Formal Verification and Validation of ERTMS Industrial Railway Train Spacing System

Alessandro Cimatti[1], Raffaele Corvino[2], Armando Lazzaro[2], Iman Narasamdya[1], Tiziana Rizzo[2], Marco Roveri[1], Angela Sanseviero[2], and Andrei Tchaltsev[1]

[1] Fondazione Bruno Kessler — IRST
[2] Ansaldo-STS

Abstract. Formal verification and validation is a fundamental step for the certification of railways critical systems. Many railways safety standards (e.g. the CENELEC EN-50126, EN-50128 and EN-50129 standards implement the mandatory safety requirements of IEC-61508-7 standard for Functional and Safety) currently mandate the use of formal methods in the design to certify correctness.

In this paper we describe an industrial application of formal methods for the verification and validation of "Logica di Sicurezza" (LDS), the safety logic of a railways ERTMS Level 2 system developed by Ansaldo-STS. LDS is a generic control software that needs to be instantiated on a railways network configuration. We developed a methodology for the verification and validation of a critical subset of LDS deployed on typical realistic railways network configurations. To show feasibility, effectiveness and scalability, we have experimented with several state of the art symbolic software model checking techniques and tools on different network configurations. From the experiments, we have successfully identified an effective strategy for the verification and validation of our case study. Moreover, the results of experiments show that formal verification and validation is feasible and effective, and also scales reasonably well with the size of the configuration. Given the results, Ansaldo-STS is currently integrating the methodology in its internal Development and Verification & Validation Flow.

1 Introduction

The verification of industrial safety critical systems is paramount. It is a particularly difficult activity because of the size and complexity of the systems. The most frequently used methods, simulation and testing, can increase the reliability of the systems, but, since they are not exhaustive, they cannot show the absence of errors. Failure in detecting an error in a safety critical system can lead to a catastrophic situation.

Formal verification has proved to provide a complete coverage. Particularly for railways critical systems, formal verification is becoming fundamental for the certification of such systems. The CENELEC EN-50126, EN-50128 and EN-50129 standards, that implement the mandatory safety requirements of the IEC-61508-7 standard for Functional and Safety, require the use of formal verification techniques to certify the correctness of the design. Despite their importance, the application of such techniques in industrial settings is by no means trivial. First, a proper verification and validation methodology has to be designed to allow for an efficient handling of the size of the system under verification. Second, the verification and validation techniques should integrate smoothly in the company internal Development, Verification & Validation Flow.

In this paper we describe an industrial application of formal methods for the verification and validation of a fragment of railways critical software. The software is called *Logica di Sicurezza* (LDS), which is a realization of the safety logic of a Level 2 European Railways Train Management System (ERTMS) developed by Ansaldo-STS. LDS is a generic distributed complex control software, which is designed to manage and control the train spacing in an ERTMS. LDS also guarantees the safety of the controlled railway network, e.g., no train collisions and proper distance among the trains. LDS is programmable and scalable, in the sense that, different train spacing control systems can be developed by instantiating the generic controller to a specific railway network configuration. For our case study, we have focused on radio block sections (RBS), which are a critical fragment of LDS that controls routes in railways. We consider instantiations of RBS on typical realistic railways network configurations.

The verification of the RBS application, carried out by Ansaldo-STS, has been relying on manual inspections and on simulations, where the simulation vectors were manually designed based on the experience of the engineers. Achieving full coverage was considered a very challenging task. We show in this paper a methodology that we have identified for applying formal verification to the considered application. This methodology is a result of a thorough evaluation of the state-of-the-art verification techniques that have shown to be highly effective in the verification of large systems. In particular, we focus on symbolic model checking for software as it is exhaustive and completely automatic, and thus allows for an easy integration within the Development and Verification & Validation Flow.

We have devised a verification flow for the verification of the considered application. We first translate the specifications of the case study, written in the VELOS language, a restricted version of the C++ language developed and used by Ansaldo-STS, into forms that can be verified by existing tools. In particular we translate such specifications into NUSMV models and sequential C programs. The translation into NUSMV models allows us to use the portfolio of advanced verification techniques, like bounded model checking [7], temporal induction [19], and CEGAR [15], that have been implemented in an extended version of the NUSMV model checker [12] developed within Fondazione Bruno Kessler (FBK). The translation into sequential C programs allows us to experiment with advanced software model checking techniques, like the eager and lazy predicate abstractions [17,4], and enables the use of off-the-shelf software model checkers for C, like CBMC [16], BLAST [4], SATABS [17], CPACHECKER [5], and KRATOS [14] (developed by FBK). We have further customized KRATOS and have extended its analysis to achieve the verification needs identified during the project.

We have carried out experiments with the above mentioned verification tools on a significant set of benchmarks obtained from the critical subset of the considered application. The experiment results allow us to devise an effective strategy, based on a combination of verification approaches, that greatly improves the efficiency of the verification. The results also provide evidence that formal verification in our industrial setting is feasible and effective. Given these results, Ansaldo-STS is currently integrating the methodology and tools in its Development and Verification & Validation Flow.

This paper is organized as follows. In Section 2 we describe LDS application. In Section 3 we present an overview of formal verification techniques. In Section 4 we

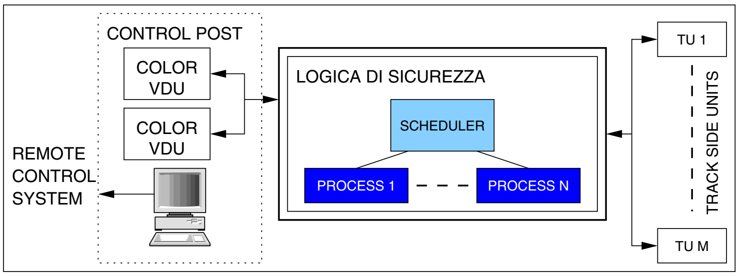


Fig. 1. The application and its environment

describe the verification methodology, along with the extensions to KRATOS. In Section 5 we show the results of our experimental evaluation. In Section 6 we discuss related work. Finally, in Section 7 we draw some conclusion, including the lessons learned, and discuss future work.

2 The Application: *Logica Di Sicurezza*

The work in this paper is concerned with the verification of a complex real-world safety critical application developed by Ansaldo-STS. The application is used to manage and control the train spacing in an ERTMS railway system. Figure 1 shows a high-level architecture of the application and its environment. Our work has focused on *Logica di Sicurezza* (LDS), a software subsystem that controls train movements and track-side equipment that is connected to track-side units, e.g., track circuits, signals and switches. LDS also implements the logical functions that can be requested by human operators, e.g., preparing the tracks for moving trains.

LDS is highly programmable and scalable: it is possible to program the modalities under which the requested logical functions are performed; and to program various configurations of track-side units. Such a distinguishing feature is achieved by means of a logical architecture composed of a *scheduler* controlling the activation of application-dependent processes. LDS is designed by specifying the processes controlled by the scheduler, which are then converted into executable code. In general, LDS can be thought of as a reactive system, acting along the following loop: read status from track-side units and input from the operator, run the processes through the scheduler to apply the control law of the logic, and write commands to actuators of the units.

The specification of processes in this architecture is non-trivial. Indeed, a railway station can have a large number of physical devices, and processes of many different kinds are required to take into account the relations and interconnections among physical devices. Moreover, although the software is completely deterministic and the possible external events (e.g. faults of peripheral devices) have been exhaustively classified, there is still a high degree of non-determinism. The software does not know if and when external events will happen, e.g., certain tasks can be requested at any time. Furthermore, the physical devices will typically react to controls with (unpredictable) delays, and may even show faulty behaviors.

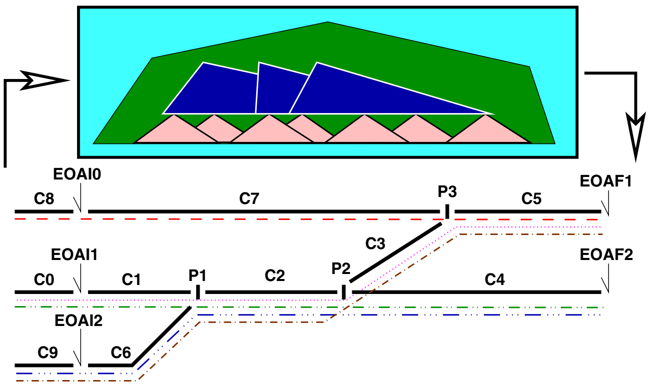


Fig. 2. A radio block section fragment of LDS and the railway layout that the fragment controls

Currently, the specification is validated by means of traditional techniques, such as simulation. Designing test cases that ensure high code coverage, as well as stimulate the whole specification to exercise all the critical functionalities, as mandated by the standards, is non-trivial. This is typically done manually, and thus often miss some critical functionalities.

2.1 The Radio Block Section

In this paper, we focus on the radio block section (RBS) fragment of LDS, a critical subsystem responsible for the allocation of logical routes to trains. In particular, as a case study, we consider the configuration corresponding to the physical layout of the railways depicted in the lower part of Figure 2 with thick dark lines. Some entities in this fragment correspond to physical entities, while some others to logical ones. A *component* is a logical reservation for a segment of line. A *point* is a logical controller for a switch. For example, the component $\mathrm{C}i$ and the point $\mathrm{P}i$ correspond, respectively, to the segment $\mathrm{C}i$ and the switch $\mathrm{P}i$ in Figure 2. A *radio block section* (RBS) corresponds to a logical route through the physical displacement. An RBS is composed of several components and points, and enclosed by an initial and final end-of-authority (EOA). In Figure 2 EOAI0, EOAI1, and EOAI2 are the initial EOA, while EOAF1 and EOAF2 are the final EOA. Figure 2 shows the ten RBS's of the physical layout in different dashed/dotted (colored) lines; each route in the figure denotes two RBS's, for left-to-right and right-to-left directions.

Figure 2 (upper part) shows the fragment of LDS that we have focused on in this activity. The (light blue) rectangle represents the scheduler, the (green) cone is the whole set of processes in LDS, the dark (blue) triangles represent the RBS's, and the light (pink) triangles represent the call graphs of the "underlying" processes. Each process consists of data and functions to modify the data. Processes are organized hierarchically according to the call graph of the whole logic of LDS, e.g., RBS processes can call the functions of sub-processes illustrated in Figure 2 by the light (pink) triangles.

2.2 Verification Properties of the RBS

Ansaldo-STS has identified five parametric properties for the RBS fragment: one for each component and four for each RBS. These properties correspond to the safety requirements of the considered application, and are formulated as invariants that have to hold at every read-scheduler-output cycle. For the case study in Figure 2, there are 12 components and 10 RBS's, and this amounts to have 52 properties. An example of a property is "no two different trains occupy the same track". An additional property is introduced to test the consistency of the physical layout configuration, so in total we have 53 properties.

Since the focus of the analysis is on the RBS's, along with the processes that they induce, we abstract away non-RBS processes, and use an abstracted scheduler that repeatedly chooses one of the ten RBS's, depending on a certain condition, starts its process by executing one of its functions, until some exiting condition is met. To avoid having a too coarse abstraction, we specified a number of assumptions, e.g., constraints over signals, that have to be satisfied by the abstracted parts. The assumptions have been thoroughly discussed, refined, and approved by Ansaldo-STS.

2.3 The VELOS Specification

A *specification* of LDS consists of (1) an *entity description* of the physical and logical entities of LDS, and (2) a *configuration* describing a particular physical layout (relation between the entities). The entity description is specified in a structured programming language, called the VELOS language, that resembles the C++ language. It contains classes that define component, point, EOA, and RBS. Like typical C++ classes, each of these classes contains member variables and member functions. The values of member variables constitute the states of the corresponding entity, while the member functions are used to modify the states of the entity. In particular, the functions of the RBS class define the logic of RBS. Member functions can contain loops that can statically be unrolled. They are also non-recursive, and thus can be inlined. Operations in these functions only involve data of types Boolean, bounded integers, or enumerations, with no pointers and no dynamic memory allocations. (These restrictions are standard for this kind of applications.) The properties to verify are expressed in a temporal logic, and are attached to the corresponding classes in the entity description.

The configuration specifies instances of the classes in the entity description as well as the relation between these instances. In particular, it describes the RBS's and entities that constitute them. Currently, both the entity descriptions and the configurations are created manually by the design engineers. The actual assembly code deployed on the physical devices is automatically generated from the specifications.

3 Verification Techniques and Tools

The problem of selecting the right techniques and tools for the verification of LDS is non-trivial. First, the techniques and tools must scale to industrial-sized designs. In particular, the chosen techniques should address the state explosion problem that is very common in such applications. Second, the right techniques are often not apparent due to a representational issue. The product development team have their own specification language that is far from the languages assumed by most verification techniques or tools. Third, the high-efficiency demand from the development team often cannot be met by existing techniques and tools. Customizations of and extensions to the existing techniques and tools, as well as a good strategy in combining them, are required to boost the performance.

In this work we appeal to symbolic (software) model checking techniques. Being completely automatic, model checking techniques can easily be integrated into the development cycle. Moreover, symbolic techniques are known to be effective in combating the state explosion problem. In what follows, we review some state of the art symbolic software model checking techniques: bounded model checking (BMC) [7] and counter-example guided abstraction refinement (CEGAR) [15]. We focus on model checking safety properties in the form of program assertion. Although, they are often used to represent requirements, they can also be used to generate execution traces that help the generation of test cases for automatic test pattern generation [22].

Software model checking has proved to be an effective technique for verifying sequential programs. In particular advances in solvers for satisfiability modulo theory (SMT) [2] have enabled for efficient Boolean reasoning and abstraction computation, which in turn have enabled SMT-based software model checkers to efficiently verify large programs with significant improvements in precision and accuracy.

In the BMC approach one verifies the program by specifying some bounds to guarantee termination. The bounds can be the number of executed statements, the depth of recursions, or the number of loop unwindings. This approach can only be used to disprove assertions (or bug finding). The seminal work on temporal induction [19] uses BMC, not only to disprove properties, but also to prove invariants.

In the CEGAR paradigm one checks if an abstraction (or over-approximation) of the program has an abstract path leading to an assertion violation. If such a path does not exist, then the program is safe. When such a path exists and is feasible in the concrete program, then the path is a counter-example witnessing the assertion violation. Otherwise, the unfeasible path is analyzed to extract information needed to refine the abstraction. For the CEGAR approach, two predicate abstraction based techniques have proved to be effective: the eager abstraction [17] and the lazy abstraction [4]. Predicate abstraction [23] is a technique for extracting a finite-state program from a potentially infinite one by approximating possibly infinite sets of states of the latter by Boolean combinations of some predicates. In each CEGAR iteration of the eager abstraction, one verifies a *Boolean program* extracted from the input program based on a set of predicates. The Boolean program consists only of Boolean variables, each of which corresponds to a predicate used in the abstraction. The lazy predicate abstraction is based on the construction and analysis of an abstract reachability tree (ART). The ART represents an over-approximation of reachable states obtained by unwinding the control-flow graph (CFG) of the program. An ART node typically consists of a location in the CFG, a call stack, and a formula representing a region or a set of data states. The formula in an ART node is obtained by means of predicate abstraction.

4 Verification Approach

We explore two directions for the verification of a VELOS specification: translation into a behaviorally equivalent NuSMV model, and translation into a behaviorally equivalent

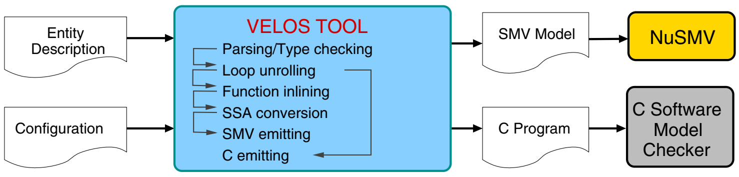


Fig. 3. Verification and validation flow

C program. The translation into a NUSMV model enables the use of a rich portfolio of verification techniques, e.g., BMC [7], temporal induction [19], CEGAR [15], all available in an extended version of the NUSMV model checker [12]. The translation into a sequential C program enables the use of mature off-the-shelf software model checking techniques and tools. Figure 3 shows our verification and validation flow.

To support our approach, we have implemented a translator, called the VELOS tool, that takes a VELOS specification as an input, and outputs a NUSMV model or sequential C programs. Independently of the final output, the VELOS tool always starts with parsing and type checking. During this phase all the syntactic and semantic errors, if any, are detected and reported to the user. The remaining steps deal with the specific target verification language. In what follows we describe the details of the translation, as well as some issues related to verifying multiple assertions using existing software model checkers. We also explain a customization of and an extension to our C model checker, KRATOS, to efficiently prove multiple assertions simultaneously.

4.1 From VELOS Specifications to NUSMV Models

To verify the VELOS specification, we have to model the application-environment loop. We model such an interaction in the NUSMV model using the synchronous semantics of NUSMV. That is, each transition in the model corresponds to a complete iteration of the loop. Note that this loop may contain an inner loop induced by the scheduler. This latter loop (also called the *scheduler loop*), however, is expected to be finite, and often its termination can be proved by means of simple syntactic criteria. For our case study, we are able to unroll completely the scheduler loop.

The translation of a VELOS specification to a NUSMV model is a complex transformation because of the difference between the paradigm of the two languages. A VELOS specification is in an imperative sequential language with control flow branches and variables assignments. NUSMV language is a transition-based data-flow language, where, for each variable, it is necessary to specify the precise transition relation between the current and next time step variables, including the frame conditions. Moreover, the flow of control in the sequential language has to be encoded explicitly in the NUSMV model using a program-counter variable.

The translation into NUSMV models first removes loops and function calls by, respectively, unrolling and inlining them. Such removals are possible due to the restriction in the VELOS specification, as described in Section 2.3. Finally, the resulting sequential program is translated into its static single assignment (SSA) [18] form, where each variable can be assigned at most once. Such a form reflects the final transition relation

```
                                  a_1 = a_0 + 2;
if (a > b) a += 2;                b_1 = a_0;
else b = a;                       a_2 = (a_0 > b_0) ? a_1 : a_0;
                                  b_2 = (a_0 > b_0) ? b_0 : b_1;
```

Fig. 4. Fragment of code (left), and corresponding SSA (right)

in the current and next time steps. For example, for the fragment of code on the left of Figure 4, the SSA conversion would generate the code on the right. In this translation we assumed the initial value of variables `a` and `b` are `a_0` and `b_0`, respectively. After the SSA conversion, the final value of every variable `V` is always `V_i` with the highest index `i`. This corresponds to the "`ASSIGN next(V) := V_i;`" NUSMV statement. In terms of the application, the value of `V_i` is the value of `V` after a complete iteration of the application-environment loop. All others `V_j` only hold values of intermediate expressions, and thus we can use the "`DEFINE V_j := expression;`" construct to avoid the introduction of explicit variables for their representation, which in turn enable NUSMV to inline them with the defined expressions.

The invariant properties in the VELOS specification are output directly as `INVARSPEC` NUSMV statements [11]. The translation also maintains a one-to-one correspondence between variables in the NUSMV model and the variable in the VELOS specification, e.g., for a class instance `I` of class `C` having class member variable `V`, a NUSMV variable `_I_V` is declared. Such a correspondence is important for examining counter-examples when some invariants fail to hold, and also for certification purposes.

To obtain NUSMV models that are amenable to be checked by NUSMV, we apply several optimizations in the VELOS tool. In particular, we reduce the number of variables and perform range analysis over possible values for variables. After this analysis, only the Boolean and enumeration types remain, and both are supported by the NUSMV language. For instance, if a variable has an initial value and is never re-assigned, than the variable can be removed completely by converting it into a constant.

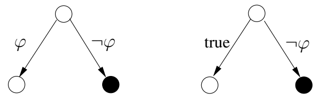
4.2 From VELOS Specifications to C Programs

The translation from VELOS specifications into C programs is simpler than the translation into NUSMV models because the language of VELOS and C are both imperative and have a lot in common. In particular, the VELOS tool converts member variables and member functions of class instances into C global variables and functions. Similar to the translation into NUSMV models, the translation into C programs also keeps a one-to-one correspondence between the variables in the resulting C programs and the variables in the VELOS specification.

The C program consists of a top-level main loop that models the application-environment loop. At the beginning of the loop body, the inputs are read and latched. As a common practice in software model checking, we use an extension of C that includes constructs to model non-deterministic data acquisition. The body performs computations according to the logic of the applications. It contains loops that are expected to be finite, and thus in principle can be unrolled. But in general, the unrolling may depend on the input acquired and on the computations performed within the loop. In particular, an inner loop of the top-level main loop corresponds to the scheduler loop.

```
int b1 = 0, b2 = 0;
assert(b1 != 0);
assert(b2 != 0);
```

(a)



(b) (c)

Fig. 5. Fragment of code (a), and assertions as branches in the control-flow graph: standard semantics (b) and assertion-as-property semantics (c)

All properties specified in the VELOS specification are translated into program assertions, and are positioned after all the function calls in the top-level loop body.

4.3 Model Checking Sequential Software with Multiple Assertions

The resulting C programs contain many program assertions. Here, we are interested in checking if *each assertion in the set can be violated.* However, if we put all assertions in a single C program, then it might be the case that some violated assertions can prevent other assertions violation from being detected. Indeed, given an assertion `assert(`φ`)`, the standard semantics requires that the execution of an assertion will pass if φ holds; otherwise a violation is detected and the execution quits. For instance, in the small program on the left of Figure 5 the second assertion violation can never be detected because the first one is always violated.

We explore two directions in addressing the problem of multiple assertions. First, for each property in the specification, we invoke the VELOS tool to generate a single separate C program such that the only assertion in the program is the property. Second, we invoke the VELOS tool to generate a single C program that contains all properties (or assertions) in the specification, but, during the analysis, we give a different semantics to the assertions such that it allows the execution to escape from assertion violations.

In the first direction, most existing software model checkers for C are readily applicable to verify the generated C programs. However, for a single specification, these model checkers have to be run many times, one run for each of the generated C programs. This bounds to be inefficient. Particularly for the CEGAR-based tools, as the generated C programs come from the same specification, these tools might perform the same abstraction-refinements on these programs.

To overcome these problems, in the second direction, we have extended the sequential analysis of KRATOS to analyze multiple assertions. First, we treat assertions as properties: given an assertion `assert(`φ`)`, the execution of the assertion can pass whether or not φ holds, but can also quit when φ does not hold. To support this semantics, we customize KRATOS to translate the assertion `assert(`φ`)` into a different kind of branch in the CFG such that the branch follows the new semantics. That is, instead of translating the assertion into the (b) branch of Figure 5, KRATOS translates the assertion into the (c) branch. With the new translation, checking for an assertion violation can still be reduced to the reachability analysis of the error node (dark node in Figure 5).

Second, we have implemented two different sequential analyses for checking multiple properties simultaneously: *all-in-one-go* and *one-at-a-time*. The all-in-one-go analysis is basically the standard lazy predicate abstraction, but instead of quitting the analysis on finding an assertion violation, the analysis continues the search for other violations. The known violated assertions are no longer considered in the successive searches. The one-at-a-time analysis, unlike the all-in-one-go, in one run checks one assertion at a time, but uses the ART constructed for checking the previous assertions to prove the subsequent assertions. On finding that an assertion `assert(`φ`)` cannot be violated, the corresponding CFG branch is strengthened by turning it into the (b) branch of Figure 5. The one-at-a-time analysis allows for performing an on-the-fly slicing with respect to the checked assertion. Such a slicing can reduce the size of symbolic expressions involved in the abstraction computations, and can also exclude predicates irrelevant to the assertion being checked. The analysis also allows for partitioning the predicates used to prove each assertion, and collecting loop invariants from the constructed ART that will be used to strengthen the successive searches.

5 Experimental Evaluation

5.1 Benchmarks

In our evaluation we consider the 10-RBS case study described in Section 2. To evaluate the scalability of the considered techniques, we create a set of benchmarks by varying the number of RBS's that can be chosen by the scheduler. The RBS's that can be chosen are called *active* RBS's. Given the 10-RBS case study, in total we have 1023 benchmarks with at least one active RBS.

The experiments have been carried out on all of the benchmarks. For presentation, in this paper we report only the statistics obtained from the experiments on two families of 10-RBS benchmarks: one family with one active RBS, subsequently called 1-active-RBS, and the other with all ten active RBS's, subsequently called 10-active-RBS. Experiments on other N-active-RBS families, for $1 < N < 10$, exhibit patterns of statistics that are similar to either that of the 1-active-RBS or that of the 10-active-RBS. The 1-active-RBS family consists of 10 benchmarks and the 10-active-RBS family consists of only 1 benchmark. As explained in Section 2, each benchmark has 53 assertions.

To experiment with NuSMV, the Velos tool generates an NuSMV model for each of the benchmarks. Each of 1-active-RBS NuSMV models consists of about 7 KLOC and has about 350 elementary Boolean variables. The 10-active-RBS NuSMV model consists of about 60 KLOC and has about 625 elementary Boolean variables.

To evaluate the software model checking approach, as explained in Section 4.3, the Velos tool has to generate a single C program for each of the assertions. Thus, for the 1-active-RBS family, we have 530 C programs, and, for the 10-active-RBS family, we have 53 C programs. Each of the generated C program is of size 40 KLOC.

5.2 Setup and Configurations

For evaluating the software model checking approach, we experimented with Kratos, Blast-2.7 [8], and CpaChecker [5] for the lazy predicate abstraction, with SatAbs-3.0 [17] (with Cadence SMV as the back-end) for the eager predicate abstraction, and with CBMC-4.0 [16] for the BMC approach. For SatAbs, Blast, and CpaChecker, we used the configurations that the tools used in the TACAS 2012 software verification competition. For CBMC, the loop unwinding was limited to the least value sufficient for detecting all assertion violations. For the direction via translation into NuSMV models, we experimented with the BMC algorithm implemented in NuSMV. (We disabled the counter-example generation and we set the bound to five steps.) To prove/disprove the assertions we ran all the software model checkers but CBMC. For CBMC and NuSMV, we focused on detecting assertion violations using BMC techniques.

We set the time limit to 500s and the memory limit to 2Gb for the experiments with the 1-active-RBS family, and we increased the time limit to one hour and the memory limit to 10Gb for the 10-active-RBS family. All experiments have been performed on a machine equipped with a 2.5GHz Intel Xeon E5420 running Scientific Linux.

5.3 Results of Experiments

The 1-active-RBS Case. The following table shows the results of experiments with the software model checkers on the 1-active-RBS family:

All Properties	Kratos	Blast	SatAbs	CpaChecker
Solved	530	0	244	312
Safe	436	0	244	218
Unsafe	94	0	0	94
Time out	0	56	286	2
Memory out	0	474	0	216
Total time	27m:26s	-	3h:58m:30s	1h:48m:30s
Max time	6.7s	-	221.3s	40.3s
Avg. time	3.1s	-	58.6s	20.8s
Max space	147.7Mb	-	454.6Mb	1.3Gb
Avg. space	74.8Mb	-	168.1Mb	985.5Mb

The row “Solved” indicates the number of benchmarks (or the number of assertions) that can be proved/disproved. The rows “Safe” and “Unsafe” indicate the number of, respectively, safe and unsafe benchmarks out of the solved ones. The rows “Time out” and “Memory out” indicate the number of benchmarks on which the tools went out of time and out of memory, respectively. The rows “Total time”, “Max time”, and “Avg. time” indicate, respectively, the total, the maximum, and the average time that the tools used for the solved benchmarks. Similarly for the rows “Max space” and “Avg. space”.

Only Kratos was able to solve all assertions. Blast was unable to verify any assertion; it either went out of time/memory, experienced failure in refinement, or crashed. SatAbs proved some assertions, but failed to disprove any. As mentioned before, SatAbs employs the eager predicate abstraction that requires it to create a Boolean program at each CEGAR iteration. Although SatAbs generates such a Boolean program efficiently, given that each benchmark is reasonably big, along with a large number of predicates in some cases, the resulting Boolean program is often too big for the back-end model checker, and thus SatAbs spends a lot of time in model checking the generated Boolean programs. Moreover, unlike Kratos that employs large-block encoding (LBE) [3] on the CFG, SatAbs uses basic-block encoding (BBE). As shown in [3], LBE tackles the problem of exploring a huge number of paths in the CFG.

Compared to CpaChecker, Kratos showed a better performance on the 1-active-RBS family. Unlike CpaChecker, Kratos performs aggressive, but cheap, static program optimizations (e.g., constant propagation, dead-code and unreachable-code eliminations) before analyzing the input program using the lazy predicate abstraction. These optimizations turns out to boost considerably the abstraction computations and the path feasibility analysis. Kratos can even prove some assertions by simply relying on these static optimizations and without performing the lazy predicate abstraction at all. CPACHECKER employs a flexible LBE on the CFG called adjustable-block encoding (ABE) [6]. The lazy predicate abstraction with ABE often suffers from memory problem because it has to keep track of the results of symbolic evaluations on the CFG paths in the ART nodes.

Focusing on bug hunting, the following table shows the performance of the model checkers in detecting assertion violations:

Unsafe Properties	KRATOS	BLAST	SATABS	CPACHECKER	CBMC	NUSMV
Solved	94	0	0	94	94	94
Time out	0	56	94	0	0	0
Memory out	0	38	0	0	0	0
Total time	6m:4s	-	-	42m:6s	22m:58s	59s
Max time	4.9s	-	-	38.9s	18.1s	-
Avg. time	3.9s	-	-	26.9s	14.7s	0.6s
Max space	145.7Mb	-	-	1.2Gb	257.9Mb	30Mb
Avg. space	121.5Mb	-	-	1.1Gb	246.8Mb	25.7Mb

The above table shows that the BMC algorithm in NUSMV performs the best in terms of run time and memory usage. Both CBMC and NUSMV are able to find all assertion violations. However, NUSMV handles enumerative types using a logarithmic encoding (see [12] for details) that turns out to reduce significantly the size of state space.

The 10-active-RBS Case. We now consider the most complex case, i.e. the 10-active-RBS family. The table below contains the accumulated results

All properties	KRATOS	BLAST	SATABS	CPACHECKER
Solved	53	0	0	8
Safe	33	0	0	8
Unsafe	20	0	0	0
Time out	0	2	52	0
Memory out	0	43	0	45
Total time	2h:36m:46s	-	-	17m:7s
Max time	553.4s	-	-	208.3s
Avg. time	177.5s	-	-	128.5s
Max space	5.2Gb	-	-	8.4Gb
Avg. space	4.5Gb	-	-	7.9Gb

while the following table focuses only on the case of unsafe properties:

Unsafe properties	KRATOS	BLAST	SATABS	CPACHECKER	CBMC	NUSMV
Solved	20	0	0	0	20	20
Time out	0	2	19	0	0	0
Memory out	0	10	0	20	0	0
Total time	26m:45s	-	-	-	2h:41m:22s	129s
Max time	85.7s	-	-	-	8m:26s	-
Avg. time	80.3s	-	-	-	8m:4s	6s
Max space	5.2Gb	-	-	-	728.1Mb	176Mb
Avg. space	5.2Gb	-	-	-	684.3Mb	176Mb

To a large extent, the above results emphasize the pattern discussed for the 1-active-RBS case (and for the intermediate n-active-RBS cases, not reported here for lack of space). Although the state-of-the-art software model checking techniques and tools are feasible for dealing with our case study, they are still far from being efficient in verifying our industrial-sized benchmarks. On the one hand, the CEGAR-based software model checking techniques can readily be used to prove or disprove assertions, but they are far less efficient than the BMC approach via the translation into NuSMV models in bug hunting. On the other hand, the BMC approach is known to be ineffective in proving assertions because one needs to know the diameter of the state space.

We also consider the impact of the new multiple-assertion analyses implemented in KRATOS. The effectiveness of these two new analyses depends heavily on the predicates used or discovered in the abstraction-refinements. In the all-in-one-go analysis, the predicates can simultaneously rule out some assertions violations, but can also clutter the search. The one-at-a-time analysis can benefit from the smaller size of the program resulting from the on-the-fly slicing, and possibly from a small number of predicates resulting from predicate partitioning. However, due to the slicing, the predicates used for solving previous assertions are often not sufficient for discharging the remaining ones. Thus, for solving the remaining assertions, the one-at-a-time analysis needs to further refine the abstraction. Overall, both techniques yield substantial speed-ups of about 80%.

Multiple assertions	KRATOS One-Per-File	KRATOS One-At-A-Time	KRATOS All-In-One-Go
Total time	2h:36m:46s	28m:46s	33m:36s
Max space	5.2Gb	6.3Gb	7.9Gb

Following the above results, we experimented with a simple strategy to further speed up the overall verification process. The idea is to combine several approaches by using cheap techniques to solve as many assertions as possible, and then use expensive ones to prove or disprove the remaining assertions. In particular, we first ran the BMC of NuSMV with a short time limit, to find as many assertion violations as possible, and then used the multiple-assertion analyses of KRATOS to solve the remaining assertions. The results are reported in the following table:

	NuSMV + One-At-A-Time	NuSMV + All-In-One-Go
NuSMV time	129 sec	129 sec
KRATOS time	560 sec	289 sec
Total time	11m:29s	6m:58s
Max space	5.4Gb	5.7Gb

Compared to using multiple-assertion analyses alone, in terms of run time, we further gained a speed-up of up to 80%. We also observe a positive impact on memory usage, with a reduction of up to 28%.

5.4 Remarks on other Experiments

The translation into NuSMV models allows for proving assertions by means of temporal induction. We experimented with the temporal induction implemented in NuSMV to first prove or disprove as many assertions as possible in the 10-active-RBS benchmark, and then use the multiple-assertion analyses to solve the rest. For this experiment, we only tried with induction of length 10. In this experiment, in addition to detecting all assertion violations as before, NuSMV was able to prove eight assertions. The following table shows that the induction slows down the verification but, with less properties to prove, Kratos, with the all-in-one-go analysis, consumes considerably less memory than before, i.e., up to 74% of reduction.

	NuSMV + One-At-A-Time	NuSMV + All-In-One-Go
NuSMV time	196 sec	196 sec
Kratos time	620 sec	254 sec
Total time	13m:36s	7m:30s
Max space	5.4Gb	0.9Gb

Finally, we remark that we also did experiments with explicit-state model checking techniques. Besides model checking, Kratos is able to encode C programs into Promela models that can be checked by the SPIN model checker [25]. However, it turned out that the size of the resulting Promela models is beyond the capability of SPIN: SPIN failed to translate the models into pan protocol analyzers.

6 Related Work

There have been numerous works that attempt to apply model checking for the verification of industrial software. A comprehensive survey can be found in [20]. Particularly for the verification of ERTMS, the work closely related to ours includes [1,13,24]. The work in [13] covers the whole safety logic of the interlocking application via manual translations into Promela, the language of the SPIN model checker. Besides verifying safety-related properties, the work also verifies liveness properties of the scheduler. Unlike our work, this work creates an under-approximation of the non-deterministic environment, and thus cannot show the absence of bugs.

Similar to [13], the work in [24] considers the whole safety logic, but only verifies bounded safety properties. The work relies on the translation into the target language accepted by the Verus model checker [10]. Both approaches in [13,24] showed poor scalability: they were applied only to small configurations with at most three processes.

The work in [1] shows the use of BMC approaches, via CBMC, to automatically generate test suites for the coverage analysis of safety-critical ERTMS. The problem of generating test suite can be reduced to the problem of verifying multiple assertions. However, due to the bound in loop unwindings, achieving full coverage is hard.

Other works related to the verification of ERTMS include [21,26,27]. The work in [27] applies theorem proving techniques to prove properties in European Train Control System specifications by means of manual encodings into Keymaera. The work in [21] adopts model-based testing, complemented with abstract interpretation techniques, for the verification of a railway signaling system. The work in [26] validates ERTMS specifications via translations into UML, and then uses Petri Nets models to generate test scenario.

Related to the multiple-assertion analysis, the work in [4] describes a technique similar to the all-in-one-go analysis in this paper. However, instead of targeting the verification of multiple properties, the goal of that work is to generate a test suite guaranteeing target predicate coverage.

7 Conclusions and Future Work

We have presented an application of model checking techniques to the verification of a significant fragment of Logica di Sicurezza, the safety logic of an ERTMS Level-2 system developed by Ansaldo-STS. We have developed a verification approach, and performed an evaluation of different verification techniques and tools. From this evaluation, we have learned two main lessons. First, even though existing verification techniques and technologies are readily applicable in the industrial settings, they might be neither efficient nor effective in verifying benchmarks coming from these settings. Moreover, a single approach alone is often not sufficient to handle the benchmarks or to satisfy the high-efficiency demand from the product development team. Second, an appropriate combination of approaches/techniques/technologies can dramatically improve the verification of industrial benchmarks. But such a combination can only be obtained by a thorough experimental evaluation on existing or new techniques.

In this work we have extended the KRATOS software model checker with two analyses that allow for checking multiple assertions simultaneously. We have successfully found a strategy that can handle our case study effectively. That is, we benefit from the efficiency of BMC for ruling out as many assertion violations as possible, and then check the rest of the assertions using the above KRATOS analyses. This strategy has proved to greatly reduce the verification effort.

For future work, given the very promising results, Ansaldo-STS is currently collaborating with FBK to integrate the methodology and approach in its internal Development and V&V Flow in order to verify the whole safety logic, with a possibility to verify the correctness regardless of the configuration.

We also plan to evaluate new model checking techniques, like Property Driven Reachability [9]. Moreover, by exploiting the functionality of NUSMV to dump verification problems in AIGER format, we plan to experiment with other model checkers.

Finally, for the certification of the application, as mandated by the standards, we will work on the certification and qualification of KRATOS and NUSMV.

References

1. Angeletti, D., Giunchiglia, E., Narizzano, M., Puddu, A., Sabina, S.: Using bounded model checking for coverage analysis of safety-critical software in an industrial setting. J. Autom. Reason. 45(4), 397–414 (2010)
2. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Art. Int. and Applications, vol. 185, pp. 825–885. IOS Press (2009)
3. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD, pp. 25–32. IEEE (2009)
4. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. STTT 9(5-6), 505–525 (2007)
5. Beyer, D., Keremoglu, M.E.: CPACHECKER: A Tool for Configurable Software Verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
6. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Bloem, R., Sharygina, N. (eds.) FMCAD, pp. 189–197. IEEE (2010)

7. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
8. Blast-2.7, `http://forge.ispras.ru/projects/blast`
9. Bradley, A.R.: Sat-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
10. Campos, S.V.A., Clarke, E.: The verus language: representing time efficiently with bdds. Theor. Comput. Sci. 253(1), 95–118 (2001)
11. Cavada, R., Cimatti, A., Jochim, C.A., Keighren, G., Olivetti, E., Pistore, M., Roveri, M., Tchaltsev, A.: NuSMV User Manual v 2.5 (2011), `http://nusmv.fbk.eu`
12. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A New Symbolic Model Checker. STTT 2(4), 410–425 (2000)
13. Cimatti, A., Giunchiglia, F., Mongardi, G., Romano, D., Torielli, F., Traverso, P.: Formal verification of a railway interlocking system using model checking. Formal Asp. Comput. 10(4), 361–380 (1998)
14. Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: KRATOS – A Software Model Checker for SystemC. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 310–316. Springer, Heidelberg (2011)
15. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
16. Clarke, E., Kroning, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
17. Clarke, E., Kroning, D., Sharygina, N., Yorav, K.: SATABS: SAT-Based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
18. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13(4), 451–490 (1991)
19. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. 89(4), 543–560 (2003)
20. Fantechi, A., Gnesi, S.: On the Adoption of Model Checking in Safety-Related Software Industry. In: Flammini, F., Bologna, S., Vittorini, V. (eds.) SAFECOMP 2011. LNCS, vol. 6894, pp. 383–396. Springer, Heidelberg (2011)
21. Ferrari, A., Magnani, G., Grasso, D., Fantechi, A., Tempestini, M.: Adoption of model-based testing and abstract interpretation by a railway signalling manufacturer. IJERTCS 2(2), 42–61 (2011)
22. Gargantini, A., Heitmeyer, C.L.: Using Model Checking to Generate Tests from Requirements Specifications. In: Nierstrasz, O., Lemoine, M. (eds.) ESEC 1999 and ESEC-FSE 1999. LNCS, vol. 1687, pp. 146–162. Springer, Heidelberg (1999)
23. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
24. Hartonas-Garmhausen, V., Campos, S.V.A., Cimatti, A., Clarke, E., Giunchiglia, F.: Verification of a safety-critical railway interlocking system with real-time constraints. Sci. Comput. Program. 36(1), 53–64 (2000)
25. Holzmann, G.J.: Software model checking with SPIN. Adv. in Comp. 65, 78–109 (2005)
26. Jabri, S., El Koursi, E., Bourdeaudhuy, T., Lemaire, E.: European railway traffic management system validation using UML/Petri nets modelling strategy. European Transp. Res. Review 2, 113–128 (2010)
27. Platzer, A., Quesel, J.-D.: European Train Control System: A Case Study in Formal Verification. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 246–265. Springer, Heidelberg (2009)