View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Florence Research

An industrial application of formal model based development: the Metrô Rio ATP case

Alessio Ferrari, Mario Papini
General Electric Transportation Systems
Via P. Fanfani, 21
Florence, Italy
{alessio.ferrari, mario.papini}@ge.com

Alessandro Fantechi, Daniele Grasso
University of Florence, D.S.I.
Via di S.Marta, 3
Florence, Italy
{fantechi, grasso}@dsi.unifi.it

ABSTRACT

The railway and metro signaling industries are currently investigating strategies for the introduction of formal model based development within their development processes.
Among the various platforms supporting this technology, the Simulink/Stateflow tool-suite has been adopted in various safety-critical domains for modeling and code generation of control systems. Despite their flexibility and ease of use, introduction of these tools for developing dependable software, and in particular signaling applications, has been often hampered by the lack of a rigorous formal semantics and by the absence of a certified code generator. This paper reports on the Simulink/Stateflow based development of the on-board equipment of the Metrô Rio Automatic Train Protection system, describing the design strategy and the approach followed in addressing weaknesses and certification issues related to the adopted tool-suite.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*; D.3.4 [**Programming Languages**]: Processors—*code generation*

General Terms

Design

Keywords

Metro Signaling, Model Based Development, Modeling Guidelines, Transport, Rail

1. INTRODUCTION

The increase in productivity and facilitation of safety assurance that the adoption of formal modeling and code generation technologies can bring in developing reliable products is described by many case studies [12]. In the metro and railway signaling domain, where the safety culture is traditionally and necessarily strong, there is increasing interest in formal methods and how they can be applied to the development of systems, with automatic train control systems being a leading candidate for these techniques. Paris Metro [4] and San Juan Metro [10] are two notable examples of this trend. Despite these successes, combining formal methods with model driven development and code generation is still at its initial stages within the signaling industry.
General Electric Transportation Systems (GETS) was commissioned for the adaptation of its SSC Automatic Train Protection (ATP) to Metrô Rio in 2008. This was a time when GETS was finishing its first large scale development project that made use of formal model based development. In an effort to improve its development process, GETS adopted the Simulink/Stateflow platform first for the development of prototypes [3] and afterwards for requirements formalization and code generation [5]. Experimentation with the code generator led to the definition of an internal set of modeling rules in the form of an extension of the MAAB guidelines [2], a stable and widely accepted standard developed by automotive companies.
The SSC - Metrô Rio ATP project provided the opportunity to refine the modeling activity toward a more formal approach. Indeed, despite the flexibility and ease of use of Simulink/Stateflow, the tools have two fundamental limitations in this type of application: the lack of a rigorous formal semantics and the absence of a certified code generator. This paper describes how these shortcomings have been addressed during the project, introducing modeling rules to reduce the languages to a semantically unambiguous set, and how design practices have been adopted to gradually achieve a formal model of the system. Quantitative results are given to show the effectiveness of the approach in terms of design error reduction and detection.

2. ATP SYSTEMS

The role of a metro signaling system is to protect trains by keeping vehicles a safe distance apart. Traditionally, the traffic along metro tracks is managed by dividing each track into segments called *block sections* or simply *blocks*, and ensuring each train 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 drivers about the status of the section that they are entering and, depending on the line topology, also about the status

of subsequent blocks. The meaning of each signal aspect can be broadly represented by three pieces of information:

Authorized speed: the speed that is permitted in the block that is being entered;
Target distance: the maximum distance that the train can move while still being protected;
Target speed: the maximum speed that the train is permitted to have over the target distance.

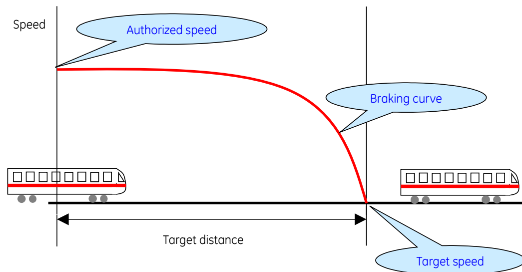

Figure 1: Authorized speed, Target distance and Target speed

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 and the protection of the line traffic independent of train operator actions.
From the architectural perspective, ATP systems are composed of wayside equipment and carborne equipment: the wayside equipment transmits information 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.

2.1 SSC Metrô Rio ATP system

The SSC Metrô Rio system consists of wayside devices, composed by an encoder and a transponder, that respectively encode and transmit a telegram that contains the data to be processed by the carborne equipment. The wayside devices are positioned close to the actual signals, and the combination of encoder and transponder is commonly referred to as an *information point*. The carborne equipment is a synchronous two out of two architecture[1] that receives the telegram data and performs the actual enforcement of train speed.
Information managed by the carborne equipment concerns the approach speed and distance for signals, but also other information typical for a metro, such as the distance to the next platform and speed reduction due to particular conditions of the line. All of this information is managed by the system as concurrent targets: for each restriction, multiple braking curves are computed to determine the most restrictive speed.

[1]Redundant system employing duplication and comparison of outputs for error detection, with no diversity.

Interaction with the driver is primarily via a touch-screen panel which displays a speedometer with the current speed and the active speed limit, and provides a set of buttons and icons to let the driver control and monitor the system. For example, one of the buttons is dedicated to the rain control function, a feature which is particular to the geographical 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 is signalled by the driver through the appropriate button on the touch-screen panel and the system reacts accordingly combining this “rainy” state with the information received from the information points.

3. MODELING GUIDELINES

Products traditionally provided by GETS, like any railway signaling application developed for Europe, shall comply with the CENELEC standards [1]. This is a set of norms and methods to be used while implementing a product having a determined safety-critical nature. The standards partition products into five different Safety Integrity Levels (SIL), from SIL-0 to SIL-4, being level 0 for non safety-related software and level 1 to 4 for increasingly safety-related software. In order to develop SIL-4 products, such as the ones GETS is traditionally providing, strong constraints are given by the CENELEC standards both on the software quality and on the process recommended practices. Although the SSC - Metrô Rio product was not going to be delivered for Europe, GETS decided to develop it with the objective of certifying it according to the CENELEC norm, since these standards remain the first reference for signaling applications and they are widely accepted outside European markets.
The CENELEC EN50128 [1] norm, specific for software of railway safety-critical systems, assesses that the code shall be developed according to coding standards to ensure traceability, structuring and readability of the code. Concerning autocoding, the guidelines ask for a validated, or proven-in-use translator. In absence of such a code generator, as is the case of Simulink/Stateflow, the compliance of automatically generated code to the guidelines is not different from that of handwritten code. The approach investigated by GETS was asking the generated code to obey the same rules about programming style and language subset which are asked for the hand-written code following the EN50128 guidelines. The idea was that only following a suitable modeling style during the model development it is possible to generate a code that is compliant with the guidelines and that can be successfully integrated with the existing one.

3.1 MAAB guidelines adaptation

The issue of modeling guidelines definition for the Simulink/Stateflow platform appeared to be already experienced by the companies of the automotive sector, that, in an effort of defining a common language between different OEMs and suppliers for models exchange and commissioning, came to the definition of the MAAB Control Algorithm Modeling Guidelines (MathWorks Automotive Advisory Board)[2],

Rule ID	Title	Priority	Restriction
db_0132	Transitions in Flowcharts	SR	M
jc_0521	Use of the return value from graphical functions	R	SR
na_0001	Bitwise Stateflow operators	SR	M
jc_0451	Use of unary minus on unsigned integers in Stateflow	R	M
na_0013	Comparison operation in Stateflow	R	M
db_0122	Stateflow and Simulink interface signals and parameters	SR	SR
db_0125	Scope of internal signals and local auxiliary variables	SR	M
jc_0491	Reuse of variables within a single Stateflow scope	R	M

Table 1: Some examples of restrictions to the MAAB guidelines

now a well established set of publicly available recommendations for modeling with Simulink/Stateflow. The guidelines, published in 2001 and afterwards revisited in 2007, resulted in being a good starting framework for GETS to define its own modeling standard targeted for code generation.
MAAB rules are focused on the design of automotive systems, and proper customizing was needed in order to adopt them in the signaling domain. The system and software architectures, the required degree of dependability and the certification processes of automotive and railway industries are quite different, and, above all, the issue of code generation is not the main focus of the MAAB guidelines, particularly oriented to model desing. It is with this purpose that MAAB rates the recommendations with a priority label (Recommended (R), Strongly Recommended (SR) and Mandatory (M)), issuing the level of importance of the rule. Great part of the guidelines needed some priority restrictions in order to ensure proper code synthesis.
The software of an on-board equipment of an ATP system, such as the SSC - Metrô Rio product, is characterized by the extensive usage of control modes logic and message analysis algorithms. These are all features that can be properly represented through state machines, and hence through discrete Stateflow models. Due to this reason, in the context of the project, only Stateflow has been adopted as specification language, while Simulink was only used as a simulation framework to allow interaction among Stateflow charts.
Indeed, it has been experimented that application of the MAAB rules to Stateflow models automatically implies the compliance of the generated code to some EN50128 or company code guidelines. For such rules the priority has been increased (see Table 1). For example, rule db_0125 states that every local data in a Stateflow model shall not be defined at *machine* (i.e., top-model) level. Since every local data that is defined at machine level is generated as global data, this rule has to be set as mandatory: global data is forbidden for SIL-4 applications. 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. More details on the MAAB guidelines adaptation to the railway signaling domain are given in a previous work [5].

3.2 Stateflow semantics restrictions

Stateflow is a graphical tool implementing a variant of Harel's hierarchical statecharts [9], normally called charts according to the Stateflow taxonomy. The complex semantics of Stateflow is not formally based, though research has been performed to define an operational semantics [8] and a denotational semantics [7] for a Stateflow subset. Along with the development of the SSC - Metrô Rio project, in order to ease an unambiguous interpretation of Stateflow models, coherent with the automatically generated code, we further extended the MAAB guidelines with a set of rules particularly oriented to restrain the Stateflow language to a semantically unambiguous subset. Indeed, the semantics of Stateflow allows modeling constructs which might be harmful from the point of view of model analysis and code generation. Below are reported three examples of additional guidelines together with a detailed explanation of their role.

ge_s_01: *Events shall not be used in Stateflow diagrams*
After the generation of an event, the control is returned to the state machine that broadcasted the event, and at code generation level this implies a recursive call to the function implementing the state machine. This might lead to the risk of an infinite recursion call, stack overflow or anyway to state-space explosion problems, with the well known drawbacks from the formal verification and analysis point of view. For this reason events are forbidden by the adopted modeling guidelines and they are simulated through variable assignments as depicted in Fig. 2. This approach preserves the sequential execution of the code, while allowing logical event implementation (each change on the variable value corresponds to an event).

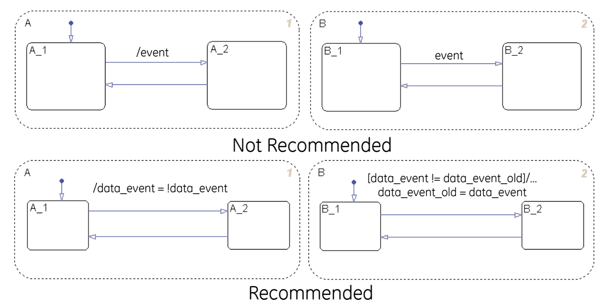

Figure 2: Events can be avoided through proper variable assignments

ge_S_02: *States and junctions shall not be used jointly*
The Stateflow language allows defining transitions between states and junctions. These are objects that have quite a different operational semantics: at each simulation step, states belonging to a single chart are mutually exclusive, while more than one junction can be traversed during the same simulation step. The behavior discrepancy between these two objects might

bring to improper combined usage. One of the well known possible hazards is *backtracking without undo*, a problem that has been explored in [11] and consisting in the possibility of traversing a path made of junctions, possibly assigning values to variables, and afterwards backtracking without restoring variable values. This problem is propagated also at code level. For this reason combined usage of states and junctions is forbidden, and junction are allowed only inside sequential function objects. Fig. 3 shows how an improper modeling can be correctly translated into an equivalent, yet safer, representation.

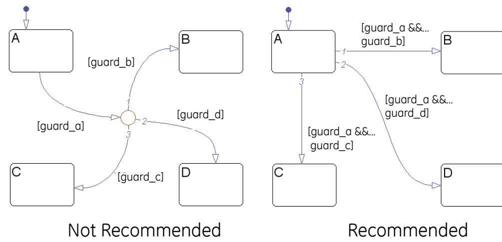

Figure 3: State/Junctions transitions can be avoided through proper modeling solutions

ge_S_03: *Outgoing transitions shall have mutually exclusive conditions on their guards*
One of the peculiar characteristics of the Stateflow semantics is the use of the *clockwise rule* to evaluate the firing of the transitions from the same state [8]. Transitions from the same state are ordered first on the form of their guards: transitions guarded by an event are evaluated before those guarded only by a condition, and unguarded transitions come last. Remaining unordered transitions from the state (i.e., transitions that have the same form of the guards) 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 ask to make this determinism explicit by using mutually exclusive condition on guards of transitions outgoing from the same state.

Other rules not detailed here have been given in order to further constrain the Stateflow semantics along the lines of the approach shown by Scaife et al. [11] for translating a subset of Stateflow into the Lustre formal language.

4. THE SSC - METRÔ RIO MODEL

On the one hand, formal modeling requires the definition of a formal language, and this has been addressed by restricting the Simulink/Stateflow language to a semantically unambiguous subset through modeling guidelines. On the other hand, when a large requirements set is involved in formal modeling, also the architecture of the model comes to be a fundamental issue. Structuring 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. Furthermore, if one is expecting to auto-generate code from the model, its structure has to take into account also the software architecture: an effort has to be made to create formal models having a structure that makes sense also in terms of the architecture of the software system.

4.1 Architecture Definition

In the context of the project, we found useful to first represent the high-level software architecture through a UML component diagram. 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.

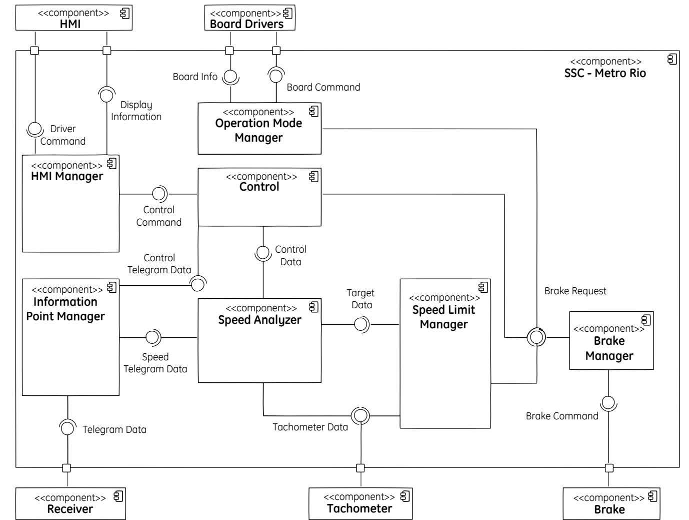

Figure 4: Simplified component diagram

The simplified component diagram of the SSC - Metrô Rio on-board application is shown in Fig. 4. 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 behav-

ior 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. 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.

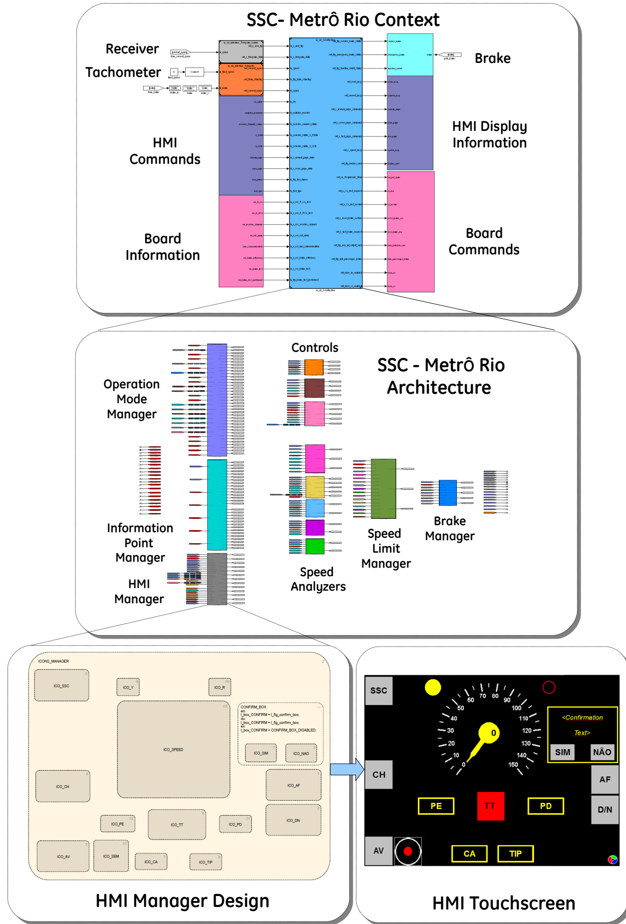

Figure 5: The multiple level hierarchical model

4.2 Derivation Approach

In order to properly formalize this kind of architecture through Simulink/Stateflow, the chosen strategy was to represent the system through a multiple-level hierarchical model (see Fig. 5). The different levels are intended for different development stages, from a more abstract to a more detailed view. A first level is defining the context, which means the interfaces with the environment in terms of input/output data. Starting from the component diagram, this level has been derived considering the boundary ports and mapping them into signals entering or exiting the Simulink blocks. This approach allowed us 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.
A 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 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 variable. This level focuses on the relationships between functional units.
A third level is actually the design level of the single Stateflow charts, each of them being 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, the system functional requirement concerning the rain control functionality (see Section 2.1) says:
When the rain function button (CH) is pressed and released within t_rain_function *milliseconds and the train is standing, the icon of the rain function button shall be lighted on and the target speed when approaching the platform shall be reduced to 40 km/h if the train is positioned outdoor.*
The requirement is decomposed as reported in Table 2.

	Requirement	Module
1	*If the rain function button (CH) is pressed and released within* t_rain_function *milliseconds, the rain event shall be raised*	HMI Manager
2	*If the rain event is raised and the train is standing, the rain function shall be activated*	Platform Control
3	*If the rain function is active and the train is positioned outdoor, the target speed shall be reduced to 40 km/h*	Platform Speed Analyzer
4	*If the rain function is active, the icon of the rain function button shall be lighted on*	HMI Manager

Table 2: Unit requirements decomposition

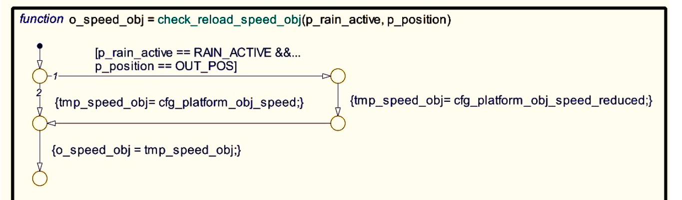

Figure 6: Example of unit requirement formalization

The first and last unit requirements in the table are apportioned to the HMI Manager, since this component is intended to manage any interaction with the driver. The second requirement is apportioned to the Platform Control, which attends to manage if the rain function shall be activated or not, and sets the status of the function itself. The third requirement (see Fig. 6 for its formal representation) is apportioned to the Platform Speed Analyzer, which is actually computing the target data for the braking curve, when this is required by the telegram data, and modifies the tar-

get speed depending on the status of the rain function and on the position (outdoor/indoor) of the train.

4.3 Results and lesson learned

Following the approach exemplified, we came to the definition of 438 unit requirements that led to the representation of 13 modules and about 150 state machines in total. The C code generated through RTW Embedded Coder resulted in 14 source files, one for each chart and one for units integration, issuing approximately 120K lines of code. Thanks to the modeling rules applied, the code synthesized resulted compliant with the EN50128 guidelines, with an acceptable degree of readability and traceability. This is actually a desirable quality not only from the certification point of view, but also from the point of view of debugging. Indeed, during development we found out that not all issues could be solved at simulation level. For example, the communication protocols with the HMI and with the telegram processing board were not part of our simulation, and direct debugging on the target was needed in order to solve problems raised by the real time constraints of the communication.
Verification activities have been performed on both the model and the code according to the approach based on Model Based Testing and Abstract Interpretation explained in a previous work [6]. Table 3 compares the results of the verification activities on SSC Metrô Rio in terms of bugs found and time spent to detect and correct the bugs, with the results of these activities on the SSC BL1 product, a previous ATP project based on model based development where only the MAAB guidelines with proper restrictions were used. The additional rules constraining the Stateflow semantics introduced for SSC Metrô Rio led to a notable reduction of bugs, while the well defined architecture derived from the novel design approach has allowed us to detect the errors in shorter time, even though, on the other hand, has increased the number of modules and the generated LOC: structuring and separating concerns necessarily affect the software size.
If we have to compare the overall development cost for the SSC Metrô Rio project with a project based on hand-crafted code, our experience tells that a developer spends 30% of the time more on modeling than on coding. Nevertheless, this greater effort is balanced by the fact that notable cost reductions are achieved in terms of verification activities (with a time reduction of about 70%, see [6]).

Project	#Modules	LOC	#Bugs	Man/H
SSC Metrô Rio	13	120K	33	**16**
SSC BL1Plus	12	40K	114	**105**

Table 3: Bug detection and correction costs for comparable projects which required a modeling cost of approximately 4 man/months

5. CONCLUSION

This paper presented the experience of a metro signaling manufacturer in employing formal model based technologies to the development of the Automatic Train Protection system of Metrô Rio. The Simulink/Stateflow tool-suite have been used to model and generate the code of the entire application software. The certification issues related to the tool-suite and the formal weakness of the languages that were used have been overcome by restricting the languages to a semantically unambiguous set and by introducing a multiple level architecture approach for deriving a formal model for the system. When compared with previous model based projects where the approach was not applied, the results in terms of number of errors deteced and in terms of time spent for correcting them are encouraging. Nonetheless, the strategy still needs improvements to ensure the consistency of the different models at the different derivation levels, where heterogeneous graphical notations are used.
We are currently exploring techniques to complete the formal development with formal verification. Specifically, solutions are under study with Simulink Design Verifier, but also with other freeware tools such as SPIN and NuSMV.

6. REFERENCES

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

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

[3] S. Bacherini, A. Fantechi, M. Tempestini, and N. Zingoni. A story about formal methods adoption by a railway signaling manufacturer. *Proceedings of FM 2006, LNCS*, 4025, 2006.

[4] A. Faivre and P. Benoit. Safety critical software of meteor developed with the B formal method and vital coded processor. In *Proceedings of WCRR99*, pages 84–89, 1999.

[5] A. Ferrari, A. Fantechi, S. Bacherini, and N. Zingoni. Modeling guidelines for code generation in the railway signaling context. In *Proceedings of 1st NASA Formal Methods Symphosium*. NASA, April 2009.

[6] D. Grasso, A. Fantechi, and A. Ferrari. Model based testing and abstract interpretation in the railway signaling context. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (to appear)*, 2010.

[7] G. Hamon. A denotational semantics for stateflow. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 164–172, 2005.

[8] G. Hamon and J. Rushby. An operational semantics for stateflow. *STTT*, 9(5-6):477–456, 2007.

[9] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[10] M. Leuschel, J. Falampin, F. Fabian, and D. Plagge. Automated property verification for large scale B models. *Proceedings of FM 2009, LNCS*, 5850:810–814, 2009.

[11] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a safe subset of Simulink/Stateflow into Lustre. In *Proceedings of the 4th ACM international conference on Embedded software*, pages 259–268, 2004.

[12] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 5(N):1–39, June 2009.