Lessons Learnt from the Adoption of Formal Model-Based Development

Alessio Ferrari[1], Alessandro Fantechi[1,2], and Stefania Gnesi[1]

[1] ISTI-CNR, Via G. Moruzzi 1, Pisa, Italy
{firstname.lastname}@isti.cnr.it
http://www.isti.cnr.it/
[2] DSI, Università degli Studi di Firenze, Via di S.Marta 3, Firenze, Italy
fantechi@dsi.unifi.it
http://www.dsi.unifi.it/~fantechi/

Abstract. This paper reviews the experience of introducing *formal model-based design* and *code generation* by means of the Simulink/Stateflow platform in the development process of a railway signalling manufacturer. Such company operates in a standard-regulated framework, for which the adoption of commercial, non qualified tools as part of the development activities poses hurdles from the verification and certification point of view. At this regard, three incremental intermediate goals have been defined, namely (1) identification of a safe-subset of the modelling language, (2) evidence of the behavioural conformance between the generated code and the modelled specification, and (3) integration of the modelling and code generation technologies within the process that is recommended by the regulations.

These three issues have been addressed by progressively tuning the usage of the technologies across different projects. This paper summarizes the lesson learnt from this experience. In particular, it shows that formal modelling and code generation are actually powerful means to enhance product safety and cost effectiveness. Nevertheless, their adoption is not a straightforward step, and incremental adjustments and refinements are required in order to establish a *formal model-based process*.

Introduction

The adoption of formal and semi-formal modelling technologies into the different phases of development of software products is constantly growing within industry [4,29,27]. 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. These applications, 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.

A. Goodloe and S. Person (Eds.): NFM 2012, LNCS 7226, pp. 24–38, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Within this context, recent years have seen the diffusion of graphical tools to facilitate the development of the software before its actual deployment. Technologies known as *model-based design* [32] and *code generation* started to be progressively adopted by several companies as part of their software process.

The development of safety-critical software shall conform to specific international standards (e.g., RTCA/DO-178B [30] for aerospace, IEC-61508 [22] for automotive and CENELEC/EN-50128 [7] for railway signalling in Europe). These are a set of norms and methods to be used while implementing a product having a determined safety-related nature. In order to certify a product according to these standards, companies are required to give evidence to the authorities that a development process has been followed that is coherent with the prescriptions of the norms.

Introducing model-based design tool-suites and the code generation technology within a standard-regulated process is not a straightforward step. 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. Concerning the tools, the norms ask for a certified or *proven-in-use* translator: in absence of such a tool, a strategy has to be defined in order to assess the equivalence between the model and the 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.

With the aim of establishing guidelines for a formal model-based development process, in this paper we review a series of relevant experiences done in collaboration with a railway signalling manufacturer operating in the field of Automatic Train Protection (ATP) systems. Inside a long-term effort of introducing formal methods to enforce product safety, indeed the company decided to adopt the Simulink/Stateflow tool-suite to exploit formal model-based development and code generation within its own development process [2,15]. The decision was followed by four years of incremental actions in using commercial tools to build a formal model-based process focused on code generation. Details of these actions, have been published elsewhere [15,17,18,16]. We are here interested instead to give a global view of the overall experience.

The paper is structured as follows. In Sect. 1 some background is given concerning *formal methods*, *model-based design* and the existing approaches integrating the two technologies. In Sect. 2 the research problem of introducing code generation from formal models in a safety-critical domain is expressed and discussed. In Sect. 3 the projects where formal model-based development has been employed are presented, together with the goals progressively achieved with respect to the main research problem. In Sect. 4 the advantages and the critical aspects of code generation are evaluated. Sect. 5 draws final conclusions and remarks.

1 Formal Model-Based Design

In 1995, Bowen and Hinchey published the Ten Commandments of Formal Methods [5], a list of guidelines for applying formal techniques, edited according to their experience in industrial projects [20]. Ten years later, the authors review their statements, and they witness that not so much have changed [6]: the industrial applications that demonstrate the feasibility and the effectiveness of formal methods are still limited, though famous projects exist, which show that the interest in these methods is not decreased. Among them, it is worth citing the Paris Metro onboard equipment [3], where the B method has been employed, and the Maeslant Kering storm surge barrier control system [33], where both the Z and the Promela notations have been used.

The comprehensive survey of Woodcock et al. [35] confirms that industries are currently performing studies on formal methods applications, but still perceive them as experimental technologies.

While formal methods have struggled for more than twenty years for a role in the development process of the companies, the *model-based design* [32] 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 allow simulations and tests of the system models to be performed before the actual deployment. The objective is not different from the one of formal methods, which is detecting design defects before the actual implementation. However, while formal methods are perceived as rigid and difficult, model-based design is regarded as closer to the needs of the developers, which consider graphical simulation more intuitive than formal verification.

This trend has given increasing importance to tools such as the SCADE suite [11], a graphical modelling environment mostly used in aerospace and based on the Lustre synchronous language, Scicos [23], an open source platform for modelling and simulating control systems, and the two tools ASCET [14] and AutoFocus [21], both oriented to automotive systems and using block notations for the representation of distributed processes.

In this scenario, the safety-critical industry has progressively seen the clear establishment of the Simulink/Stateflow [25] platform as a *de-facto* standard for modelling and code generation. The Simulink language uses a block notation for the definition of continuous-time dynamic system. The Stateflow notation is based on Harel's Statecharts [19] 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, while tools such as Simulink Coder [25] and TargetLink [12] support automatic source code generation from the models. These features, strengthened by the large amount of associated toolboxes to analyse the different aspects of an application, has enabled a cross-domain spread of the platform.

Nevertheless, 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 industries and researchers toward an integration between model-based design and formal techniques [1,29]. The main goal is to 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 [26]. 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 [4]. The most complete, integrated methodology is probably the one currently practiced by Rockwell Collins [27]. The process implemented by this company of the avionic sector starts from Simulink/Stateflow models to derive a representation in the Lustre formal language. Then, formal verification is performed by means of different engines, such as NuSMV and Prover, followed by code generation in C and ADA.

The main contribution of the current paper with respect to the related work is the in-depth focus on the code generation aspect, together with the evaluation of the advantages given by the introduction of this technology in a medium-size company. Our objective is to give a clear picture of how the adoption of code generation affects the overall development process.

2 Problem Statement

The company considered in this paper operates in the development of safety-related railway signalling systems. Inside an effort of adopting formal methods within its own development process, the company decided to introduce system modelling by means of the Simulink/Stateflow tools [2], and in 2007 decided to move to code generation [15].

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.

In order to achieve this goal, the company contacted experts from academia, expected to give guidance and support along this paradigm-shift in the development process. This paper reviews a series of experiences done in this collaboration, inside a four year reasearch activity started at the end of 2007, with the aim to address the following:

Problem Statement
Define and implement a methodology for the adoption of the code generation technology from formal models by a railway signalling company

During the research activity, the problem statement has been decomposed into the following sub-goals.

Goal 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 and auto-coding are adopted, the generated code shall conform to the same standard asked to the hand-crafted code. Hence, the identification of a safe subset of the adopted modelling language is required for the production of code compliant with the guidelines and that can be succesfully integrated with the existing one.

Goal 2 - generated code correctness. Safety-critical norms ask 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 to perform the verification activities at the level of the abstract model, minimizing or automating the operations on the code.

Goal 3 - process integration. Product development is performed by companies by means of processes, which define a framework made of tasks, artifacts and people. Introduction of new technologies in an established process requires adjustments to the process structure, which 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.

3 Projects and Achievements

Addressing the problem statement issued above started with the objective of introducing model-based design and code generation within the development process of the company. The specific projects, summarized in the following, were subsequently selected as test-benches for the incremental introduction of such technologies. Each goal expressed in Sect. 2 is evaluated according to its progressive refinement during the projects.

The first experiments have been performed during **Project 1**, involving the development of a simple Automatic Train Protection (ATP) system.

ATP systems are typically embedded platforms that enforce the rules of signaling systems by adding an on-board automatic control over the speed limit imposed to trains along the track. In case of dangerous behaviour acted by the driver (e.g., speed limit or signalling rules violation) the system is in charge of regulating the speed by enforcing the brakes until the train returns to a safe state (i.e., the train standing condition or a speed below the imposed limit).

During Project 1, an ATP system was developed from scratch with the support of the Simulink/Stateflow tool-suite. A Stateflow model was designed in

collaboration with the customer in order to define and assess the system requirements [2]. This experience, completed with the successful deployment of the system, allowed the assessment of the potentials of modelling for prototype definition and requirements agreement.

The actual research on code generation started when the hand-crafted system was already deployed and operational. The Stateflow model, formerly employed for requirements agreement, has been used as a prototype platform for the definition of a first set modelling language restrictions in the form of *modelling guidelines* [15]. The model passed through a refactoring path according to the guidelines defined, and proper code synthesis of the single *model units* was achieved through the Stateflow Coder[1] tool.

At the end of 2007, the system evolved in a new version. 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 the product remained an hand-crafted system also in its new version, since a proper V&V process for its certification against regulations was not defined yet.

Project 2, involved an ATP system about ten times larger in terms of features compared with Project 1. With this project, the company put into practice its acquired experience with code generation. The set of internal guidelines edited during Project 1 has been integrated with the public MAAB recommendations[2] [24] for modelling with Simulink/Stateflow.

Furthermore, a preliminary process for code verification has been defined [18]. The defined process was structured as follows. First, an internally developed tool was used to check modelling standard adherence, a sort of static analysis performed at model level. Then, functional unit-level verification was performed by means of a two-phase task made of *model-based testing* [13] and *abstract interpretation* [9]. The first step checks for functional equivalence between model and code. The second step, supported by the Polyspace tool [10,25], is used to assess the absence of runtime errors. Due to the timing of the project, both guidelines verification and model-based testing have been only partially employed. The process had to be adjusted with *ad-hoc* solutions, mostly based on traditional code testing, in order to address the problem of a non-certified code generator.

Project 3, concerning an ATP system tailored for metro signalling, has been the first complete instance of a formal development process [17,16]. Besides the already adopted technologies, a hierarchical derivation approach has been employed. Simulink and Stateflow are proper tools to represent the low-level aspects of a system, while they offer poor support for reasoning at the software architecture level. Therefore, we resorted to adopt the UML notation to model the software architecture of the system. The approach starts from such an UML

[1] The tool is currently distributed as part of Simulink Coder.

[2] Set of guidelines developed in the automotive domain for modelling with Simulink/Stateflow. The current version of the MAAB recommendations is 2.2, issued in 2011. The project adopted the 2.0 version, issued in 2007.

representation, and requires deriving unit-level requirements and a formal representation of them in the form of Stateflow diagrams. During the project, the modelling guidelines have been updated with a set of restrictions particularly oriented to define a formal semantics for the Stateflow language (see Sect. 3.1). These restrictions have enforced the formal representation of the requirements.

A new code generator, Real Time Workshop Embedded Coder[3], has been introduced, which permitted to generate also the integration code between the different generated units. With the previously adopted Stateflow Coder, code integration was performed manually. The adoption of the new generator allowed a further automation and speed-up of the development.

Within the project, 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* [8] with abstract interpretation. Translation validation has been performed with an internally developed framework. This framework supports *back-to-back* [34] model-code execution of unit level tests, and the assessment of consistency between model and code coverage. Abstract interpretation with Polyspace has been performed with a strategy analogous to the one already applied for Project 2.

Project 3 has also marked the start of the first structured experiments with formal verification by means of Simulink Design Verifier [25]. The evaluation was particularly oriented to verify whether this technology, employed at the level of the model units, could actually replace model-based unit testing with a substantial cost reduction. The first results have been encouraging. The experiments have shown that about 95% of the requirements can be verified with the tool to achieve a cost reduction of 50% to 66% in terms of man/hours [16]. The remaining requirements, for which this cost gain cannot be achieved, can be verified through model-based testing. The company is currently devising strategies to systematically employ formal verification in the development process.

Many of the development, verification and certification issues related to formal model-based development appeared only during its actual deployment: the goals planned at the beginning of the research have been addressed after progressive tuning of the strategy across the different projects. Table 1 summarizes the technologies incrementally introduced during the projects. *Italics* indicate partial adoption of a technology or partial achievement of a goal.

3.1 Goal 1 - Modelling Language Restriction

The first goal was to identify a proper subset of the Simulink/Stateflow language: the idea was that C code generated from models in this subset would be compliant with the guidelines defined by the company in accordance with the quality standard required by the norms. With Project 1, this problem was addressed by first analysing the violations of the quality standard issued by the code generated from the original model. Then, sub-models have been defined, on which the evaluation could be performed more easily. The translation of

[3] Currently renamed Simulink Coder.

Table 1. Summary of the results achieved during the projects

Year	**Project**	**Technologies** (Full or *Partial* Adoption)	**Goal**
2007-2008	Project 1	Modelling guidelines (25) *Code generation (Stateflow Coder R2007b)*	1
2008-2010	Project 2	Modelling guidelines + MAAB (43) Code generation (Stateflow Coder R2007b) *Guidelines verification* *Model-based testing* Abstract interpretation (Polyspace 7.0)	1 *2* *3*
2009-2011	Project 3	Modelling guidelines + MAAB (43) Semantics restrictions UML + hierarchical derivation Code generation (RTW Embedded Coder R2010a) Translation Validation Abstract interpretation (Polyspace 8.0) *Formal Verification (Simulink Design Verifier R2010a)*	1 2 3

single graphical constructs, and of combination of them, have been evaluated and classified. Proper modelling guidelines have been defined in order 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 [15].

With Project 2, where code generation has been actually employed for the development of the whole application logic software, a more systematic study has been performed. The preliminary set of guidelines 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 needed. Actually, in the automotive sector a set of accepted modelling rules equivalent to the MISRA [28] ones for C code had emerged, that is, the MAAB guidelines [24], defined by OEMs[4] and suppliers of the automotive sector to facilitate model exchange and commissioning. The preliminary set was extended by adapting the MAAB guidelines to the railway domain. This new set, composed of 45 guidelines in total, prompted further restrictions. These restrictions were not only limited to enforce generation of quality code, but were also oriented to define well-structured models.

A further step was performed during Project 3: in order to ease a formal analysis and a formal representation of the requirements, it was decided to complete the modelling style guidelines by restricting the Stateflow language to a semantically unambiguous set. To this end, the studies of Scaife et al. [31], focused on translating a subset of Stateflow into the Lustre formal language, have been used. These studies brought to the definition of a formal semantics for Stateflow [17], which constrains the language to an unambiguous subset. A set of guidelines has been defined for enforcing the development of design models in accordance to this subset of the language. The models produced are semantically independent from the simulation engine, and a formal development process could actually take place.

[4] Original Equipment Manufacturers.

3.2 Goal 2 - Generated Code Correctness

The second goal was to address the problem of a non-certified, neither proven-in-use, translator. The objective was to ensure the code to be fully compliant with the model behaviour, and to guarantee that no additional improper functions are added during the code synthesis phase. The approach adopted, preliminarily defined during Project 2, but refined and fully applied only on Project 3, consisted in implementing a model-based testing approach known as *translation validation* [8], and completing it with static analysis by means of abstract interpretation [9].

Translation validation 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. 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 and justified.

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. 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. The main feature of the tool is to detect runtime errors by performing static analysis of the 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 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 within Project 2, a two-steps procedure has 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.

3.3 Goal 3 - Process Integration

The third goal was integrating the modelling and code generation technologies into a coherent development process. Also concerning this issue, a sound process was finally achieved only with Project 3, after incremental adjustments. The

introduction of modelling and the need to ensure consistency between models and code, has prompted changes also to the verification and validation activities, which had to be tailored according to the new technology. On the other hand, it has allowed working 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 Fig. 1. The process embeds two verification branches: one for the activities performed on the models, the other for the tasks concerning source code and system.

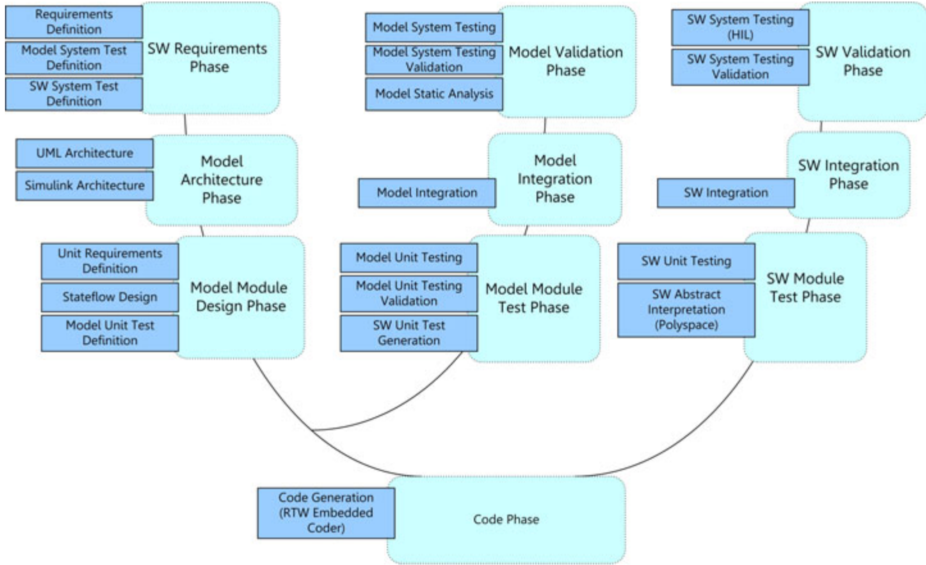

Fig. 1. Overview of the formal model-based development process adopted

From system-level software requirements, tests are defined to be perfomed both at model-integration level and at system-level (**SW Requirements Phase**). Then, a UML architecture is defined in the form of a component diagram. The diagram is then manually translated into a Simulink architecture (**Model Architecture Phase**). During the design phase, system requirements are decomposed into unit requirements apportioned to the single architectural components. Furthermore, the Stateflow models are defined according to these unit requirements, following the style-guidelines and the semantics restrictions (**Model Module Design Phase**).

Functional unit testing (**Model Module Test Phase**) and system testing (**Model Validation Phase**) are performed on the models by using the Simulink simulator before generating code through RTW Embedded Coder (**Code Phase**). After code generation, translation validation is performed, followed by static analysis by means of abstract interpretation (**SW Module Test Phase**). The application code is then integrated with operating system and drivers (**SW Integration**

Phase), and *hardware-in-the-loop* (HIL) is used to perform system tests according to the system requirements (**SW Validation Phase**). The whole process is supported by coherent documentation: this is auto-generated by means of Simulink Report Generator, a Simulink toolbox, using the comments edited by the developers on the models.

4 Lessons Learnt

Code generation was introduced following the intuition that defining a formal model of the specifications, and automatically producing code, allows speeding-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.

Abstraction. Models require working 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 Project 2 and Project 3, gives the advantage of defining test scenarios at component level without disrupting the model structure.

Expressiveness. Graphical models are closer to the natural language requirements. At the same time, they are an unambiguous mean to exchange or pass artefacts among developers. This observation has been enlightened by the Project 3 experience, where the project passed from the hands of its first main developer to another developer within one month only

Cohesion & Decoupling. The generated software is composed by modules with higher internal cohesion and better decoupling. Interfaces among functionalities are based solely on data, 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 when developing with time-to-market constraints.

Uniformity. The generated code has a repetitive structure, which 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. Therefore, 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 procedure (see Sect. 3.2) resulted profitable on the generated code only, since systematic analysis on hand-crafted code was made harder by its variable structure and programming style.

Traceability. Software modules are directly traceable with the corresponding 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 defect, that produced it. The formal development approach introduced, with the support of RTW Embedded Coder,

has allowed defining navigable links between the single code statements and the requirements.

Control. The structured development has given greater control over the components, producing in the end software with less bugs already before the verification activities, as witnessed by the bug reduction evaluation measured during Project 3 (from 10 to 3 bugs per module) [17].

Verification Cost. When passing from traditional 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% [18]. The recent experiments with formal verification have shown that this cost can be further reduced by 50-66% [16].

The main drawback encountered in introducing code generation has been the *size* and overall *complexity* of the resulting software. Though these aspects were not complicating the verification activities, they 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. The reaction time is influenced by the main execution time, which resulted four times higher in the first experiments. In the discussed case, the hardware upgrade actually solved the problem. However, with the design of new, more complex systems, this issue has to be taken into account while defining the hardware architecture.

The hardware designer shall consider that the code is larger in size, and there is less flexibility in terms of optimizations at source level (we recall that optimizations at compiler level are not recommended for the development of safety-critical 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 have 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.

Preliminary experiments with formal verification applied at unit-level have shown that this technology might considerably reduce the verification cost for the majority of the requirements. However, further analysis is required before introducing formal verification as part of the process.

Some lessons have been learned also from the *knowledge transfer* point of view. The research activity has been performed according to the following research management model.

On one side there is a research assistant who comes from the university and is fully focused on the technology to be introduced. On the other side there is an internal development team, which puts the research into practice on real projects when the exploratory studies are successful.

The results obtained across these four years would have not been possible through intermittent collaborations only. Moreover, they would have been hardly achieved if just an internal person would have been in charge of the research. In order to separate the research from the time-to-market issues, the independence

of the research assistant from the development team has to be preserved. Large companies can profit from dedicated internal research teams, or even entire research divisions. Instead, medium-size companies often have to employ the same personnel for performing research explorations, which are always needed to stay on the market, and for takeing care of the day-by-day software development. We argue that the research management model adopted in the presented experience, based on an academic researcher independently operating within a company, can be adapted to other medium-size companies with comparable results.

5 Conclusion

The research activity reported in this paper started with the objective of introducing the formal design and code generation technologies within the development process of a railway signalling manufacturer. At the end of the experience, these techniques have radically changed the whole process in terms of design tasks and in terms of verification activities. In particular, formal model-based design has opened the door to model-based testing, has facilitated the adoption of abstract interpretation, and has allowed performing the first successful experiences with formal verification.

This methodology shift required four years and three projects to be defined and consolidated. Most of the implications of the introduction of code generation could not be foreseen at the beginning of the development, but had to be addressed incrementally. This tuning has been facilitated by the flexibility of the toolsuite adopted: given the many toolboxes of Matlab, there was no need to interface the tool with other platforms to perform the required software process tasks (e.g., test definition, tracing of the requirements, document generation)[5].

However, we believe that the success of the experience has been mainly driven by the research management model followed. The presence of an independent reseacher operating within the company has been paramount to ensure that research was performed without pressure, while research results were properly transferred to the engineering team. The experience showed that also for medium-size companies, such as the one considered in this paper, it is possible to perform research when a proper model is adopted.

Research is essential 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.

The considerations made in this paper have mainly concerned a formal model-based design process based on commercial tools, in a given application domain.

[5] The reader can note that most of the tool support referred in this paper comes from a single vendor. It is not at all the intention of the authors to advertise for such vendor. However, we have to note that interfacing with a single vendor is a preferential factor for industry, and in this case has influenced the choice of the tools.

Assuming different tools and different application domains, it is not so immediate that the same considerations still hold. As future work, we are launching the study of a similar development process based on UML-centered tools: in this case the flexibility will not be given by an integrated toolsuite, but by the Unified Modelling notation itself, even if open-source or free tools will be adopted. Different application domains will be addressed as well.

Acknowledgements. The authors thank Daniele Grasso and Gianluca Magnani from General Electric Transportation Systems for the effort put in several of the reported experiences. This work has been partially funded by General Electric Transportation Systems with a grant to University of Florence.

References

1. Adler, R., Schaefer, I., Schuele, T., Vecchié, E.: From Model-Based Design to Formal Verification of Adaptive Embedded Systems. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 76–95. Springer, Heidelberg (2007)
2. Bacherini, S., Fantechi, A., Tempestini, M., Zingoni, N.: A Story About Formal Methods Adoption by a Railway Signaling Manufacturer. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 179–189. Springer, Heidelberg (2006)
3. Behm, P., Benoit, P., Faivre, A., Meynadier, J.-M.: Météor: A Successful Application of B in a Large Project. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 369–387. Springer, Heidelberg (1999)
4. Bochot, T., Virelizier, P., Waeselynck, H., Wiels, V.: Model checking flight control systems: The Airbus experience. In: ICSE Companion, pp. 18–27. IEEE (2009)
5. Bowen, J.P., Hinchey, M.G.: Ten commandments of formal methods. IEEE Computer 28(4), 56–63 (1995)
6. Bowen, J.P., Hinchey, M.G.: Ten commandments of formal methods...ten years later. IEEE Computer 39(1), 40–48 (2006)
7. CENELEC. EN 50128, Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems (2011)
8. Conrad, M.: Testing-based translation validation of generated code in the context of IEC 61508. Formal Methods in System Design 35(3), 389–401 (2009)
9. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
10. Deutsch, A.: Static verification of dynamic properties. Polyspace Technology, white paper (2004)
11. Dormoy, F.X.: Scade 6: a model based solution for safety critical software development. In: ERTS 2008, pp. 1–9 (2008)
12. dSPACE. Targetlink (December 2011), `http://www.dspaceinc.com`
13. El-Far, I.K., Whittaker, J.A.: Model-based software testing. Encyclopedia of Software Engineering 1, 825–837 (2002)
14. ETAS. Ascet (December 2011), `http://www.etas.com`
15. Ferrari, A., Fantechi, A., Bacherini, S., Zingoni, N.: Modeling guidelines for code generation in the railway signaling context. In: NFM 2009, pp. 166–170 (2009)

16. Ferrari, A., Grasso, D., Magnani, G., Fantechi, A., Tempestini, M.: The Metrô Rio ATP Case Study. In: Kowalewski, S., Roveri, M. (eds.) FMICS 2010. LNCS, vol. 6371, pp. 1–16. Springer, Heidelberg (2010); journal special issue (to appear, 2012)
17. Ferrari, A., Grasso, D., Magnani, G., Fantechi, A., Tempestini, M.: The Metrô Rio ATP Case Study. In: Kowalewski, S., Roveri, M. (eds.) FMICS 2010. LNCS, vol. 6371, pp. 1–16. Springer, Heidelberg (2010)
18. Ferrari, A., Magnani, G., Grasso, D., Fantechi, A., Tempestini, M.: Adoption of model-based testing and abstract interpretation by a railway signalling manufacturer. IJERTCS 2(2), 42–61 (2011)
19. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987)
20. Hinchey, M.G., Bowen, J.: Applications of formal methods. Prentice-Hall (1995)
21. Huber, F., Schätz, B., Schmidt, A., Spies, K.: Autofocus: A Tool for Distributed Systems Specification. In: Jonsson, B., Parrow, J. (eds.) FTRTFT 1996. LNCS, vol. 1135, pp. 467–470. Springer, Heidelberg (1996)
22. IEC. IEC-61508, Functional safety of electrical/electronic/programmable electronic safety-related systems (April 2010)
23. INRIA. Scicos: Block diagram modeler/simulator (December 2011), `http://www.scicos.org/`
24. MAAB. Control algorithm modeling guidelines using Matlab, Simulink and Stateflow, version 2.0 (2007)
25. MathWorks. MathWorks products and services (December 2011), `http://www.mathworks.com/products/`
26. Meenakshi, B., Bhatnagar, A., Roy, S.: Tool for Translating Simulink Models into Input Language of a Model Checker. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 606–620. Springer, Heidelberg (2006)
27. Miller, S.P., Whalen, M.W., Cofer, D.D.: Software model checking takes off. Commun. ACM 53(2), 58–64 (2010)
28. MISRA. Guidelines for the use of the C language in critical systems (October 2004)
29. Mohagheghi, P., Dehlen, V.: Where is the Proof? - A Review of Experiences from Applying MDE in Industry. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 432–443. Springer, Heidelberg (2008)
30. RTCA. DO-178B, Software considerations in airborne systems and equipment certification (December 1992)
31. Scaife, N., Sofronis, C., Caspi, P., Tripakis, S., Maraninchi, F.: Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. In: EMSOFT, pp. 259–268. ACM (2004)
32. Selic, B.: The pragmatics of model-driven development. IEEE Software 20(5), 19–25 (2003)
33. Tretmans, J., Wijbrans, K., Chaudron, M.R.V.: Software engineering with formal methods: the development of a storm surge barrier control system revisiting seven myths of formal methods. Formal Methods in System Design 19(2), 195–215 (2001)
34. Vouk, M.A.: Back-to-back testing. Inf. Softw. Technol. 32, 34–45 (1990)
35. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.S.: Formal methods: Practice and experience. ACM Comput. Surv. 41(4) (2009)