ADOPION OF CODE GENERATION
BY A RAILWAY SIGNALLING
MANUFACTURER

Alessio Ferrari

Ph.D. Coordinator
Prof. Giacomo Bucci

Advisors
Prof. Alessandro Fantechi
Prof. Enrico Vicario

XXIII Ciclo – 2009-2011
Abstract

This dissertation is the result of a three years research experience at General Electric Transportation Systems (GETS), Intelligent Systems division of Florence. GETS is a well known railway signalling manufacturer that started practicing formal methods in 2002, with exploratory projects performed in collaboration with the Computer Engineering department (D.S.I. - Dipartimento di Sistemi e Informatica) of the University of Florence. At the end of 2007, the company decided to introduce the code generation technology within its development process, and contacted experts from the university to implement industrially feasible strategies to address this goal. The work presented here reports on the results achieved in the context of this collaboration.
# Contents

Introduction iv  
1 Background 1  
1.1 Formal model-based design ........................................ 1  
1.2 Model-based testing .................................................. 4  
1.3 Abstract interpretation .............................................. 7  
1.4 Automatic Train Protection (ATP) systems .................... 10  
1.5 CENELEC standards .................................................. 12  
1.6 Problem statement .................................................. 14  

2 Early Applications 17  
2.1 From SCA to SSC .................................................... 17  
2.2 The SSC BL1 and BL1 Plus projects .............................. 21  
  2.2.1 Required properties ............................................. 23  
  2.2.2 Code generation and evaluation ............................ 24  
  2.2.3 Guidelines definition and refactoring ..................... 25  
  2.2.4 Second evaluation ............................................. 27  
  2.2.5 Model evolution ................................................. 29  

3 SSC Baseline 3  
3.1 Software development principles ................................ 33  
3.2 Guidelines formalization .......................................... 36  
  3.2.1 Model architecture guidelines ............................. 38  
  3.2.2 Stateflow guidelines .......................................... 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>Guidelines verification</td>
<td>42</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Model Verifier</td>
<td>44</td>
</tr>
<tr>
<td>3.4</td>
<td>Model-based testing</td>
<td>47</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Results</td>
<td>50</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Evolution</td>
<td>52</td>
</tr>
<tr>
<td>3.5</td>
<td>Abstract interpretation</td>
<td>54</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Polyspace first step</td>
<td>57</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Polyspace second step</td>
<td>60</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Results</td>
<td>61</td>
</tr>
<tr>
<td>3.6</td>
<td>Results</td>
<td>67</td>
</tr>
<tr>
<td>4</td>
<td>The Metrô Rio Project</td>
<td>71</td>
</tr>
<tr>
<td>4.1</td>
<td>Project story</td>
<td>71</td>
</tr>
<tr>
<td>4.2</td>
<td>Development process</td>
<td>73</td>
</tr>
<tr>
<td>4.3</td>
<td>Hierarchical architecture definition</td>
<td>75</td>
</tr>
<tr>
<td>4.4</td>
<td>Modeling guidelines update</td>
<td>79</td>
</tr>
<tr>
<td>4.5</td>
<td>Translation validation and 2M-TVF</td>
<td>83</td>
</tr>
<tr>
<td>4.6</td>
<td>Results</td>
<td>85</td>
</tr>
<tr>
<td>5</td>
<td>Summary and Discussion</td>
<td>90</td>
</tr>
<tr>
<td>5.1</td>
<td>Projects and technologies</td>
<td>90</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Goal 1: modeling language restriction</td>
<td>93</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Goal 2: generated code correctness</td>
<td>94</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Goal 3: process integration</td>
<td>95</td>
</tr>
<tr>
<td>5.2</td>
<td>Lesson learned</td>
<td>97</td>
</tr>
<tr>
<td>Conclusion</td>
<td></td>
<td>101</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>105</td>
</tr>
</tbody>
</table>
The adoption of formal and semi-formal modelling technologies into the different phases of development of software products is constantly growing within industry [12] [80] [77]. Designing model abstractions before getting into hand-crafted code helps highlighting concepts that can hardly be focused otherwise, enabling greater control over the system under development. This is particularly true in the case of embedded safety-critical applications such as aerospace, railway, and automotive ones, that, besides dealing with code having increasing size and therefore an even more crucial role for safety, can often be tested only on the target machine or on ad-hoc expensive simulators.

Within this context, recent years have seen the diffusion of graphical tools such as the SCADE suite [31], mostly used in aerospace, ASCET [35] and AutoFocus [60], both oriented to automotive systems, and Simulink/Stateflow [71], a general-purpose platform integrated in the Matlab environment.

Simulink/Stateflow can be actually considered a de-facto standard for the development of embedded control applications. The Simulink language uses a block notation for the definition of continuous-time dynamic systems [62], while the Stateflow notation is based on Harel’s Statecharts [52] and supports the modelling and animation of event-based discrete-time applications. The
integration of the two languages allows a flexible representation of hybrid systems [48], while tools such as Stateflow Coder [72], Real Time Workshop (RTW) Embedded Coder [73] and TargetLink [33] support automatic source code generation from the models. These features, together with the large amount of associated toolboxes to analyse the different aspects of an application, has enabled a cross-domain spread of the platform.

The General Electric Transportation Systems (GETS), Intelligent Systems division of Florence [45] is a well-known railway signalling manufacturer leading in the field of Automatic Train Protection (ATP) systems. Inside a long-term effort of introducing formal methods to enforce product safety, the company decided to adopt the Simulink/Stateflow tool-suite to exploit model-based development and code generation within its own development process [5] [42].

Products traditionally provided by GETS, like any railway signaling application developed for Europe, shall comply with the CENELEC standards [18] [19]. This is a set of norms and methods to be used while implementing a product having a determined safety-critical nature. In order to certify a product according to CENELEC, companies are required to give evidence to the authorities that a development process coherent with the prescriptions of the norm has been followed.

Introducing the Simulink/Stateflow tool-suite and the code generation technology within a CENELEC based process is not a straightforward step. The code used in railway systems shall conform to specific quality standards, and normally the companies use coding guidelines in order to avoid usage of improper constructs that might be harmful from the safety point of view. When modelling is adopted, the generated code shall conform to the same standard asked to the hand-crafted code. Concerning the tools, the norms ask for a
certified or *proven-in-use* translator: in absence of such a tool, as in the case of the Simulink/Stateflow platform, a strategy has to be defined in order to assess the equivalence between the model and generated code behaviour. The modelling and code generation technologies are then required to be integrated with the established process, that shall maintain its coherence even if changes are applied.

In order to address these issues, in 2007 the company decided to contact experts from the Computer Engineering department of the University of Florence. The work presented in this dissertation reports on the results achieved in the context of this collaboration within three years of applied research. The outcomes will be presented in chronological order, linking them to the GETS projects where code generation has been experimented: from the first exploratory projects where the actual goals were not clear yet [5] [42], until the systems for which a formal model-based process has been established [43] [40] [41].

The rest of the dissertation is structured as follows. In chapter 1 an overview is given concerning the industrial application context, and a problem statement is clearly expressed. In chapter 2 a short review of the previous experience of GETS with formal methods is given and the first exploratory project, namely SSC Baseline 1, is described. Chapter 3 reports the story of SSC Baseline 3, the first ATP project where code generation was actually employed. Chapter 4 describes the Metrô Rio project, where the previous experiences have been profitably used to establish a complete formal model-based development process. Chapter 5 summarizes the lesson learned and the future study directions planned by the company.
1.1 Formal model-based design

The term *formal method* identifies a set of mathematically rigorous practices to support the development of software intensive systems [21] [11] [79]. It is common to consider three application levels to which formal methods can be employed: formal specification, formal development and machine checked proof. Formal methods that embrace all the three levels are development frameworks composed by a formal language, a verification strategy and a set of tools to translate a specification into an implementation in a rigorous manner [27]:

- the formal language is an unambiguous notation with well-defined syntax and semantics used to express a system specification;
- the verification strategy is a machine-automated mechanism that allows to assess that the system specification conforms to desired system
Formal model-based design

- properties;

- the translation tools are intended as a development process composed of consistent refinement phases, in which the verified specification is progressively transformed into more detailed representations of the system, until the actual implementation.

Of course this dictionary-like definition hardly applies to reality, and with the name formal method are often identified techniques that focus only on one or two of the formalization levels [79]. Among the traditional techniques that can be considered formal methods *tout-court*, it is worth citing the B method [99], the VDM (Vienna Development Method) [64] and the RAISE (Rigorous Approach to Industrial Software Engineering) [82] method. Examples of well-established techniques that focus only on part of the whole formal process are instead Z [102], PVS (Prototype Verification System) [85], HOL (Higher Order Logic) [46] and the more recent Coq [9].

All these techniques can claim a successful history in academia, but their employment at industrial level, even for safety-critical products, is still more the exception than the rule. In 1995, Bowen and Hinchey published the Ten Commandments of Formal Methods [13], a list of guidelines for applying formal techniques, edited according to their experience in industrial projects [57]. Ten years later, the authors review their statements, and they witness that not so much have changed [14]: the industrial applications that demonstrate the feasibility and the effectiveness of formal methods are still limited, though famous projects, like the Paris Metro onboard equipment [36], where the B method has been employed, and the Maeslant Kering storm surge barrier control system [107], where both the Z and the Promela [58] notations have been used, show that the interest is not decreased. The recent, comprehensive survey of Woodcock et al. [112] confirms that industries are currently performing studies on formal methods applications, but still perceive them as experimental technologies.
While formal methods have struggled for twenty years for a role in the development process of the companies, the *model-based design* [101] paradigm has gained ground much faster. The defining principle of this approach is that the whole development shall be based on graphical model abstractions, from which an implementation can be manually or automatically derived. Tools supporting this technology allows to perform simulations and tests of the system models before the actual deployment: the objective is not different from the one of formal methods, that is detecting design defects before the actual implementation, but while formal methods are perceived as rigid and difficult, model-based design is view as closer to the needs of developers, that consider graphical simulation as more intuitive than formal verification.

This trend has given increasing importance to tools such as the SCADE suite [31], a graphical modelling environment based on the Lustre synchronous language, Scilab/Scicos [100], an open source platform for modelling and simulating control systems, ASCET [35] and AutoFocus [60], both using block notations for the representation of distributed processes and embedding advanced verification capabilities. In this scenario, the safety-critical industry has progressively seen the clear establishment of the Simulink/Stateflow [71] platform as the *de-facto* standard for modelling and code generation. This success is mainly due to the several features of the tool-suite and to the effective and engineering-friendly modelling languages. The large number of built-in blocks provided by Simulink, together with the notable capabilities of Stateflow Statecharts, allows fast development of prototype applications which can be simulated and directly analysed with the support of the Matlab environment. On the other hand, since the languages and tool-suite are not formally based, their full employment for the development of safety-critical applications poses challenges from the verification and certification point of view: how to ensure that the generated code is compliant with the modelled application? How to integrate model-based practices with traditional
certified processes? These are all questions that started pushing the industries and researchers toward an integration between model-based design and formal techniques [1] [80], and take profits from the flexibility of the first and the safety assurance of the latter, going toward the definition of formal model-based design methods.

Large size companies have been the first to employ formal model-based practices. Already in 2006, Honeywell started defining an approach for the translation of Simulink models into the input language of the SMV model checker [75]. Airbus has used the model checking capabilities of the SCADE suite for ten years in the development of the controllers for the A340-500/600 series, including the Flight Control Secondary Computer and the Electric Load Management Unit [12]. The most complete, integrated methodology is probably the one currently practiced by Rockwell Collins [77], that starts from Simulink/Stateflow models to derive a representation into the Lustre formal language, and then perform code generation (in C and ADA) and formal verification through different engines such as NuSMV and Prover.

This dissertation reports the story of a medium size railway signalling company in putting into practice formal model-based strategies using the Simulink/Stateflow tool-suite. With the progressive formalization of the process, the company has found profitable to introduce also other technologies as a substantial completion of the development, namely model-based testing and abstract interpretation.

1.2 Model-based testing

Model-based testing (MBT) is a general term that means employing models to support any task related to the testing activities of an application [34] [17] [55]. In [56] a classification of the different MBT approaches is given according to the modelling language adopted, while an extensive introduction
to methods and tools can be found in [108]. We adopt the classification of [89], where four main strategies are identified:

- the most common one, often used in the literature for defining MBT itself [28], starts from the code to extract an abstract model of it. The model is then used to define abstract test cases (i.e., test cases at model level made of sequences of input and expected output) that are then transformed into actual test cases for the implementation under test;

- with the second strategy, the model to define tests is created manually, starting from the requirements and the software specification that have been used to produce the code;

- the third approach uses a common model to generate code and to produce tests. As noticed in [90], if the model used for automatic code synthesis is rather close to the actual software implementation, its usage for testing activities adds little improvements in terms of effort reduction during test definition. Nevertheless, repeating the execution of tests at model level and at code level helps increasing confidence on the translator, which is a central issue for safety-critical software, where every translation step, from model to source code to binary code, has to be ensured to be fully consistent. The methodology discussed in this dissertation is inspired to this third strategy;

- a last, more complex, approach uses two models: one, called design model, is used for code generation, and the other one, called test model and defined at a more abstract level, for test case generation. As presented in [3], where the Alloy language is used for the test model and the Z language is used for the design one, this approach shows promising results for detecting requirements flaws. However, the effort required for building two models complicates the adoption of this strategy by industry, for which time-to-market is also a pressing issue.
The resistance of the industry to cope with the cost of the paradigm shift is actually extended to MBT in general: only in the latest years we have experienced a greater interest of companies towards this technology. Early industrial applications of MBT have been surveyed in [88]. The authors state that the method is mature enough for industry, but the collected papers do not give any evidence concerning cost reduction in comparison with traditional testing. Among the methodologies adopted in the first industrial applications of MBT, the approach presented in [91], dealing with the smart cards domain, stands out. The CASE tool AutoFocus is used for behavioural modelling, and tests are generated through symbolic execution based on Constraint Logic Programming (CLP). The same approach is applied in [90] on an automotive network controller, to enlighten the benefits of MBT in detecting requirements errors.

The other sectors where MBT has been employed more recently, and valuable industrial research papers have been published, are the telecommunications, the automation and the aerospace ones. In the telecommunications domain, the work presented in [54] shows an application of MBT to a Wireless Application Protocol Gateway developed by Ericsson, where timed automata are used as specification language and a tool based on the Uppaal model checker generates the abstract test suite. Previous works [15] [68] had already shown the interest of the telecommunication industry to the MBT technology. The ABB Group, from the automation domain, constrained by process standards such as the IEC 61508, has recently started to deal with the integration within its development process of an automated testing approach that uses UML and TTCN-3 [103]. As in our work, the problem is analysed from a process point of view, which is a common issue in any standard regulated domain. Concerning flight critical software, [86] presents a novel strategy developed at the NASA Ames Research Center, that uses Simulink as specification language, and extract tests from a Java symbolic representation of
the model, to test the software that is auto-generated from the Simulink specification with Real-Time Workshop (RTW) Embedded Coder. This last state-of-the-art experience gives an alternative way of combining code generation and model-based testing, another common issue in the development of safety-critical systems.

1.3 Abstract interpretation

In industrial software verification and validation processes, dynamic analysis techniques such as testing, that are mainly focused on checking functional and control flow properties of the code, are normally complemented with static analysis, that aims at automatically verifying properties of the code without actually executing it. The properties checked by static analysers range from coding standard adherence and programming style checking, to absence of runtime errors. Examples of techniques traditionally adopted for static analysis are data flow analysis [44], program slicing [111], constraint solving [2], and, with an increasing spread in the latest years, abstract interpretation. Abstract interpretation [24] [25] [26] is based on the theoretical framework developed by Patrick and Radhia Cousot in the seventies. However, due to the absence of effective analyses techniques and to the lack of sufficient computer power, only after twenty years software tools have been developed to support it, so that applications of the technology at industrial level could take place.

Companies of the automotive and aerospace sector have applied abstract interpretation to evaluate worst-case execution time (WCET) of programs [106] [8] and for stack overflow prevention [38] [94], adopting both ad-hoc techniques and commercial software such as the WCET tool aiT and the StackAnalyzer package, distributed by AbsInt. Our focus is the application of the technology in the analysis of source code for runtime error detection,
Abstract interpretation

which means detecting variables overflow/underflow, division by zero, dereferencing of non-initialized pointers, out-of-bound array access and all those errors that, might them occur, would bring to an undefined behaviour of the program.

In this context, abstract interpretation is a methodology that aims at ensuring the correctness of a program in terms of runtime errors. Since the correctness of the source is not decidable at the program level, the tools implementing abstract interpretation work on a conservative and sound approximation of the variable values in terms of ranges of values, and consider the state space of the program at this level of abstraction. The problem boils down to solve a system of equations that represent an over-approximate version of the program state space. Finding errors at this higher level of abstraction does not imply that the bug also holds in the real program. The presence of false positives after the analysis is actually the drawback of abstract interpretation that hampers the possibility of fully automating the process. Uncertain failure states (i.e., statements for which the tool cannot decide whether there will be an error or not) have normally to be checked manually and several approaches have been put into practice to automatically reduce these false alarms. Besides academic studies on statistical approaches for false positive ranking [67] [65] and strategies based on refinements of the abstract interpretation analysis [95] [49], it is an interest of safety-critical systems manufacturers to address this problem in an effective manner.

In [16] the first experimentation of NASA with Polyspace [30] for the analysis of the Mars Exploration Rover flight software is presented. The presence of a 20% of warnings to be manually checked, that requires a time consuming activity on programs of several hundreds of thousands of lines of code (from now on, KLOC), and the slow performances of the tool, led the organization to develop a prototype program called C Global Surveyor, particularly targeted for buffer overflow detection [109]. The precision achieved did not
outperform the one of Polyspace, this suggesting that the bound is possibly inherently related to the abstract interpretation framework itself. However, the possibility of tailoring the tool to the type of software analysed, an option that was not available with a proprietary software like Polyspace, radically decreased the analysis time. In the avionics sector, successful experiments for the suppression of warnings [29] have been performed using the tool Astrée [10], currently distributed by AbsInt. Thanks to a tight collaboration between application and tool developers they have achieved a reduction to zero of the number of false alarms issued by the tool. An alternative approach, that consists in complementing the Polyspace analysis with bounded model checking by means of CBMC, has been experimented in the automotive domain [87]. The authors state that the strategy reduces of about 70% the warnings issued by Polyspace (no distinction between true and false alarm is given). The main weakness of this study is the fact that they bound the execution time of Polyspace to seven hours only, since they focus on the cost of the bounded model checking step, but this allows them to give only partial conclusions on the overall approach.

Though experimented on open-source software only, and oriented solely to buffer overflow detection, it is worth mentioning also the recent study presented in [66], that combines abstract interpretation with symbolic execution by means of satisfiability modulo theory solvers, and is able to filter out 68% of the false alarms on average. The strategies applied in [29], [87] and [66] are based on multiple-step refinement approaches that are conceptually close to the one presented in this work: after the first analysis, constraints are deduced and further analyses are performed.

Compared to the other methodologies reported in the literature, the main strengths of the strategy that will be presented along with this dissertation are: 1) it is currently implemented in the development process of the company, and therefore it is an assessed approach; 2) we use one single commer-
Automatic Train Protection (ATP) systems

The role of a railway signalling system is to protect trains by keeping vehicles a safe distance apart. Traditionally, the traffic along railway and metro tracks is managed by dividing each track into segments called block sections or simply blocks, and ensuring each train not to cross a given block section unless the block is clear of other trains and it is safe to do so. Signals are placed at the beginning of each block to inform the driver about the status of the section that they are entering. In a manner that is analogous to street traffic lights, if the line ahead is free the signal will display a permissive aspect (green traffic light), and the train is allowed to cross the block, if the section is occupied by another train, the signal will be restrictive (red traffic light), and the driver is required to stop the vehicle. Trains are significantly heavier than cars, and as such, stopping a train going at its maximum speed requires a significant amount of space. In order to ensure a safe braking distance, the driver needs to be alerted before approaching a red signal, and this is normally issued by protecting the section behind through a yellow signal (be prepared to stop in our car traffic analogy). Given three blocks in the direction of travel, a normal, simplified sequence of aspects will be green, then yellow followed by red. In particular line topologies, different combination of the three basic aspects on the same signal can be used (e.g., red/yellow, yellow/green), and the logical role of these aspects often depends from the country/customer specific signalling system adopted. However, the meaning of each aspect can be broadly represented by three pieces of information (see Fig. 1.1):

- **Authorized speed**: the speed that is permitted in the block that is being entered;
Automatic Train Protection (ATP) systems

- **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.

The expected behavior of the train is therefore determined by a sequence of signal aspects that are logically chained together and each ring of the chain issues an appointment to the next ring.

![Figure 1.1. Authorized speed, target distance and target speed](image)

Automatic Train Protection (ATP) systems are typically embedded platforms that enforce the rules of signalling systems by adding an on-board automatic control over the speed limit allowed to trains along the track, thereby ensuring the safety of movement of the trains. A signalling system by itself has only the role of an information system and safety can only be ensured if the train operator follows the indications provided, much as a street intersection is only safe if all drivers obey the traffic lights. An ATP system assumes the active role of ensuring consistent protection of the line traffic independent of train operator actions, by ensuring that authorized speed, target distance and target speed are respected.

From the architectural perspective, ATP systems are composed of wayside equipment and carborne equipment: the wayside equipment transmits infor-
mation about the authorized speed, the target distance and the target speed, according to the aspect of a specific signal; the carborne equipment receives this information, and determines the instantaneous speed limit by computing the braking curve that the train is required to follow in order to maintain safety. The computation of the braking curve involves many parameters that depend both on the railway line, such as the presence of switches and the grade of the track, and the train, such as the weight and length of the vehicle.

The projects on which this research is focused concern onboard equipments of ATP platforms: the software complexity of these systems, mainly characterized by control-modes logic and message analysis algorithms, make them good candidates for the application of formal model-based development.

1.5 CENELEC standards

In the context of European railway signalling and control, CENELEC (Comité Européen de Normalisation en ÉLectronique et en ÉleCtrotechnique) is the institution that issue standards for the safety certification of products. There, safety is defined as the absence of unacceptable risk levels [18]. CENELEC EN 50128 [19] is the norm that specifies the procedures and the technical requirements for the development of programmable electronic devices to be used in railway control and signalling protection. This norm is part of a family, and it refers only to the software components and to their interaction with the whole system (see figure 1.2). The basic concept of the norm is the SSIL (Software Safety Integrity Level): the higher is the level of a system, the more serious are the consequences of a failure. Integrity levels range from 0 to 4, where 0 is the lower level, which refers to software with no effects on the safety of a system, and 4 is the maximum.

Quantitative verification of safety cannot be conducted on software, since
it is not possible to reliably quantify failure rates and the contribution software has given to an occurred accident. For these reasons software is verified using qualitative techniques: the EN 50128 norm defines an elaborated life cycle for the definition of specifications, the development and the verification of the software. The norm encourages the usage of models and formal methods in every phase of the software development cycle, starting from the design to the verification. The rationale is that models are more related to abstract concepts than the technologies used for their implementation into code, and are therefore closer to the domain of the problem. On the other hand, sufficiently detailed models can be used for automatic generation of code: automatic code generation, according to EN 50128, requires that the code generator itself is somehow trusted. The norm lists two criteria that can be used to achieve trust in model-to-code translator. The first criterion requires a certification for the tool: in this way the translator will be qualified for the usage in a safety-critical development process. The second criterion (proven-in-use translator) is applicable to those translators that have been used in many previous applications and that have been monitored during
their usage to assess their trustfulness. Since any code generator available for the Simulink/Stateflow tool-suite is not certified according to the EN 50128, the only possibility was to fulfil the requirements of the proven-in-use criterion. This entails a workflow of several tasks of bug recordings and continuous reviews. As a core part of the process to achieve the proven-in-use property for the translator, we found useful to introduce a technique named translation validation [23]. This consists in verifying the functional equivalence between models and generated software by executing the same tests on the model and the code, and afterwards performing further structural analysis to ensure that no additional functionality has been introduced. The MBT testing approach that will be finally adopted as the first phase of our verification process is an implementation of the translation validation technique.

Abstract interpretation is not currently part of the recommended practices of the EN 50128 norm, since this has been published before abstract interpretation became a mature tool supported technique. However, due to the evident benefits that it can bring in terms of runtime errors detection, companies started practising it as a completion of the verification process to enforce the safety of its products. As a result of this growing interest, the Polyspace tool has been recently certified for usage in the context of the EN 50128 norm. The abstract interpretation approach finally adopted as a completion of the verification process shows how the company has employed Polyspace in its application domain.

1.6 Problem statement

General Electric Transportation Systems (GETS) is a well known railway signalling systems manufacturer leading in Automatic Train Protection (ATP) systems technology, and traditionally operating in the CENELEC regulated market. Inside an effort of adopting formal methods within its own deve-
opment process, GETS decided to introduce system modelling by means of the Simulink/Stateflow tools [5], and in 2007 chose to move to code generation [42]. In order to achieve this goal, the company decided to contact experts from the computer engineering department of the University of Florence. The collaboration was indeed a tradition, though, until that time, mainly limited to thesis projects. This was the first time that higher-quality support from graduated personnel was required: formal modelling with automatic code generation were seen as breakthrough technologies for managing projects of increasing size, and for satisfying the requirements of a global market in terms of product flexibility.

This dissertation is the result of a three years research activity started at the end of 2007, aiming at addressing the following:

**Problem Statement**

Defining and implementing a methodology for the adoption of the code generation technology by a railway signalling company

During the research, the problem statement has been decomposed into the following sub-goals:
1. Modelling language restriction

The code used in safety-critical systems shall conform to specific quality standards, and normally the companies use coding guidelines in order to avoid usage of improper constructs that might be harmful from the safety point of view. When modelling is adopted, the generated code shall conform to the same standard asked to the hand-crafted code. Hence, it is required the identification of a safe subset of the adopted modelling language to allow the production of code that is compliant with the guidelines and that can be successfully integrated with the existing one.

2. Generated code correctness

The EN 50128 norm asks for a certified or proven-in-use translator. In absence of such a tool, like in the case of the available code generators for Simulink/Stateflow, a strategy has to be defined in order to ensure that the code behaviour is fully compliant to the model behaviour, and no additional improper functions are added during the code synthesis phase. The objective is performing the verification activities at an abstract model-level, minimizing or automating the operations on the code.

3. Process integration

Product development is performed by companies by means of processes, that define a framework made of tasks, artifacts and people. Introduction of new technologies in an established process requires adjustments to the process structure, that shall maintain its coherence even if changes are applied. This is particularly true in the case of safety-critical companies, whose products have to be validated according to normative prescriptions. Hence, a sound process shall be defined in order to integrate modelling and code generation within the existing process.
2.1 From SCA to SSC

GETS started the first experiments with formal methods in 2002. In that period, the market of railway signalling was rapidly moving in Europe from a kind-of protected, national based market, to a free market. Before, the design of new equipments was carried on as a single team between the railway operator and the equipment providers. The evolution of the business framework had clearly separated the roles of the operator, which issues equipment specifications, and providers, which implement the specification, but also needs to produce addressing the global market. Hence, the specification themselves gained more importance, in particular with respect to the possibility to have unambiguous, formally defined, specifications. Together with other contextual issues (discussed in [5]), this was the main reason that pushed the company to start practicing formal methods.
In the first experiments, the Specification and Description Language (SDL) [61] was used to model an already produced system named SCA (Sistema Conta Assi, axle counter system) [4][37], with specific attention to the issues of validation coverage [6] and of code generation [7]. Though modelling with SDL allowed a formal methods culture to start to consolidate inside GETS, it was not felt that this was the definitive choice, both for some difficulties emerged with the language itself (the asynchronous nature of communication, inherited by the original mission of SDL to describe communication protocols, and some other characteristics of the messages management have been perceived as difficulties by the designers) and for the not clear future of the language and its support tools, which were going to be merged into the UML 2.0 world.

Following the trends of the international railway signalling arena, mainly inside the Eurointerlocking effort, later experiments have switched to Statecharts, at first in their Statemate [53] dialect. The experiments consisted in the formal specification of a railway signalling system for the objects detection in level crossing areas. The system, named PAI-PL, was developed and homologated SIL 4 by GETS using a customer paper-based requirements specification. During the experiment, that specification was translated in a Statemate model and analysed using the related model checker tools. The results showed both that formal methods could be used in specification activities and that could also permit to find mistakes or ambiguous aspects in requirements. However, also Statemate met the same doom of SDL. The cost of the tools for developing with Statemate was considered too high for the actual benefits that could have brought and the design department was more keen to adopt more flexible tools that could aid several aspects of the design, and not only the specification by statecharts of the discrete behaviour of a system.
A new candidate that appeared to satisfy the cost/benefit target of the company was the Stateflow component of the Matlab environment. Again, the PAI-PL product was used as reference for the experimentation. The activities showed that, though the semantics of the language was rather poor and with a lower degree of formalism compared with the one of Statemate, modelling with Stateflow was perceived as more intuitive. Furthermore, the large amount of toolbox available in the Matlab environment to model and analyse the different aspects of a system was felt very attractive.

Stateflow was actually adopted for the first time in the design of a brand new system with SSC (Sistema di Supporto alla Condotta, driving support system), while previous experiences were mainly playing with the specifications of systems already in production. SSC is the Automatic Train Protection (ATP) system developed by GETS for the Railway Italian Network (RFI), currently deployed over 2000 kilometers of Italian secondary lines. Development activities started in 2002 to satisfy the customer needs of a prototype of a new train stop system. The main functionality is to stop the train in case of SPAD (Signal Passed at Danger, also called as train trip), which is known to be the most common cause of serious railway accidents: after each restrictive signal (i.e., yellow signal), the system has to check the signal recognition by the pilot (through a dedicated panels button) and otherwise stop the train. These basic functionalities required an architecture (described in figure 2.1), composed by two subsystems: a wayside one, that had to process the signal aspect information and transmit it to the carborne equipment, and the carborne equipment itself, which has to receive the information transmitted, acquire input from the pilot through a panel and enforce braking, when necessary.

The logic of the system is in principle simple, but it was decided to develop a formal model of the system to better understand through simulation...
From SCA to SSC

Figure 2.1. SSC system architecture

activities the interaction between the wayside and the carborne equipments. But the real usefulness of the model appeared only few months since its birth, when the simple train stop system started to evolve to become the current ATP. Indeed, the message transmitted from wayside to onboard equipment was bearing more and more information and new complex functionalities were introduced. The requirements from the client changed so quickly that it was very difficult to consolidate them and, especially, to assure their consistency. In this scenario it was decided to improve the model developed and to use it for the managing and for the formal specification of the system requirements.

The model allows for the reproduction of the real dynamics of the system in terms of onboard acquisition of the messages transmitted by the wayside components and in terms of their processing by the onboard equipment. Moreover the model includes the carborne panel interaction, the odometer information processing and the braking enforcing. In other words, the model implements the high level logic of the onboard equipment and its interaction with the environment. The Simulink tool has been used to model all the interfaces of the onboard system: the panel, the speed information, the message transmitted by the onboard equipment, the braking system etc. A single Stateflow diagram has been used to develop the logic of the system: it is made up by functional blocks (one or more states connected by arcs), each
of them representing a functionality of the system.

The requirement manager used the Stateflow model as a prototype to animate the natural language requirements and agree on them with the costumer. At the same time, the model was used by the developers as a support for writing the actual application software.

2.2 The SSC BL1 and BL1 Plus projects

At the end of 2007 the V&V department of GETS started to investigate the possibility of introducing the code generation technology inside its development process. The initial idea was trying to substitute existing hand-crafted software components with auto-generated ones, in order to understand the challenges related to the paradigm shift, and to identify how far actually was the objective of having a complete auto-generated system. It was chosen to perform the experiments using the existing SSC logic model, developed in the previous years as a prototype for the definition of the system requirements. For this model, there was already an implemented hardware/software system, so that a consistent comparison between hand-crafted and auto-generated code could take place.

Of course, the model was developed primarily to support interaction with the costumer and without considering the issues related to code generation in a standard-regulated context. The code of embedded safety-critical platforms shall conform to high-quality standards: if the system incurs in a failure, there must be a way to trace the failure back to the fault that produced it, and the code shall be easily analysable to allow that. The software shall give evidence of a structured and controlled development, with well-defined architecture, proper comments and supporting documentation. As required by the EN 50128 norm, GETS employs a set of coding guidelines.
for developing applications in C, that restrict the language to a safe subset. The C syntax allows usage of constructs whose employment shall be restricted on the software of safety-critical systems. For example, the guidelines ask the developer not to use dynamic memory allocation (i.e., the malloc and free functions): the data memory shall have a fixed size with variables and structures having a pre-determined position. With dynamic memory allocation, it is not possible to test all the possible behaviors in terms of memory access, in particular it is not well-defined the behavior in case of a memory leak. Another example is the switch construct: each switch construct shall always be terminated with a default label, and each non-empty switch-clause shall be terminated with a break. Different forms of the construct are witnesses of improper structure of the program, and poorly controlled control-flow.

Other guidelines adopted by GETS concern the architecture of the program. A maximum size is given for each C file in terms of lines of code: smaller software modules having a well defined role. Usage of global variables is not recommended, since, besides being indicative of unstructured programming, they highly decrease the maintainability of the code.

The actual execution time of the generated code was another issue to take into account. ATP systems do not have hard real-time constraints, however they are reactive systems that, might a failure occur, shall activate the brakes in a limited amount of time in order to reach the safe state (i.e., the train standing condition). The reaction time is influenced by the execution cycle of the software, that in the SSC platform was bounded to 100ms, a time interval evaluated according to the overall fault-reaction time required by the non-functional requirements of the system. Compilers can optimize the assembly code generated from the C code to achieve better performances, but optimization shall be turned off when compiling code of SIL 4 software, in order to avoid optimizations to introduce additional behaviors. Therefore, also the generated code is expected to be already optimized at source code
level, and its response time shall be analogous to the one of hand-crafted code.

The question was: how to obtain from the SSC model an auto-generated code that was showing the same quality properties and performances of the hand-crafted one, and that was conforming to the GETS coding standard? The idea was that one could work at model level to obtain a code that was comparable with the one written by the developers. The approach adopted consisted in first defining a set of properties expected from the generated code, identify a set of modelling guidelines that could allow the satisfaction of the properties, and then refactor the SSC model according to the guidelines.

2.2.1 Required properties

The selected properties the code was expected to satisfy are listed below.

SRS Compliance: the code shall be compliant with the Software Requirements Specification (SRS) document, satisfying all functional and non-functional requirements for the system. This property encloses all the dependability attributes, typical of any safety-critical applications, belonging to the non-functional requirements set.

Performance: the performance of the code in terms of real-time response to the stimuli shall be comparable with those of the hand-crafted code.

Structuring: the code shall be well-structured, partitioned into modules and composed by functions implementing well defined tasks.

Traceability: the code shall be traceable with respect to the requirements. It shall be simple to identify the software functions implementing a specific requirement or requirements set.
Readability: the code shall be readable and comparable with hand-crafted code. Name of the variables shall be self-explaining, expressions shall be simple, indentation shall be appropriate.

Testability: the code shall be testable. Access to every function shall be enabled just through well-defined interfaces.

SW Coding Guidelines Compliance: the code shall conform to the GETS coding standard.

2.2.2 Code generation and evaluation

Source C code was generated from the SSC model using the Stateflow Coder R2007b tool. The choice of the code generator adopted in this phase was driven by the fact that only Stateflow had been used to specify the system logic. The generated code consisted of a single C file of 12 KLOC, with one interface function and a large set of global variables corresponding to the input and output of the Stateflow chart.

The integration of this code with the existing one required a surgical activity on the latter. It was required to identify the mapping between the global variables generated and the variables in the original code, and to define an adapter layer to allow proper interaction between drivers and logic. There was not much expectation that the experiment would succeed, due to the chaotic structure of the overall cut-and-paste artefact.

Functional tests were performed on the integrated system, and surprisingly, except for little bugs due to type and neutral values conversions in the adapter layer, the behavior of the logic was conforming to the functional requirements, and the 100ms threshold on the execution time was also respected. The first two expected properties were satisfied. On the other hand, the code lacked of the other five properties. With a single interface function implementing all the system behavior, it was not structured at all. Together with the
large presence of global variables, the absence of structure jeopardized the
testability of the code and influenced the traceability, since without distinct
functions, unit tests and apportionment of the requirements are prohibitive
tasks. The indentation resulted appropriate, and the name of the variables
corresponded to the names adopted in the model, but the readability was
complicated by the total absence of comments: what was made clear by the
graphical support in the model, appeared unintelligible once the graphics
disappeared.

Great part of the GETS coding constraints were violated. No dynamic allo-
cation was used, but many of the switch statements were lacking the default
label, and no defensive-programming control was adopted in case of opera-
tions that could lead to overflow.

The fact that, regardless of its substantial quality leaks, the code was working
properly, encouraged the prosecution of the research. The design department,
strongly suspicious about automatic code generation in the early phases of
the research, started to give valuable support to the activity.

2.2.3 Guidelines definition and refactoring

The objective was now obtaining a Stateflow model that was functionally
equivalent to the existing one, but showing the required quality properties.
The original model was too large to be suitable for an experimentation in-
volving loops of re-modelling and code generation. We decided to focus on
the properties violated by the model and we defined sub-models on which
experimentation could be performed more easily, observing the translation
of the single graphical constructs, and combination of them, to avoid the
violations experienced.

The activity led to the definition of a preliminary set of 25 guidelines for
The SSC BL1 and BL1 Plus projects creating models with Simulink/Stateflow targeted for code generation. The rules involved model structure, chart design and configuration of the code generator. Among them, we report those ones that had the highest impact in the refactoring of the SSC model.

RULE 1. *Each functional unit shall define input and output data and shall be implemented through a single Stateflow chart:* the main drawback of the original model was the monolithic structure. Having multiple charts communicating through input/output variables enables a better separation of concerns. At code generation level, this aids also unit testing: each Stateflow chart is translated into a single file with a unique interface function having the same input and outputs of the Stateflow chart from which it has been generated.

RULE 5. *Variables having machine-level scope are not admitted:* machine-level variables (i.e., variables defined at the level of the root Simulink diagram) are shared among different Stateflow charts belonging to the same Simulink model and they can be accessed by any component of the model. At code generation level, they are translated into global variables, which, besides being a normally not-recommended practice, are not allowed by the GETS coding standard.

RULE 8. *Implement sequential functions through Function graphical objects and not through state machines:* the repetitive structure of the generated code, made of sequences of nested switch/case statements, was due to the usage of state machines to implement any behavior of the system, including sequential functions that a developer would implement as if/else blocks. The Function graphical objects can be executed as actions by any state, and are actually translated into sequence of if/else blocks.

Following the first rule, the original Stateflow chart was separated into charts communicating through input/output variables. An overview of the transfor-
mation for a generic functionality $F$ is depicted in figure 2.2. The approach consisted in the following steps:

1. Identify, with the support of a domain expert, which were the functionalities that could be separated one from the other and define a chart for each one of them;

2. Identify the variables of the original model used by the functionality, and associate them to the chart;

3. Associate a scope to the variables of each chart: if the variable was used by the function only, it was declared local to the Stateflow chart; if the variable was only read by the function, it was defined as input for the Stateflow chart (i.e., non-modifiable); if the variable was written, it was defined as output for the chart (possibly read by other charts or directly passed the environment as system output).

2.2.4 Second evaluation

The final model consisted in 10 stateflow charts, one for each identified functionality. As expected, this time the quality properties resulted satisfied by the generated code. Though the amount of generated code, still around 12 KLOC, was more than twice the one of the original hand-crafted code of SSC, it resulted well structured, readable and traceable. When performing functional testing on the integrated system, it appeared that the execution time for the control cycle was still below the 100 ms threshold, but its average value was increased from less than 10ms for the original SSC model, to about 40ms. This increment, related to the substitution of the global variables with functions parameters, was at first considered negligible, since no hazard could anyway raise from an execution time that was still below the limit. The integration testing showed that this analysis was erroneous.
In SSC, when the first telegram is received, a lamp of the driver dashboard is lighted on. During the tests of the reception and telegram analysis mechanism, this lamp was lighted on with a delay of about 2 seconds. This intolerable delay in handling the message was at first inexplicable: the generated logic was analysing a message in one single cycle, and the cycle time was below 100ms. The problem was in the driver, that was passing the message to the software logic in packets of one byte for each main cycle, making the message completely available only after 39 cycles. In the original hand-crafted system and in the first model, this was leading to a delay of less than 500ms, while in the new auto-generated software, the cumulative runs were bringing to the visible delay of 2 seconds. In order to avoid the problem, the driver was changed so that the system could process a message in a single main cycle.
The solution solved a pre-existing problem, but the experience showed that the challenges related to the automatic code generation where not only limited to the quality of the code produced, but they were also involving performance issues.

2.2.5 Model evolution

In short time, the custumer started requiring further modifications to the SSC system, now called SSC Baseline 1 (SSC BL1), to distinguish it from its planned successor, SSC Baseline 2 (SSC BL2), which actually has never seen birth. New features were going to be added and old functionalities were going to be modified. Soon, the additional requirements brought to a completely new system, named SSC BL1 Plus. During this evolutionary process, the original model and the refactored one were both updated consistently, to assess the robustness of the latter to further changes. The new structure showed to be flexible enough to support the modifications, and the original model was abandoned after the requirements manager gained confidence with the new one. At that time, SSC BL1 Plus was still an hand-crafted system, since a V&V process for the generated code was not defined yet.
The chance to experiment code generation in the actual development process came in 2008, when GETS was involved in a bid of the Italian rail infrastructure company for the development of a new platform, named SSC Baseline 3 (SSC BL3). The system was deemed to integrate into a single on-board platform the functionality of SSC, whose deployment over the secondary lines was now completed, and SCMT (Sistema di Controllo Marcia Treni), the ATP system deployed over the main Italian lines.

The objective of the new SSC BL3 platform was allowing interoperability among the two systems, so that trains could pass from a secondary line equipped with SSC to a main line equipped with SCMT without having to manually switch the protection system or, worse, to change locomotive. The new system should have managed these transition areas, making the actual mode switching transparent and, at the same time, ensuring continuous train protection. SCMT is harmonised with the European Rail Traffic Management System (ERTMS) standard for implementing interoperability between
European rail networks: making interoperability with secondary lines would have allowed a full continuity within the overall network.

As SSC, SCMT is composed by a wayside sub-system and a carborne sub-system (see figure 3.1). The wayside sub-system consists of an encoder, that encodes the information to be sent to the train, and of so-called balises, that are antennas mounted between the rails. When the train passes over the balise, this is energized and sends the telegram formatted by the encoder to the carborne sub-system. This one performs the message interpretation, and accordingly provides train control and enforcement of the train speed.

Though the system is in principle similar to SSC, there are architectural and functional differences that make it much more complex. For example, each information point consists of two, three or four balises, placed about three meters one from the other, replicating the same content to increase the availability of the system. Compared to SSC, where there is no duplication of the information, this redundant configuration requires higher processing power to acquire and evaluate messages that come within shorter time intervals. Furthermore, the SCMT system includes also another train protection system, named RSC (Ripetizione dei Segnali in Continua, continuous cab signalling), that controls the train speed according to the track circuit codes.

Besides the intermittent information coming from the balises, the system acquires also the continuous information of the track circuits, and shall protect the train taking into account both data. As a last indicator of the size of the SCMT system, consider that each telegram is 1024 bit long, while SSC messages are 152 bit long. This difference, even if partly influenced by the more compact information representation in SSC, confirms that the data to be managed by SCMT, and the number of functionalities implemented to process these data, are by far larger than those of SSC.

Despite the differences, from the driver point of view the final system should
have appeared like a single one. The two electronic panels for driver interaction of SSC and SCMT, originally equipped with mechanical buttons, were going to be merged in a single touch-screen panel. Overlapping human machine interface functions, such as for example the brake rearm one, provided by both SSC and SCMT, would have been implemented in a single touch-button. Driver interaction procedures previously belonging to SCMT only, such as for example the one for requiring confirmation for the release of the traction cut-off after a train trip, would have been exported also to SSC.

Of course the in-field deployment is not perfectly partitioned into SSC-only and SCMT-only areas, but there are some overlapping zone, where the two systems have to coexist. Very often also the RSC system is involved in the transitions, and there are areas equipped with both SSC and RSC, but without SCMT balises. The complexity of the in-field configurations required the definition of an integrated software, since managing all these situations would have been prohibitive using hardware only. This software would have been auto-generated for the behavioral logic part, while hand-crafted code would have been used for operating system and drivers.

A new, more powerful, hardware platform was to be defined, since, besides the fact that the new software was estimated to be about 10 times big-
ger than the original SSC BL1 Plus, the experiments with code generation had shown that the performance could decrease with respect to hand-crafted code. Still to address the new performance requirements, an auxiliary DSP CPU, with optimized hand-crafted code only, would have been dedicated to the first-stage reception of the messages coming from the balises.

3.1 Software development principles

Before the start of the new platform development, GETS had already the SSC BL1 Plus hardware/software and a model of its logic refactored for code generation. Assumed that the code generation technology was considered the only possible choice to complete the project in reasonable time to participate to the bid, there were two possibilities at software level: building a model of the logic of the larger SCMT system and integrating the SSC logic into this one, or choosing to keep the two models separated and defining a third model to manage operating modes switching in transition areas. This second option was the preferred one. The idea was to keep the two logics as independent as possible, delegating the complexities of the integration to a separate component, that was baptized SSC/SCMT Manager. This was going to implement also those functions that were shared among the two systems, such as panel control, braking management and error management. Though initially considered valuable, the possibility of using the already tested code of SSC BL1 Plus together with the auto-generated code of the rest of the system was soon abandoned. The chance to exercise the overall system logic with Simulink, using all the visual debugging capabilities of the tool, was considered worth loosing the verification activities performed on the SSC BL1 Plus deployed system.

With SSC BL1, requirements were defined together with the model, without the target of code generation. Then, the model was refactored and incremen-
tally modified according to the new requirements to assume the SSC BL1 Plus form. SCMT introduced a new development scenario: a set of requirements (already defined by the customer for a code-oriented implementation) to be formalized with a model.

The SCMT on-board system specification consists of 20 documents and about 1200 requirements. Luckily, these requirements show a structure that was facilitating a representation in Simulink/Stateflow consistent with the modeling guidelines defined during the previous research. Each system function was already associated with a well-defined set of module-level requirements, that could be apportioned to a single Stateflow chart. Furthermore, the relation of each function with the other ones of the system was represented through Data Flow Diagrams (DFD), precisely defining the input and the output of each module. Figure 3.2 reports the context-level DFD of the SCMT system, as it appears in the specification [96]. The interface data, labeling the diagrams transitions, could be easily mapped into input/output structures for the Stateflow charts. As one could infer, such requirements were really detailed, and, though facilitating the module-level implementation, they were not giving any support in understanding what the overall platform was supposed to do at system level. The implementation of the models gave a consistent help in this direction. The role of part of the requirements appeared clear only after animating the interaction among the modules in the Simulink environment.

An additional, further development scenario was given by the SSC/SCMT Manager: a model conforming to the modeling guidelines to be defined together with the requirements. Following the successful prototyping experience of SSC BL1, it was decided to use a single Stateflow chart for its modeling, with the idea of possibly refactoring it in case its functionalities would appear too extensive to be contained in a single module.
Software development principles

Figure 3.2. Context-level DFD of the SCMT system
3.2 Guidelines formalization

Unlike the previous experiments, this time the generated code would have been used as part of a SIL-4 product, and new issues came concerning the guidelines defined during the experience with SSC BL1 Plus. This set of rules, though valuable, was not considered exhaustive, and, in order to let the assessor accept the technology, a more systematic study should have been performed. A comparison with the experience of other safety-critical domains was required and the internal guidelines should have been bound to an assessed framework of development rules. It was required a set of accepted modelling rules equivalent to the MISRA ones for C code, in which the internally defined guidelines could be integrated. Furthermore, one has to consider that part of the models were going to be outsourced, and therefore a more strict and unambiguous set was needed, to avoid misunderstandings with the supplier.

The automotive sector, that embraced code generation long time before the railway one, had already experienced issues similar to the ones GETS was now facing. The MAAB (MathWorks Automotive Advisory Board) Control Algorithm Modeling Guidelines [74] is a well established set of publicly available rules for modelling with Simulink/Stateflow. This set of recommendations has been developed by a group of OEMs and suppliers of the automotive sector with the objective of enforcing and easing the usage of the MathWorks tools within the automotive industry. The guidelines have been published in 2001 and afterwards revisited in 2007 in order to integrate some additional rules developed by the Japanese division of MAAB [63].

The scope of the current edition of the guidelines ranges from model maintainability and readability to code generation issues. The rules are conceived as a reference baseline and therefore they need to be tailored to comply with the characteristics of each industrial context. Customization of these recommendations has been performed for the automotive control systems domain in order to enforce code generation with the TargetLink code gen-
erator [32]. The MAAB guidelines have been found profitable also in the aerospace/avionics sector [81] and they have been adopted by the MathWorks Aerospace Leadership Council (MALC), that involves the most important defense and aerospace companies.

The MAAB guidelines have been developed to address the need of the automotive industries for a common language in model-based development using the MathWorks tools. GETS decided to adopt these rules as a baseline for defining its own modelling standard according to the needs of the railway signaling systems domain. The set of guidelines defined during the SSC BL1 project were going to be re-evaluated and integrated in the MAAB framework.

Railway signaling systems software, and in particular the code of Automatic Train Protection (ATP) systems, is characterized by the extensive usage of control modes logic and message analysis algorithms. These kind of features can be easily modeled through Stateflow state machines, while Simulink blocks are more suitable for numerical algorithms [74]. During the SSC BL1 development, this observation had suggested the possibility of adopting Stateflow as the only tool for modelling the software, and use Simulink as a framework for allowing Stateflow Charts integration and to simulate the external environment. This approach was adopted also for the new project.

The MAAB guidelines are mainly composed by Naming Conventions, Model Architecture, Simulink and Stateflow recommendations. The same structure has been followed by GETS while defining its own guidelines. The majority of the work has been focused on Model Architecture and Stateflow rules. GETS already provided its own Naming Conventions for software and these conventions have been translated into modelling rules according to specific analysis of the generated code. Since only Stateflow blocks were used for
the purpose of code generation, only MAAB Simulink rules dealing with model structure have been adopted and included in the Model Architecture rules set. MAAB recommendations for Model Architecture and Stateflow have been analyzed and enhanced to produce the GETS modelling standard adopted for the project.

3.2.1 Model architecture guidelines

Guidelines on Model Architecture concern the hierarchical development of Simulink/Stateflow models. The MAAB recommendations belonging to this set are quite general and do not give much support in developing the architecture of a system and moreover do not cover issues related to code generation. More detailed architectural guidelines have been developed to fit the railway signaling domain and address the code generation problem [42]. The most important ones are reported below.

RULE GE_A_1 - View Partitioning. The system shall be partitioned into three hierarchical view: Context View, Architecture View and Design View. The Context View describes the interaction with external systems. It is implemented through Simulink and it is linked to the Architecture View through a Model Reference block. The Architecture View describes the interaction between functional units. It is implemented in a Simulink model and it is constituted by interacting Stateflow Charts. The Design View represents the single functional units and it is implemented through Stateflow.

RULE GE_A_2 - Context. The overall system shall define its Context View in terms of just variables and buses. No direct connection with custom driver code is allowed at any level of the model.

This rule enables the possibility of generating the code from the system and afterwards connecting the input variables with the sensor drivers (e.g., odometer, operator dashboard) and the output variables with the
actuators drivers (e.g., braking system, operator display), facilitating the integration phase.

RULE GE_A_3 - Design View. Each functional unit shall define input and output data and shall be implemented through a single Stateflow Chart. This rule helps concurrent development, since developers can implement Stateflow Charts separately, once the interface with the other components is given. At code generation level this enables easy unit testing: each Stateflow Chart is translated into a single file with a unique interface function having the same input and outputs of the Stateflow chart from which it has been generated.

Note that the last rule reported here was already identified during the SSC BL1 project, but with BL3 it is integrated in a more formal and strict development framework.

3.2.2 Stateflow guidelines

MAAB recommendations given for Stateflow modelling concern Stateflow constructs usage and patterns implementation. Each one of these rules is rated with a priority label (Recommended (R), Strongly Recommended (SR) and Mandatory (M)), issuing the level of importance of the recommendation. Even though the guidelines are rather detailed, they need some priority restriction and modification to be suitable for the railway signaling context. Table 3.1 summarizes the rules that have been selected from the Stateflow recommendation set of MAAB and the priority restrictions or possible modifications issued.

Priority restrictions are in general given for those rules having some impact on code generation. For example, rule db_0125 states that every local data in a Stateflow Chart shall be defined at Chart level and not at Machine level. Since every local data that is defined at Machine level is generated as global
<table>
<thead>
<tr>
<th>Rule ID</th>
<th>Title</th>
<th>Priority</th>
<th>P. Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_0129</td>
<td>Stateflow transition appearance</td>
<td>SR</td>
<td>Modified</td>
</tr>
<tr>
<td>db_0133</td>
<td>Use of patterns for Flowcharts</td>
<td>SR</td>
<td>M</td>
</tr>
<tr>
<td>db_0132</td>
<td>Transitions in Flowcharts</td>
<td>SR</td>
<td>M</td>
</tr>
<tr>
<td>jc_501</td>
<td>Format of entries in a State block</td>
<td>R</td>
<td>SR</td>
</tr>
<tr>
<td>jc_0521</td>
<td>Use of the return value from graphical functions</td>
<td>R</td>
<td>SR</td>
</tr>
<tr>
<td>na_0001</td>
<td>Bitwise Stateflow operators</td>
<td>SR</td>
<td>M</td>
</tr>
<tr>
<td>jc_0451</td>
<td>Use of unary minus on unsigned integers in Stateflow</td>
<td>R</td>
<td>M</td>
</tr>
<tr>
<td>na_0013</td>
<td>Comparison operation in Stateflow</td>
<td>R</td>
<td>M</td>
</tr>
<tr>
<td>db_0122</td>
<td>Stateflow and Simulink interface signals and parameters</td>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>db_0125</td>
<td>Scope of internal signals and local auxiliary variables</td>
<td>SR</td>
<td>M</td>
</tr>
<tr>
<td>jm_0011</td>
<td>Pointers in Stateflow</td>
<td>R</td>
<td>M</td>
</tr>
<tr>
<td>db_0151</td>
<td>State machine patterns for transition actions</td>
<td>SR</td>
<td>Modified</td>
</tr>
<tr>
<td>db_0148</td>
<td>Flowchart patterns for conditions</td>
<td>SR</td>
<td>M</td>
</tr>
<tr>
<td>db_0149</td>
<td>Flowchart patterns for condition actions</td>
<td>SR</td>
<td>M</td>
</tr>
<tr>
<td>db_0134</td>
<td>Flowchart patterns for If constructs</td>
<td>SR</td>
<td>M</td>
</tr>
<tr>
<td>db_0159</td>
<td>Flowchart patterns for case constructs</td>
<td>SR</td>
<td>M</td>
</tr>
<tr>
<td>db_0135</td>
<td>Flowchart patterns for loop constructs</td>
<td>R</td>
<td>M</td>
</tr>
</tbody>
</table>

Table 3.1. MAAB Guidelines Restrictions
data by Stateflow Coder, this rule has to be set as mandatory: global data are forbidden by the GETS software coding standard. Another example is the rule jc_0451 which regulates the usage of unary minus on unsigned integers. Unary minus shall be forbidden for unsigned integers to avoid possible overflow errors even if no warning is issued by the code generator.

Some rules needed to be replaced or modified when considered in the GETS context. For example rule db_0132 states that transitions entering states are allowed if they are directed to sub-states. This is not permitted by GETS modeling rules, since allowing a transition between states at different hierarchical level reduces the separation of concerns and moreover the generated code results in a switch/case statement without a default condition, violating one of the pre-existing GETS coding rules. Other GETS specific guidelines have been added to address the need for GETS coding rules compliance, and the final set consisted in 43 modeling rules. These additional recommendations range from limitation in the number of states for each Stateflow Chart to restriction in function objects usage. Examples of additional rules are reported below.

RULE GE_S_01 - States and Junctions. States and junctions shall not be used jointly

RULE GE_S_02 - Path Connections. Avoid path connections in function objects

RULE GE_S_03 - Maximum Number of States. Each state machine shall be composed of maximum 10 states having the same hierarchical level

Violation of the first two rules generates code with additional local variables with names depending on the code generator, decreasing the readability degree of the software. The third rule is used to limit the maximum number of case statements for each switch/case block.
3.3 Guidelines verification

During the integration of the internal guidelines in the MAAB framework, GETS started also to investigate the possibility of automatically verifying the compliance of the models to the rules. This activity is basically equivalent to the static analysis that is normally performed on safety-critical code, using tools such as QA-C [92] to check programming standard adherence. Model-driven development practices with strict modeling guidelines are a recent technology, and the number of available industrial tool that perform static analysis on models is rather limited. A survey was performed by GETS to analyse the state-of-the-art, and evaluate which tool could be eligible to be integrated in the development process.

From the survey, it resulted that the approaches adopted by the available tools for guidelines verification on Simulink/Stateflow models are two: M-script based and meta-model based. The tools implementing the first approach use the M-Scripts, which are functions written in the Matlab programming language and interpreted by the Matlab environment, to verify the model properties. The M-Script invokes Matlab interface functions to retrieve informations about the models, and then it performs cross-comparison of the retrieved information with reference values, to check whether a given property is satisfied or not.

The meta-model based approach uses a meta-model of the Simulink/Stateflow diagram to work at a higher level of abstraction with respect to the one offered by the M-Scripts. A first variant of the approach is the one used by MATE [104] [105], that enables the analysis and modification of the actual model through pattern-search operations and refactoring on a visual meta-model. A second variant is the one used by the OSLO tool [84], that provides the possibility of defining the modeling guidelines through a formal language (e.g., O.C.L., Object Constraint Language) and verify them on the meta-model.
A preliminary analysis showed that the meta-model based approach was still at embryonic stage, with the MATE development temporarily suspended, while M-script based tools already had commercial representatives used at industrial level. Hence, the analysis was focused on this second type of tools, and three products have been considered for possible adoption, namely Simulink Model Advisor (SMA) [71], Model Examiner (MXAM) [76] and MINT [70]. The first one is distributed by the Mathworks, and it is fully integrated in Matlab, while the other two are third-party products. The evaluation of the tools has been performed according to their usability, number and type of built-in rules and other structural and performance-related properties. The results of the analysis are summarized by table 3.2.

<table>
<thead>
<tr>
<th>Property</th>
<th>SMA</th>
<th>MXAM</th>
<th>MINT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usability</td>
<td>−</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>Number of built-in rules</td>
<td>−</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>Extensibility</td>
<td>+</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>Matlab integration</td>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>Accuracy</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Report flexibility</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>Effectiveness</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 3.2. Evaluation of the style-checking tools (+ positive evaluation, - negative evaluation)

SMA, though having quite a poor user interface, has the advantage of being part of Matlab itself. This enables a more fluent workflow during the verification. For example, it gives the user the possibility of visualizing the exact point of the model where the violation to a certain rule occurred, a feature that was not supported by the other tools. On the other hand, the built-in rules concern Simulink models only, and Stateflow, which was the focus of the SSC BL3 models, is only marginally considered.

MXAM is the most mature of the tools in terms of usability and available
Guidelines verification

functionalities. Its GUI is partitioned into three panels that respect the different phases of usage: configuration (i.e., rules selection), execution and report. It collects guidelines from three different public sets: MAAB, MISRA AC TL [78] and dSpace TargetLink [32]. However it does not include the desired group of MAAB rules, and it is rather hard to extend with new guidelines, since these have to be implemented according to a strict format that narrow the flexibility of the rules implementation. This drawback affects also the third tool, MINT, which is fully focused on the verification of MAAB rules and implements 13 out of 43 of the modeling guidelines belonging to the required set.

Though each tool had desirable properties, none of them was considered fully satisfying. In any case, one would have to extend the rules available, to integrate the GETS guidelines. The usage of modeling inside the company being still at its early stages, the tool was required to be extremely flexible, not to have to be linked to third-party assistance for possible extension. Furthermore, one has to consider the normative context: none of the tools was certified according to the CENELEC norm, and the full knowledge of the implementation details of a software, and of the development process adopted, is crucial in the verification tasks of a safety-critical system. For these reasons, it was decided to design and implement a new style-checking tool that could better satisfy the requirements of the company.

3.3.1 Model Verifier

The development of the software, baptized Model Verifier (MV) and written in the Java language, followed a strict V-based process made of requirements definition, UML architecture and design, and integration testing. To be conservative, we have chosen to follow the stream of the commercially available applications, that is, to develop the tool according to the M-script paradigm.
The software was supposed to interact with the Matlab environment, core for the verification of the guidelines through M-Script, hence it was decided to structure the application according to the front-end/back-end model. The front-end is the part of the software that directly interacts with the user, while the back-end collects all the components that process the front-end output. According to this architecture, the final application was composed by:

- a GUI allowing the user to configure parameters for the verification of the guidelines on the model. The GUI is the front-end of the application, and its output toward the back-end are the configuration parameters;

- a set of rules implemented through M-Script and executable by means of the Matlab environment, that is the back-end of the application.

Model Verifier invokes Matlab through JMI (Java Matlab Interface), produced and distributed by the MathWorks. Through this interface, it requires the execution of an M-Script that incorporates the implementation of a check. M-scripts are written in a C-like language, and have all the power of any high-level programming language. Figure 3.3 reports the implementation of the check for the ge.0032 guideline, asking each data to be set with an initial value. One can see that the verification of a rule boils down to a search in the objects contained in the model and on an evaluation of desired properties on these objects. The M-Script interacts with the rest of the environment simply returning an object called mResult, which embeds pointers to all the objects violating the guideline. Once the front-end GUI receives the pointers, it can display the violations directly on the models.

From the GUI point of view, the development was inspired to the three-task interface of MXAM, with a Configuration, Execution and Report panel, separating the phases of the verification, since this form was found rather
Seven rules belonging to the guidelines have been implemented in the form of M-Scripts, together with a simple template for defining new rules to extend the tool. Complete implementation of the rules and execution of unit tests, though planned, was postponed. This was influenced by the fact that the tool was developed in the context of a thesis, and its completion would have required some effort from the internal team, that was instead completely focused on the development of SSC BL3. We have to remember that GETS was involved in a bid for the development of the platform, and the result of the bid was strictly related to how fast the company would have built the overall system. In this scenario, further introduction of new technologies could have delayed the development. However, still in the context of the thesis, performance tests were executed to check that the new tool was giving cost
improvements with respect to a visual revision of the models. The results were actually encouraging, since, for a test set of eleven models, it resulted that the time required for a visual inspection was 40 hours, while the time employed by Model Verifier was less than 1 hour and a half.

### 3.4 Model-based testing

The introduction of model-based development with the SSC BL3 project made necessary the adoption of new technologies for unit testing. Developing executable models and testing code would have been senseless: verification has to be made at the same abstraction level used during the design. Before the adoption of code generation, GETS used to perform structural unit testing according to the all-paths\(^1\) coverage criterion. The tests were derived from the structure of the code, with the objective of covering all the possible paths of the function to be tested. This white-box testing approach, though systematic and easy to apply, was requiring very high costs without giving too much added value. With the introduction of the generated code, the size and complexity of the software took a rapid leap: the average number of paths for each module increased by three times (from an average of 119 to more than 380 paths). The increased complexity of the control flow made unfeasible the use of former methods based on structural testing. On the other hand, the code should have been verified anyway, since the Stateflow Coder translator was not certified according to the EN 50128 norm, and its output was not ensured to be free from additional or incorrect behaviors introduced by the generator.

In order to address the issue raised on the verification side, GETS decided to investigate the possibility of adopting the model-based testing (MBT) technology as part of the SSC BL3 process. In general, model-based testing

---

\(^1\)evaluated following boundary interior method [59]
Model-based testing makes use of the behavioral part of the models, normally state transition systems or finite state automata (as in our case) to produce tests for the Implementation Under Test (IUT), which is the actual software derived from the models. Test cases production can be both manual or automatic, and might be driven by requirements based test specifications or by structural coverage goals. Though the effort of building abstract models and derive tests is not negligible, recent studies have shown how this approach allows a greater effectiveness in terms of errors detection with respect to the traditional approaches based on deriving tests directly from the code [90].

The adoption of model-based testing should have addressed two goals at the same time: verification of the models and verification of the code. In the context of SSC BL3, GETS defined an approach to tackle the two issues, consisting in a model/code back-to-back (B2B) testing strategy [110] where both the model and the related generated code are tested using the same stimuli as inputs, and the numerical results obtained as output are checked for equivalence.

In order to assess that the specification is compliant with the expected behavior of the system, a test suite is manually derived for the single Stateflow chart, with a set of tests defined according to the requirement coverage criterion: for each unit requirement at least a unit test is provided in the form of an input data sequence, in order to verify, through simulation and observation of the output, that the requirement has been correctly translated into the Stateflow formalism.

For example, consider the following requirement of the SSC BL3 specification [97]:

**UC 13.49.** *From the initial state, the Red Crossing function shall return to the inactive state, if the current speed is higher than 30 km/h or a code different from AC is received from the RSC system*
The requirement has been formalized in the Red Crossing module, as represented by the snapshot of figure 3.4. In order to verify the correct modeling of the requirement, the state machine has to be exercised with a test scenario that first allows to reach the initial state (STATO_CMT_INIZIALE), and then drives the return to the inactive state (STATO_CMT_DISATTIVO) by issuing a speed higher than 30 km/h or a code value different from AC. Note that this is a kind of test scenario performed at unit level, and the success of the test depends on the correct behavior of the unit during subsequent time steps with variations on input data. It is rather different from a common unit test that one would perform on a hand-crafted C function, for which the verifier normally executes the function once with specific parameters, without considering its time-dependent behavior. In this sense one could evaluate our model unit tests as multiple hand-crafted unit tests with evolving input data sequences.

A tool called Test Observer [47] [39] has been developed to automatically translate the unit tests manually provided for the Stateflow model, into test scenarios for the generated code, which represents our IUT. The tool registers the test execution during the simulation in terms of input and output Simulink time-series (Simulink data objects made of \((time, value)\) tuples for each variable), and directly translates the time-series into given input/expected output matrices for the generated module. This consists of a single file having a unique interface function for each module. Another tool, called Test
Integrator, creates a main file embedding the registered given input, the expected output and the model generated code. For each test case, the tool produces an executable file that checks whether the current output equals the expected output for the whole input sequence. The code is also instrumented by means of the weaving technology to collect coverage data during the test. If the execution is performed without errors, it can be stated that, for the unit test, the generated code is consistent with the Stateflow specification.

3.4.1 Results

Model-based testing in the form presented has been put into practice on the first release of the SCMT and SSC/SCMT Manager models. SSC was initially left out from the unit testing activity, since its functionalities were going to change soon with the integration and with new requirements coming from the customer. 327 test scenarios were provided and 162 hours were employed for the overall B2B testing task. A cost comparison with the structural tests previously performed on the hand-crafted code is hardly feasible, since, as highlighted before, the types of tests are radically different and the features of the code as well. The number of tests provided for hand-crafted C code are normally much more (some thousands), and their unit cost is lower (12 minutes against half an hour). However, one has to consider that the model scenario are equivalent to test sequences embedding large sets of traditional unit tests. If one would have decided to practice traditional testing on the generated code, it would have needed more than 8000 tests to cover all the paths, with impracticable cost.

The new procedure supported by the modelling technology was perceived by the developers as much more effective in detecting bugs and exercise the behaviour of the system. In terms of tests, the models were giving several advantages compared with hand-crafted functions:
there were bigger modules, with clearly identified functionalities where system-level requirements could be easily traced and verified;

- the definition of test scenario on hand-crafted C code, equivalent to those performed on the models, would have required the isolation and integration of several C modules, with consequent cost increase;

- the models showed higher cohesion: since the interaction among functions was performed only through data exchange, there was no cross-calls among the modules, and a simpler control-flow. Previously, the cross dependency of hand-crafted code functions was often complicating the definition of unit-tests, since many stubs were required to isolate a module and perform proper tests.

No functional discrepancy was revealed by the re-execution of the tests on the code, and this outcome confirmed the validity of the adoption of the Stateflow Coder code generator. A comparison with the coverage obtained at code level showed that there was a slight mismatch with the model coverage, being the former always lower than the latter. A deeper analysis showed that the difference were caused by three main reason:

- the code generator introduces defensive programming constructs in case of possible overflow on embedded Matlab functions; the corresponding C code is expected not to be executed in case of proper functioning of the program;

- the guidelines adopted require each state machine to have a (unique) initial default transition. These transitions correspond to the default statement of the switch-case construct that represents the translation of the state machine: in case of proper functioning of the software, that portion of the code shall not be traversed;

- depending on the options adopted for code generation, it can happen that part of the instructions of the model are not translated as sepa-
rated C functions, but are expanded in-line multiple times inside the code to optimize the execution. Therefore it is acceptable that such code portions are partially exercised locally in each instance, since they are completely exercised if all instances are taken into account.

These considerations allowed to assess that the code generator was not introducing additional behaviours in the code, except for those ones concerning defensive programming or linked to optimizations of the code generator.

3.4.2 Evolution

The MBT strategy contributed to obtain a complete working model within few months. Afterwards the development scenario evolved in such a way that the role of unit tests lost its importance. The hardware, operating system and drivers were available for integration testing, and the company had the chance to perform the first test drives of the system. Integration tests were defined, but at the same time new functionalities were required, since the in-field experience was constantly showing new scenarios that were not taken into account by the original specifications. An hardware-in-the-loop (HIL) simulator was developed to perform integration testing, and to execute the in-field scenarios coming from the test drives. The development was now focused on the small adjustments on the models to address the new issues coming from the railway scenarios, and turned into a frantic ping-pong between the in-field engineers, testing new software releases on the train, and the development team, fixing the models and performing integration testing on the HIL simulator before issuing new releases. With these incremental modifications the system assumed its final operating form.

Though the platform was properly working and respecting the system requirements, its certification was requiring unit tests to be performed. Restarting the testing task at the end of the development was considered not to
Model-based testing
give so much added value: the integration tests successfully performed on the
HIL simulator were traced over all the original and new requirements, and
therefore the 100% requirements coverage objective was satisfied by the tests.
The evidence of complete structural coverage, that would have assessed that
the evolution did not introduce additional and not evaluated behaviours, was
nevertheless missing. The idea was to re-execute on the integrated models
the tests defined on the HIL simulator, and evaluate the structural coverage
at single chart level, defining new tests in case the coverage would result
incomplete. The metric adopted was decision coverage, since the norm is
requiring only the execution of all statements, and the approach was consid-
ered a fair enhancement compared to the norm prescriptions. In figure 3.5 is
depicted the procedure defined to address the goal, which can be summarized
as follows:

- translation of the tests defined for the HIL simulator into tests for the
  integrated Simulink model, in the form of Excel files;

- batch execution of all the tests on the model and computation of the
decision coverage through the Simulink Model Coverage tool;

- evaluation of the coverage: if the decision coverage for a single chart
  composing the overall model is below 100%, new tests are required for
  the module;

- definition of new tests: the not covered decision are visually inspected
  on the model and, if they can be stimulated at system level, a system
test shall be defined, otherwise a unit test shall be defined.

Note that evaluation of the coverage was performed only at model level, since
the previous data on coverage comparison had already assessed the equiva-
lence between model and code coverage.
3.5 Abstract interpretation

With the project, the company introduced also another verification technology named abstract interpretation [24] [25] [26]. Unit testing alone, whether model-based or code-based, cannot cover all the possible behaviors of the code in terms of control flow and data flow. Most notably, it lacks in detecting all those run-time errors that might occur only with particular data sets, such as division by zero and buffer overflow. This issue has actually remained unsolved, and normally stemmed through boundary analysis strategies, until the tools for static analysis by means of abstract interpretation resulted mature for industrial usage, that means only in the latest five years.
Abstract interpretation

Experimentation with abstract interpretation started right before the beginning of the project, using the Polyspace tool [30], distributed by the MathWorks. From an industrial perspective, having the same producers for several tools employed in the development process gives more confidence on their compatibility, and simplifies the interface with the tool providers. Furthermore, at that time, the tool was going to be certified according to the EN 50128 norm, and this also encouraged its adoption. Though the first experiments started in the context of a thesis, as for Model Verifier, in this case the fortune was different: once experienced the power of the tool and the evident benefits related to its usage, the company has forged ahead with the integration of the tool into the development process.

Polyspace analyses the C code and detects the statements that could produce errors during the execution of the code. The main runtime errors that Polyspace allows to identify are:

- access to not initialized variables (NIVL e NIV);
- access to not initialized pointers (NIP);
- illegal memory access through pointers (IDP);
- out-of-bound array access (OBAI);
- arithmetic overflow and underflow (OVFL, UVFL);
- non-terminating loop and non-terminating call (NTL e NTC).

The tool presents its results through chromatic marks on the analysed code:

- green, if the statement can never lead to a runtime error;
- orange, if the statement can produce an error under certain conditions;
- red, if the statement leads to a runtime error in every run;
• grey, if the statement is not reachable.

In order to perform static analysis on the code, the tools based on abstract interpretation techniques build an abstract domain that represents an over-approximation of the real domain. The abstraction process might bring to the generation of false positives during the verification: this behaviour is caused by errors raised in those runs which are allowed only in the extended domain, but not in the original one. For this reason, it is essential, for the adoption of this technique, to define a well-structured process that permits to reduce the cost of the analysis of false positives that represents the price to pay to obtain the exhaustive verification of the code behaviour.

In order to address the problem and, at the same time, to obtain a substantial improvement of the confidence on the correctness of the code, GETS decided to adopt a two step process (see figure 3.6). The first step is performed with a very large over-approximation set. The second one capitalizes the information obtained by the analysis of the previous one, and executes the verification with the use of a finer approximation set.

![Figure 3.6. Polyspace process overview](image)
3.5.1 Polyspace first step

The purpose of the first step is mainly to detect systematic runtime errors (red), that is, errors which arise in all the runs considered in the verification, and unreachable statements (grey). Examples of systematic runtime error are infinite loops, out of bound array accesses and usage of not initialized variables.

Although unreachable code might seem to be a low severity problem, in our experience grey marks are often indications of erroneous modelling of the specifications: a code block that is never executed might be the translation of an unreachable state in a Stateflow chart. Only in some limited cases, grey marks are related to additional defensive-programming instructions introduced by the translator to maintain control even in presence of completely unexpected input, which may be due to hardware or software faults. For example, the default statement in a switch/case block (which is the natural translation of a state-machine), is likely to be never executed, and therefore will always be marked with grey. Obviously, these grey marks do not harm the safety of the code because they represent collectors used to handle unexpected behaviours.

The first step is performed using all the possible over-approximations settings provided by Polyspace. Since these approximations are used in order to be sure that the analysed runs include all the actual runs, the results obtained are not selective enough. The large set of spurious runs that are analysed in the step leads to an outstanding number of orange checks.

For example, in figure 3.7 the statement highlighted by the arrow could raise a runtime error, in particular the function could access a memory location which is outside the bound of the array named buffer.

In the example we consider that the function is static and that in the analysed module there is a function call to the get_value procedure. In this condition Polyspace does not generate an automatic call to the function, but
Abstract interpretation

makes use of the existing call to verify the function code. Otherwise, the code reads a location of the array `buffer` indexed by the global variable `index`. In the first step, Polyspace automatically initializes all the global variables with full-range values. For this reason, when the tool analyses the statement highlighted in figure 3.7, it finds that for some values of the variable `index` there is an out of bound access to the array. The result suggests that on the values that the variable `index` can assume, narrower bounds have to be introduced in order to reduce orange marks.

In the example, Polyspace signals a possible erroneous behaviour on the array bracket, but the actual cause of the orange mark is the full range initialization of the variable `index`. It is on this variable that one has to work in order to avoid the warning. This situation is similar for any orange mark coming from the first step: in order to narrow the approximation for the subsequent step of Polyspace, each orange mark has to be related to the cause that produced it. The generic classes of causes that generate the orange marks might be related to the overapproximation issued by the tool or by usage of code constructs that are not managed by the tool itself. We have partitioned the possible causes into 6 classes:

- Full-interleaving: the tool automatically generates the function calls for the public procedures of the module under test if they are not invoked
by other functions defined in the same module. All the possible interleaving of the automatic function calls are analysed in the verification;

- Static variables initialization: the static variables defined in the module in every run are initialized with all the values of their type range (in the following we will refer to this kind of approximation as the full-range initialization);

- Global variables initialization: the global variables defined in the module are managed in the same way of static variables;

- Full-range parameters: the formal parameters of the function for which the tool generates the call are initialized at full-range;

- Absolute address: usage of absolute memory addresses for operations directly related to the system hardware;

- Initialization of local variables through pointers: if a variable whose scope is local to a function is initialized by an external function, the tool issues an orange mark.

An analyst with a minimum proficiency with the tool can easily evaluate the orange marks and quickly classify their causes.

In the case of the example, the analyst recognizes the orange mark on a bracket as referring to the global variable initialization setting, so can pinpoint the variable that has been initialized full range. Another case might be the one in which one module has two interface functions, the first to initialize static variables, and the second to actually perform the functionalities required to the module (this is actually the normal structure of the generated code). In the actual usage of the program, the initialization function will always be called before the other one. However, the tool will issue orange marks on all the static variables used by the execution function: due to the full-interleaving over-approximation, the tool assumes that the second function might be called before the initialization one, leaving the static variables
without an initial value. Also in this case, the analyst recognizes a bunch of oranges on static variables, and can associate them to the full-interleaving class. Then, he gives constraints to the tool concerning the order of execution of the functions.

As exemplified, the identified classes are used to define input constraints to be given to the tool to restrict the analysed abstract domain of the program. Sometimes, editing the configuration file that define the constraints might require advices from the developers, since the analyst is often not aware of the actual domains of the variables, or of the program context in which a certain function is used. However, we experienced that the analyst is much more independent if (s)he has to deal with the automatically generated code, since the repetitive structure of the software simplifies the review task.

3.5.2 Polyspace second step

The second Polyspace step, performed with the restrictive settings, allows a finer approximation of the real domain of the program and then a reduction of the number of false positives. At the end of this step, the remaining orange marks are due to the complex interactions between variables that cannot be constrained by simply introducing finer approximation bounds.

As an example, consider the code segment reported in figure 3.8, that describes a typical software procedure present in the railway signalling context: it deals with a train receiving messages from the car-borne equipment at every given distance, in proximity of a so called information point. Every time the train passes by the information point and receives a message, the code assigns the current value of the space, maintained in the variable current_space, to the variable last_msg_space. Once the train gets by the information point, it uses the procedure in figure 3.8 to compute the space covered from the last
message received. Polyspace produces an orange mark that signals the risk that the described statement could raise an underflow. This orange mark is reported also in the second step of the Polyspace verification, because in this case the constraint on the possible values assumed by the variables does not handle the particular bound that makes impossible the underflow.

According to our experience, the overall time employed for the configuration and set-up activities is 20% more than the time Polyspace takes to actually execute the two steps. The most time consuming task is the first review of the orange marks, that takes about the 48% of the overall time required for the whole process. Due to the low number of residual oranges after the second step (normally about 2.6% of the total), the cost of the second review is basically negligible. Nevertheless, one has to consider that the absolute overhead of the orange review is acceptable: about 5 minutes for each orange, in average. The generated code is characterized by a limited number of different classes of motivations for the orange marks, and this makes most of the review a rather systematic activity.

3.5.3 Results

Polyspace has been employed for the verification of both hand-crafted and generated SSC BL3 code. First, one module of the first group and one of the second group were evaluated, to identify how the structural differences
among the artifacts of the different code production approaches could have affected the implementation of the abstract interpretation task.

For the hand-crafted code, it was chosen to evaluate the driver module controlling the ethernet communication, considered a fair representative of its group. From the first, unconstrained, Polyspace step, resulted 158 oranges, classified in figure 3.9 according to the error issued by the statement. In figure 3.10 (a), the identified oranges have been classified according to the approximation classes that possibly produced them. Three of the four classes are involved, namely full range static/global variables, full range parameters and absolute address usage.

![Figure 3.9. Classification of orange marks according to the error (hand-crafted code)](image)

Figure 3.10 (b) presents a further refinement of the analysis: the 47 orange marks related to the full range static/global variables class depend on missing constraints over 10 variables, while the 25 oranges traced to the full range parameters class depend on the constraints over other 10 variables. The other 86 oranges, caused by absolute address usage, are not considered, since they depend on properties of the code that cannot be verified through the usage of the tool, related to the actual hardware in use.
The selection of the constraints to add in the second Polyspace step is supported by the classification of oranges obtained during the unconstrained analysis. In particular, it was required to identify the actual range of the input parameters and of the static/global variables, and giving the latter an initialization value.

As one can deduce from the chart, the classification effort and the second Polyspace phase seemed not to be so useful, since great part of the oranges were the ones caused by absolute addresses, and could not be disambiguated. The remaining oranges could have been checked by hand without executing the second step. This intuition was confirmed by the analysis performed on 99 hand-crafted modules of the BL3 system. As shown in figure 3.11, that collects representative results on 15 modules, the simple unconstrained Polyspace step resulted already quite effective. In the greatest part of the situations, more than 90% of the code was ensured to be free from runtime errors already after the first step. On the other hand, the few oranges were spread over all the possible causes, and this was making the classification effort rather useless. For this reason, in the context of SSC BL3, it was decided to perform only the first unconstrained step on the hand-crafted code, and provide evaluation of the residual oranges by hand.
During the Polyspace execution, two red errors were actually found on the hand-crafted code: one concerning the wrong usage of a variable that was not initialized yet, and the other caused by an illegal access to a memory location for a wrong array index. It has to be noticed that, while the first error could be detected also by common static analyzers, the second one would have required dynamic testing with boundary analysis, a task that was usually not performed by GETS.

Concerning the generated code, the module selected for the preliminary evaluation was the one managing all the functions related to the interaction of the SSC with the train driver. A number of 206 oranges was identified after the first step, 204 of which related to non-initialized values and only two to illegal pointer access. Compared with hand-crafted code, not only the number of error classes were reduced, but also the number of overapproximation classes: according to figure 3.12 (a), only the wrong interleaving of function call and
the full range parameters classes were involved. The further refinement of the analysis showed that the actual constraints required to disambiguate the oranges were only four (see figure 3.12 (b)). The first two concern the function interleaving: the sequence of the function calls should have been explicitly stated. The second two concern the initialization of input and output variables, that in the generated code are embedded in two separate structures required as input parameters by each function of the module. After the introduction of the constraints, the result of second step were really impressive. The whole code was green, without remaining orange marks to disambiguate. In this case the classification task and the constraint definition seemed to be worthy.

The two-step Polyspace analysis performed on the SSC software component showed that, after the first step, the average number of oranges was slightly higher compared with hand-crafted code (about 87% against more than 90%). However, as in the case of the module considered in the preliminary evaluation, the number of classes of causes was again limited to full-interleaving and full-range parameters initialization. For this reason, in the case of the generated code, it was decided to apply the complete two-step process.
Abstract interpretation

The experience with abstract interpretation evidenced advantages of generated code in terms of verification activities. Its simple and repetitive structure enables easier analysis, and the possibility of automating it. The orange marks are always caused by the same overapproximations and can be discarded in a systematic way: the two-step process applies effectively in these situations, and eases the actual error discovery. If it appears that an orange does not belong to the classes of causes common to the generated code, it is likely to be an error.

With hand-crafted code the developer is free to use complex and interwoveved design that might ease its job, but complicates the activity of the verifier. A systematic approach is hardly applicable when there is no uniformity in the structure, and if each orange mark has a different cause, it has to be evaluated by hand.

Another point is the interaction between developers and verifiers. Great part of the oranges normally issued can be eliminated by clearly identifying the range of the input variables of a module and configuring the tool to constrain the range of such variables. In the generated code it is easy for the verifier to identify the input and output data of a module, since they are always embedded in two separate structures passed as parameters to the functions. Therefore, before performing the second Polyspace step, he can ask the developer the proper constraints on the range of these parameters and then he can work independently. With hand-crafted code, the verifier has to inspect the code to understand which are the data used as input by a function (i.e., data that are only read) and those one that are output (i.e., data that are written). Furthermore, very often pointer to functions and void pointers are used in the hand-crafted code, and this really complicates the activity of defining constraints on the input parameters. Therefore, the interaction with the developers has necessarily to be tighter, bringing to higher verification costs. This analysis is reflected by the results of the SSC BL3 project:
the Polyspace analysis on the hand-crafted code required 298 hours, while
75 hours only were employed for analyzing the generated one (the overall
amount of hand-crafted and generated code was basically the same, if one
considers the hand-crafted code of the auxiliary unit).

As a final indicator of the generated code advantages, one has to consider
that no red error was issued by this one, while two errors appeared in the
hand-crafted code. This was seen as a confirmation that performing software
simulation before deployment was actually ensuring early error detection.
The limit of this evaluation is that the two code production approaches were
employed to implement different component of the system, having distinct
roles. Hand-crafted software was used for the operating system and the
drivers. In these cases, performance issues require usage of hardly verifiable
constructs such has void pointers or shared variables. Furthermore, access to
absolute addresses, whose correctness can not be verified with Polyspace, is
inherently required by the system drivers. Generated code was used for the
logic part, that has less strict performance constraints, and, being indepen-
dent from the hardware, does not use absolute addresses. On the other hand,
the different employment counterbalances the verification cost discrepancies:
operating system and drivers are stable, while the behavioral logic changes
for any products. It is therefore acceptable to have higher verification cost
for the first and lower for the latter.

3.6 Results

The final version of SSC BL3 consisted in about 30 KLOC of hand-crafted
operating system and driver code and 62 KLOC of generated code, partitioned
between the three software components as reported in table 3.3. It
has to be noticed that the SSC/SCMT Manager, originally consisting in one
module only, has been progressively split into five modules reaching almost
half the size of the rest of the system. This is in line with the development story reported and with the initial development choices taken: while test-drives where issuing new transition scenarios, all the logic modifications were applied to the SSC/SCMT Manager, since it was decided to keep the two system completely decoupled. Looking backward and focusing on the size of SSC compared with the one of the SSC/SCMT Manager, it would have been probably a better choice to integrate SSC inside SCMT.

<table>
<thead>
<tr>
<th>Component</th>
<th>Modules</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSC/SCMT Manager</td>
<td>5</td>
<td>16479</td>
</tr>
<tr>
<td>SCMT</td>
<td>20</td>
<td>37038</td>
</tr>
<tr>
<td>SSC</td>
<td>12</td>
<td>8270</td>
</tr>
</tbody>
</table>

Table 3.3. LOC for the SSC BL3 generated software components

Despite alternative architectural decisions that could have been taken, it is out of doubt that the adoption of modelling and code generation technologies was the core choice that allowed the company to complete the development of the first version of the platform within one year, and win the tender for the development of SSC BL3. In this phase, the verification approach put into practice for the first version had a crucial role: the integration of model-based testing and abstract interpretation showed to have reduced the cost of unit-level verification activities with respect to the previous technique based on all-paths structural testing. The results of table 4.3 show that the new approach reduces the verification cost of 70%, even with code having a higher complexity in terms of path number. At the same time we have obtained a verification accuracy that cannot be achieved with traditional testing.

<table>
<thead>
<tr>
<th>Verification Process</th>
<th>Modules</th>
<th>Paths</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural Testing</td>
<td>19</td>
<td>2274</td>
<td>728</td>
</tr>
<tr>
<td>MBT + Abs. Int.</td>
<td>21</td>
<td>&gt;8000</td>
<td>227 (162 + 75)</td>
</tr>
</tbody>
</table>

Table 3.4. Comparison of verification cost
Besides the advantages given by the simulation capabilities, that enforced the confidence on the logical soundness of the system before deployment, the introduction of modelling made possible the partial outsourcing of the development. In a large company, personnel recruitment requires a long process, that hardly combines with the time-to-market requirement, and, as in the case of SSC BL3, outsourcing is the only way to get the things done in short time. At the same time, it is often hard to define the interfaces between the outsourced parts of the system and the internally developed components. With modelling, this interface definition process is simplified by the inherent separation of concerns of the Simulink/Stateflow platform, that allows to partition software components in blocks having formally defined interface, decoupling in particular the logic part of the system from the operating system and drivers. The outsourcing was facilitated also by the development tool-suite adopted: though the practice with Simulink/Stateflow was at its initial stage within the company, this is a de-facto standard in the development of embedded systems, and it is therefore easy to find a supplier with some experience with the tool-suite. Integration of the internal guidelines with the public MAAB guidelines also had a positive influence in defining a common language with the supplier.

Though enabling a faster and more formal development, the introduction of model-based design posed challenges on the verification and certification-related activities. Many difficulties were found in defining a feasible verification process, that could be accepted by the certification authority. Further complications came with the internal re-organization of the company, that was asking the development team to perform the verification activities with an independent internal group. The tasks planned by the V&V team at the beginning of the SSC BL3 project, where now in the hands of people with a development-oriented background, that associated lower priority to verifi-
Results

cation activities like model-based testing and modelling style-checking. This was one of the reasons why the MBT approach was not further applied in the project after the first version, and the Model Verifier tool had no current usage.

The main problem related to certification of the product has been raised by the code generator. Since it was not qualified according to the EN 50128 norm, the assessment authorities require at least a proven-in-use translator. This property, besides activities of bug-reporting and strict version control, that are carried by the tool distributor, requires the successful application of the tool to several projects, and SSC BL3 was the first project where Stateflow Coder was applied by GETS. On the other hand, SSC BL3 had about one year and a half of operation, and six stable releases of the software installed and exercised on the train. At the end of July 2010, 316 locomotives were equipped with the platform. The idea was therefore to use this experience to qualify the tool in the context of the project only, establishing the proven-in-use property only for SSC BL3: the different versions can be seen as subsequent projects where the code generator was successfully employed. This strategy has been considered acceptable but not sufficient by the assessors, and further activities are currently required, concerning precisely what was left out during the development, namely guidelines verification and model-based unit testing.

Though the time required by the certification task is consuming the cost savings given by the model-based design, we have to remark that this is the first project where the new approach was applied: the lessons learned during the assessment, and the progressive tailoring of the process required costs that will have their return in future projects.
Chapter 4

The Metrô Rio Project

4.1 Project story

At the end of 2008, GETS was commissioned for the adaptation of its SSC ATP platform to the metro of Rio de Janeiro, managed by the Metrô Rio company. This was the first time that GETS had to deal with a system for underground transportation, but the preliminary analysis of the requirements showed that the issues of this domain were not different from the ones of railway signalling. Though the main ATP principles were not changing from trains to metro, there were differences that had to be tackled and proper customizing of the product was required. Compared to trains, the normal speed of metros is lower, stops are more frequent and signalling is simpler. Furthermore, the customer was requiring additional functionalities not provided by the original SSC system. For example, one of the requirements was related to the rain control function, a feature which is particular to the geo-
graphical context of the application. Since Rio de Janeiro is a tropical area, it encounters long periods of torrential rains. The metro line is mostly exposed to the elements, while platforms are mostly covered. During the rainy season the train is subject to notable slip/slide phenomena under braking. For this reason, in the case of rain, the system shall protect to a more restrictive braking curve when approaching a platform to minimize spin/slide effects by reducing the braking effort that is requested. The presence of rain shall be signalled by the driver through an appropriate button on the touch-screen panel and the system shall react accordingly combining this “rainy” state with the information received from the information points.

The new project, baptized SSC Metrô Rio, started during the first verification stages of SSC BL3, when the modeling and code generation technologies were already showing their advantages in terms of cost reduction and product safety, and were considered already part of the established development process. In principle, it would have been possible to adapt the SSC BL1 Plus component of SSC BL3 to the new platform. However, the model was highly focused on the Italian signaling approach and was coming from several modifications required by the integration in the SSC BL3 system. Though modeling had shown its effectiveness in supporting frequent requirements changes, each adjustment unavoidably brings to an artifact that is less generic and therefore less adaptable. For these reasons it was decided to define new requirements and a new model from scratch. The idea was using the previous experience to create a model for SSC Metrô Rio that was generic enough to allow its components to be adapted to different metro applications. A sort of root model, from which new instances could be created according to possible customer needs.

Concerning code generation, the company adopted the more customizable RTW Embedded Coder in place of Stateflow Coder. During SSC BL3 develop-
Development, the company had performed experiments with this tool, that was initially not adopted because the code produced was perceived as too complex compared to the one of Stateflow Coder. Further analysis showed that well-structured and readable code could be produced with a proper tailoring of the code generation options provided by the tool. Concerning the modeling guidelines, the transition was basically painless, since all the modeling rules developed for the previous tool resulted in being applicable also for the new one. RTW Embedded Coder allows the generation of glue code for the integration of the modules, while with Stateflow Coder one has to perform this operation by hand: though initially seen as a minor disadvantage, it became then evident how this task was actually really ticklish, especially when new updates are frequent. The adoption of the new code generator allowed a further speed-up of the development, since the integration of the generated code into the system was now only a matter of transferring files on the target machine project.

4.2 Development process

The project provided the opportunity to consolidate the model-based development process and to refine the modeling activity toward a more formal approach. While with SSC BL3 many issues related to the verification and certification of the product were encountered during the actual development, and required ad-hoc solutions, with SSC Metrô Rio a structured, complete process was defined since the beginning. Fig. 4.1 summarizes the overall process structure. Embedded in 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. According to this decomposition, system requirements are partitioned into mutually exclusive sets of unit re-
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. As for the previous projects, 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. In SSC BL3 already an extension of the MAAB guidelines was used, aimed at enhancing the readability, maintainability and structuring of the code, while with SSC Metrô Rio additional restrictions have been introduced to further constrain the Stateflow semantics to an unambiguous set of constructs. Unit tests have been defined in the form of scenarios at Stateflow model level, using an enhanced version of the B2B testing framework that automatically executes the test suite on the Stateflow automaton and on the generated code to ensure functional coherence between model and software behaviour. The confidence on the correctness of the generated code is increased with the PolySpace tool for abstract interpretation, that completes the unit-level verification activities. Finally, system tests are performed on an ad-hoc train simulator with
4.3 Hierarchical architecture definition

When a large requirements set is involved in formal modeling, a well defined architecture of the model can help in clarifying which are 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.

![Figure 4.2. Simplified component diagram](image-url)
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 [40] (a simplified version is represented in figure 4.2). 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 the internal structure, a centralized software architecture has been chosen, with the Operation Mode Manager enabling/disabling all the other components according to the current state of the system, and managing downgraded and faulty situations. The Information Point Manager and the Human Machine Interface (HMI) Manager are the components taking care of controlling the system state according to the information coming from the external interfaces. When new telegrams are received they are first processed by the Information Point Manager to ensure data consistency, and afterwards forwarded to the internal components. The HMI Manager implements all the functionalities related to the interaction with the driver, controlling the touchscreen according to the state of the other functional units. A set of Speed Analyzer components is defined which process the information coming from the Information Point Manager and translate them into different authorized speed limits, target speed and distances. All these data are collected by the Speed Limit Manager, computing the braking curves of the different targets and determining the current speed limit. Control components are in charge of modifying the behavior of the Speed Analyzers according to particular information coming from both the HMI Manager and the Information Point Manager. In case of dangerous behavior acted by the driver (i.e., speed
limit violation) or system fault, the Brake Manager is in charge of commanding brakes according to the controls coming from the other functional units.

Figure 4.3. The multiple level hierarchical model

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 figure 4.3), according to the three different views identified during the SSC BL3 project, and defined in the model architecture guidelines set. The different levels are intended for different development stages, from a more abstract to a more detailed view.

**SSC Metrô Rio Context**: the 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 con-
Hierarchical architecture definition

considering the boundary ports and mapping them into signals entering or exiting the Simulink blocks. This approach allowed us defining 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 data, touch-screen buttons, telegram data, etc.), to perform interactive testing of the model.

**SSC Metrô Rio Architecture:** the second level represents the internal software architecture in terms of interacting functional units modeled through Stateflow charts. For each one of the components of the original UML component 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.

**SSC Metrô Rio Design:** the third level is actually the design level of the single Stateflow charts, each of them 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 mutually exclusive sets of unit requirements, to identify the requirements apportioned to each single Stateflow chart. For example, consider the system functional requirement concerning the control over the unauthorized passing of a red signal (normally called *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 4.1. 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, that enforces any brake condition, and the fifth requirement is apportioned to the HMI Manager, controlling the interaction with the driver.

Figure 4.4 shows the formal representation of the ICO_TT state machine, modeling the fifth requirement of table 4.1, together with the corresponding generated code.

### Table 4.1. Unit requirements decomposition (train trip function)

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

4.4 Modeling guidelines update

Stateflow implements a variant of Harel’s hierarchical statecharts [52], normally called charts according to the Stateflow taxonomy. The complex se-
semantics of Stateflow is not formally based, though research has been performed to define an operational semantics [50] and a denotational semantics [51] 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 with a set of rules oriented to restrict the use of the Stateflow language to a semantically unambiguous subset. With reference to the Stateflow language notation defined in [50], in table 4.2 we represent the subset of Stateflow adopted for the SSC Metrô Rio project that has been identified following the approach shown by Scaife et al. [98] 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 [98], use of events implies generation of recursive code that might lead to the risk of an infinite recursion call, stack overflow or anyway to state-space explosion problems. For this reason events are forbidden by the adopted modeling guidelines and they are simulated through variable
<table>
<thead>
<tr>
<th>Original set</th>
<th>Restricted subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>state $s$</td>
<td>state $s$</td>
</tr>
<tr>
<td>active state $s_a = \emptyset</td>
<td>s$</td>
</tr>
<tr>
<td>junction $j$</td>
<td>junction $j$</td>
</tr>
<tr>
<td>path $p = \emptyset</td>
<td>s.p$</td>
</tr>
<tr>
<td>event $e$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>destination $d = p</td>
<td>j$</td>
</tr>
<tr>
<td>action $a$</td>
<td>action sequence $a = \emptyset</td>
</tr>
<tr>
<td>condition $c$</td>
<td>condition $c$</td>
</tr>
<tr>
<td>transition $t = (e_t, c, a, a, d)$</td>
<td>state transition $t = (e, c, a, p), c \neq \emptyset, p \neq \emptyset$</td>
</tr>
<tr>
<td>transition list $T = \emptyset</td>
<td>T</td>
</tr>
<tr>
<td>composition</td>
<td>composition</td>
</tr>
<tr>
<td>$C = OR(s_a, p, T, SD)</td>
<td>AN D(boolean, SD)$</td>
</tr>
<tr>
<td>state definition $sd = ((a, a, a), C, T_i, T_o, J)$</td>
<td>state definition $sd = ((a, a, a), C, T_i, T_o)$</td>
</tr>
<tr>
<td>state definition list $SD = {s_0 : sd_0; \ldots; s_n : sd_n}$</td>
<td>state definition list $SD = {s_0 : sd_0; \ldots; s_n : sd_n}$</td>
</tr>
<tr>
<td>junction definition list $J = {j_0 : T_j0; \ldots; j_n : T_jn}$</td>
<td>junction definition list $J = {j_0 : T_j0; \ldots; j_n : T_jn}$</td>
</tr>
</tbody>
</table>

Table 4.2. Restricted Stateflow subset compared with the original language
assignments as depicted in figure 4.5. In Stateflow every chart is executed according to a deterministic sequence (State A executes always before State B) and therefore any race condition is avoided. This approach preserves the sequential execution of the code, while allowing logical event implementation (each change on the variable value corresponds to an event).

Transitions have been distinguished between state transitions $t$ and junction transitions $t_j$: it is not allowed to define transitions between states and junctions. These are objects that 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 [98], a problem consisting in the possibility of traversing a path made of junctions, possibly assigning values to variables, and afterwards backtracking without restoring variable values. Figure 4.6 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* [50]. 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. This implies that 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 condition on guards of transitions outgoing from the same state.

![Diagram](image.png)

Figure 4.6. State/Junctions transitions can be avoided through proper modeling solutions

### 4.5 Translation validation and 2M-TVF

Model-based unit testing was introduced in the first stages of the SSC BL3 development. Due to the evolution of the project, the approach was then abandoned for the new versions of the platform. This choice created considerable problems from the certification point of view, with the issue of a trusted code generator still unsolved. With SSC Metrô Rio, the company decided to resume the strategy, with the objective of implementing all the tasks required by the certification authorities for the usage of a non-certified code generator in the context of safety-critical applications.
The technique adopted, which is an enhancement of the previously defined approach, is known as *translation validation* [23]. The approach comprises two steps. The first one is again a form of B2B testing, where both the model and the related generated code are tested using the same stimuli as inputs, and the numerical results obtained as output are checked for equivalence. The second one consists of an additional evaluation to grant the absence of unwanted and unexpected behaviours introduced by the model-to-code translation process. This evaluation is basically the comparison of the measures of structural coverage reached on both the code and the model. The two phases could be seen respectively as a duplicated black-box testing (output comparison) plus a duplicated white box testing (coverage comparison).

A framework called *2M-TVF*, that stands for Matlab Model Translation Validation Framework, has been developed to perform the model-based testing and the code validation process. The framework, whose workflow is depicted in figure 4.7, 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 tests execution, and then it is compiled and linked to obtain the executable. Automatic 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. We estimated that 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.

4.6 Results

The SSC Metrô Rio model in its first version was composed by 13 Stateflow charts for a total amount of approximately 70 KLOC of generated code. The development of the model was performed by a single developer within 4 months, in tight collaboration with the requirements manager. This interaction gave consistent help in consolidating the requirements: the model was used as a prototype to assess the correctness and soundness of the specification, like in the SSC BL1 experience. At the same time, the model was the source of the software that would have been deployed in the actual machine, and this improved the traceability between system requirements and the actual code. The system requirements were decomposed into unit requirements, and each one of them was apportioned to a Stateflow chart, from which code was generated. The adoption of RTW Embedded Coder, that produces navigable links between the source code statements and the corresponding model constructs, completed the bottom-up traceability evidence. Starting from a
single statement, one could reach the construct of the formal specification that produced it, and, from this one, one could reach the natural language unit requirement, and finally the system requirement, from which the model was defined.

The unit-test suite, provided according to the functional requirements coverage, consisted of 238 manually defined test-cases that covered 100% of functional unit-level requirements on the models. Other test cases were provided to reach 100% of decision coverage on the models.

The validation framework 2M-TVF was used to perform the comparison of the behaviour of models and related code: the models, together with the test suites, served as inputs for the framework that automatically performed the validation of the code for each module and gave the measures of coverage on both model and code.

The advantages of the new unit testing approach can be expressed in terms of bugs found and time saved during the verification phase, also thanks to the automation of the verification activities.

The modifications introduced in the development of SSC Metrô Rio ATP did not have a negative impact on the abstract interpretation analysis phase: the results obtained by the execution of the first Polyspace step shows a similar behaviour with respect to the ones obtained in the context of BL3 project, as depicted in figure 4.8. Also the results of the second step are coherent with the BL3 ones. Furthermore, the adoption of the new guidelines and the multi-level architecture led to a better structured generated code, with a substantial improvement for the human verification activities after the Polyspace analysis. Indeed, in the SSC Metrô Rio ATP project we have experienced a sensible reduction of the analysis time for the justification of orange marks: from 3 hours and a half in average for each module employed in the BL3 project, to 3 hours and ten minutes.
Table 4.3 compares the results of the verification activities performed on SSC Metrô Rio with the results of SSC BL1 Plus, where model-based development was also employed. In SSC BL1 Plus, the extended modelling guidelines, the 2M-TVF validation framework and the abstract interpretation technique were not yet introduced. Since the two projects are comparable in terms of decision coverage (see DC column) achieved with the tests, and therefore the code behaviour has been consistently exercised, it can be hypothesized that the considerable reduction of detected bugs (from 10 to 3 bugs per module) is due to the improvements of the development process.

In the last column of table 4.3 we have considered the overall time required to pinpoint the bugs and provide corrections. The well defined architecture derived from the multi-layer approach has allowed to reduce this time from 55 minutes to half an hour per bug in average.

Though the automation of the MBT phase through the 2M-TVF framework
### Results

<table>
<thead>
<tr>
<th>Project</th>
<th>#Modules</th>
<th>LOC</th>
<th>DC</th>
<th>#Bugs</th>
<th>Man/H</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSC Metro Rio</td>
<td>13</td>
<td>70K</td>
<td>100%</td>
<td>33</td>
<td>16</td>
</tr>
<tr>
<td>SSC BL1 Plus</td>
<td>12</td>
<td>8K</td>
<td>93%</td>
<td>114</td>
<td>105</td>
</tr>
</tbody>
</table>

Table 4.3. Bug detection and correction costs for comparable projects (modeling cost of approximately 4 man/months)

has improved the validation task in comparison with the SSC BL3 project, the bottleneck of the activity resulted the manual definition of tests for the models, that takes about the 60-70% of the cost of the verification process. In order to address this issue, the company is currently evaluating strategies for test generation that can be considered compatible with the current process. On the other hand, also formal verification by means of Simulink Design Verifier [69] is evaluated as an option to complete the formal development. However, the current version of the EN 50128 norm does not assess that formal verification can replace testing, and this is one of the reason why there is not so much pressure in this direction.

After its first version and the completion of the first verification process, the SSC Metrô Rio project passed to another developer, who had already some experience with Simulink modelling, but without confidence with code generation techniques. After less than one month, he was able to have a complete control of the model and apply the modifications required by the costumer for the subsequent versions. In the company, this learning ability was never experienced for projects that were using solely hand-crafted code. The formal approach adopted, supported by the further refinement of the guidelines, showed to be useful also for creating a sort of independence between developers and projects, which is a not negligible advantage both from the company and from the employees point of view. The former is ensured to have a formal knowledge base that can be handled by different developers,
while the latter is not required to give life-long support for the artifacts he has developed.
5.1 Projects and technologies

The research activity reported in this dissertation started with the objective of introducing the code generation technology within the development process of a railway signaling manufacturer. Table 5.1 summarizes the technologies progressively introduced during the projects reported in this dissertation.

The first experiments have been performed on the SSC project, the ATP system deployed over 2000 km of the Italian secondary lines. The Stateflow model developed by GETS for the first version of the system, namely SSC BL1, has been used as a prototype platform for the definition of the first set of modelling guidelines. The model, formerly employed only for requirements agreement with the customer, went through a refactoring path according to the rules, and proper code synthesis of the single model units was achieved.
Projects and technologies

<table>
<thead>
<tr>
<th>Year</th>
<th>Project</th>
<th>Technologies (Full or Partial Adoption)</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007-2008</td>
<td>SSC BL1</td>
<td>Modelling guidelines (25)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>SSC BL1 Plus</td>
<td>Code generation (Stateflow Coder R2007b)</td>
<td></td>
</tr>
<tr>
<td>2008-2010</td>
<td>SSC BL3</td>
<td>Modelling guidelines + MAAB (43)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code generation (Stateflow Coder R2007b)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Guidelines verification (Model Verifier 1.0)</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Model-based testing</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Abstract interpretation (Polyspace 7.0)</td>
<td></td>
</tr>
<tr>
<td>2009-2010</td>
<td>SSC Metró Rio</td>
<td>Modelling guidelines + MAAB (43)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Semantics restrictions</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UML + hierarchical derivation</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code generation (RTW Embedded Coder R2010a)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Translation Validation (2M-TVF 1.0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Abstract interpretation (Polyspace 8.0)</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1. Summary of the results achieved during the projects

through the Stateflow Coder tool. At the end of 2007, SSC BL1 evolved in SSC BL1 Plus, and the refactored model substituted the original one for the definition of the new system specifications. Still, code generation was not employed in the actual development process, and SSC BL1 Plus remained an hand-crafted system, since a proper V&V process was not defined yet.

With the SSC BL3 system, the platform that integrates the features of SSC and SCMT (the system deployed over the main Italian lines), the company put into practice its experience with code generation. The set of internal guidelines has been integrated with the public MAAB recommendations for modelling with Simulink/Stateflow, and a preliminary process for code verification has been defined. The planned process consisted in using an internally developed tool, Model Verifier, to check modelling standard adherence, and then performing functional unit-level verification by means of a two-phase task made of model-based testing and abstract interpretation. The first step checks for functional equivalence between model and code, while the second step is used to assess the absence of runtime errors. Due to the evolution of the project, both guidelines verification and model-based testing have been
Projects and technologies

only partially employed, and the process had to be adjusted with *ad-hoc* solutions to address the problem of a non-certified code generator.

SSC Metrô Rio was the project that allowed a complete implementation of a formal development process. Besides the technologies previously adopted, a hierarchical derivation approach has been employed, which, starting from an UML representation of the system, allows to derive unit-level requirements and a formal representation of them. The guidelines have been updated with a set of recommendations particularly oriented to define a formal semantics for the Stateflow language.

A new code generator, RTW Embedded Coder, has been introduced, which permitted to generate also the integration code between the different generated units, thus accelerating the development. With the project, also the goal of ensuring correctness of the generated code in absence of a certified translator has been addressed by combining a model-based testing approach known as translation validation [23], with abstract interpretation [24] [25]. Model-based testing has been performed with the support of 2M-TVF, an internally developed framework that allows back-to-back [110] model-code execution of unit level tests and ensures consistency between model and code coverage. Abstract interpretation with Polyspace has been performed with a strategy analogous to the one already applied for SSC BL3.

Many of the development, verification and certification issues related to the new technology appeared only during the actual development: the goals planned at the beginning of the research have been addressed after progressive tuning of the strategy across the different projects.
5.1.1 Goal 1: modeling language restriction

The first goal was identifying a subset of the Simulink/Stateflow language that generates C code that is compliant with the guidelines defined by the company, according to the quality standard required by the EN 50128 norm. With SSC BL1, this issue was addressed by first analysing the violations issued by the code generated from the original model, and then by defining sub-models on which experimentation could be performed more easily, observing the translation of the single graphical constructs, and combination of them, to avoid the violations experienced. The activity led to the definition of a preliminary set of 25 guidelines for creating models targeted for code generation.

With the SSC BL3 project, where code generation has been actually employed for the development of the whole logic software, a more systematic study has been performed. The preliminary set had in fact the limit of being derived from a specific model, and could lack of generality. A comparison with the experience of other safety-critical domains was required and the internal guidelines should have been bound to an assessed framework of development rules. It was required a set of accepted modelling rules equivalent to the MISRA ones for C code, in which the internally defined guidelines could be integrated. The preliminary set was extended by adapting to the railway domain the public MAAB guidelines defined by OEMs and suppliers of the automotive sector to facilitate model exchange and commissioning. This new set, composed by 45 guidelines in total, issued further restrictions that were not only limited to allow generation of quality code, but were also oriented to define well-structured models.

A further step was performed during the SSC Metrô Rio project: in order to ease formal analysis and formal representation of the requirements, it was decided to complete the modelling style guidelines by restricting the Stateflow language to a semantically unambiguous set. The studies of Scaife et al. [98], focused on translating a subset of Stateflow into the Lustre formal lan-
guage, have been used to define a formal semantics for Stateflow. With this approach, the models defined are independent from the simulation engine, and a formal development process could actually take place.

5.1.2 Goal 2: generated code correctness

The second goal was addressing the problem of a non-certified, neither proven-in-use, translator. The objective was ensuring the code to be fully compliant with the model behaviour, and guarantee that no additional improper functions are added during the code synthesis phase. The approach adopted, preliminarily defined during the SSC BL3 project, but refined and fully applied only on SSC Metrô Rio, consisted in implementing a model-based testing approach known as translation validation, and completing it with static analysis by means of abstract interpretation.

Translation validation, implemented in its final version through the 2M-TVF tool, consists of two steps: (1) a model/code back-to-back execution of unit tests, where both the model and the corresponding code are exercised using the same scenarios as inputs and results are checked for equivalence; (2) a comparison of the structural coverage obtained at model and at code level. The first step ensures that the code behaviour is compliant with the model behaviour, while the second one ensures that no additional function is introduced in the code: tests are performed until 100% of decision coverage is obtained on the models, if lower values are obtained for the code, any discrepancy must be assessed.

Model-based testing with translation validation ensures equivalence between model and code, but cannot cover all the possible behaviours of the code in terms of control-flow and data-flow, and, in particular, it lacks in detecting all those runtime errors that might occur only with particular data sets, such as division by zero and buffer overflow. For this reason translation validation has been completed with abstract interpretation by means of the
Polyspace tool, aimed at detecting runtime errors by performing static analysis of the source code. Since the correctness of the source is not decidable at the program level, the tools implementing abstract interpretation work on a conservative and sound approximation of the variable values in terms of intervals, and consider the state space of the program at this level of abstraction. Finding errors in this larger approximation domain does not imply that the bug also holds in the real program. The presence of false positives after the analysis is actually the drawback of abstract interpretation that hampers the possibility of fully automating the process. Already with SSC BL3, a procedure of two steps have been defined for the usage of the tool to address the problem of false positives: (1) a first analysis step is performed with a large over-approximation set, in order to discover systematic runtime errors and identify classes of possible false positives that can be used to restrict the approximation set; (2) a second analysis step is performed with a constrained abstract domain, derived from the first analysis, and the number of uncertain failure states to be manually reviewed is drastically reduced.

5.1.3 Goal 3: process integration

The third goal was integrating the modelling and code generation technologies into a coherent development process. Also for this issue, a sound process was finally achieved only with SSC Metró Rio, after subsequent adjustments. The introduction of modelling and the need to ensure consistency between models and code, has issued changes also on the verification and validation activities, that had to be tailored according to the new technology. On the other hand, it has allowed to work on a higher level of abstraction, and different methods and tools have been combined to achieve a complete formal development.

The final process is an enhanced V-based development model, as depicted in figure 5.1. The process embeds two verification branches: one for the activi-
ties performed on the models, the other for the tasks concerning source code and system.

From system requirements, tests are defined to be performed both at model-integration level and at system-level. Then, an UML architecture is defined in the form of a component diagram. The diagram is then translated into a Simulink architecture. During the design phase, system requirements are decomposed into unit requirements apportioned to the single architectural components, and Stateflow models are defined according to these unit requirements following the style-guidelines and the semantics restrictions. Functional unit testing is performed on the models before generating code through RTW Embedded Coder. Then, translation validation is performed, followed by static analysis by means of abstract interpretation. The application code is then integrated with operating system and drivers, and
hardware-in-the-loop is used to perform system tests according to the system requirements. The whole process is supported by coherent documentation: this is mainly auto-generated by means of Simulink Report Generator, using the comments edited on the models.

5.2 Lesson learned

Code generation was introduced following the intuition that defining a formal model of the specifications and automatically producing code allows to speed-up the development, while ensuring greater correctness of the code at the same time. The intuition has been actually confirmed by the practice. The modelling and code generation showed the following advantages with respect to hand-crafted code:

- models allow to work at a higher level of abstraction, and they can be manipulated better than code. The model-based testing approach, in the two versions put into practice during SSC BL3 and SSC Metrô Rio, has allowed to define test scenarios at component level without disrupting the model structure, an approach that would be impracticable on hand-crafted code;

- graphical models are closer to the natural language requirements and they are at the same time an unambiguous mean to exchange or pass artefacts among developers. This observation is enlightened by the SSC Metrô Rio experience, where the project passed from a developer to another within one month only with very little support;

- the generated software is composed by modules with higher internal cohesion and better decoupling: interfaces among functionalities are based on data only and the control-flow is simplified since there is no cross-call among different modules. Decoupling and well-defined interfaces have helped in easing the model outsourcing, which is a relevant
aspect in the development of products that have to tackle time-to-market issues;

- the generated code has a repetitive structure, that facilitates the automation of the verification activities. When strict modelling guidelines are defined, one could look at the generated code as if it would be the software always written by the same programmer: any code analysis task can be tailored on the artificial programmer’s design habits. As a witness for this observation, consider that the full two-step Polyspace process resulted profitable on the generated code only, since systematic analysis on hand-crafted code was made harder by its variable structure and programming style;

- software modules are directly traceable with specific blocks of the modelled specification. Traceability is a relevant issue in the development of safety-critical systems, since any error has to be traced back to the process task, or artefact cause, that produced it. The formal development approach introduced, with the support of RTW Embedded Coder, has allowed to define navigable links between the single code statements and the requirements;

- the structured development has allowed to achieve greater control over the components and, finally, to have software with less bugs already before the verification activities, as witnessed by the bug reduction evaluation performed during the SSC Metrô Rio project (from 10 to 3 bugs per module);

- the introduction of the new development process has allowed the reduction of verification activities cost, while ensuring greater confidence on the product safety. When passing from code unit testing based on structural coverage objectives to testing based on functional objectives aided with abstract interpretation, it was possible to reduce the verification cost of about 70%.

SUMMARY AND DISCUSSION

98
Lesson learned

The main drawback found in introducing code generation has been the size and overall complexity of the resulting software. Though this was not complicating the verification activities, it posed challenges from the performance point of view: ATP systems do not have hard real-time constraints, however they are reactive systems that, might a failure occur, shall activate the brakes in a limited amount of time in order to reach the safe state (i.e., the train standing condition). The reaction time is influenced by the main execution time, that resulted four times higher in the first experiments. The change of the hardware actually solved the problem, but with the design of new, more complex, systems, this issue has to be taken into account. From the hardware architecture side, one has also to consider the fact that the code is larger and one has less flexibility in terms of optimizations at source level (we recall that optimizations at compiler level are not allowed on SIL-4 systems): when designing the platform, a larger amount of memory has to be planned if one wants to employ code generation.

Though consistent cost improvements has been achieved on the verification activities, manual test definition is still the bottleneck of the process, requiring about 60-70\% of the whole unit-level verification cost. Formal verification is supposed to address this problem for the validation of models with respect to safety properties, but, in our experience, adopting this technology for all the functional, non-safety related, requirements is not applicable. An approach for test generation shall be foreseen when applying model-based development, but this requires the definition, or the extraction, of a separate higher-level model, from which tests can be derived for the model used for code generation. Examples of such strategies are reported in the literature [3] [86], but the technology is not well-established yet to be easily applied at industrial level. At the moment, the best option is having a library of verified components that can be combined to produce a complete system logic: in this way unit tests are performed only once and one can focus on integration.
Some lessons have been learned also from the process management point of view. The research activity has been performed according the following model: a research assistant coming from the university, fully focused on the technology to be introduced, and the internal development team, that puts into practice the research on real projects when this is successful. The results obtained across these three years would have not been possible through intermittent collaborations only. On the other hand, they would have been hardly achieved even if an internal person would have been in charge of the research. The independence of the research assistant from the development team has to be preserved, in order to separate the research from the time-to-market issues. The experience showed that also for average size companies, such as the signalling division of GETS, it is possible to perform research when a proper model is adopted, and this is indispensable to address the new market requirements. Along with the experience reported here, the company started to enlarge its business, previously focused in Italy, towards foreign countries, such as Sweden, China, Kazakhstan and Brazil, and the introduction of formal model-based development had actually played a relevant role to support this evolution.
Conclusion

This dissertation reports the experience of introducing formal model-based design and code generation by means of the Simulink/Stateflow platform [71] in the development process of a railway signalling manufacturer, namely the General Electric Transportation Systems (GETS) [45], Intelligent Systems division of Florence. The company operates in a CENELEC [18] [19] regulated framework, and adopting commercial, non qualified tools as core part of the development activities poses hurdles from the verification and certification point of view. First, a safe-subset of the modelling language has to be identified, in order to produce source code that is compliant with the quality standard required by the norms. Second, the behavioural equivalence between the generated code and the modelled specification has to be ensured. Third, the modelling and code generation technologies have to be integrated within the process recommended by the regulations.

The three issues have been gradually addressed throughout different industrial projects performed by the company, with the support of the Computer Engineering department of the University of Florence.

During the SSC Baseline 1 (SSC BL1) exploratory project (2007-2008) [42], a first set of 25 modelling guidelines for the production of quality code has
been identified, through the analysis and refactoring of a pre-existing model used for requirements elicitation and formalization. This set had the limit of being focused on the experience of one platform only: a comparison with the background of other safety-critical domains was required and the internal guidelines should have been bound to an assessed framework of development rules, before using the technology on real products.

In the context of the SSC Baseline 3 (SSC BL3) project (2008-2010) [42][43], the modelling guidelines have been integrated with the MAAB style-guidelines [74], a publicly available set of rules edited by OEMs and suppliers of the automotive domain for model exchange and commissioning. The new set was composed by 43 guidelines, and was used for the actual development of the platform logic.

With the employment of code generation for real products, the issue of the equivalence between the code and the intended behaviour expressed by the model had to be taken into account. A unit-testing verification process based on two step has been defined. The first step consists in performing model/code back-to-back testing [110], that means executing functional unit tests on the models and the corresponding generated code and automatically evaluating coherence of the results. This step ensures correctness of the code with respect to the functional requirements, and, at the same time, guarantees that model and code are equivalent. The second step performs static analysis by means of abstract interpretation through the Polyspace tool [30]. This technology allows to systematically detect runtime errors: these cannot be discovered by means of unit testing only, since dynamic testing cannot exercise all the possible behaviours of the code in terms of control-flow and data-flow.

With the new testing approach, the company reduced the unit-level verification costs of about 70% [47][39]. The benefits gained from the verification point of view stabilized the code generation technology within the company,
but a formal development process was yet to be defined.

The SSC Metrô Rio project (2009-2010) allowed to address also this final issue [40] [41]. The integration of high-level formalisms, namely the UML notation, with Simulink/Stateflow allowed to define a hierarchical derivation approach for the development of models. On the other hand, the Stateflow language, core of the model logic, was further restricted to a semantically unambiguous subset. Having a formal specification language ensures that the models defined are independent from the simulation engine, and a formal development process can actually take place. With SSC Metrô Rio, the first step of the unit testing approach defined for SSC BL3 was improved using the translation validation technique [23], and introducing the internally developed 2M-TVF framework to implement the methodology.

The main result obtained with the employment of a formal development process has been the greater control over the models, that brought to a substantial bug reduction (from 10 to 3 bugs per module) and to improved code-model-requirements traceability, witnessed by the reduction of the time required for bug detection and correction (from 55 minutes to half an hour per bug).

Though improvements in costs and product safety have been achieved with the new process based on formal modelling and code generation, the bottleneck of the development is still the manual unit tests definition, that takes about the 60-70% of the cost of the verification process. Formal verification strategies, such as model checking [22] or theorem proving [20], are supposed to address this problem for the validation of models with respect to safety properties, however, in our experience, adopting these technologies for all the functional, non-safety related requirements is not applicable. An approach for test generation shall be foreseen based on the definition, or the extraction, of a separate higher-level model, from which tests can be derived for the
model used for code generation. Examples of such methods are reported in the literature [3] [86], but the technology is not well-established yet to be easily applied at industrial level. At the moment, the option considered by the company is having a library of verified components that can be combined to produce a complete system logic: in this way unit tests are performed only once and verification can be focused on integration level verification. The current activities concerning the adoption of SysML modelling [83] for the management of the whole product life-cycle are heading toward this direction.


BIBLIOGRAPHY  107


