

Code Generator Certification: A Test Suite-oriented Approach

Ingo Stürmer¹, Mirko Conrad¹

Abstract: Code generators are increasingly used in an industrial context to translate graphical models into executable code. Since this code is often deployed in safety-related environments, the quality of the code generators is of high importance. The use of test suites is a promising approach for gaining confidence in the code generators' correct functioning.

This paper gives an overview of such a practice-oriented testing approach for code generation tools. The main application area for the testing approach presented here is the testing of optimisations performed by the code generator. The test models and corresponding test vectors generated represent an important component in a comprehensive test suite for code generators. As regards the actual certification situation for development tools, the test suite proposed could be a substantial contribution to current certification practises.

1. Introduction

The development of embedded software has become increasingly complex and abstraction appears to be the only viable means of dealing with this complexity. In the automotive sector, the way embedded software is developed has changed in that executable but abstract graphical models are used at all stages of development, from specification up to implementation as well as for testing (*model-based development*, cf. [SCD+04]).

While in the past such models were implemented manually by the programmer, some recent approaches allow the automatic generation of efficient code directly from the software model via so-called *code generators* [JB03], such as TargetLink by dSPACE [DS04] or the Real-Time Workshop by The MathWorks [MW04]. A code generator is essentially a compiler in that it translates a source language (a graphical modelling language, e.g. Simulink or Stateflow [MW04]) into a target language (a textual programming language, e.g. C or Ada). Code generators are examples of dependable software tools upon which software designers rely. However, at present, they are not as mature as C or ADA compilers which have been proven in use and thus their output must be checked with almost the same, expensive effort as for manually written code.

In order to increase confidence, the use of test suites, which make it possible to check compilers systematically, is also a promising approach for code generators. When testing a code generator which translates graphical models into a textual target language, one has to deal with two main testing objectives: First, how one can determine a set of appropriate input models suitable for covering code generator functionality and, second, which the

¹ DaimlerChrysler AG, Research Information and Communication, Alt-Moabit 96A, 10559 Berlin,
✉ {Ingo.Stuermer | Mirko.Conrad}@DaimlerChrysler.com

right stimuli (input data) are for these models in order to check the correctness of the code generation process.

2. Tool Certification and Testing

Although the application of software development tools generally has many advantages, they can also introduce errors. Therefore, various standards provide requirements and / or guidelines for the application of such tools.

Avionic standards such as DO-178B [RTCA92] encourage the *qualification* of code generation tools. Qualifiable code generators such as SCADE [Es03], which endorse a certification of the application software, do exist. However, they only make it possible to reduce the amount of some of the verification activities but do not allow them all to be omitted completely. In addition, their source language is not as popular as Simulink / Stateflow and they perform only a limited amount of optimisations.

The qualification of a development tool can be treated similarly to the certification of the application software itself. Thus, qualifying a development tool such as a code generator does not mean proving its correctness. Instead, it is important to gain sufficient confidence in its correctness [Edw99]. Despite the progress made in the area of formal methods, *dynamic testing* has proved to be one of the most important techniques for this purpose and is also a recognized and established measure in the area of compiler certification (i.e. language conformance certification, cf. [Ton99]). But standard testing technologies for these tools do not yet exist.

Both functional (black-box) as well as structural (white-box) testing techniques should be taken into account (cf. e.g. [RTCA92]). A direct, brute-force application of one of the two basic test approaches to the code generator is impossible or at least disproportionately expensive:

- According to Myers [Mye79], it is impossible to perform an exhaustive black-box test of a compiler and therefore also of a code generator. This would require test cases representing all valid source programs / models and test cases for all invalid source programs / models.
- In order to achieve sufficient structural coverage for a level A system², testing would have to cover every instruction and modified condition / decision coverage (MC/DC coverage) would have to be adopted. This has proved to be disproportionately expensive for the qualification of a complex development tool. Santhanam [San02], for instance, points out that the GNAT compiler for Ada 95 consists of 625,000 lines of Ada and C code. Santhanam estimates further, that structural testing of the compiler would require 58 person years of effort.

Accordingly, the essential task during code generator testing is the determination of a finite, manageable number of error-sensitive test cases. This ultimately determines the scope and quality of the test.

² This is the most critical category in an avionics system. A system failure prevents continued safe flight and landing.

On the surface, test cases for a code generator are valid and invalid graphical models ('*test models*'). We can assume that the code generator is working correctly if:

- invalid test models are rejected by the code generator, i.e. are not translated into C code (invalid test cases) and
- valid test models are translated by the code generator and the code generated from this behaves in a 'functionally equivalent' way (valid test cases).

For the invalid test cases, test models are sufficient. They simply have to be rejected by the code generator. For valid test cases the situation is more complex: In order to check *functional equivalence between model and code*, the test model and the C code generated from it have to be stimulated with the same sets of input data ('*test vectors*') and the respective system reactions have to be compared. So in general, *code generator test cases* are graphical models (test models) which have to be stimulated by appropriate sets of input data (test vectors).

Test vectors and system reactions are not considered to be constant values but rather time-variable data courses. Upon comparison, the system reactions of model and C code to one and the same test vector must be sufficiently similar.

The reader should consider that the models are designed on a host computer with a different arithmetic (floating point) to the target environment, which is often implemented with fixed-point arithmetic for reasons of efficiency. Thus, both hardware types deal with different precisions in number representation as well as different execution speeds. Furthermore, there are distinctions between both hardware types dealing with boundary values such as 'not a number', or numerical instabilities for very large or small numbers (e.g. underflow). Due to these differences it would not be advantageous to demand absolute equality between the system reactions of the model and the code. Rather, the signals observed must have a certain similarity relation to each other. A detailed discussion of this similarity relation and the resulting term of functional correctness is beyond the scope of this paper. Those interested may refer to [CFP03] for more information on this subject.

Based on these considerations, the following chapter outlines the proposed code generator testing approach and looks at the systematic selection of the test cases, i.e. of the test models and corresponding test vectors.

3. Code Generator Testing Approach

This chapter surveys the overall principles of the proposed code generator testing approach. For a more detailed explanation, the reader should refer to [SC03]. The main activities are shown in Figure 1 and are described in the following:

- (A) A formal specification of a code generator transformation is created as a graph rewriting rule. We used graph transformation rules for this purpose since they give us a clear understanding of how patterns of the input graph are replaced and transformed into code.

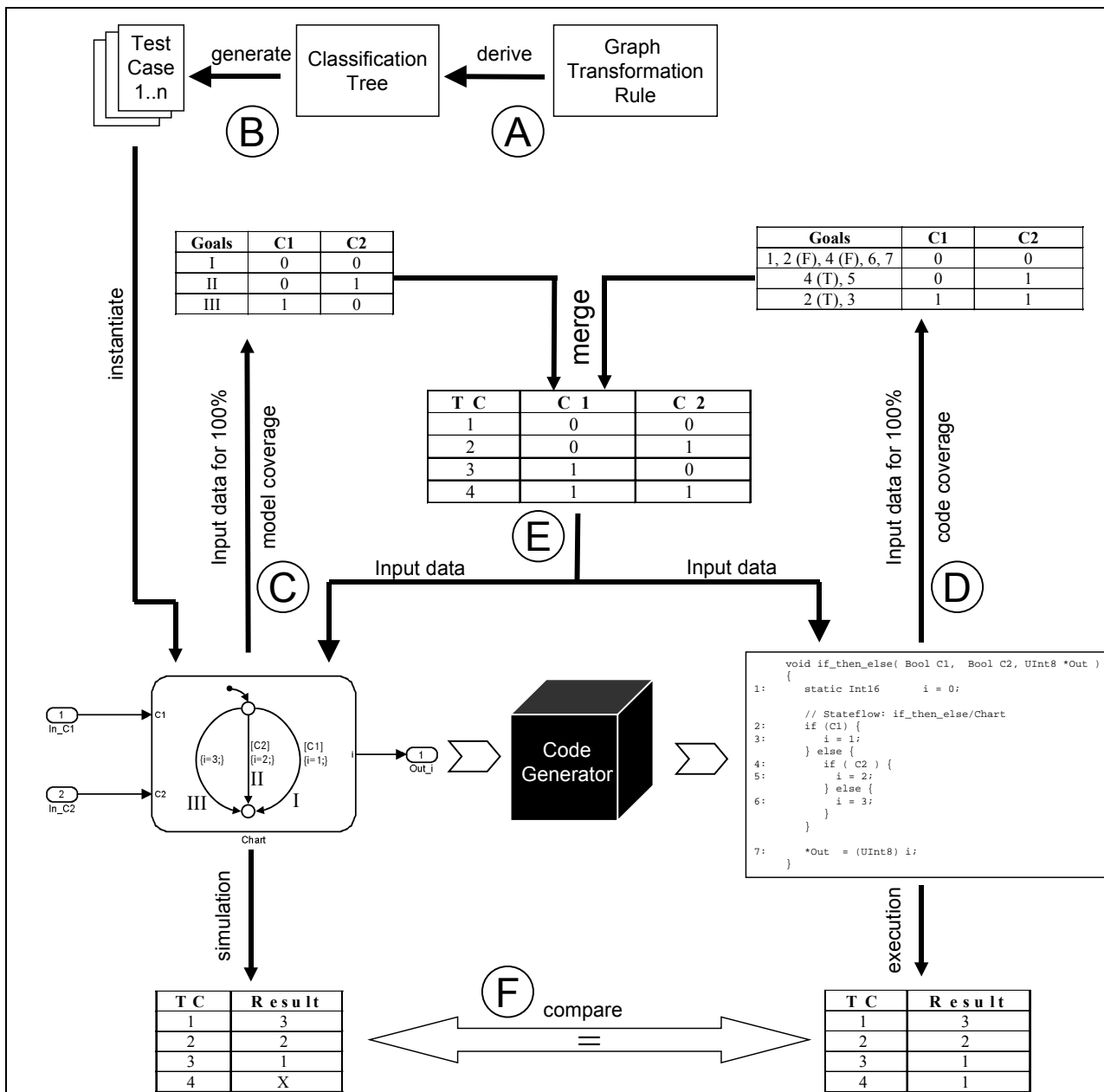


Figure 1: The Code Generator Test Approach

- (B) The formal specification is then used as a blueprint to describe the possible input domain of a transformation rule with the Classification-tree Method [GG93]. With the test models derived from the classification-tree we now have representative input models to cover the code generator's functionality with regard to a specific transformation. However, before we can observe the code generator's correct behaviour, we need the right input data to stimulate these models.
- (C) In order to stimulate all possible 'simulation pathways' through a given test model, we employ structural coverage metrics (branch coverage) on model level. Automated test vector generation with regard to these coverage measures is used to find a selection of input data which achieves full structural model coverage [BCSW03]. This can, to a large extent, be automated using tools such as Reactis [Re04].
- (D) After code generation has been carried out, a similar approach is followed on code level: this time structural testing is used to create a second set of test vectors which guarantees complete structural coverage (again: branch coverage) of the C code

generated. In this case, automation can be achieved with the aid of the evolutionary structural test tool [WSB01].

- (E) After test vector sets for model and code coverage have been generated, both test data sets are merged together. This is necessary because the control flow of the model and the code could be different. On the one hand, optimisation techniques could omit or melt branches of the model. On the other hand, the code generator could produce additional code, e.g. for protecting a division operation for a possible division by zero. Nevertheless, both the test model and the autocode are then stimulated one after the other with the resulting amount of input data (*back-to-back test*).
- (F) Finally, the model and the code outputs are compared. If these are sufficiently similar for one and the same test vector, this is an indication that the code generator and the other tools used (e.g. compiler, linker) are working correctly. If, however, they are (substantially) different, one can conclude that this is due to an incorrect implementation of the code generator, a problem with one of the other tools involved, a faulty test model or an incomplete specification of the optimisation (incorrect graph transformation). The comparison of system reactions can be automated by using the MEval tool [IT04].

4. Conclusion

In this paper, a practice-oriented test approach for the verification of a code generator's correct functioning was outlined. The main benefits of this testing approach could be summed up as follows:

- Test models for the code generator are designed systematically, based on formal specifications of the code generator transformations. This is an enhancement of existent approaches which are based on ad-hoc generated sets of test models.
- The necessary inputs for these test models as well as for the code generated are created by means of structural testing. This ensures high structural coverage on both model and code level. This approach goes beyond the results achieved with randomly generated test data [TRB+99].
- The test process could be automated to a large extent, such that the test approach could be easily applied to new code generator versions.

The main application area for the testing approach presented here is the testing of optimisations performed by the code generator. The test models and corresponding test vectors created in this way represent an important component in a comprehensive test suite for code generators. This test suite also comprises other elements such as low level tests for the individual basic blocks [cf. e.g. JB03] of the graphical modelling language and also complex customer models. In the opinion of the authors, this kind of comprehensive test suite could play a central role in the application-independent safeguarding of code generator deployment. As regards the actual certification situation for development tools, automated test suites based on a formal specification such as graph rewriting rules, could have an important impact on the acceptance of code generation tools in the context of safety-related software development for embedded systems, as has been shown in this paper.

5. References

- [BCSW03] Baresel, A., Conrad, M. Sadeghipour, S. and Wegener, J.: The Interplay between Model Coverage and Code Coverage. Proc. of 11th Int. Conf. on Software Testing, Analysis and Review (EuroSTAR '03), Amsterdam, Dec., 2003.
- [CFP03] M. Conrad, I. Fey, and H. Pohlheim, Automatisierung der Testauswertung für Steuergeräte-software. VDI-Berichte, Vol. 1789, VDI Verlag, 2003, pp. 299-315.
- [DS04] dSPACE: TargetLink – Production Code Generator at <http://www.dspace.com>, 2004.
- [Edw99] Edwards, P. D.: The Use of Automatic Code Generation Tools in the Development of Safety-related Embedded Systems. Eur. Conf. on Vehicle Electronic Systems, ERA report 99-0484, June 1999
- [Es03] Esterel Technologies: SCADE at <http://www.esterel-technologies.com>, 2003
- [GG93] Grochtmann, M., and Grimm, K. Classification Trees for Partition Testing. Software Testing, Verification and Reliability, 1993, pp. 63-82.
- [IT04] ITPower Consultants: MEval at www.it-power.com, 2004.
- [JB03] M. Jungmann, M. Beine: Automatische Code-Generierung für sicherheitskritische Systeme. Automotive Electronics, Heft II/2003.
- [MW04] The MathWorks: Simulink, Stateflow and Real-Time Workshop at <http://www.martworks.com/products>, 2004.
- [Mye79] Myers, G.J.: The Art of Software Testing. John Wiley & Sons, New York, 1979
- [Re04] Reactive Systems Inc: Reactis Simulator / Tester at www.reactive-systems.com, 2004.
- [RTCA92] RTCA, Software Considerations in Airborne Systems and Equipment, DO-178B, Requirements and Technical Concepts for Aviation, Inc., 1992.
- [San02] V. Santhanam, The Anatomy of an FAA-Qualifiable Ada Subset Compiler, Proc of the ACM SigAda Annual Int. Conference (SigAda 2002), Houston, TX (USA), Dec. 8-12, 2002.
- [SC03] Stürmer, I.; Conrad, M.: Test Suite Design for Code Generation Tools, Proc. of 18th Int. Automated Software Engineering Conference, 2003, pp. 286-290.
- [SCD+04] H. Schlingloff, M. Conrad, H. Dörr, C. Sühl: Modellbasierte Steuergerätesoftwareentwicklung für den Automobilbereich. Proc. of Automotive - Safety & Security, Stuttgart, 6.-7. October 2004
- [Ton99] Tonndorf, M.: Ada Conformity Assessments: A Model for Other Programming Languages? ACM SIGAda Ada Letters, Vol. XIX (3), pp. 89-99, 1999
- [TRB+99] S. Töppe, S. Ranville, D. Bostic, Y. Wang: Practical Validation of Model Based Code Generation for Automotive Applications. 18th AIAA/IEEE/SAE Digital Avionics System Conference, St. Louis (USA), Oct. 1999.
- [WSB01] Wegener, J.; Stahmer, H.; Baresel, A.: Evolutionary Test Environment for Automatic Structural Testing. Special Issue of Information and Software Technology, Vol. 43,2001, pp. 851-854.