Science of Computer Programming 78 (2013) 828–842

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

The Metrô Rio case study

Alessio Ferrari [a,*], Alessandro Fantechi [a], Gianluca Magnani [b], Daniele Grasso [b], Matteo Tempestini [b]

[a] *Dipartimento di Sistemi e Informatica (D.S.I.), Universitá degli Studi di Firenze, Via di Santa Marta 3, 50139 Firenze, Italy*

[b] *General Electric Transportation Systems, Via Pietro Fanfani, 21, 50127 Firenze, Italy*

ARTICLE INFO

Article history:
Received 8 February 2011
Received in revised form 10 April 2012
Accepted 15 April 2012
Available online 25 April 2012

Keywords:
Formal methods
Model-based development
Code generation
Railway

ABSTRACT

This paper reports on the Simulink/Stateflow based development of the on-board equipment of the Metrô Rio Automatic Train Protection system. Particular focus is given to the strategies followed to address formal weaknesses and certification issues of the adopted tool-suite. On the development side, constraints on the Simulink/Stateflow semantics have been introduced and design practices have been adopted to gradually achieve a formal model of the system. On the verification side, a two-phase approach based on model-based testing and abstract interpretation has been followed to enforce functional correctness and runtime error freedom. Formal verification has been experimented as a side activity of the project.

Quantitative results are presented to assess the overall strategy: the effort required by the design activities is balanced by the effectiveness of the verification tasks enabled by model-based development and automatic code generation.

1. Introduction

Industrial applications of formal methods and model-based development for railway signaling systems are discussed in many case studies. The Paris Metro [1], the SACEM system [2], and the San Juan metro [3] are past and recent examples of successful stories about the usage of these technologies in the railway domain. In this paper, we offer a further insight into the actual industrial usage of formal methods. To this end, we present the experience of a railway signaling company, namely the railway signaling division of General Electric Transportation Systems (GETS), in adopting formal specification and development techniques by means of Simulink/Stateflow for the development of the Metrô Rio Automatic Train Protection (ATP) system.

GETS was commissioned for the adaptation of its SSC[1] ATP to Metrô Rio at the end of 2008. This was a time when the company was finishing its first large scale development project that made use of formal model-based development with Simulink/Stateflow. Introducing this tool-suite within a safety-critical process is not a straightforward step: the lack of a formal semantics of the language and the absence of a certified code generator require strategies to be defined in order to have a sound and safe development. The project SSC–Metrô Rio provided the opportunity to improve and assess the process practices experimented in previous projects to address these issues. Indeed, GETS adopted the Simulink/Stateflow platform first for the development of prototypes [4] and afterwards for requirements formalization and code generation [5]. Experimentation with the code generator led to the definition of an internal set of modeling rules in the form of an extension

* Corresponding author.
E-mail address: alessiofer@gmail.com (A. Ferrari).

[1] Sistema di Supporto alla Condotta — Driving Support System.

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.04.003

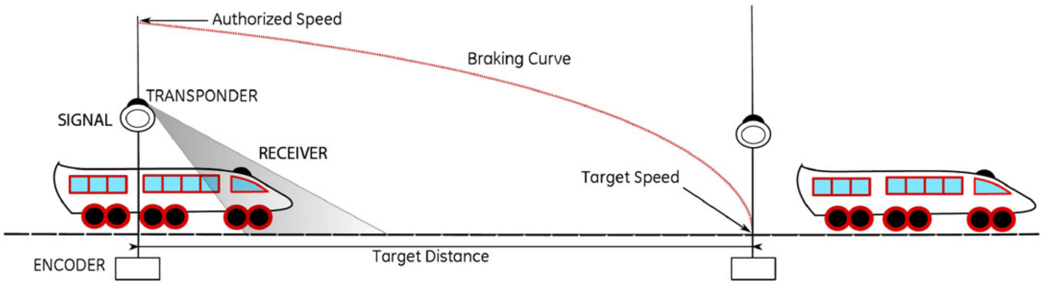


Fig. 1. Authorized speed, target distance and target speed.

of the MAAB guidelines [6], a stable and widely accepted standard developed by automotive companies. With SSC–Metrô Rio, additional rules have been introduced to constrain the language to a semantically unambiguous set. Furthermore, a hierarchical derivation approach has been defined in order to gradually achieve a formal model of the system. Concerning verification of models and generated code, an enhancement of the two-phase approach presented in [7] has been adopted: control-flow and functional properties have been verified through model-based testing, while static analysis by means of abstract interpretation has been used to check data-flow properties. The model-based testing activity has been performed through a code validation framework that executes the same test cases both at model level and at code level. The framework automatically verifies consistency of the test results for each model unit. This idea basically settles the problem of having a qualified code generator, since certification of conformity can be ensured each time the code is synthesized from a model.

The successful application of the presented process, witnessed by encouraging results in terms of cost reduction and safety assurance, has actually opened the door to formal verification. Experiments have been performed in the framework of the SSC–Metrô Rio project with the tool Simulink Design Verifier to explore the possibility of substituting part of the functional unit testing activities with formal verification.

The rest of the paper is structured as follows. In Section 2 the generalities of ATP systems are introduced. Section 3 gives an overview of the overall process. In Sections 4 and 5 the formal design phase is described and the semantics restrictions to the Stateflow language are discussed. Section 6 illustrates the code verification approach adopted, and Section 7 presents the experiments performed with formal verification. In Section 8 we review other reports concerning the application of formal verification within industrial settings where model-based design is employed. The last section draws general conclusions on the experience.

This article is an extended version of a conference contribution [8], augmented with the description of the results obtained by practicing formal verification in the context of the project. Compared to the original paper, despite small adjustments through all the sections, the current one adds Section 7 together with updated conclusions edited according to more recent results.

2. SSC–Metrô Rio ATP system

The role of a metro signaling system is to protect trains by keeping vehicles a safe distance apart. Traditionally, the traffic along metro tracks is managed by dividing each track into segments called *block sections* or simply *blocks*, and ensuring each train does not enter a given block section unless the block is clear of other trains. Signals are placed at the beginning of each block to inform the drivers about the status of the section that they are entering. The meaning of each signal aspect can be broadly represented by three pieces of information (see Fig. 1):

Authorized speed: the speed that is permitted in the block that is being entered;
Target distance: the maximum distance that the train can move while still being protected;
Target speed: the maximum speed that the train is permitted to have over the target distance. If the next block is occupied by another train, as in Fig. 1, the target speed is zero.

Automatic Train Protection (ATP) systems, such as the SSC–Metrô Rio one, are typically embedded platforms that enforce the rules of signaling systems by adding an on-board automatic control over the speed limit allowed to trains along the track. This ensures the safety of movement of the trains and the consistent protection of the line traffic independently of train operator actions.

The SSC–Metrô Rio system consists of carborne equipments and wayside devices. The latter are composed by an encoder and a transponder, which respectively encode and transmit a telegram containing the data to be processed by the carborne equipment. The wayside devices are positioned close to the actual signals, and the combination of encoder and transponder is commonly referred to as an *information point*. The carborne equipment receives the telegram data and performs the actual enforcement of train speed. Information managed by the carborne equipment concerns the approach speed and distance to the signals, but also other information typical for a metro, such as the distance to the next platform and speed reductions due to particular conditions of the line. All of this information is managed by the system as different targets; for each restriction, multiple braking curves are computed to determine the most restrictive speed. Interaction with the driver is primarily via a

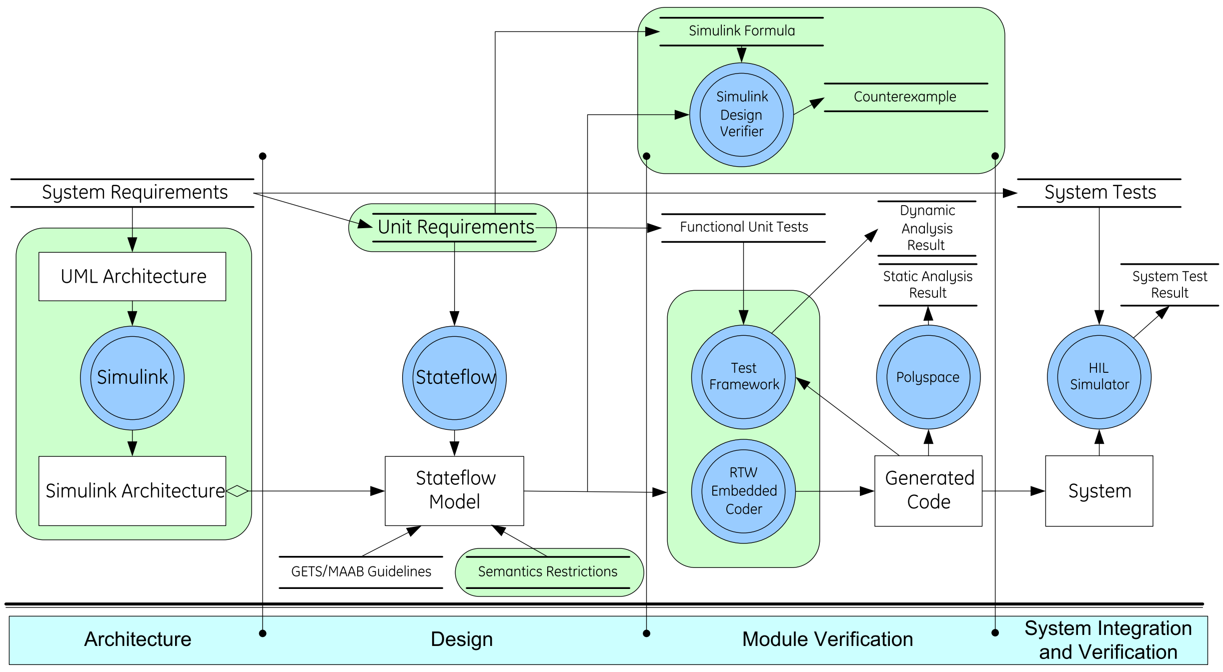


Fig. 2. Process overview.

touch-screen panel, which displays a speedometer with the current speed and the active speed limit, and provides a set of buttons and icons to let the driver control and monitor the system.

3. Process overview

In this paper we focus on the approach followed for the development of the carborne equipment of the SSC–Metrô Rio. The process adopted represents an application of model-based practices to a V-based life-cycle. Four phases are considered as the core of the system development: architecture, design, module verification and system integration/verification. In the context of the project, also formal verification has been introduced as experimental verification activity, although not yet consolidated into the currently practiced development process. Architecture and design activities concerning the project have been reported in a previous paper [9], while the verification steps represent an enhancement of the approach presented in [7].

Fig. 2 summarizes the overall process structure. Embedded in gray rounded boxes are the novel elements that have been introduced for this project. Starting from system requirements and using domain knowledge, a functional architecture in the form of a UML component diagram has been defined consisting of independent functional units (see Section 4). According to this decomposition, system requirements are partitioned into disjoint sets of unit requirements to be apportioned to the functions. The UML architecture is then translated into a Simulink architecture and the unit requirements are formalized in terms of Stateflow finite-state automata. The software of an on-board equipment of an ATP system, such as the SSC–Metrô Rio product, is characterized by the extensive usage of control modes logic and message analysis algorithms. These are all features that can be properly represented through state machines, and hence through discrete Stateflow models. Due to this reason, in the context of the project, only Stateflow has been adopted as specification language, while Simulink was only used as a simulation framework to allow interaction among Stateflow charts. Stateflow models have been designed using a safe subset of the language in order to ensure proper code synthesis. Already in previous projects an extension of the MAAB guidelines was used, aimed at enhancing the readability, maintainability and structuring of the code. With SSC–Metrô Rio, additional restrictions have been introduced to further constrain the Stateflow semantics to an unambiguous set of constructs (Section 5). Concerning code generation, the company adopted the more customizable RTW Embedded Coder in place of the previously used Stateflow Coder: the transition was basically painless, since all the modeling rules developed for the previous tool resulted in being applicable also for the new one. Unit tests have been defined in the form of scenarios at the Stateflow model level (Section 6.1). To this end, an internally developed framework has been used that automatically executes the same test suite on the Stateflow automaton and on the generated code to ensure functional coherence between model and software behavior. Confidence in the correctness of the generated code is increased with the Polyspace tool for abstract interpretation (Section 6.2), which completes the unit-level verification activities. Finally, system tests are performed on an ad-hoc train simulator with hardware in the loop.

Formal verification (Section 7) has been experimented at module-level using Simulink Design Verifier (DV), a proprietary proving engine that uses Simulink/Stateflow both as modeling and as property representation language. Unit requirements have been translated into Simulink formulae against which Stateflow models have been verified.

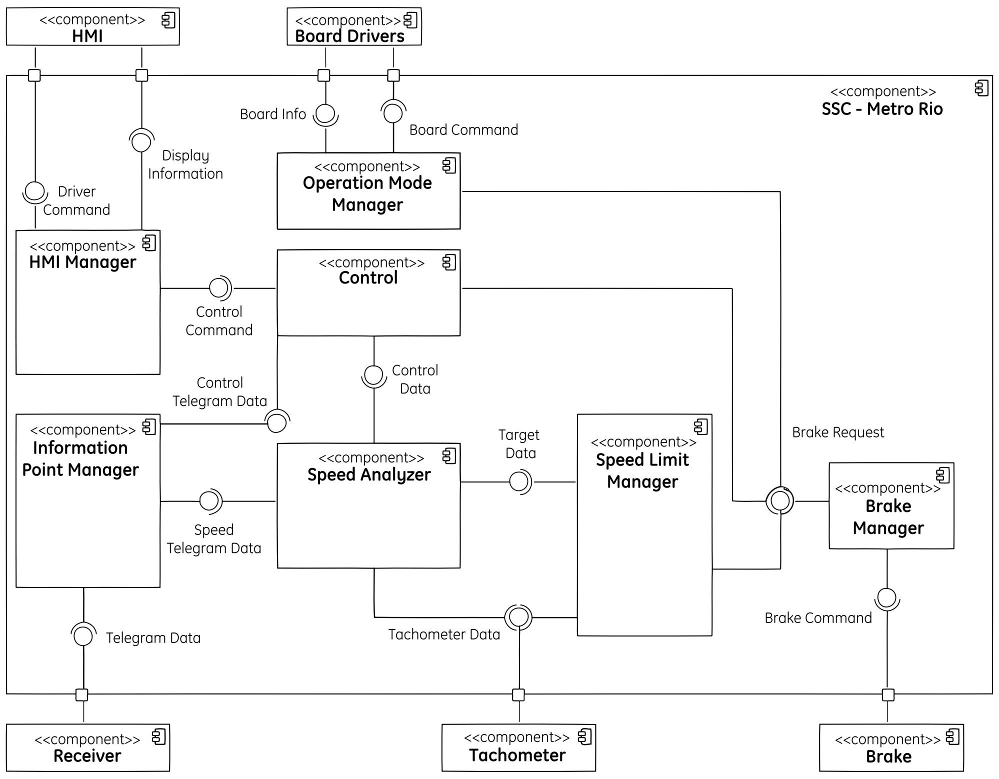


Fig. 3. Simplified component diagram.

4. Hierarchical architecture definition

When formalizing a large set of requirements, a well defined architecture of the model can help identify the components of the system and how they are interconnected, bridging the gap between requirements definition and component design. When automatic code generation is adopted, the architecture of the model is reflected in the software: an effort has hence to be made to create formal models having a structure that makes sense also in terms of the architecture of the software system.

Though being powerful languages, Simulink and Stateflow are not suitable to represent the high-level architecture of a system. Therefore, a more abstract approach was required in order to identify the semantic relationships between the software functional units, to build up the conceptual architecture of the application. In the context of the project, we found useful to first represent the high-level software architecture through a UML component diagram (a simplified version is represented in Fig. 3). UML component diagrams focus on the interfaces and dependencies of the functional units. Each component is basically defined by a set of implemented interfaces, a set of required interfaces and a set of dependencies. In the diagram, the external components represent the software drivers that interface the system to external devices, such as the tachometer and the braking command device, while the internal components are the system functions (for a better insight refer to [9]).

In order to properly formalize this architecture through Simulink/Stateflow, the chosen strategy was to represent the system through a multiple-level hierarchical model (see Fig. 4). The different levels are intended for different development stages, from a more abstract to a more detailed view. A first level is defining the context, which means the interfaces with the environment in terms of input/output data. Starting from the component diagram, this level has been derived considering the boundary ports and mapping them into signals entering or exiting the Simulink blocks. This approach allowed us to define the borders of the software system, which can be treated as a black box completely defined by its input/output signals. As part of this model we introduced other blocks simulating the actual interfaces (tachometer, touch-screen, telegrams, etc.) to perform interactive testing of the model.

A second level represents the internal software architecture in terms of interacting functional units modeled through Stateflow charts. For each of the components of the original diagram, a Stateflow chart has been defined having the same input/output interfaces in terms of variables: each required interface becomes a set of input variables, while each implemented interface becomes a set of output variables. Note that no function is implemented through Simulink, used only as a simulation framework, and no block with continuous dynamics is used in our approach. Any analysis problem that might be related to a hybrid semantics is therefore discarded.

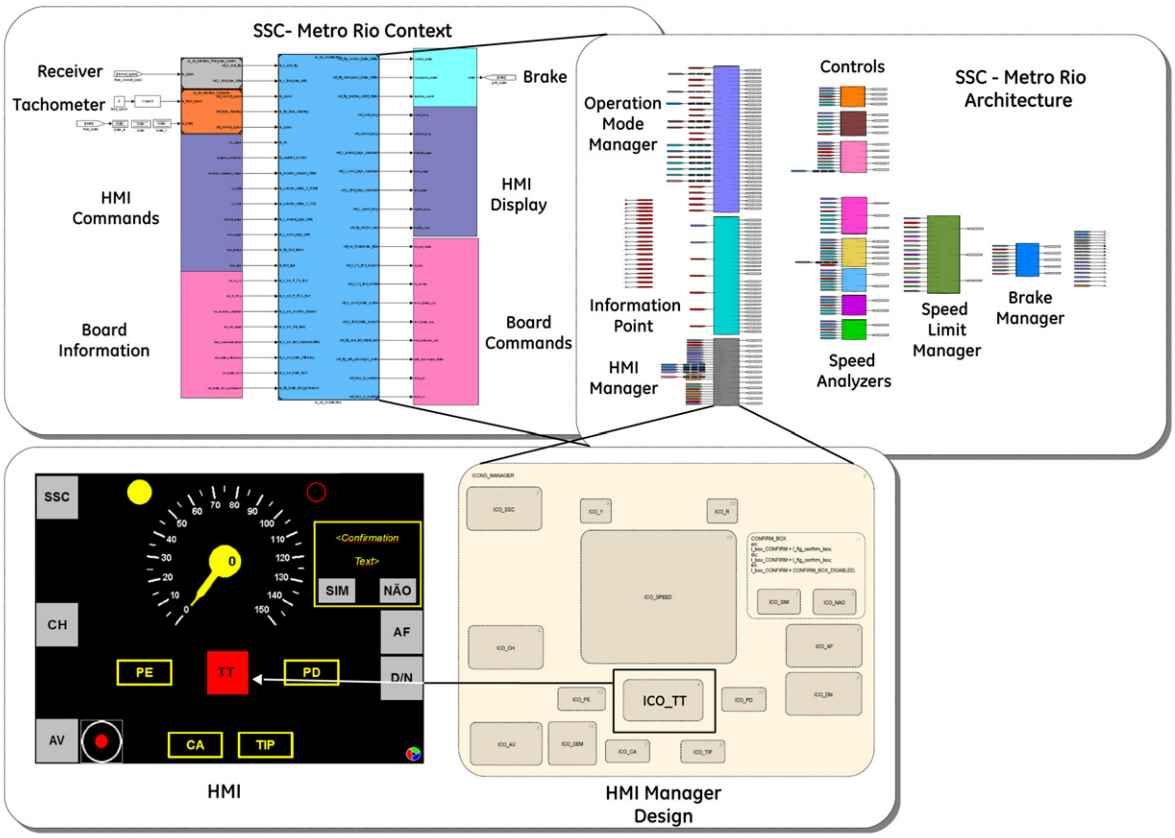


Fig. 4. The multiple level hierarchical model.

Table 1
Unit requirements decomposition.

	Requirement	Module
1	*If an information point with authorized speed equals to zero is received, the system shall raise the Train Trip (TT) event*	Information Point Manager
2	*If the TT event is raised, the TT procedure shall be activated*	Red Control
3	*If the TT procedure is active, it shall remain active until the train is not standing*	Red Control
4	*If the TT procedure is active, the brake shall be activated*	Brake Manager
5	*If the TT procedure is active and ICO_TT is invisible, ICO_TT shall start blinking*	HMI Manager

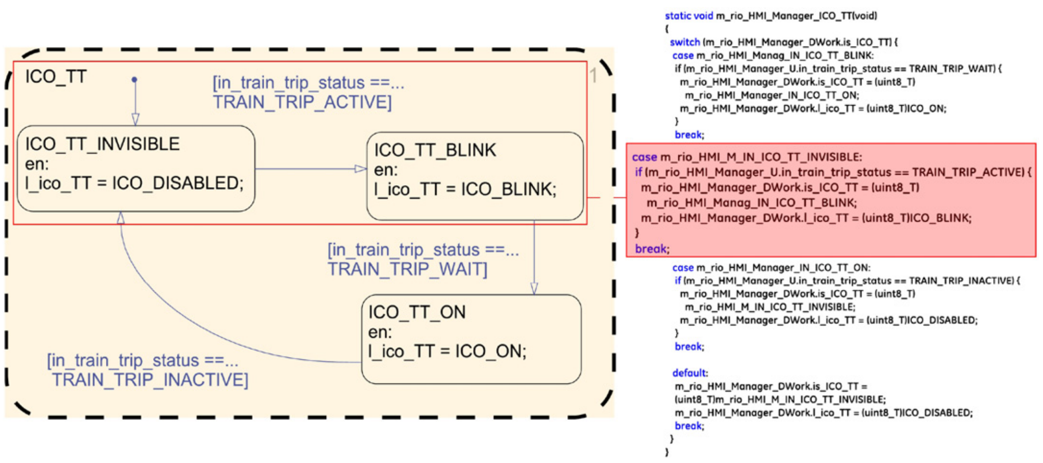
A third level is actually the design level of the single Stateflow charts, each of which is structured into parallel state machines formally modeling the system functional requirements. In order to derive such a formal model from the system requirements written in natural language, we first decomposed them into disjoint sets of unit requirements to identify the requirements apportioned to each single Stateflow chart.

The decomposition of the requirements is performed by domain experts. First the modules involved in the implementation of the system requirement are identified. Then, the role of these modules is expressed through unit requirements solely in terms of the input and output variables of the module. Given the natural language nature of system and unit requirements, logical equivalence among them cannot be ensured.

As an example, consider the system functional requirement concerning the control over the unauthorized passing of a red signal (normally called the Train Trip function): *When a red signal is passed without authorization, the system shall brake the train and the Train Trip icon (ICO_TT) shall blink on the display until the train comes to a standstill.*

The requirement is decomposed as reported in Table 1.

The first unit requirement is apportioned to the Information Point Manager, since this component is intended to interpret the telegram data and to forward events to the other functions. The second and third requirements are apportioned to the Red Control, which manages authorized and unauthorized passing of red signals. The fourth requirement is apportioned to the Brake Manager, which enforces any brake condition, and requirement 5 is apportioned to the HMI Manager, controlling the interaction with the driver.



```
static void m_rio_HMI_Manager_ICO_TT(void)
{
 switch (m_rio_HMI_Manager_DWork.is_ICO_TT) {
  case m_rio_HMI_Manag_IN_ICO_TT_BLINK:
   if (m_rio_HMI_Manager_U.in_train_trip_status == TRAIN_TRIP_WAIT) {
    m_rio_HMI_Manager_DWork.is_ICO_TT = (uint8_T)
     m_rio_HMI_Manager_IN_ICO_TT_ON;
    m_rio_HMI_Manager_DWork.l_ico_TT = (uint8_T)ICO_ON;
   }
   break;
case m_rio_HMI_M_IN_ICO_TT_INVISIBLE:
 if (m_rio_HMI_Manager_U.in_train_trip_status == TRAIN_TRIP_ACTIVE) {
  m_rio_HMI_Manager_DWork.is_ICO_TT = (uint8_T)
   m_rio_HMI_Manag_IN_ICO_TT_BLINK;
  m_rio_HMI_Manager_DWork.l_ico_TT = (uint8_T)ICO_BLINK;
 }
 break;
  case m_rio_HMI_Manager_IN_ICO_TT_ON:
   if (m_rio_HMI_Manager_U.in_train_trip_status == TRAIN_TRIP_INACTIVE) {
    m_rio_HMI_Manager_DWork.is_ICO_TT = (uint8_T)
     m_rio_HMI_M_IN_ICO_TT_INVISIBLE;
    m_rio_HMI_Manager_DWork.l_ico_TT = (uint8_T)ICO_DISABLED;
   }
   break;

  default:
   m_rio_HMI_Manager_DWork.is_ICO_TT =
   (uint8_T)m_rio_HMI_M_IN_ICO_TT_INVISIBLE;
   m_rio_HMI_Manager_DWork.l_ico_TT = (uint8_T)ICO_DISABLED;
   break;
 }
}
```

Fig. 5. Example of unit requirement formalization.

Table 2
Restricted Stateflow subset compared with the original language.

Original set	Restricted subset
State s	State s
Active state $s_a = \oslash_s \mid s$	Active state $s_a = \oslash_s \mid s$
Junction j	Junction j
Path $p = \oslash_p \mid s \cdot p$	Path $p = \oslash_p \mid s \cdot p$
Event e	$\oslash$
Destination $d = p \mid j$	$\oslash$
Action a	Action sequence $a = \oslash_a \mid ba \cdot a$
	Declarative action da
	Basic action $ba = da \mid J$
Condition c	Condition c
Transition $t = (e_t, c, a, a, d)$	State transition $t = (c, a, p),\ c \neq \oslash,\ p \neq \oslash_p$
	Junction transition $tj = ((c \mid a), j)$
	Junction transition list $TJ = \oslash_{TJ} \mid tj \cdot TJ$
Transition list $T = \oslash_T \mid t \cdot T$	Transition list $T = \oslash_T \mid t \cdot T$
Composition	Composition
$C = OR(s_a, p, T, SD) \mid AND(boolean, SD)$	$C = OR(s_a, p, T, SD) \mid AND(boolean, SD)$
State definition $sd = ((a, a, a), C, T_i, T_o, J)$	State definition $sd = ((a, a, a), C, T_i, T_o)$
	where the conditions of the transitions in T_o are mutually exclusive
State definition list	State definition list
$SD = \{s_0 : sd_0; \ldots; s_n : sd_n\}$	$SD = \{s_0 : sd_0; \ldots; s_n : sd_n\}$
Junction definition list	Junction definition list
$J = \{j_0 : T_0; \ldots; j_n : T_n\}$	$J = \{j_0 : TJ_0; \ldots; j_n : TJ_n\}$

Fig. 5 shows the formal representation of the ICO_TT state machine, modeling requirement 5 of Table 1, with the corresponding generated code.

5. Modeling guidelines

Stateflow implements a variant of Harel's hierarchical statecharts [10], normally called charts according to the Stateflow taxonomy. The complex semantics of Stateflow is not formally based, though research has been performed to define an operational semantics [11] and a denotational semantics [12] for a Stateflow subset. Along with the development of the SSC–Metrô Rio project, in order to achieve more easily an unambiguous interpretation of Stateflow models, coherent with the automatically generated code, we further extended the MAAB guidelines adopted in previous projects. In particular, we added a set of restrictions oriented to limit the use of the Stateflow language to a semantically unambiguous subset. With reference to the Stateflow language notation defined in [11], in Table 2 we represent the subset of Stateflow adopted for the SSC–Metrô Rio project (actual restrictions are highlighted in *grey*). This subset has been identified following the approach shown by Scaife et al. [13] for translating a subset of Stateflow into the Lustre formal language.

The main restriction concerns the elimination of events from the language, since, as pointed out in [13], events are possibly dealt with by code generators through the introduction of recursive code. This might lead to the risk of an infinite recursion call, stack overflow or to state-space explosion problems. For this reason events are forbidden by the adopted modeling guidelines and they need to be simulated through variable assignments as depicted in Fig. 6. In Stateflow every

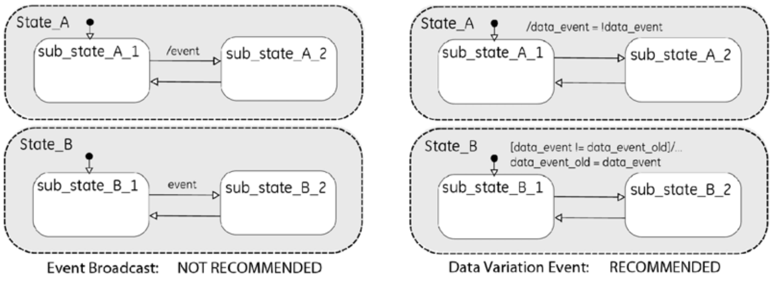

State_A
sub_state_A_1
/event
sub_state_A_2
State_B
sub_state_B_1
event
sub_state_B_2
Event Broadcast: NOT RECOMMENDED
/data_event = !data_event
[data_event != data_event_old]/...
data_event_old = data_event
Data Variation Event: RECOMMENDED

Fig. 6. Events can be avoided through proper variable assignments.

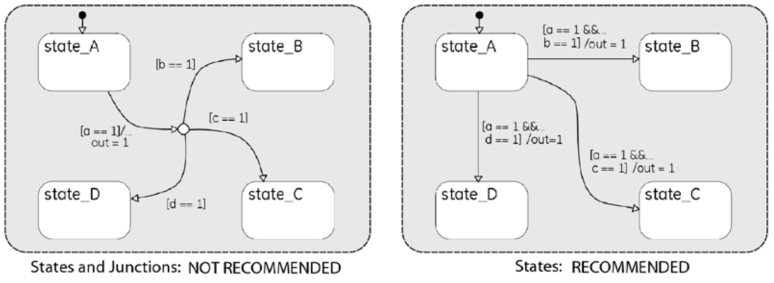

state_A
state_B
state_C
state_D
[b == 1]
[c == 1]
[a == 1]/... out = 1
[d == 1]
States and Junctions: NOT RECOMMENDED
[a == 1 &&... b == 1] /out = 1
[a == 1 &&... d == 1] /out=1
[a == 1 &&... c == 1] /out = 1
States: RECOMMENDED

Fig. 7. State/Junctions transitions can be avoided through proper modeling solutions.

chart is considered to be executed according to a deterministic sequence (State A is always traversed by the execution before State B) and therefore any race condition is avoided. This approach preserves the sequential execution of the code, while allowing for logical event implementation (each change on the variable value corresponds to an event).

Transitions have been distinguished between state transitions t and junction transitions tj; in the adopted guidelines it is not allowed to define transitions between states and junctions. Indeed, junctions and states have a different operational semantics; at each simulation step, states belonging to a single *OR* composition are mutually exclusive, while more than one junction can be traversed during the same step. The behavior discrepancy between the two objects might bring to improper combined usage. One of the well known possible hazards is *backtracking without undo* [13], a problem consisting of the possibility of traversing a path made of junctions, possibly assigning values to variables, and afterwards backtracking without restoring variable values. Fig. 7 shows how an improper modeling can be correctly translated into an equivalent, yet safer representation. The restriction concerning the state definition object, besides eliminating junction definition lists from states, requires the conditions over outgoing transitions to be mutually exclusive in order to avoid Stateflow to evaluate firing of transitions according to the *clockwise rule* [11]. This rule implies that transitions are ordered by their graphical appearance; the first transition is the one whose edge starts closest to the upper left corner of the source state, and the others follow clockwise. Hence, transitions naturally perceived as non-deterministic by the user, and interpreted as non-deterministic in other formal statechart languages such as Statemate, are actually deterministic. For this reason we require to make this determinism explicit by using mutually exclusive conditions on guards of transitions outgoing from the same state.

6. Code verification approach

Traditionally GETS has used, as the main approach to verification of code units, white-box testing based on path coverage. This approach has been revealed to be almost unfeasible due to the high structural complexity of the automatically generated code. A two phase verification process [7] was defined to address this shortcoming: the first phase implements model-based testing to verify the functional requirements coverage, the second phase employs abstract interpretation [14] to statically enforce runtime errors detection. For the application to the Metrô Rio ATP the first phase has been improved to obtain an implicit validation of the code generator as well. The verification activity is completed by final system tests with hardware in the loop. In this paper we do not describe this last step since it is not strictly related to formal modeling.

6.1. Model-based testing and code validation

Products traditionally developed by GETS, like any railway signaling application developed for Europe, shall comply with the CENELEC standards [15]. This is a set of norms and methods to be used while implementing a product having a determined safety-critical nature. The CENELEC EN 50128 considers two alternative strategies to assure the correctness of the tools that produce the code that is finally embedded in the product, such as compilers or automatic code generators: the first one requires the code generator to be validated, and the second one is based on the so called *proven in use* property. However, both strategies could not be applied for the adopted code generator: the validation of the generator is unfeasible,

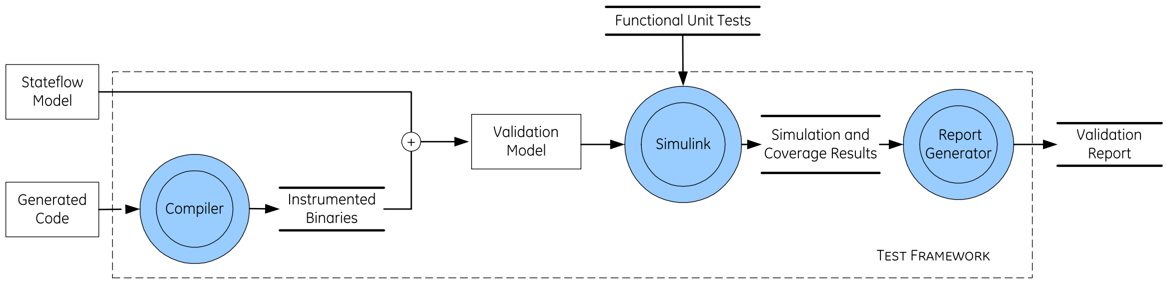


Fig. 8. 2M-TVF test framework.

since the source code is proprietary and no information is given about its development process; the translator could anyway be considered as a *proven in use* tool, since it is used by a certain number of industries that operate in safety critical contexts. Furthermore, the creation of a specialized tool, and its validation, is not in line with the company strategy, which completely resorts to the usage of commercial tools.

The approach used in the context of the Metrô Rio project is inspired by the one presented in [16], called Translation Validation[2]: this approach is not focused on the code generator itself, but on the inputs and the outputs of the code generation process and on their comparison. The validation of the generated code is performed through two phases: the first one consists of running the same extensive test suite (defined starting from the modeled requirements, with a 100% requirements coverage), on the models (by simulation) as well as on the generated code (by dynamic testing). In the first phase of the validation of the generated code, a functional/black box testing is performed, where both model and code are stimulated with the same inputs, and their outputs are compared for equality. If there are differences between the model outputs and the code outputs, they shall be assessed. A successful outcome of this phase gives confidence that the model and the generated code will show the same behavior in response to the same stimuli. In our experience, no differences were found executing the tests on the model and the code; this increased the confidence in the code generator GETS used to translate from Matlab models to embedded code.

However, it is still necessary to demonstrate that unexpected and unwanted behavior has not been introduced during the translation process. The second phase of the validation of the generated code is carried out to address this question: after the execution of the tests, the structural coverage percentage is measured on the models as well as on the code, and the measures are compared. As stated in [16], if the code coverage obtained after test execution is less than the model coverage, then some unwanted functionality might have been introduced by the translator. In order to perform this evaluation, it is necessary to use comparable metrics for model and code coverage [18]. In our case we chose to use decision coverage for the model and branch coverage for the code.[3] Since a branch is the outcome of a decision, branch coverage is defined as the ratio between the number of decision outcomes that have been exercised by a test suite and the total number of decisions in the code. The choice of those metrics is due to the fact that the CENELEC EN 50128 requires at least statement coverage, which is automatically implied when full branch coverage is achieved; furthermore, since no continuous dynamic Simulink blocks are used in our modeling approach, the decision for those metrics is appropriate. A framework called *2M-TVF*, which stands for Matlab Model Translation Validation Framework, has been developed to perform the model-based testing and the code validation process. The framework works under the Simulink environment, and in order to carry out the validation process, allows for the definition of a validation model that includes both the Matlab model to be tested and the related generated code, embedded in a Simulink block. The code is previously instrumented to permit the evaluation of the coverage after the test's execution, and then it is compiled and linked to obtain the executable (See Fig. 8.). The definition of the test suite is a manual activity: the tests are originated according to the requirement coverage criterion applied to the unit requirements that have been defined during the design phase of the system development. Automatic functional test generation techniques were not used due to the fact that they generally require models with a higher level of abstraction than the ones we use to generate code. The time needed to devise the tests, by a domain expert, is of the same order of magnitude than the time needed to model the system.

2M-TVF is totally automated: starting from the system under test (SUT) and a reference to the test suites, the framework uses RTW Embedded Coder to generate the code for the SUT and then it creates the validation model. The tests are then used as input for the validation model, and a report is visualized after the execution of every test. The report contains information about the result of the comparison of the model and code outputs and a detailed section on the coverage metrics obtained on both model and code. A subsequent analysis of the report is needed to assess every mismatch between outputs or coverage values.

By implementing the approach on many applications that use the same generator, it is possible to achieve the *proven in use* property for the code generator and, at the same time, to validate the generated code of every single application.

[2] Translation Validation refers to a process that involves the comparison between code and models behavior as a model-based conformance testing. The term Translation Validation has been previously used in literature to address a similar problem in [17].

[3] Decision coverage and branch coverage are synonymous. It was decided to maintain both definitions since the former is used in code-testing context, while the latter is used in model-testing contexts.

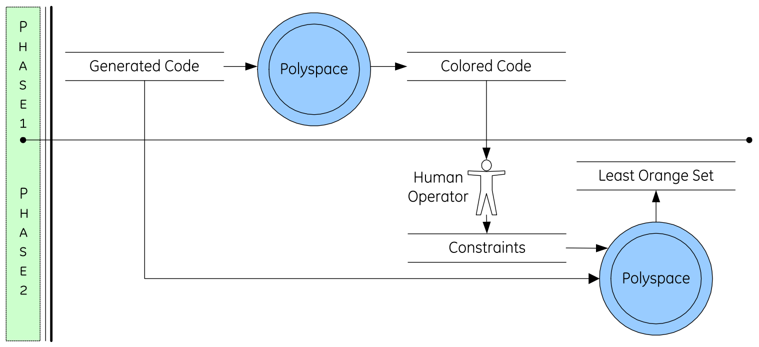


Fig. 9. Abstract interpretation phase.

6.2. Abstract Interpretation

Abstract Interpretation is a static analysis method that is able to infer dynamic properties of the code and to detect runtime errors and faulty states of the program without executing the code. The method defines an overapproximation of all the program reachable states, in order to check all the possible program runs without actually executing it. However, the computed overapproximation includes combinations of values and calls that cannot occur in any actual execution of the program. Consequently, tools for abstract interpretation may lead to false positives caused by the analysis of program runs possible in the overapproximation, but which do not belong to the real domain of the code. The tool chosen to perform abstract interpretation is Polyspace [19]. Polyspace works on C code and highlights possible runtime errors through the use of chromatic marks on statements:

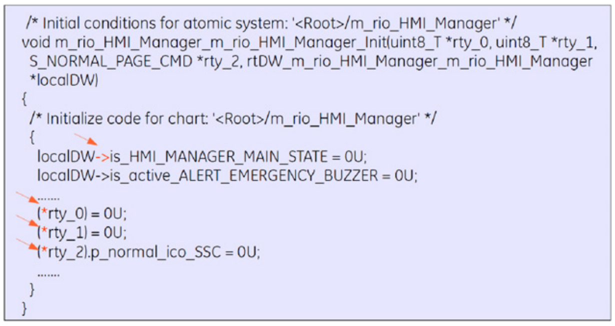
- *green* if the statement can never lead to a runtime error;
- *orange* if the statement leads to a runtime error in at least one analyzed program run;
- *red* if the statement leads to a runtime error in every analyzed program runs;
- *grey* if the statement is not reachable.

The critical issue in using Polyspace is the mining of actual code errors from the high number of orange warnings caused by overapproximation. GETS has adopted a two step process (see Fig. 9) in order to significantly reduce the orange checks that have to be manually reviewed. With the first step the code is quickly verified using a large overapproximation set. In the second step a finer approximation set is applied using the information obtained from the previous step. The first step is useful to detect systematic runtime errors (red) and unreachable statements (grey). Since no constraints are given in this analysis step, the set-up time spent is negligible. On the other hand results are not selective enough about the orange warnings, and, in order to define the constraints for the subsequent step, each orange warning has to be associated to the cause that could have produced it. An analyst with minimum proficiency with the tool can easily evaluate the orange marks and quickly define the classes of causes they belong to, although in this step it is still difficult and time consuming to identify those orange warnings that are actually false positives. The identified classes give sufficient information to suggest input constraints to be given to the tool to restrict the analyzed abstract domain of the program. Examples of loose input constraints are related to interleaving of function calls and range of program variables. The tool generates an automatic call for each interface function of the module and initializes the input parameters with full-range values. Since the tool has no information about the correct order of invocation of the interface functions, it verifies all possible interleavings of the generated calls. In Fig. 10 the four orange warnings, highlighted by the arrows, are caused by the function call generation performed by the tool. The input parameters are initialized with full-range values and consequently the tool detects that the pointers could be outside their bounds. The information obtained in the first step is used to identify the part of the approximation that needs to be refined. In the example reported in Fig. 10, value ranges for the input parameters need to be narrowed in the following analysis step. The second verification step, performed with restrictive settings, allows a finer approximation of the real domain of the program and a reduction of the number of false positives. The analyst can quickly check the small number of false positives and in the end is able to state that the code is free from runtime errors.

It should be noticed that the use of two verification steps does not produce a high overhead. Our experience, as shown by the results given in the next section, confirms that the review performed on the first phase is simplified by the fact that the generated code is characterized by a limited number of classes of causes of orange warnings, while the second verification phase normally results in a low number of warnings.

6.3. Results

The SSC–Metrô Rio ATP system consists of 13 Stateflow models for a total amount of approximately 120K lines of code. For each Stateflow model, unit test cases have been manually provided according to the functional requirement coverage. The test suite consists of 238 test cases, which cover 100% of the functional requirements on the model. Further test cases



```
/* Initial conditions for atomic system: '<Root>/m_rio_HMI_Manager' */
void m_rio_HMI_Manager_m_rio_HMI_Manager_Init(uint8_T *rty_0, uint8_T *rty_1,
 S_NORMAL_PAGE_CMD *rty_2, rtDW_m_rio_HMI_Manager_m_rio_HMI_Manager
 *localDW)
{
 /* Initialize code for chart: '<Root>/m_rio_HMI_Manager' */
 {
  localDW->is_HMI_MANAGER_MAIN_STATE = 0U;
  localDW->is_active_ALERT_EMERGENCY_BUZZER = 0U;
  .......
  (*rty_0) = 0U;
  (*rty_1) = 0U;
  (*rty_2).p_normal_ico_SSC = 0U;
  .......
 }
}
```

Fig. 10. Oranges caused by the automatic initialization of input function parameters.

Table 3
Bug detection and correction costs for comparable projects (modeling cost of approximately 4 man/months).

Project	#modules	Loc	#bugs	Man/h
SSC–Metrô Rio	13	120K	33	16
SSC–BL1Plus	12	40K	114	105

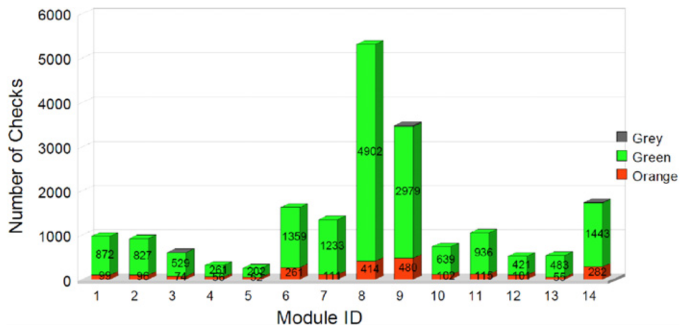


Fig. 11. Results of the first Polyspace verification step.

were added in order to reach 100% of the decision coverage on the models. The full requirements coverage assures that each requirement is satisfied by the model behavior; the full decision coverage assures that each decision in the model is exercised without causing unexpected responses by the model. The test framework provides for exercising the generated code with the test cases previously performed on the model. Table 3 compares the results of the verification activities on SSC–Metrô Rio in terms of bugs found and time spent to detect and correct the bugs, with the results of these activities on the SSC BL1 product, a previous ATP project based on model-based development where only the MAAB guidelines with proper restrictions were used. Although the new system has considerably increased the number of lines of code, the guidelines refinement led to a notable reduction of bugs while the well defined architecture derived from the novel design approach has allowed the detection of the errors in shorter time.

The first step of the Polyspace verification phase has detected no red errors, as shown in Fig. 11.

Although many oranges have been detected, thanks to the characteristics of the generated code, it has not been time expensive to classify these warnings according to the kind of approximation that supposedly produced them. Indeed, due to the disciplined use of modeling guidelines, the generated code has a high number of simple structures and has well-defined module interfaces, which have helped to confine the causes of orange marks to the two only classes, already mentioned, of wrong interleaving of function calls and automatic initialization of input function parameters.

The second step of the Polyspace phase has led to only few orange warnings, and most modules turned out to be entirely green. The results have been compared with the ones obtained on another previous project where Polyspace was first applied, but where modeling guidelines were less restrictive. As in the previous project, the orange marks detected in the first step are approximately 15% of the total number of checks for each module (see Fig. 11), but the time spent to classify the oranges and to determine the constraints for the second step has been considerably reduced thanks to the well defined structure of the generated code (Table 4).

Though the automation of the model-based testing phase through the 2M-TVF framework has improved the validation task in comparison with previous projects, the bottleneck of the activity resulted from the manual definition of tests for the models, which takes about 60%–70% of the cost of the verification process. In order to address this problem, the company has explored the possibility of substituting part of the functional test activity with formal verification.

Table 4
Comparison of costs for classification and constraints definition.

Project	#modules	Oranges	Analysis	Constraints
SSC–Metrô Rio	13	2298	38 h	6 h
SSC–BL3	21	1753	58 h	14 h

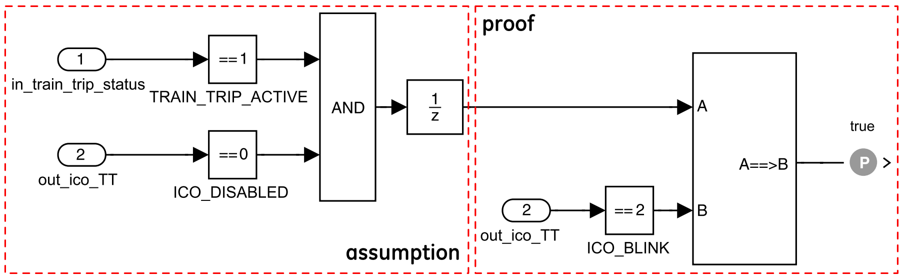


Fig. 12. Fifth requirement of Table 1 represented in the form of a Simulink circuit.

7. Formal verification

The CENELEC EN 50128 norm states that formal verification techniques, such as model checking [20] and theorem proving [21], are highly-recommended practices for safety-related software. Despite the norm guidelines and the successful history of formal verification in academia, the adoption of these technologies by companies is still more the exception than the rule. Formal verification tools are perceived as complex and esoteric, requiring highly specialized personnel that are able to deal with formal languages and temporal logic. Moreover, there is still no clear evidence on the actual cost reduction that can be achieved with respect to traditional testing.

With SSC–Metrô Rio, GETS decided to perform a systematic experimentation with formal verification by means of Simulink Design Verifier (DV), a test generation and property proving engine based on Prover Technology [22], which uses a proprietary algorithm based on Bounded Model Checking (BMC) with SAT-solvers [23] combined with K-induction [24] techniques. It was not the first time that the company practiced formal verification. In previous projects, the two open source model checkers, SPIN and NuSMV, were evaluated for the verification of railway interlocking equipments [25], and Statemate ModelChecker was used in the development of a system for object detection in level crossing areas [4]. In this case the choice fell upon DV, since the tool uses Simulink/Stateflow as modeling languages and it is fully integrated in the Matlab environment. These aspects have been considered as favorable elements to disrupt the reluctance of the developers towards formal verification.

The focus is again on units, and verification is performed at the level of the Stateflow models. The main objective was to understand if, with the new formal development process, the formal verification task could replace the costly unit testing task.

Verification through DV is performed by translating the property that one wishes to verify into a formula expressed in the Simulink language. In the case of SSC–Metrô Rio, the properties are the unit requirements we obtained through the system functional requirements decomposition. The formula derived from the property has the form of a graphical circuit where the variables considered by the property are connected by Simulink blocks implementing logical (*AND*, *OR*, *NOT*, etc.), arithmetic ($+$, $-$, $\times$, etc.) and time delay operators. The engine verifies that the formula is globally true for every execution path of the Simulink/Stateflow model. The property is interpreted as if both the *A* and *G* operators of the popular CTL (Computation Tree Logic) paradigm would be prefixed to it; it is basically an invariant. If the property is violated, a counterexample showing a failing execution is given in the form of a test case for the model.

As an example of property representation, consider the fifth unit requirement of Table 1, related to the Train Trip function of the system. This requirement can be represented in the form of a Simulink circuit as depicted in Fig. 12. Following the terminology of DV, the first part of the formula represents the *assumption*, which is a set of hypotheses on the state of the model (i.e., the value assumed by the input or output variables of the module). The second part of the formula represents the *proof*, which is a set of expected values on the output that one wants to hold whenever the hypotheses are met. The time delay operator of the assumption is used to ensure that the proof is checked one execution step after the verification of the assumption. The requirement has an equivalent CTL representation: $AG((in\_train\_trip\_status = 1 \wedge out\_ico\_TT = 0) \rightarrow AX(out\_ico\_TT = 2))$.

Such an equivalent CTL representation is not easy to give for more complex requirements, which can however be represented in the *assumption/proof* form. Consider again the Train Trip function: once the train comes to a standstill after the emergency brake is issued by the system, the Train Trip icon is supposed to stop blinking. Then, the driver pushes the icon for at least one second, the icon disappears, and the system releases the brake so that the train can leave again. We evaluate the following sub-requirement involved in this function and apportioned to the HMI Manager:

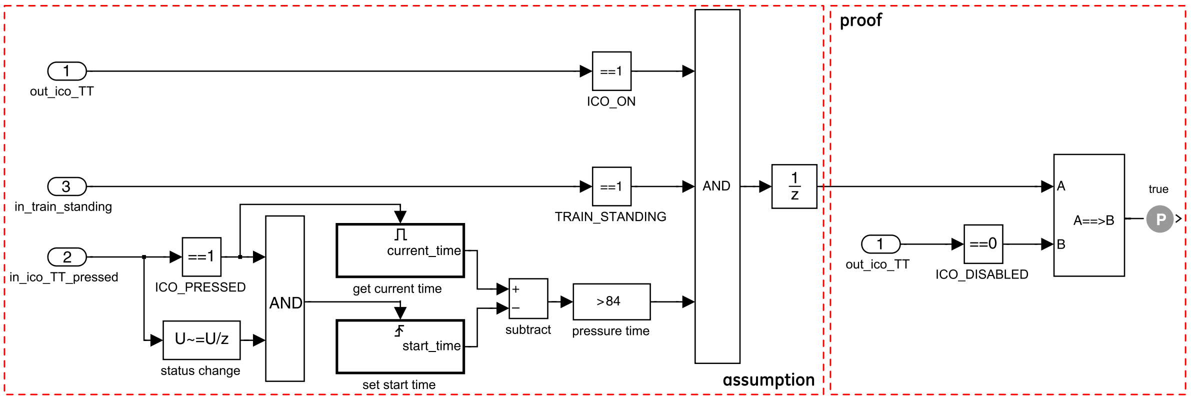


Fig. 13. Unit requirement with timer in the form of a Simulink circuit.

If the Train Trip icon (ICO_TT) is visible and not blinking, the train is standing and ICO_TT is pressed for at least one second, ICO_TT shall become invisible.

The corresponding Simulink circuit is represented in Fig. 13.

In this requirement a timer variable is involved: the button associated to the icon shall remain pressed for at least one second before ICO_TT can change its state. In the assumption part of the formula, two subsystems are used to evaluate the duration of pressure. This is supposed to be more than 84 main cycles (each main cycle has a minimum duration of 12 ms, therefore 1000 ms/12 ms $\approx$ 84) to enable the proof part of the formula. The CTL representation of this property is in principle feasible, but requires the usage of 84 nested sub-formulae, each using the next operator.

As one can infer from Table 1 and from the examples, all the unit requirements used in this context have the form *if* $\langle \Phi \rangle$, $\langle \Psi \rangle$. Φ is a condition sentence that might involve entities related to input and output variables in the model, while Ψ is an action sentence that involves entities related to output variables only. The translation of these requirements into the *assumption*/*proof* form is rather natural, and, in our requirements set, we have identified three classes of requirements, according to the structure of the formula resulting from the translation:

- requirements where Φ involves input variables only. These can be translated into a formula $assumption(I)/proof(O)$, where I and O are (possibly empty) subsets respectively of the input and output variable sets of the Stateflow model to be verified. This class of requirements includes requirements 1, 2 and 4 of Table 1. Invariants that are independent from the input also belong to this class.
- requirements where Φ involves both input and output variables. These can be translated into a formula $assumption(I, O')/proof(O'')$ where O' and O'' are (possibly disjoint) sub-sets of the output set O. This class includes requirements 3 and 5 of Table 1.
- requirements having any of the previous forms, but where the Φ or Ψ require some value to hold for a certain amount of time. These can be translated into a formula $assumption(*, T_{\Phi})/proof(*, T_{\Psi})$, where T_{Φ} is a set of timers on the variables of Φ, and T_{Ψ} is a set of timers on the variables of Ψ. The requirement associated to the circuit of Fig. 13 belongs to this class.

Requirements that do not belong to any of the classes above are not verifiable with DV. Among these requirements we consider both non structural requirements, normally verified through model inspection even when model-based testing is applied, and requirements that involve adherence of the output to complex mathematical functions, such as the ones related to the computation of the braking curve. In this case, the verification approach adopted has been model duplication and simulation; a continuous model of the curve has been created, and its behavior has been simulated and compared to the discrete model used for code generation.

The verification process discussed above has been conducted over a sample of the unit requirements. Actually, out of the 490 unit requirements, 377 were verifiable with the presented approach: 80 sample requirements have been extracted from this set.

Our experiments have shown that requirements belonging to the first class need 10 min of set-up time on average, while for the second class we reach 15 min. The verification time is in both cases negligible (less than one minute). The third class requires a longer set-up time, around 40 min for each requirement, mostly due to the complexity of expressing relationships between events in the timeline through memory blocks or time delay operators. In many cases the formal verification incurs state space explosion problems and fails to terminate, hence these requirements are likely to need to be verified through simulation and testing.

Table 5 reports the number of requirements belonging to each class for the SSC–Metrô Rio project. It can be noticed that most requirements belong to the first two classes, which required limited set-up and verification time. Requirements belonging to the third class are limited and are mainly related to HMI interaction (button pressure, icon display timers, etc.).

Table 5
Classification of requirements according to the associated formula.

	Class 1	Class 2	Class 3	n.v.
# requirements	285	73	19	113

According to these results, a new verification process based on formal verification can be foreseen. Requirements belonging to the first two classes can be formally verified with a cost that is 50%–66% lower than the one required for testing (each test needs 30 min on average). Requirements of the third class and non-verifiable cases can be treated with the previously used approaches: model-based testing or inspection. Unfortunately, except for the time elapsed in the verification and the results of the tests, DV does not give any further information about the process, and thus the user is unable to estimate precisely which is the actual cost of the verification, for example in terms of memory consumption.

The main threat to the validity of our evaluation is given by the fact that the models were already verified by means of testing; no further error was discovered during the formal verification phase. Our estimates do not consider the iterative task of adjusting models or formulae whenever a counterexample is issued. Nevertheless, we believe that the cost of these adjustments can be compared to the cost of fixing a bug detected by testing.

The introduction of formal verification in the established development process is still in the evaluation stage. Despite the encouraging results in terms of costs, there are other issues that have to be addressed, such as the qualification of the tools and the integration of the approach with the other process tasks. The company is currently defining solutions in these directions.

8. Related work on model checking within model-based design

In a large part of the safety-critical systems industry, the model-based design approach has emerged as the main paradigm for the development of software, often exploiting the possibility to automatically generate the code from the models. Formal verification by model checking on the models is still apparently less popular in industry. Indeed, many case studies and pilot projects have been presented in the literature, but, also due to confidentiality, only a few reports, such as those cited in the following, give interesting data on the overall adoption of formal verification by model checking in the development process.

One experience of model checking within model-based design in the avionics domain is the one done at Airbus [26], which has many points in common with the one we presented, notwithstanding the different application domain and the different environment employed, SCADE from Esterel Technologies. As in the case of Mathworks Simulink Design Verifier, Esterel Technologies SCADE Design Verifier is also built on top of the proprietary very efficient SAT solver by Prover Technology. Also in this case, an observer based approach to property expression is adopted, this time using the SCADE synchronous logic blocks and gates. Formal verification proved effective in the early discovery of violations in models of flight control functions: such violations could not be detected by early tests performed by simulation runs on the SCADE models, since these early tests did not consider highly dynamic aspects, e.g., unusual aircraft trajectories, which are usually taken into account only by later test on a flight simulator; however, the earlier violations are detected, the cheaper their correction. A challenge to formal verification was posed by some models which employed temporal counters, which produce enormous state spaces; one of these function was solved in 48 computation hours. Counterexamples (that is, detected erroneous executions) have been reported to be 50 to 160 cycles long, witnessing the intricacy of the conditions that lead to the error.

A prominent experience, again in the avionics domain, is the one by Miller and Cofer [27], where both Simulink/Stateflow and SCADE were used as model-based development environments, and several translators have been developed to apply different model checkers, among which SMV, NuSMV and, again, Prover, to models coming from different modeling tools. The first phases in this experience have confirmed that model checking is much more effective than testing in finding errors, and that BDD-based SMV and NuSMV are capable of dealing with very large state spaces. This has been actually proved by a parallel verification conducted by two independent verification teams: while the model checking team found 12 errors, the testing team was unable to find any errors, despite spending half of the time spent by the model checking team. Challenging issues, like the treatment of models that include floating point arithmetic, have also been studied by defining suitable abstractions to deal with the complexity of floating point arithmetic and the implied huge size of the state space. Due to the resulting loss of precision, model checking once again has demonstrated itself very valuable for debugging, but not at demonstrating correctness.

We can say that the reported experiences show several convergences, such as the use of a few commercial model-based development environments, and related formal verification engines, which allow an expression of properties by means of the same (graphical) formalisms in which the models are designed, a feature particularly appreciated by designers. Also common conclusions are that formal verification is actually far from being a single push-button experiment. It is rather an iterative process, and it is far more likely able to detect errors in the model than to certify the absence of errors.

9. Conclusion

In this paper we presented the formal model-based process adopted by a railway signaling manufacturer for the development of the on-board equipment of the Metrô Rio ATP system. According to our experience, the cost of formal modeling is 30% higher than manual coding.[4] This workload increase is partly due to the fact that graphic editing is inherently slower than textual editing, and partly to the training cost required by the technological shift. Nevertheless, the case study shows that this greater effort is payed back by the cost reduction of the code verification activities (about 70% in total, with respect to a manual coding based process) and by the increased confidence on the product safety and quality. Indeed, we showed that, through the formal approach introduced with SSC–Metrô Rio, we have been able to reduce by 71% the number of bugs found. At the same time, we reduced the average time required to detect and correct each bug from almost one hour to less than 30 min. These achievements are tangible and quantified benefits that demonstrate that moving from a code-centric process to a formal model-based one is *feasible* and *effective* in an industrial context.

The cost of formal modeling could be possibly decreased through the adoption of a modeling language with a simpler semantics than Simulink/Stateflow. However, such a choice should take into account the actual tool support for the chosen language. It should be possible to simulate and visually animate the models in order to assess the intended system behavior. Furthermore, an integrated platform with multiple toolboxes should be provided to integrate formal modeling within the overall system development. Unfortunately, simpler formal languages currently lack such a tool support, and this is the main reason why companies prefer to adopt semi-formal languages, which are more complex, but offer adequate aid for model animation and process integration.

The presented approach is the result of a long-term effort in introducing formal methods in the development process within the company. All the technologies used in the development of SSC–Metrô Rio, namely formal modeling, code generation, model-based testing and abstract interpretation, passed a three-stage exam before becoming an internal standard. First, they have been evaluated on an actual product, but offline w.r.t. the development process, then they have been introduced as part of pilot projects and finally they have been refined and adopted. A complete summary of the lessons learnt from these experiences has been published in [28]. Formal verification, experimented as a a side task in the Metrô Rio project, is in the first stage. The possibility of a further reduction of the unit-level verification costs encourages the company to invest in this direction, and activities are currently underway to prepare the subsequent stages.

References

[1] A. Faivre, P. Benoit, Safety critical software of meteor developed with the B formal method and the vital coded processor, in: Proc. of WCRR'99, 1999, pp. 84–89.

[2] G. Guiho, C. Hennebert, Sacem software validation, in: Proc. 12th Int. Conf. on Software Engineering, ICSE'90, IEEE Computer Society Press, Los Alamitos, CA, USA, 1990, pp. 186–191.

[3] M. Leuschel, J. Falampin, F. Fritz, D. Plagge, Automated property verification for large scale *b* models, in: FM 2009: Formal Methods, in: LNCS, vol. 5850, Springer, 2009, pp. 708–723.

[4] S. Bacherini, A. Fantechi, M. Tempestini, N. Zingoni, A story about formal methods adoption by a railway signaling manufacturer, in: FM 2006: Formal Methods, in: LNCS, vol. 4085, Springer, 2006, pp. 179–189.

[5] A. Ferrari, A. Fantechi, S. Bacherini, N. Zingoni, Modeling guidelines for code generation in the railway signaling context, in: Proc. 1st NFM Symposium, 2009, pp. 166–170.

[6] Mathworks Automotive Advisory Board (MAAB), Control Algorithm Modeling Guidelines Using Matlab, Simulink and Stateflow, Version 2.0, 2007.

[7] A. Ferrari, G. Magnani, D. Grasso, A. Fantechi, M. Tempestini, Adoption of model-based testing and abstract interpretation by a railway signalling manufacturer, IJERTCS 2 (2) (2011) 42–61.

[8] A. Ferrari, D. Grasso, G. Magnani, A. Fantechi, M. Tempestini, The Metrô Rio ATP case study, in: Formal Methods for Industrial Critical Systems, in: LNCS, vol. 6371, Springer, 2010, pp. 1–16.

[9] A. Ferrari, A. Fantechi, M. Papini, D. Grasso, An industrial application of formal model based development: the Metrô Rio ATP case, in: Proc. 2nd Int. Workshop on Software Engineering for Resilient Systems, SERENE'10, 2010.

[10] D. Harel, Statecharts: a visual formalism for complex systems, Sci. Comput. Program. 8 (1987) 231–274.

[11] G. Hamon, J. Rushby, An operational semantics for Stateflow, Int. J. on Software Tools for Technology Transfer (STTT) 9 (5–6) (2007) 447–456.

[12] G. Hamon, A denotational semantics for Stateflow, in: Proc. 5th ACM int. conf. on Embedded software, EMSOFT'05, ACM, New York, NY, USA, 2005, pp. 164–172.

[13] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, F. Maraninchi, Defining and translating a safe subset of Simulink/Stateflow into Lustre, in: Proc. 4th ACM Int. Conf. on Embedded Software, EMSOFT'04, ACM, New York, NY, USA, 2004, pp. 259–268.

[14] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proc. 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL'77, ACM, New York, NY, USA, 1977, pp. 238–252.

[15] European Committee for Electrotechnical Standardization, CENELEC EN50128, Railway Applications — Software for Railway Control and Protection Systems, 1997.

[16] M. Conrad, Testing-based translation validation of generated code in the context of IEC 61508, Form. Methods Syst. Des. 35 (2009) 389–401.

[17] S.M. Pnueli, A.E. Singerman, Translation validation, in: Proc. 4th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, in: LNCS, vol. 1384, Springer, 1998, pp. 151–166.

[18] A. Baresel, M. Conrad, S. Sadeghipour, J. Wegener, The interplay between model coverage and code coverage, in: Proc. 11th Eur. Int. Conf. on Software Testing, Analysis and Review, EuroSTAR'03, 2003.

[19] A. Deutsch, Static verification of dynamic properties — Polyspace white paper, 2004.

[20] E.M. Clarke, O. Grumberg, D.E. Long, Model checking and abstraction, ACM Trans. Program. Lang. Syst. 16 (1994) 1512–1542.

[21] C.-L. Chang, R.C.-T. Lee, Symbolic Logic and Mechanical Theorem Proving, 1st Edition, Academic Press, Inc., Orlando, FL, USA, 1997.

[4] The cost difference has been empirically measured over the effort (man/hour) spent for the two activities in similar projects. We do not mean this measure to be accurate.

[22] P.A. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, O. Åkerlund, Designing safe, reliable systems using scade, in: ISoLA, 2004, pp. 115–129.
[23] E. Clarke, A. Biere, R. Raimi, Y. Zhu, Bounded model checking using satisfiability solving, Form. Methods Syst. Des. 19 (2001) 7–34.
[24] M. Sheeran, S. Singh, G. Stlmarck, Checking safety properties using induction and a SAT-solver, in: Formal Methods in Computer-Aided Design, in: LNCS, vol. 1954, Springer, 2000, pp. 127–144.
[25] A. Ferrari, G. Magnani, D. Grasso, A. Fantechi, Model checking interlocking control tables, in: Proc. 8th FORMS/FORMAT Symposium, 2009.
[26] T. Bochot, P. Virelizier, H. Waeselynck, V. Wiels, Model checking flight control systems: the Airbus experience, ICSE Companion (2009) 18–27.
[27] W.M.W. Miller, S.P., D.D. Cofer, Software model checking takes off, Commun. ACM 53 (2) (2010) 58–64.
[28] A. Ferrari, A. Fantechi, S. Gnesi, Lessons learnt from the adoption of formal model-based development, in: NASA Formal Methods, in: LNCS, vol. 7226, Springer, 2012, pp. 24–38.