

Code Generator Testing in Practice

Ingo Stürmer¹, Mirko Conrad¹

DaimlerChrysler AG Research and Technology
Alt-Moabit 96 A, D-10559 Berlin
{Ingo.Stuermer, Mirko.Conrad}@DaimlerChrysler.com

Abstract: This paper provides an overview of 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.

1 Introduction

In the automotive sector, the way embedded software is developed has changed, in that executable graphical models are used at all stages of development, from the first design phase up to implementation (model-based development). Some recent approaches allow the automatic generation of efficient code directly from the software model via so-called *code generators*, 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) into a target language (a textual programming language). At present, code generators 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. When testing a code generator which translates a graphical source language (e.g. Simulink or Stateflow [MW04]) into a textual target language (e.g. C or Ada), two main testing issues arise. First, how to determine a set of appropriate input models suitable for covering code generator functionality and, second, which the right stimuli (input data) are for these models in order to check the correctness of the code generation process.

2 Code Generator Testing

Despite the progress made in the area of formal methods, *dynamic testing* continues to represent one of the most important techniques for assuring software quality.

¹ The work described was partially performed as part of the IMMOS project funded by the German Federal Ministry of Education and Research (project ref. 01ISC31D).

Dynamic testing is considered to be the execution of a test object on a computer with selected test data. The aim of this execution is to check - in a real operating environment - whether or not the test object behaves like a defined test reference. In the case of code generator testing, test cases are graphical models (*test models*) which have to be stimulated by appropriate test data sets (*test vectors*). The correct functioning of the code generator, i.e. the correct transfer of the models into C code, can be ensured using *back-to-back tests* between the test models and the code generated from them. 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, and *valid test models* are translated by the code generator and the code generated from this behaves in a 'functionally equivalent' way. In order to check functional correctness, the test model and the C code generated from it are stimulated with the same test vectors and the respective system reactions are compared. In doing so, test data sets and system reactions are not considered to be static 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. Those interested may refer to [CFP03] for more information on this subject.

The following chapter outlines the code generator testing approach and looks at the systematic selection of the test cases, i.e. of the test models and corresponding test vectors, as well as the automatic generation of test models.

3 Code Generator Testing Approach

This chapter surveys the overall principles of the proposed code generator testing approach (for a more detailed explanation, please refer to [SC03]). The main tasks 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. In a different approach, these transformation rules could also be used for a compact description of code generator behaviour [BKS04].

(B) The graph transformation rule 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 generated from the classification-tree with a so-called *model generator* (MG, see section 3.1), 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 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. This can, to a large extent, be automated using tools such as Reactis [Re04].

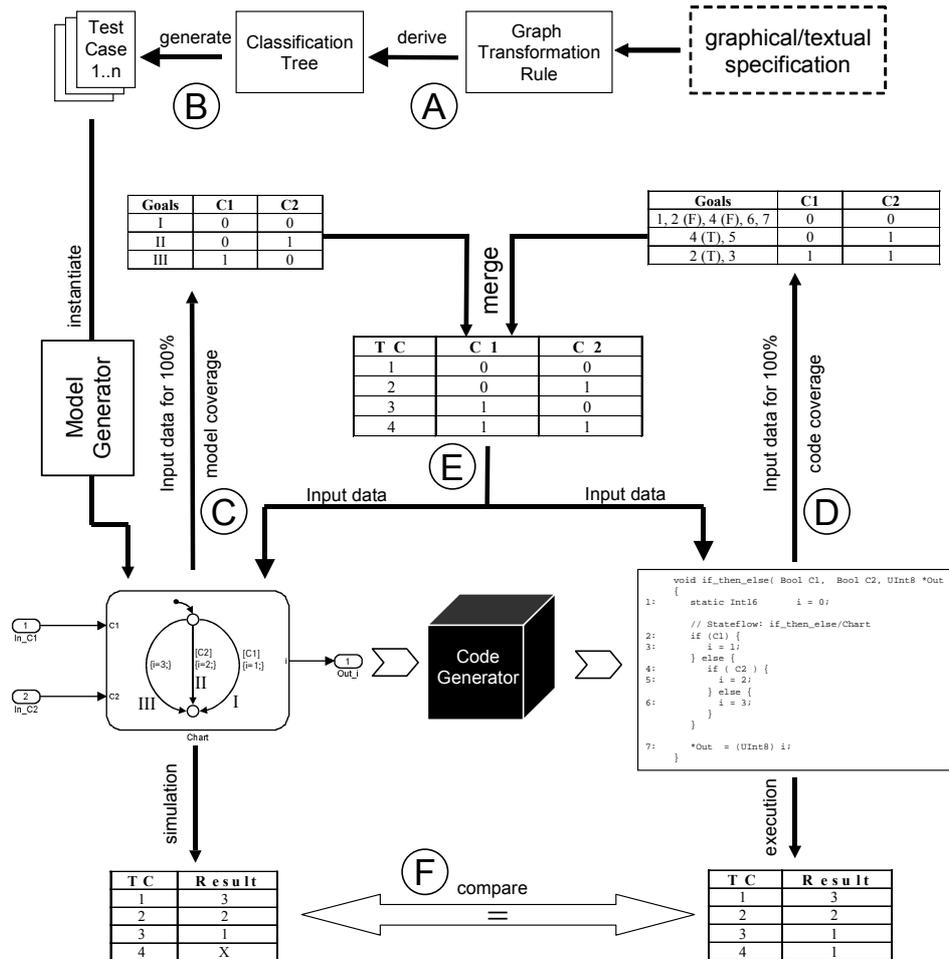


Figure 1: The Code Generator Test Approach

(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 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].

3.1 Automatic Model Generation

In [SC03] the authors state that the number of possible test models which could be derived from the classification tree is very high (a few hundred is not uncommon). We therefore developed a so-called *model generator* which uses the concept of meta-model transformation for generating Simulink or Stateflow models from the classification-tree automatically (see Figure 2). For that purpose, the XML-representation of the classification-tree is transformed with the graph-transformation tool GREAT from Vanderbilt University [Kar03]. The transformation phase is followed by the generation of executable models with the help of SimEx [IT04], a tool which allows the conversion of Simulink or Stateflow models from or into XML representations.

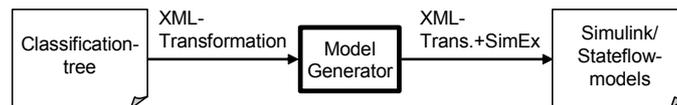


Figure 2: Principles of the Model Generator

4 Conclusion

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

Second, the necessary inputs for these test models as well as for the generated code are generated 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 [Toe99]. Third, 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 to test 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 as well as complex customer models. In the opinion of the authors, such a comprehensive test suite would play a central role in the application-independent safeguarding of code generator deployment.

References

- [BKS04] Baldan, P.; König, B.; Stürmer, I.: Generating Test Cases for Code Generators by Unfolding Graph Transformation Systems. To appear in Proc. of 2nd Int. Conference on Graph Transformation, Springer-Verlag, LNCS, 2004.
- [CFP03] Conrad, M.; Fey, I.; Pohlheim, H.: Automatisierung der Testauswertung für Steuergerätesoftware. VDI-Berichte, vol. 1789, VDI Verlag, 2003, pp. 299-315.
- [DS04] dSPACE: TargetLink – Production Code Generator v1.3p2. <http://www.dspace.com>, 2004.
- [Edw99] Edwards, P.D.: The Use of Automatic Code Generation Tools in the Development of Safety-Related Embedded Systems. Vehicle Electronic Systems, European Conference and Exhibition, ERA Report 99-0484, 1999.
- [GG93] Grochtmann, M.; Grimm, K.: Classification Trees for Partition Testing. Software Testing, Verification and Reliability, 1993, pp. 63-82.
- [IT04] ITPower Consultants, MEval and SimEx. <http://www.it-power.com>, 2004.
- [JB03] Jungmann, J.; Beine, M.: Automatische Code-Generierung für sicherheitskritische Systeme. Automotive Electronics, Heft II/2003.
- [Kar03] Karsai, G. et. al.: On the use of Graph Transformation in the Formal Specification of Model Interpreters. Journal of Universal Computer Science, Vol. 9(11), 2003, pp. 1296-1321.
- [MW04] The MathWorks: Simulink, Stateflow and Real-Time Workshop. <http://www.mathworks.com/products>, Website, 2004.
- [Re04] Reactive Systems Inc: Reactis Simulator / Tester. <http://www.reactive-systems.com>, Website, 2004.
- [SC03] Stürmer, I.; Conrad, M.: Test Suite Design for Code Generation Tools. Proc. of 18th Int. Automated Software Engineering Conference, 2003, pp. 286-290.
- [Toe99] Toepe, S. et. al.: 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.