specification holds, the process ends. Otherwise, we can create a new tentative specification until either we prove the rule or we give up and consider analyzing the source code. Although we may have to run some tentative specifications before we reach a valid one, we may save some time and effort because the model checker response time is short compared to the time needed to analyze the source code. In the example given above, we could start from the tentative specification (X99-1RWCK $\rightarrow$ **X** $\neg$X99-AG). Since the model checker would say that the specification does not hold, we could try (X99-1RWCK $\rightarrow$ **XX** $\neg$X99-AG). That specification holds, so we would have successfully saved time and effort on not analyzing the code.

In brief, we spent only few hours writing the correct LTL specifications for the problem at hand. Table 2 shows the results of that work. The second column shows the LTL specification used for checking the rules and the third column gives the bound for model checking, i.e. the number of cycles the software has to run in order to either go to the safe state or to be compliant with the invariant.

One interesting point is the origin of the **X** operator used at the beginning of specifications of groups 2.1, 3, and 4. These groups are responsible for checking the input-independent invariants. It means that the LTL specifications depend only on the variables calculated by the software. According to the model we are using, the values of those variables are completely undefined before the first software cycle. Therefore, it is not possible to guarantee any rule concerning their values before the first iteration. The **X** operator in front of the formula reflects this argument. The LTL specification of group 2.1 has another **X** operator at the right-hand side of the $\rightarrow$ operator. It expresses the fact that the software is written in a way that the assignments of the lock commands (LS variables) are executed before the assignments of the signals (G variables). Hence, the state of the signals will be reflected at the lock command only at the next cycle.

Table 2
LTL specifications for the rule groups

Rule	LTL specification	k
1	(!_1L05TP & !_1L04TP) -> X X (!_1L05-1L04-35 & !_1L05-1L04-50 & !_1L05-1L04-68) and 39 others in the same format.	2
2.1	X ((_X99-AG | _X99-BG | _X99-CG | _X99-DG) -> X (!_X99-1LS &!_X99-2LS))	2
2.2	(_X99-1RWCK | _X99-2RWCK) -> X !((_X99-AG & _X99-CG) | (_X99-AG & _X99-DG) | (_X99-BG & _X99-CG) | (_X99-BG & _X99-DG))	1
2.3	(_X99-1RWCK & _X99-2RWCK) -> X !(_X99-AG | _X99-BG | _X99-CG | _X99-DG)	1
2.4	(_X99-1RWCK -> X !(_X99-AG | _X99-CG)) (_X99-2RWCK -> X !(_X99-BG | _X99-DG))	1
2.5	!(_X99-1NWCK & _X99-2NWCK) -> X !((_X99-AG & _X99-CG) | (_X99-AG & _X99-DG) | (_X99-BG & _X99-CG) | (_X99-BG & _X99-DG))	1
3	X !(_X99-AG & _X99-BG) X !(_X99-CG & _X99-DG)	1
4	X ((!_X99-AG) -> (!_1L05-1L06-50 & !_1L05-1L06-68)) X ((!_X99-BG) -> (!_1L07-1W06-35 & !_1L07-1W06-50 & !_1L07-1L06-68)) X ((!_X99-CG) ->(!_2L07-2W06-35 & !_2L07-2W06-50 & !_2L07-2L06-68)) X ((!_X99-DG) -> (!_2L05-2W06-50 & !_2L05-2W06-68))	1

4.4 Experiments

All the rules were efficiently verified. Table 3 shows the group (G) that was verified, the number (n) of properties in that group and the depth (k) of the specifications. It also shows the memory needed by NuSMV (Mb) and the time in seconds that it took to complete the verification of all the group properties. We used a computer with an Athlon 2800+ CPU.

It is possible to observe from Table 3 that the used technique circumvented the state explosion problem. Moreover, the model checker achieved fast verification times. Considering that it was a very positive result, we tried the verification of the same classes of rules to a larger section. The new experiment was a much more complicated track section, having about 800 digital inputs, 1150 state variables, nearly 60 track-circuits, 7 points, and 18 signals.

We spent about four hours to write the new LTL specifications. The results are shown in Table 4.

Table 3
Verification of the first section

G	n	k	Mb	t(s)
1	40	2	33	17.5
2.1	1	2	25	0.8
2.2-2.5	5	1	21	0.9
3	2	1	19	0.4
4	4	1	19	0.6

Table 4
Verification of the second section

G	n	k	Mb	t(s)
1	43	2	83	42.1
2.1	2	2	65	5.5
2.1	3	3	77	19.5
2.2-2.5	9	1	54	5.4
3	9	1	50	3.9
4	14	1	50	4.7

5 Related work

The results described above are consonant with the successful results reported by [8] and [6]. The later achieved verification times of fractions of a second for a system of about 2000 variables and $k = 1$. Both system size and verification times are close to ours. [7] successfully verified the control software of two subway stations using a BDD-based model checker and proving specifications written in a format close to ours, although the specifications were written in CTL rather than LTL. During our experiments, we wanted to discover if we would also be able to verify our properties using BDDs. Unfortunately, NuSMV was not able to verify any specification, so we decided to experiment

with another easily available SMV implementation called Cadence SMV [1], trying to verify one property of each of the four groups defined above. The results for the simpler section ranged from 25 seconds of execution time and 65 Mbytes of memory for verifying a property of group 1 to a failure in the verification of the property of group 4, due to lack of memory. No specification of the larger section could be verified. Hence, although the good results reported by [7] with BDD-based model checking, we concluded that SAT-based BMC seems to be much more promising for our domain.

Like us, [6] was not able to verify its system with BDDs and proposed BMC. However, it suggests a different way of verifying the safety properties: instead of writing a specification using temporal logic, it proposes the definition of new variables that are appended to the end of the program. Those variables represent system invariants and the value of each one is a function of the current and next values of other variables. The domain expert engineers should create the invariants based on their interpretation of the safety requirements. The rationale for this approach is that the authors believe that using a language familiar to the expert domain engineers is crucial for their proposal to be useful. We dispute that. Firstly, this approach limits the properties that can be verified to depth 0 or 1 formulas, unless the developer creates new intermediary variables to store older values (entangling the verification and bringing unwanted manual intervention). Secondly, we concluded that it is easy to write down LTL specifications for the properties to be verified, since we only need the **X** LTL operator as well as the traditional Boolean operators. Thirdly, we are afraid that the chance of common mode errors increases with that approach, because the expert domain engineers are responsible for many tasks: interpreting the specification stated in natural language, implementing the program, and defining the invariants in the same language of the control software. The verification may fail if either the interpretation of the requirements is incorrect or the wrongly implemented code is used somehow in the definition of the invariants. Another drawback is that it makes it difficult for others to check whether the invariants are correctly stated. Thus, we think that a better approach would be stating the safety properties either as condition/safety pairs or as invariants in the specification document. The developer would have to write the LTL formulas and run the model checker in order to verify the properties. The LTL formulas would be implementation-dependent, but could be easily assessed by independent reviewers.

6 Conclusion and future work

We have presented the results of model checking two different actual sections of a subway track. The main problem that we could have faced was the state explosion problem. However, experience showed that we were able to verify all the rules in relatively short time, with no need for further abstractions, which would demand more manual interference on the process as well as the possibility of inserting errors into the verification. An interesting point is that although we used a model checker, we actually did not need one. A SAT solver would suffice, because it is not possible to verify a temporal formula of depth k in less than k applications of the transition relation. Hence, there is no use in trying to find a counter-example smaller than k. However, we decided to use NuSMV because it easily represents the model to be verified and is freely available. Another point in favor of the model checker is that the time spent on finding counter-examples smaller than the depth of the formulas is not a problem, given its fast response time. We conclude that, for the specific kind of subway control software that was examined, an automatic verification process based on model checking and on the described approach seems to be feasible.

This work opens some paths for future works. One of them is the automatic verification of software functional requirements on this particular domain. Examples of functional requirements are the conditions for a route approval, the conditions for cancelling a route, and how the system must depict a point position while it is moving. Another essential point to be studied is to understand the coverage degree that can be achieved with that kind of verification, its costs, and benefits to the project.

References

[1] Cadence SMV. Cadence SMV and its documentation can be downloaded from http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of TACAS 1999*, number 1579 in LNCS, page 193. Springer-Verlag, 1999.

[3] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C35(8):677–691, 1986.

[4] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[5] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[6] G. Dipoppa, G. D'Alessandro, R. Semprini, and E. Tronci. Integrating automatic verification of safety requirements in railway interlocking system design. In *Proceedings of HASE 2001*, pages 209–219. IEEE Computer Society, 2001.

[7] C. Eisner. Using symbolic model checking to verify the railway stations of Hoorn-Kersenboogerd and Heerhugowaard. In *Proceedings of CHARME 1999*, number 1703 in LNCS, pages 97–109. Springer-Verlag, 1999.

[8] W. Fokkink. Safety criteria for Hoorn-Kersenboogerd railway station. Logic Group Preprint Series 135, Utrecht University, 1995.

[9] M. Huber and S. King. Towards an integrated model checker for railway signalling data. In *Proceedings of FME 2002*, number 2391 in LNCS, page 204. Springer-Verlag, 2002.

[10] K. McMillan. *Symbolic Model Checking: An Approach to the State Explosian Problem.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[11] K. Winter. Model checking railway interlocking systems. In M. Oudshoorn, editor, *Proceedings of Australian Computer Science Conference (ACSC 2002)*, number 24(1), pages 303–310. Australian Computer Science Communications, 2002.

[12] K. Winter and N. Robinson. Modelling large railway interlockings and model checking small ones. In M. Oudshoorn, editor, *Proceedings of the 26th Australian Computer Science Conference (ACSC 2003)*, pages 309–316, 2003.