

Code Generation for Safety-Critical Systems – Open Questions and Possible Solutions

Ines Fey
Carmeq GmbH

Ingo Stürmer
Model Engineering Solutions

Copyright © 2008 SAE International

ABSTRACT

The approach taken in developing embedded software in the automotive field has shifted towards the paradigm of using executable graphical models at all stages of development: from the initial design phase through to implementation (model-based development). Models are designed with common graphical modeling languages, such as Simulink / Stateflow from The MathWorks. New approaches allow for the automatic generation of efficient controller code from the Simulink and Stateflow models directly via *code generators*, such as TargetLink by dSPACE or the Real-Time Workshop/Embedded Coder by The MathWorks. The usage of a code generator can lead to significant improvements in productivity in the software implementation phase. Furthermore, the level of quality gained by early quality assurance at the model level can also lead to higher quality code. Automotive software is often deployed in safety-critical systems and therefore cannot 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, even when using a code generator that is proven to be 'correct-by-construction', it is impossible to avoid generating erroneous code in every given case. Inappropriate modeling or faulty configuration of the code generator, for example, can lead to erroneous generated code.

This paper discusses experiences gained by the authors from previous projects with DaimlerChrysler and automotive supplier how code generators and the code they generate can be safeguarded through tool certification with regard to the safety standards that are relevant to the automotive industry. Specific, tool-related problems will be discussed and illustrated with practice-relevant examples.

MODEL-BASED DEVELOPMENT

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.

In the first design stage, a *behavioral model* (or physical model) is created, which is derived from the requirements specification (see Fig. 1). The behavioral 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 behavioral 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 behavioral 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 behavioral 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 behavioral 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.

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 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).

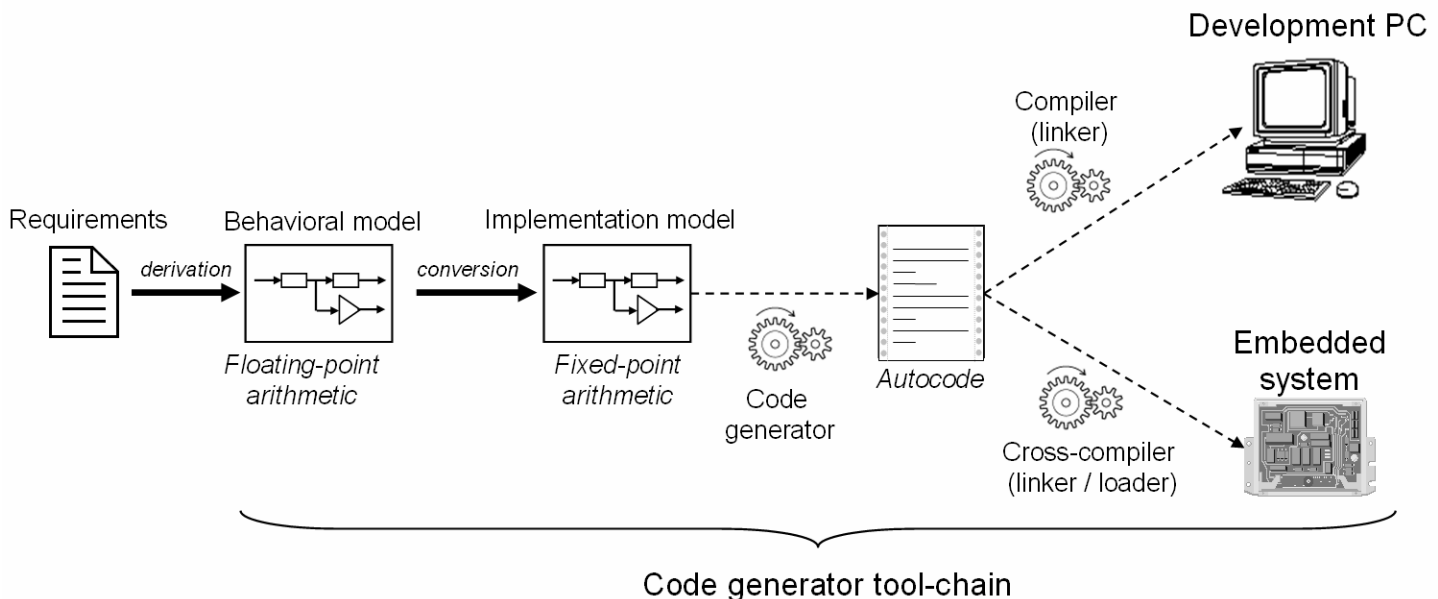


Fig. 1: Model-based code generation for embedded systems

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 [UO04]. If the manual verification process at the code level can also be reduced, savings of up to 50% are reported. This conforms to 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. Due to 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 is 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.

Model-based code generation is a new paradigm that makes new demands on the development process and throws up 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