Certification of Model-based Code Generators – Open Problems and Possible Solutions

Ingo Stürmer

Certification of Model-based Code Generators – Open Problems and Possible Solutions

Ingo Stürmer[1], Model Engineering Solutions, Friedrichstr. 50, 10117 Berlin, Germany

Abstract: Model-based development and automatic code generation have become an established approach in embedded software development for both the automotive and avionics sectors. The use of a code generator can lead to significant improvements in productivity in the software implementation phase. Moreover, early quality assurance at the model level can lead to a higher level of code quality. However, automotive or avionic software is very often deployed in safety-critical systems and as a result, may not contain errors. In this context it is crucial that the use of a code generator and its tool chain (editor, compiler, linker, loader, etc.) does not incorporate errors in the target system and leave them undetected. In general, this cannot be fully avoided even when using a code generator proven to be 'correct-by-construction'. Inappropriate modeling or the faulty configuration of the code generator could, for example, lead to erroneous generated code. This paper discusses how code generators and generated code can be safeguarded by means of tool certification (also termed qualification in the avionics sector) in respect to safety standards that are relevant for the automotive and avionics sectors. Specific, tool-related problems will be discussed and illustrated with practice-relevant examples; possible solutions for safeguarding model-based code generators will be presented.

Keywords: Tool certification, qualification, model-based development, test suite, guidelines, checks.

1. Model-based Code Generation

In model-based development (MBD), the seamless use of executable models is characteristic of function and control system design and the subsequent implementation phase. This means that models are used throughout entire control system development: from the preliminaries to the detailed design. Such models are designed using popular graphic modeling languages, such as Simulink and Stateflow from The MathWorks [1].

In the first design stage, a *physical model* is created, which is derived from the requirements specification (see Fig. 1). The physical model describes the behavior of the control function to be developed, containing transformation algorithms related to continuous input signals as well as incoming events or states. These algorithms are usually described using floating-point arithmetic.

Since the physical model focuses on the design of the control function and on checking its functional behavior with regard to the stated requirements, it cannot serve directly as a basis for production code creation. Implementation details, which are the prerequisite for automatic coding, are not considered here. Therefore the physical model needs to be manually revised by implementation experts with regard to the requirements of the production code (e.g. function parts are distributed between different tasks). For example, in order to enhance the model from a realization point of view, the floating-point arithmetic contained in the physical model is adjusted to the fixed-point arithmetic used by the target processor. If fixed-point arithmetic is used, the model must be augmented with necessary scaling information in order to keep imprecision in the representation of fixed-point numbers as low as possible. Apart from the change in the type of arithmetic, it may be necessary to substitute certain model elements that are not part of the language subset supported by a particular code generator. Furthermore, it is often necessary to restructure the behavioral model with regard to a planned software design.

The result of this evolutionary reworking of the physical model is what we call an *implementation model*. The implementation model can be used as a basis for (A) manual coding by a software developer (not shown in Fig. 1), or (B) automatic code generation with a code generator (see Fig. 1). The implementation model contains all the information that is needed for code generation and enables the creation of efficient C code by the code generator. Code *efficiency* is vital, due to the limited resources of the embedded system running the generated code. Code generators that are capable of translating MATLAB Simulink and Stateflow models into efficient production code include TargetLink [2][1] and the Real-Time Workshop/Embedded Coder [1].

Depending on the development stage and purpose, the code is generated for the development computer, in most cases a standard PC (Fig. 1, right). In this case, a classical compiler / linker combination is

[1] TargetLink uses its own graphical notation for code generation (TargetLink blockset), which is based on the Simulink modeling language.

used to translate the generated code into an executable. For the target hardware (typically an evaluation board similar to the embedded system), a cross-compiler is required. Here a linker and loader build and load the binary code onto the embedded device. The tool chain established by the modeling tool (editor and simulator), the tools for model-to-code translation (e.g. code generator, (cross-) compiler, linker, loader), and finally the target hardware itself make up the code generation tool chain (Fig. 1).

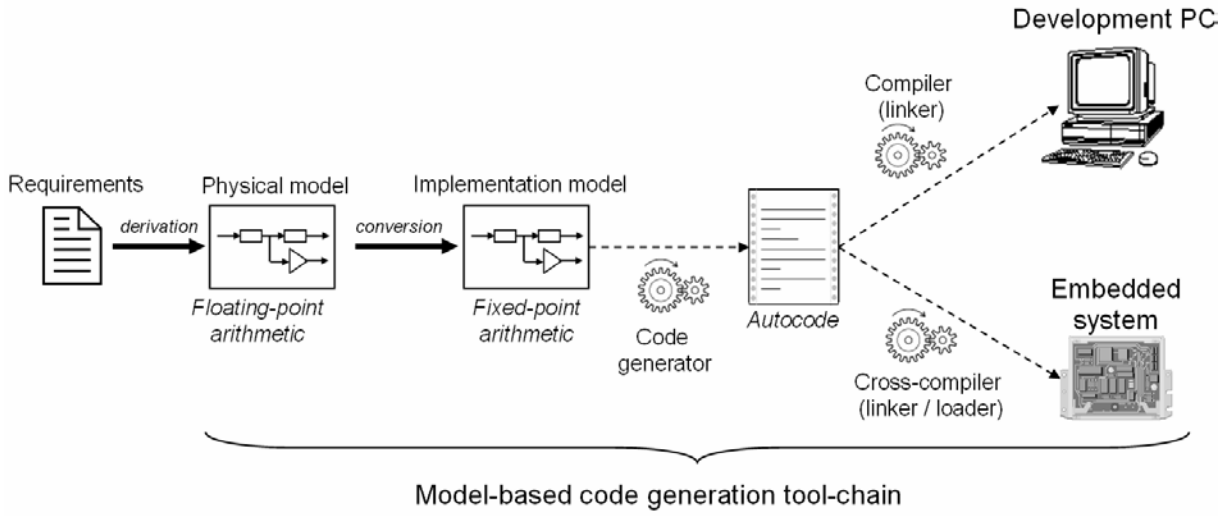

Figure 1: Model-based code generation

Model-based code generation is one of the principal advantages of model-based development. The use of a code generator leads to significant productivity improvements in the software implementation phase. Individual studies have shown a reduction in software development time by up to 20% through code generation [5]. If the manual verification process at the code level can also be reduced, savings of up to 50% are reported. This conforms with internal information provided by other users. In summary, productivity can increase by up to 50% compared with traditional manual coding. Moreover, the level of quality gained by early quality assurance at the model level can lead to high quality code, provided that the code generator works reliably. As a result of these characteristics, there is a strong industrial demand for code generators.

Some people still express reservations about using a code generator for safety-related software development. This may be due to the fact that code generators are not as mature as established C or ADA compilers. The technological risk of a code generator can be high, because they (1) are used by a relatively small group of developers, and (2) face a high rate of technological innovation causing new versions to appear in short cycles. As a result, formal proof of code generator correctness is unfeasible in practice. For this reason, productivity improvements that can be achieved with model-based code generation tools cannot be fully exploited. The generated code must still be checked with the same time and cost-intensive effort as manually written code, even though intensive quality measures have already been spent on the model. A survey of quality assurance methods for model-based development with code generation is provided in [3].

Model-based code generation is a new paradigm that makes new demands on the development process and poses additional questions concerning the certification of systems. An important and generally accepted approach for increasing confidence in code generation is to use a certified or qualified code generator. Apart from correctness aspects, there are many benefits to be expected from using a certified code generator:

- Higher quality of generated code (number of errors, readability, traceability) in comparison to manual code.
- Consistent level of code quality.
- Consistency between specification and software.
- Reusability of executable specifications in subsequent projects.
- Internal development of software in organizations / departments with an insufficient number of implementation specialists.
- More efficient realization of the implementation phase in software development.

Code generation relies on software that can obviously never be completely free from faults or errors. This is the main reason for the ongoing debate regarding the use of code generation for safety-related systems. However, the use of code generators is recommended from the point of view of

the development process, and there is some advice to be found in the different safety standards (elaborated in Chapter 4). We will, however, first focus on specific model design problems, which directly influence the quality of the code generated from the implementation model.

2. Problems with Model-based Code Generation

There are many sources of error that can be identified within the model-based code generation process. One example are *design errors*, which are caused by inappropriate design of the (physical) model with respect to its functional requirements or due to misunderstandings regarding the semantics of the modeling language (a survey on all possible errors in the model-based code generation tool chain is provided in [21]). The misuse or misunderstanding of a modeling language used for code generation will be discussed next.

Example 1

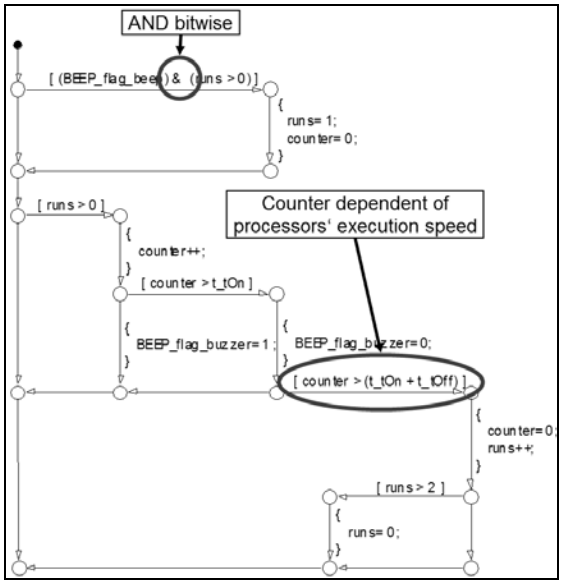

Figure 2: Buzzer realized as a flowchart (variant A)

The following two realizations of a buzzer exemplify the need for quality assurance methods on the model design level. The first realization (Figure 2) was created at an early development stage of a typical embedded software development project for an automotive system. The model contains several problems that are not obvious at first glance (especially for flowchart novices). However, the two possible and semantically comparable realizations of a buzzer are presented in Figure 2 (flowchart, variant A) and Figure 3 (Stateflow, variant B). The generation of an acoustic warning (beep) is controlled by the flag BEEP_flag_beep. When initially activated, the flag for beep activation BEEP_flag_buzzer is turned to true (1) during t_tON. After t_tON the flag is turned back to false (0). The buzzer beeps three times. In variant A, which was created at an early development stage of the project, the buzzer is realized as a flowchart diagram that typically consists of transitions and connective junctions. A connective junction defines a decision point between possible paths of a transition, whereas a transition can bear a complex label for checking specific conditions (e.g. runs > 0) and for performing actions (e.g. counter++;). This modeling paradigm is comparable to if-then-else control structures used in e.g. C. The appropriate use of flowcharts for automatic controller code implementation often produces efficient C code with little overhead. Modeling with flowcharts, however, allows quasi free C code programming. The assignments in the curly braces, for example, can contain an arbitrary number of C statements. As a consequence, classical programming errors can occur, such as the faulty use of a bitwise AND (&) instead of a logical one (&&). Such human sources of error are often neglected or underestimated in the literature, e.g. [7]. Inappropriate modeling must be considered as an additional error source. Timing behavior, for example, such as the fractions of time the buzzer is activated or disabled, is realized by incrementing the local counter variable. This kind of modeling temporal behavior is often used by Stateflow novices. However, when using the model for production code generation, such modeling of temporal behavior implies intrinsic errors. Timing is no longer independent of the processor's execution speed or the time for computations assigned by the operating system respectively. In variant A, the flowchart realization of the buzzer is more technical rather than intuitive, since the functionality of the buzzer is, in principle, state-based. Essentially, the expected improvements in efficiency on the code level are largely compensated for by the aforementioned disadvantages.

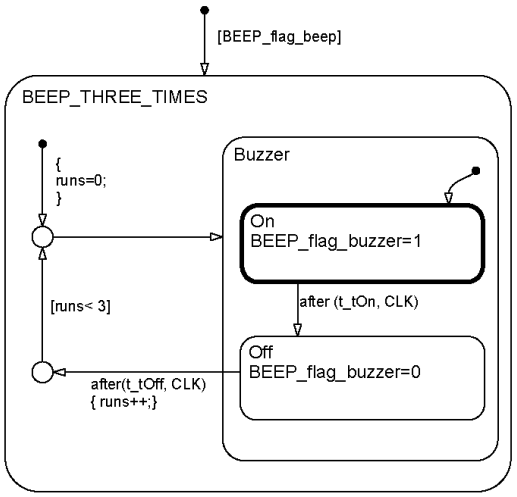

Figure 3: Buzzer realized in Stateflow (variant B)

The realization of the buzzer as a Stateflow state machine (Figure 3) is easier to understand intuitively. In variant B, temporal behavior is modeled by using Stateflow's after operator, which allows a precise definition of how long a state is activated, e.g. when the transition from state On to Off is executed. Timing is 'incremented' by using the Stateflow CLK event. In doing so, timing is independent of the processor's execution speed or resource allocations of the operating system. In contrast to variant A, this variant is easier to parameterize (e.g. for double beeps). In summary, we can see that on the one hand, modeling by using states and events increases the understandability and readability of the model, while on the other hand, the use of states and events is problematic in projects where resource consumption (e.g. required RAM/ROM size) is of decisive importance, since both produce a large overhead.

For the reasons stated above, a compromise between understandable modeling and code efficiency must be found. Modeling guidelines must clarify the advantages and disadvantages of different modeling techniques and patterns. It is also advisable to use a 'safe-subset' of the modeling language in order to prevent problematic patterns in generated code, which cannot be fully avoided even when using a code generator that is considered to be 'correct-by-construction'. An example for this statement is given next.

Example 2

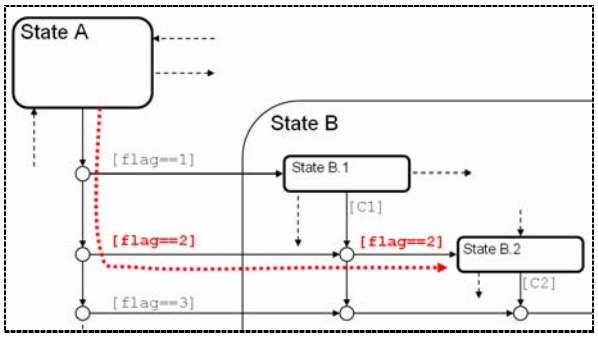

Figure 4: Stateflow chart with potential 'dead' transition path

Figure 4 shows a simplified detail of a complex Stateflow chart. This chart consists of 2 super states (A and B) as well as two sub-states (B1. and B.2). There are several transition paths originating from state A. One path goes from state A to sub-state B.2 (highlighted with a red dashed line). Within this pathway, two conditions are checked whether the condition *flag* equals two. At first glance, this way of modeling a control flow does not seem problematic. But a correct code generator translates the pattern shown in Figure 4 into the code pattern shown in Figure 5.

As we can see, the code generator generated a correct, but not optimized, code pattern that identically reflects the behavior of the model. Only analytical quality assurance methods on code level, e.g. structural testing or a code review, can detect such problematic patterns on code level. This means

```
If (flag==2){
     if (flag==2){
          State_A=0;
          State_B2=1;
          State_B2_en();
     } else {
          State_A=0;
          State_B2=1;
          State_B2_en();
     }  // DEAD CODE!!!
}
```

Figure 5: Code pattern generated from Figure 4

that even having a correct or certified code generator does not imply that quality assurance methods on code level can be fully omitted. A typical solution to prevent the use of such modeling patterns is the application of modeling guidelines and checking the models with regard to modeling guideline compliancy (elaborated in Chapter 5.2).

3. Code Generation for Safety-related Systems

The trend towards increasing model-based application development also extends to safety-relevant systems. However, more stringent developmental requirements must be fulfilled if such systems should be subject to certification. The principal expectation is that the methods used to design the system are all state-of-the-art, thus everything possible has been done to ensure that safety is guaranteed. General requirements on the development of safety-related systems are usually summarized in standards, such as DO-178B for avionic software, or IEC 61508 for E/E systems. In addition to providing certification of the system under development, these standards serve to define more or less specific demands for using specific tools for automating development steps.

Both software tools and development tool chains offer potential sources of error. Normally errors produced by development tools have no direct influence on the product itself. However, this is not the case for tools like compilers and code generators. When developing safety-relevant systems, it is particularly important to ensure that the use of a code generator and its tool chain (editor, compiler, linker, loader, etc.) does not incorporate errors into the target system and leave them

undetected. Certification institutions therefore never certify source code on its own, but only in connection with the corresponding machine codes and usually when integrated in the whole system.

3.1. Advantages of Code Generator Certification

This is not sufficient reason to dispense with code generation in the first place. Other reasons, especially those relating to process improvement and development efficiency, underscore the need for qualified code generators:

1. We are faced with continual tool improvement: this means a high rate of updates, since code generation for production is a relatively new technology.
2. Taking this into account, the generated output of the tool has to be checked with the same time-consuming effort as code that has been written manually, even though the types of errors produced by code generation are completely different to those found in manual coding.
3. This drastically reduces potential improvements in productivity that can be brought about by automatic code generation.

In general, the qualification of a software tool has the objective of compiling evidence of the tool's high-quality output, thus ensuring the use of the tool in the development of systems with safety-related aspects.

Advantages that can be expected from the application of a qualified code generator, provided that a 'safe subset' of the modeling language is used(!), are as follows:

- Decreased overall effort spent on verification and validation activities, e.g. (a) eliminating source code reviews for generated code parts, and (b) eliminating or minimizing the effort needed for unit testing.
- Improved productivity of software development, especially in the software implementation phase.
- Ensuring the controllability of rapid iterations in development.
- Consistent or higher code quality despite the aforementioned points.

How should we therefore safeguard the deployment of a code generator in the model-based development process? In such a way that allows errors induced into the development product to be discovered or completely circumvented, without resulting in a considerable reduction in efficiency of the development process.

This is answered in part by existing safety standards, such as IEC 61508 [4] and DO-178B [12]. However since additional measures should be considered for safeguarding code generation, this is discussed in the next chapter.

4. Safety Standards Applicable for Tool Certification

The various applicable safety standards that we use today were published some time ago, so they give no explicit guidance for the application of model-based development and the utilization of code generation. Future versions such as RTCA/DO-178C will be expanded with respect to this development paradigm. In any case, the general requirements with respect to product certification or development tool qualification can be adopted. The latter aspect is of major interest here.

4.1 RTCA/DO-178B

RTCA/DO-178B (*Software Considerations in Airborne Systems and Equipment Certification Requirements*) is an international standard, which is mandatory in all avionic software development. This standard focuses on the software development process.

The qualification of a development tool can be handled in a similar way to the certification of the application software itself. In other words, qualifying a development tool such as a code generator does not mean proving its correctness. Instead, it is merely important to gain sufficient confidence in its correctness. In this context, the prerequisite for qualification is an assessment, which shows the conformance of the tool development process with the development requirements of the standard. Tool qualification is closely connected to the development and certification of a system and its criticality. This means that a tool that is issued with all its development documents in conformity with the standard could be seen as a qualifiable tool.

4.2. IEC 61508

The International Electrotechnical Commission (IEC) has submitted the international safety standard IEC 61508 [4], which is primarily concerned with safety-related control systems incorporating electrical / electronic / programmable electronic safety-related systems (E/E/PES). It also provides a framework, which is applicable to safety-related systems, irrespective of the technology on which those systems are based (e.g. mechanical, hydraulic, or pneumatic). For specific demands, the standard has been adapted to a particular industry as is currently the case in the automotive industry for certifying safety-related (software-based) E/E/EPS parts.

In the context of IEC 61508, only compilers (translators) have so far been regularly subject to tool certification procedures. Compilers are assessed in two different ways: (1) the compiler or translator is certified against its respective language or process standards, (2) compilers or translators are assessed by their increased confidence from use

(fitness for purpose). In general, however, compilers are not certified in any way with regard to safety[2].

4.3. Examples of tool Certification w.r.t. Safety Standards

Qualifiable code generators *w.r.t. DO-178B*, such as SCADE, which endorse a certification of the application software, do exist. They make it possible to reduce the total amount of the verification activities, without allowing them all to be omitted completely. However, their source language is not as popular as Simulink / Stateflow, and they perform only a limited number of optimizations. The latter is crucial for the automotive sector, where developers must deal with strict resource limitations

ASCET-SD [LBB+97], developed by ETAS, is the first code generator for automotive embedded control systems that is certified as being fit for its purpose for SIL 3 according to IEC 61508 [10]. To gain this certificate, TÜV inspectors performed an in-depth analysis of the ETAS tool chain to understand the purpose of its use and the tool's development process. Based on this knowledge, the inspectors created a test plan according to IEC 61508 SIL 3. This test plan aims to access the 'fitness for purpose' of the code generator and includes, for example, formal characteristics of the documentation, software requirements specification, the test as part of design, development and integration, verification and validation (V&V). Following this test plan, the tool developer could show, for instance, "the existence of conclusive evidence for correct code generation."

5. Possible Solutions for Code Generator Certification

Certification authorities such as the technical supervisory agencies TÜV (Technischer Überwachungsverein / German Technical Inspection Authority) or the British Standards Institution generally consider the qualification of an optimizing compiler for safety-relevant software as sophisticated. On one hand, it is a particularly complex tool and on the other, its behavior when applied to optimizations cannot be clearly comprehended. Code generators can be viewed in the same way, since the main focus of their work is on code optimization, at least in automotive applications. Moreover, code generators are not as mature as established compilers, which have been proven to be reliable in use and whose correct functioning is constantly validated by numerous programmers.

Challenges from the tool supplier and the development management point of view are as follows:

- At present, insufficient regulations and a lack of common rules for tool qualification hinder the tool supplier in defining an approved, long-term validation strategy.
- A prerequisite for a valid qualification process is that all development information must be available. However, tool suppliers are usually unwilling to make their internal know-how public.
- The focus of a tool supplier's work is on the development itself and not on qualification procedures.
- The qualification procedure is conducted for a particular tool. This means that qualifying a code generator does not consider the compiler, linker, etc. These tools either need to be qualified themselves or their results have to be checked manually each time they are used.
- The costs of a qualification procedure are fairly high, at least when following the RTCA DO-178B approach.

Considering the aforementioned qualification issues, there is obviously a need to define different approaches to efficiently safeguard code generation.

5.1. Code Generator Certification via a Test Suite

Since the use case of code generators is quite similar to that of compilers, it could be helpful to adapt the current practice for compiler validation.

The utilization of test suites for the certification of compilers has been proven in use for Ada and C compilers. Although this does not fulfill the qualification requirements stipulated in DO-178B, it appears to be an efficient way of facilitating sufficient compliance with quality and reliability requirements. A general approach for validating a code generator (including examples) is documented in [8]. This approach could be used, for example, to generate those parts of such a test suite, which focus on optimization criteria. An important initiative of automotive manufacturers and suppliers are currently working on a test suite for code generators, see [9].

The advantages of utilizing a test suite as a qualification approach for code generators are as follows:

- Cost-effective qualification and validation approach.
- Modification and patches of the tool can be handled; 'Delta Certificates' can be assessed more easily and at lower cost compared to a complete qualification.
- The input language of the code generator must not necessarily be considered completely. A subset can be defined and incrementally expanded.
- A common view of the content and application of the test suite could emerge. At least if the major

[2] [4], part 7, Appendix C4.3

constraints and a basic set of tests become public and even standardized.

5.2. Use of a 'Safe Subset'

IEC 61508-3, for example, highly recommends the use of a language subset for software development in accordance with SIL3 and SIL4. However, since IEC 61508 was written with traditional, (hand-coded) software development processes in mind, the standard does not cover advanced embedded software development technologies such as model-based design and production code generation. A "safe subset" for modeling languages such as Simulink or Stateflow is often defined in general accepted model design guidelines such as those defined by the MathWorks Automtoive Advisory Board [14]. However, the MAAB guidelines do not directly address model patterns for code generators. Publicly available guidelines such as the MAAB guidelines are often supplemented by in-house sets of modeling guidelines, for example the Daimler *Model-based Development Guidelines* [15]. These are are hosted via the *e-Guidelines Server* [16], a web-based infrastructure for publishing and centrally administering different sets of guidelines. The Daimler guidelines provide a sound basis for checking the models used for production code generation. The adoption of such guidelines can significantly improve the efficiency of generated code.

Apart from having guidelines for model-based development available, it is also important to check and ideally repair models with regard to guideline compliancy automatically. Static model analysis tools exist for the automatic verification of models with respect to modeling guideline compliancy. The purpose of these tools is to establish a framework in which to perform checks. The check itself is a program that verifies a particular guideline.

Commercial static model analysis tools that are available for MATLAB Simulink, Stateflow, and TargetLink models are the Simulink *Model Advisor* [17], *MINT* [18], or the *Model Examiner* (M-XAM) [19]. With such tools, model analysis is carried out by executing a selection of MATLAB M-scripts. The checks either come with the tool or can be supplemented as user-written checks. The results of model verification using the checks are compiled into reports. These show all detected guideline violations linked to the model. In a final step, the faulty model patterns must be investigated and, where necessary, corrected by the developer: a highly time-intensive and error-prone task. Case Studies have shown, that this task can be significantly reduced up to 50% by using automatic model repair functionality with tools such as the *Model Examiner* or *MATE* [20]. Up to 90% of model rework with regard to modeling guidelines compliancy can be saved by using automatic repair functions with user feedback (so far only available with tool such as *MATE*).

Advantages of using and enforcing modeling guidelines for production code generation are:

- Deveopers can built upon expert know-how proven to be feasible in practice
- It is ensured that a safe subset of the modeling language is used
- Increase comprehensibility (readability) of a model
- Facilitate maintenance
- Ease testing, reuse, and expandability
- Simplify the exchange of models between OEMs and suppliers

6. Code Generation in Practice

Model-based development of automotive applications has proved its effectiveness over the last 10 to 15 years. Furthermore, the utilization of code generation has begun to become established for series production in conjunction with the further development of optimizing code generators. Although the use of code generation appears natural in model-based development, project experience has shown that it **cannot** be considered a pure 'push button' approach; rather the whole development process has to be adjusted towards the aim of production code generation. This starts with the availability of appropriate modeling and design guidelines as well as company and project-specific guidelines, and ends with sophisticated quality assurance measures, cf. [21].

Two major aspects must be taken into account when moving towards production code generation:

- A consummate physical model is not necessarily an implementation model suitable for code generation. However, the physical model usually matches the structure of the specification of the function to be implemented.
- Code safety/efficiency is strongly dependent on the use of appropriate modeling constructs.

Thus, a conflict between comprehensibility and code efficiency arises. In practice, a compromise between both important aspects must be found in each specific project.

Essentially, experts who are skilled with respect to the modeling paradigm are required. The influence of their knowledge on the quality of developed code is comparable to the influence of the programmer's experience in code-based development. Overheads of generated code compared with manual code are usually caused by improper modeling or incorrect configuration of the code generator.

As a result, the application of code generation for safety-critical systems is a feasible option, even

when there is no general means of qualification at hand.

6. Conclusions

Since model-based development began to become widespread in the application development of different automotive domains, code generation has become more and more common.

This trend has thrown up a number of questions, which address the whole application development, especially when safety-critical systems are under development. More specifically, we are faced with the question as to whether tools used for automating manual transformation work need to be qualified or somehow validated in accordance with a defined procedure. At present, safety standards like DO-178B only request this if a verification of the generated results is omitted. The definition of a dedicated test suite for code generator validation in combination with the use and enforcement of a safe subset for model design appears another promising approach. In any case, it is reasonable to implement further measures to safeguard code generation. The main challenge facing the use of code generation is not qualification of the tool, but still a combination of process adoption and project member training.

7. References

[1] The MathWorks Inc (product information), www.mathworks.com/products, 2007.

[2] *dSPACE*. TargetLink 1.3p2: Production Code Generator. http://www.dspace.com, 2007.

[3] *Stürmer, I., Weinberg, D., and Conrad, M.*: "Overview of Existing Safeguarding Techniques for Automatically Generated Code", Proc. of 2nd Intl. ICSE Workshop on Software Engineering for Automotive Systems (SEAS'05), St. Louis, Missouri, USA, May. 21, pp. 1-6, 2005.

[4] *IEC 61508*: Functional Safety of Electrical /Electronic/Programmable Electronic Safety-Related Systems, International Electrotechnical Commission, 1999.

[5] Ueda, T., Ohata, A., "Trends of Future Powertrain Development and the Evolution of Powertrain Control Systems", *Proc. of 30th Int. Congress on Transportation Electronics (Convergence 2004)*, Detroit, Michigan, USA, SAE #2004-21-0063, pp. 439-449, Oct 2004.

[6] Authors: "*Title of Paper*", Name of the conference, Location (Town, Country), Year of conference.

[7] Edwards, P.D. "The Use of Automatic Code Generation Tools in the Development of Safety-Related Embedded Systems", *Vehicle Electronic Systems, Europ. Conference and Exhibition*, Jun 1999.

[8] Stürmer, I., Conrad, M., Dörr, H., Pepper, P.: Systematic Testing of Model-based Code Generators, IEEE Transactions on Software Engineering, Vol. 33(9), Sep 2007.

[9] Beine, M., Eisemann, U., Wewetzer, C.: Quality Assurance Aspects and Activities in Automotive Model-based Development, Automotive Safety & Security, Jul 2006.

[10] Junker, F., Glöe, G.: "Guaranteed Product Safety According to the IEC 61508 Standard", RealTime, Vol. 1, pp. 28-29, 2003.

[11] Lefarth, U., Baum, U., Beck, T., and Zurawka, T., "ASCET-SD – Development Environment for Embedded Control Systems", Proc. of IFAC Symposium on Cumpter Aided Control System Design, Gent, Apr 1997.

[12] RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment Certification", Requirements and Technical Concepts for Aviation, Inc., Dec 1992.

[13] Conrad, M., "Using Simulink and Real-Time Workshop Embeddd Coder for Safety-critical Automotive Applications", Proc. of Dagstuhl Seminar MBEES '07, pp. 41-50, 2007.

[14] MathWorks Automotive Advisory Board, Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow, Version 2.0", 2007.

[15] Daimler Model-based Development Guidelines (DCX:2007). http://www.eguidelines.de, internal, 2007.

[16] Model Engineering Solutions, http://www.model-engineers.com, e-Guidelines Server, 2007.

[17] The Mathworks Inc, Simulink Model Advisor, www.mathworks.com/products, 2007.

[18] Ricardo, Inc., *MINT,* http:/www.ricardo.com/mint, 2007.

[19] Model Engineering Solutions, The Model Examiner M-XAM, www.model-engineers.com, 2008.

[20] Ameluxen, C., Legros, E., Schürr, A., Stürmer, I., "Checking and Enforcement of Modeling Guidelines with Graph Transformations", Proc. of Application of Graph Transformations with Industrial Relevance (AGTIVE), LNCS, Springer, 2007.

[21] Fey, I., Stürmer, I.: Quality Assurance Methods for Model-based Development: A Survey and Assessment. SAE World Congress, SAE Doc. #2007-01-0506, Detroit, 2007.