

The Interplay between Model Coverage and Code Coverage

André Baresel
DaimlerChrysler AG
Alt-Moabit 96a
10559 Berlin / Germany
andre.baresel@daimlerchrysler.com

Mirko Conrad
DaimlerChrysler AG
Alt-Moabit 96a
10559 Berlin / Germany
mirko.conrad@daimlerchrysler.com

Sadegh Sadeghipour
IT Power Consultants
Gustav-Meyer-Alle 25
13355 Berlin / Germany
sadegh@itpower.de

Joachim Wegener
DaimlerChrysler AG
Alt-Moabit 96a
10559 Berlin / Germany
joachim.wegener@daimlerchrysler.com

Abstract

Executable graphical models are used throughout the model-based development process for embedded controls. The test process accompanying model-based development can profit from the existence of such executable models in various ways. For instance, in addition to traditional code coverage analysis the executable model can also be subjected to structural coverage metrics on model level. This paper gives the details and results of our experience with the deployment of model coverage metrics and the application of test vector generators and coverage analyzers.

1 Introduction

Automotive manufacturers' and suppliers' need to reduce development times and costs has resulted in a paradigm shift toward the model-based development of software embedded in automotive electronic control units (ECUs). This means that executable models, designed with popular graphical modeling languages, are used throughout the development process. These models then form the 'blueprint' for the automatic or manual coding of the ECU software.

The test process accompanying model-based development (model-based testing) can profit from the existence of such executable models in many ways. The executable model can already be simulated, tested and analyzed by the development engineer. As a result, errors can be detected at an early stage and eliminated at low cost. Since the executable models could and should be exploited as an additional, comprehensive source of information for testing, model-based development allows and, at the same time, requires new approaches to software testing. One of various ways in which the test process can profit from the existence of such executable models is the possibility of applying structural coverage criteria not only at code level but also at model level.

The determination of structural coverage metrics at code level (code coverage) as a part of the test process is considered to be best practice in industrial software development and is required by many development standards (see e.g. [13]). By using coverage criteria at model level the known benefits of structural coverage, namely controlling the test depth and detecting coverage holes in given test suites [23] [27] [12] [10], can be exploited in an early stage of software development. Consequently, the effectiveness and quality of testing can be measured as early as executable models are available.

Coverage metrics at code level can be control flow or data flow oriented. The coverage is measured with respect to certain program elements which are executed during the test. For instance, in the case of branch coverage these are program branches. In contrast, structural coverage metrics at

model level (model coverage) are, at best, currently being pilot tested [10]. Here, the number of model elements covered by the test is related to the total number of these elements. In the case of state coverage for Stateflow diagrams, such elements are then, for instance, Stateflow states.

A structural coverage metric can be utilized in two different ways: Firstly, it can be used as a test adequacy criterion, i.e. to decide whether a given test set is complete or adequate with respect to that criterion (acceptor for black-box tests). Secondly, it can be an explicit specification for test case or test sequence selection. It then plays the role of a test selection criterion (generator for white-box tests) [27].

To be beneficial to a real software development process, which may involve software with thousands of lines of code or models with thousands of blocks or states, a coverage metric must be suitable for automated collection and analysis [23]. Since structural metrics based on control flow are most suitable for automated collection and analysis [23], we will limit the scope of the paper to these types of metrics. Moreover, we will limit our scope to Simulink / Stateflow [25] [22] and C, because they are the most popular graphical modeling and programming languages respectively for embedded automotive software.

Tool support for use as test adequacy criteria is provided by coverage analyzers. Control flow based C code coverage analyzers exist for years and are quite common [18]. In contrast, model coverage analyzers for Simulink / Stateflow models have only become available to a broader circle of users since the year 2000. These tools now comprise the Simulink / Stateflow Model Coverage Tool [16], Reactis Tester [20], and Beacon Tester [4].

To support the application of coverage metrics as test selection criteria, test input generators are indispensable. Due to internal system states only repeated, sequential calls of the test object can reach a high structural coverage of embedded control software. Therefore, it is often not sufficient to generate test vectors for a single call of the test object. Instead, sequences of such test vectors (test sequences) have to be generated [2]. The respective tools are referred to as test vector / test sequence generators in the following. They enable an automatic generation of test suites with a high degree of either model or code coverage [20] [26]. From the tester's point of view, the automatic generation of test vectors / test sequences saves a large amount of time. The tester can concentrate on analyzing the correctness of the test results – a task which cannot be accomplished by the generators.

Only limited experience has been gained in the use of model coverage. Experience with the use of coverage metrics for Simulink / Stateflow models as well as with the application of model coverage analyzers and test vector / test sequence generators is very rare. A few exceptions known to the authors comprise [1], [10] and [2].

For this reason, the authors have carried out some experiments concerning the deployment of coverage metrics for Simulink / Stateflow models as part of model-based testing. The following questions were of central importance:

- ◆ Which possible applications exist for model coverage measurements within the context of model-based development?
- ◆ Which correlation exists between structural model and code coverage criteria?
- ◆ Can model coverage metrics supplement or, in part, replace the measurement of code coverage?
- ◆ How can model coverage analyzers and test vector / test sequence generators be used to improve model-based testing?

The remainder of the paper is structured as follows: Section 2 introduces structural coverage criteria at model and code level. Section 3 describes the procedure and the results of the experiments mentioned above. Section 4 analyzes the experimental results, and section 5 summarizes the previous sections and introduces a model-based test strategy.

2 Structural coverage criteria

In the context of model-based development two kinds of structure can be seen as a structural coverage measure for testing software. On the one hand, the coverage of the model structures can be considered, on the other hand, coverage metrics can be determined for the source code structures.

Both coverage at code as well as at model level can be considered to be instances of a more general concept of structural coverage. The determination of structural coverage in order to control the test depth or test end is based on the assumption that test coverage rises as the number of tests increases, in so far as the test coverage is not yet complete and the newly added tests do not repeat tests already existing. The definition of new tests aims at covering structural elements which have not yet been tested by the existing tests. This selection metric leads to a more focused and efficient test process (in comparison to the random test). The attainment of a certain amount of structural coverage is, in practice, often used as a necessary, but not adequate, test termination criterion. 100% structural coverage can usually not be attained in practice, as a certain portion of the structural units is not executed during the program run. This portion, however, is negligible (see [15], [9]). The following sections describe the structural coverage measurements at model and code level used during the experimental investigations made by the authors.

2.1 Model coverage criteria

There are no general model coverage criteria known and the few available model coverage tools are presently being introduced into industrial praxis. The Model Coverage Tool [16], utilizable since the availability of Matlab R12 in 2000, allows the determination of different coverage criteria for Simulink / Stateflow models during the model test. The authors also used the test vector / test sequence generator tool Reactis Tester in their experiments. This tool generates test sequences for Simulink / Stateflow models and has been available since 2002.

Most of the coverage metrics provided by the Model Coverage Tool and Reactis Tester tend to be control flow oriented. Those of them which have been used in the experiments presented in this paper, are described below.

2.1.1 The Model Coverage Tool's coverage criteria

The coverage metrics provided by the Model Coverage Tool used in our experiments are decision coverage (D1) and condition coverage (C1).

Decision Coverage (D1). In order to determine decision coverage (D1), the execution of blocks, which serve as decision points, is analyzed. The number of possibly and actually executed alternatives is calculated for each decision point. For the determination of decision coverage with Matlab R12.1, Switch, Abs, Logic and While blocks, triggered / enabled subsystems, as well as Stateflow states and transitions are drawn upon. Figure 1 shows, as an example, the different pathways through a Switch block and lists the test goals which have to be reached in order to achieve full coverage.

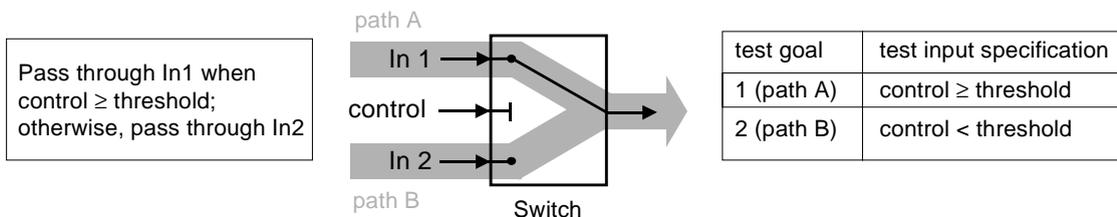


Figure 1. Decision coverage on model level - pathways through a Switch block and test goals

Note that the decision coverage is a control flow related metric, and the acronym D1 for this coverage metric does not correspond to the naming convention used for code coverage criteria mentioned

in section 2.2. In order to avoid confusion, the coverage metrics referred to in the tables of section 3 have the prefix “M_” for model and the prefix “C_” for code coverage.

Condition Coverage (C1). In order to determine condition coverage (C1) the coverage of the logical predicates is analyzed for logical Simulink blocks (e.g. Logic blocks, Combinatorial Logic blocks) and Stateflow transitions. A test achieves 100% C1 coverage at model level when each logical block and each Stateflow transition have been evaluated as *true* and as *false* at least once.

2.1.2 Reactis Tester’s coverage criteria

Reactis Tester’s coverage criteria are rather tool-specific and have been defined separately based on Simulink and Stateflow. The coverage metrics used in our experiments are branch coverage for Simulink as well as state coverage and condition action coverage for Stateflow [20].

Branch Coverage. The branch coverage metric considers a block with conditional behavior covered if all conditional behavior has been exercised at least once. Blocks considered for this purpose are: Dead Zone, Logical Operator, MinMax, Multiport Switch, Relational Operator, Saturation, and Switch.

State Coverage. This metric tracks which states have been entered at least once.

Condition Action Coverage. This metric tracks which transition segments have had their condition actions executed at least once. If a segment has no condition action, then the segment is considered covered when its condition has evaluated to true at least once.

2.2 Code coverage criteria

Coverage measurements at code level can be control or data flow oriented.

Typical control flow oriented code coverage metrics are statement, branch and path coverage. These coverage criteria define which program path, statements or conditions have to be executed in order to achieve full coverage of the software under test. The different methods of control flow related testing differ in the definition of the nature and frequency of the execution of paths or program constructs.

Data flow related test procedures define the corresponding criteria using the access to information structures contained in the program sequence, for example variables, fields and lists. The idea behind data flow oriented testing is to check the interaction between value assigning and value using statements. Differences between the individual methods arise in the part of the data flow interactions which has to be executed in order to achieve complete coverage.

Whilst the use of control flow related coverage criteria (Cx) at code level is widespread in practice, data flow related coverage measurements (Dx) are used less often due to their difficult automation. Prevalent control flow oriented coverage measurements are statement coverage (C0), branch coverage (C1), path coverage (C4) and different variations of condition coverage [17]. The individual coverage criteria are partially ordered. For instance, the branch coverage encloses statement coverage (see e.g. [11]).

In the experiments we limited our scope to C0 and C1 code coverage, because these metrics were supported by the deployed tools and were comparable to the model coverage metrics used.

Statement coverage (C0). During the statement test, the aim is to test the test object in such a way that each of its statements is executed at least once [17]. The statement coverage metric relates the number of statements executed during the test to the number of executable statements in the program code.

Branch coverage (C1). The aim of the branch test is to run through each program branch of the test object at least once. Here, a program branch is roughly defined as a possible route from the program start, or from a branching of the control structure, to the next control structure or program end. The branch coverage metric is defined as the ratio of the branches run through to the total number of branches present in the source code [11]. Figure 2 shows, as an example, the different branches emerging from an If-statement and lists the test goals which have to be reached in order to achieve full coverage.

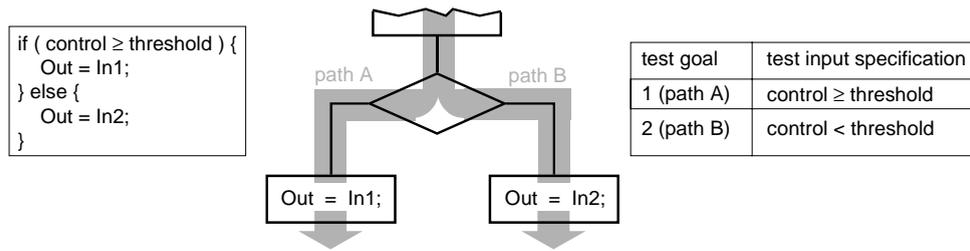


Figure 2. Branch coverage on code level – control flow graph and test goals

3 Experimental procedure and results

The objective of the experiments was to determine the applicability of the model coverage metrics, their relationship to the code coverage metrics and to assess the usability of test vector / test sequence generators on both model and code level.

3.1 Overall procedure, test objects

The tests were carried out at both model level and code level. The test objects comprised three different modules (subsystems) of a Simulink / Stateflow modeled automotive body control system of real size and complexity (Enable_A, Enable_S, and Ctrl_S). Enable_A and Enable_S contained both Simulink as well as Stateflow portions, Ctrl_S contained Simulink portions exclusively. Table 1 shows selected complexity measures of the Simulink / Stateflow models tested. The measures concern general as well as coverage-specific features.

Table 1. Information on the complexity of the Simulink / Stateflow models used as test objects

Test object	Size in KB	Blocks	Sub-systems	Inputs	State-flows	Stateflow states	Stateflow transitions	Switch blocks	Relational operators	Logic blocks
Ctrl_S	126	175	4	28	0	0	0	3	9	25
Enable_S	293	214	11	15	3	13	30	5	5	11
Enable_A	201	154	7	11	3	11	20	3	5	6

The test objects were subjected to a model test. Existing black-box test scenarios were taken as a basis from which test data were gained in the form of signal waveform over time. In total, 36 black-box test scenarios with an average length of 11 sec. were included in the investigations. During the model test, decision and condition coverage were determined with the Model Coverage Tool from Matlab R12.1.

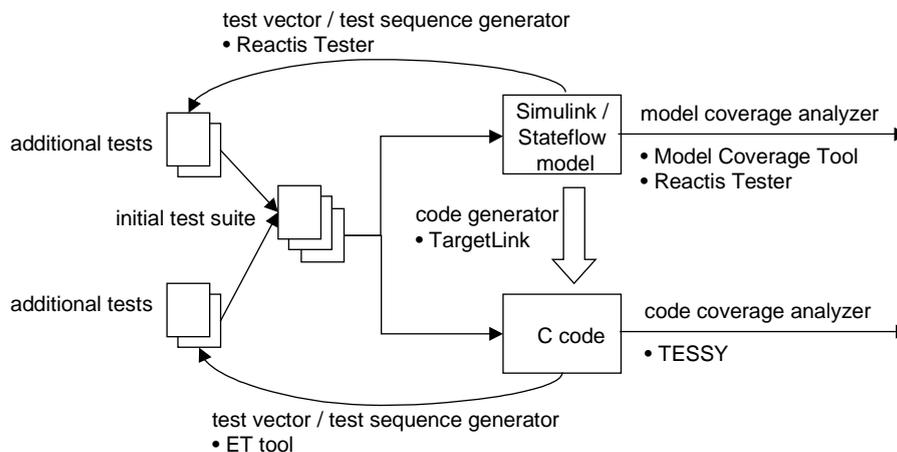


Figure 3. Experimental procedure

Further, aiming at maximum model coverage, test data were automatically generated by Reactis Tester [20]. The coverage reached by the generated test data was then compared with the coverage reached by the black-box test.

Following the model test, floating-point C code was created for the test objects with the aid of the code generator TargetLink [24]. Consequently, the behavior of the test objects on model and code level was equivalent. The test data from the model test served as stimuli for the renewed test execution (code test) with the aid of the test system TESSY [18]. The statement and branch coverage at code level (C_C0 and C_C1) were thus determined. As on model level, test data aiming at maximum code coverage were automatically generated by the evolutionary testing tool ET [26], and the code coverage reached was compared with the coverage reached by the black-box test. Figure 3 illustrates the experimental procedure.

3.2 Measuring model and code coverage

Table 2 summarizes the experimentally determined coverage of the three test objects at model and code level. For each module it contains the sum of the coverage attained with the black-box tests. The model coverage was measured with the Model Coverage Tool and the code coverage by TESSY using the same data. The coverage criteria used on model level are decision coverage (M_D1) and condition coverage (M_C1), and those used on code level are statement coverage (C_C0) and branch coverage (C_C1). Table 2 also shows the coverage measured by Reactis Tester using the test data of black-box tests. The relevant coverage criteria of Reactis Tester are branch coverage (shown as M_Branch), state coverage (shown as M_State) and condition action coverage (shown as M_CondAct).

Table 2. Model coverage reached by the black-box test;
coverage measured by Reactis Tester and TESSY with the same data

Test object	Test	Model Coverage		Model Coverage			Code Coverage	
		with Model Coverage Tool [%]		with Reactis Tester [%]			with TESSY [%]	
		M_D1	M_C1	M_Branch	M_State	M_CondAct	C_C0	C_C1
Ctrl_S	Σ black-box	100	69	70	n.a.	n.a.	100	100
Enable_S	Σ black-box	88	85	95	80	72	88	89
Enable_A	Σ black-box	54	66	91	80	21	61	58

3.3 Automatic test vector / test sequence generation for model coverage

Reactis Tester automatically generates test sequences in the form of signal waveforms over time in order to reach maximum coverage of model structures. The test data generation procedure is divided into two stages: random and targeted. During the targeted phase the tool tries to 'guess' good inputs that will put the model into states which increase the coverage degree. Details of the algorithms used by Reactis Tester have not been published.

The generation of test data by Reactis Tester depends on some parameters such as the maximum number and steps of generated test sequences, test sequences to be preloaded, how long the probes are initially, how often it should try extending a probe before giving up, etc.

3.3.1 Details of experiments

Table 3 shows the best model coverage reached by Reactis Tester after executing several tests with different test generation parameter values. Table 3 also shows the model coverage measured with the Model Coverage tool as well as the code coverage measured with TESSY using the same test data generated by Reactis Tester. The tests with the Model Coverage Tool and TESSY were accomplished in order to be able to compare the tool-specific coverage reached by Reactis Tester with the more general coverage criteria used by Model Coverage Tool as well as with the standard coverage criteria on code level.

Table 3. Best model coverage reached by Reactis Tester; coverage measured by Model Coverage Tool and TESSY with the same data

Test object	Test	Model Coverage		Model Coverage			Code Coverage	
		with Model Coverage Tool [%]		with Reactis Tester [%]			with TESSY [%]	
		M_D1	M_C1	M_Branch	M_State	M_CondAct	C_C0	C_C1
Ctrl_S	'best' white-box	100	80	81	n.a	n.a.	100	100
Enable_S	'best' white-box	94	96	100	100	97	98	93
Enable_A	'best' white-box	87	91	100	100	85	87	85

Table 4. Number of test goals to be covered and number of test steps probed by Reactis Tester

Test object	Number of test goals			Number of test steps
	M_Branch	M_State	M_CondAct	
Ctrl_S	74	n.a	n.a.	20500
Enable_S	78	13	58	20500
Enable_A	46	11	39	20500

By comparing Table 2 and Table 3 it can be seen that the M_D1 and M_C1 model coverage as well as the C_C0 and C_C1 code coverage of Enable_A and Enable_S increased by generating test data automatically. Due to the high coverage of Ctrl_S reached during the black-box tests an optimization was only achieved for C1 model coverage (condition coverage M_C1). Table 4 shows the number of test goals (Simulink branches, Stateflow states and condition actions) to be covered and the number of whole test steps that has been probed by Reactis Tester.

3.4 Automatic test vector / test sequence generation for code coverage

Evolutionary Testing is an innovative technique for automating software tests by using metaheuristic search methods. One of the main application areas is the automatic generation of test suites which achieve full structural coverage (evolutionary structural testing).

3.4.1 Brief introduction to evolutionary structural testing

Evolutionary algorithms (EA) have been used to search for data for a wide range of applications. EA is an iterative search procedure using different operators to copy the behavior of biologic evolution. When using EA for a search problem it is necessary to define the search space and the fitness function. The algorithms are implemented in GEATbx, a widely used tool box [19].

Evolutionary testing uses EA to generate test vectors / test sequences automatically. In the case of evolutionary structural testing, the goal is to find a test suite which achieves full structural coverage for a given coverage criteria.

The general idea is a separation of the test into test goals and the use of EA to search for test data fulfilling the test goals. Each partial goal represents a set of program elements to be executed in order to achieve full coverage of the structural test metric selected, i.e. each single statement represents a partial goal when using statement coverage (C_C0) on code level. The definition of a fitness function, that represents the test goal accurately and supports the guidance of the search, is a prerequisite for the successful evolutionary test.

For evolutionary algorithms, an encoding of the individuals generated (mapped to test vectors) has to be defined. In order to address the requirement for generating sequences of test inputs the authors selected an approach which generates input sequences as a list of separate input elements. The length of this list is not defined in the test object and is, in general, not limited for control systems. A straightforward approach is to specify an upper bound to the length of the input sequences manually. A search space is formed with the information on the sequence length by replicating all the input parameters in such a way that a separate value is provided for each call and each parameter. Every individual generated by the EA represents the data of one sequence of calls of the test object. The fitness function which is decisive for successful optimization, is based on static control flow information as well as dynamic data monitored during the test execution. There are different ways

of evaluating the monitoring data and transforming them into a fitness value. One of the basic ideas is to determine the distance in the executed conditions of the software tested. In this way the search is directed with information on incorrectly evaluated conditions. The optimal solution of the search is the execution of the test goal. Details of the optimization approach used here can be found in [14], [21] and [26].

3.4.2 Details of experiments

The experiments were performed with an extended version of the automatic structural test system ET. The GEATbx [19] is the optimization component used by the system. The authors applied the standard settings for this class of problem (namely, 6 subpopulations with 50 individuals each, linear ranking, a selection pressure of 1.7, migration and competition between subpopulations). A real valued representation is used. The subpopulations employ different search strategies by using different settings for recombination (discrete and line recombination) and mutation operator (real valued mutation with differently sized mutation steps – large, medium and small).

The C code generated from the three Simulink / Stateflow moduls Ctrl_S, Enable_S and Enable_A was tested using the extended ET system. Table 5 provides an overview of the complexity measures for the test objects on the code level. The test object *Ctrl_S* is relatively small but is generated from a state diagram containing flags in all conditions.

Table 5. Information on the complexity of the test objects

Test object	Size	LOC	Nodes	param.	if-then	conditions	nesting level
Ctrl_S	16kB	220	20	28	5	6	2
Enable_S	54kB	800	140	15	51	70	10
Enable_A	44kB	520	86	11	39	56	8

Table 6 summarizes the coverage achieved for the three test objects by the test sequences generated using the ET system. The coverage degrees reached by the same data on model level, measured using the Model Coverage Tool and Reactis Tester, are also shown in Table 6.

The number of test goals (statements and branches contained in program code) to be covered and the number of individuals (test vectors) generated by the ET system are shown in Table 7.

Table 6. Code coverage reached by the ET system;
coverage measured by Reactis Tester and the Model Coverage Tool with the same data

Test object	Test	Model Coverage		Model Coverage			Code Coverage	
		with Model Coverage Tool [%]		with Reactis Tester [%]			with ET System [%]	
		M D1	M C1	M Branch	M State	M CondAct	C C0	C C1
Ctrl_S	Σ white-box	88	54	61	n.a.	n.a.	95	91
Enable_S	Σ white-box	97	98	99	100	93	99	98
Enable_A	Σ white-box	82	95	100	100	67	95	92

Table 7. Number of test goals to be covered and number of individuals generated by the ET system

Test object	Number of test goals		Number of individuals
	C C0	C C1	
Ctrl_S	17	23	650000
Enable_S	135	196	550000
Enable_A	87	126	495000

By comparing Table 2 and Table 6 it can be seen that the M_D1 and M_C1 model coverage as well as the C_C0 and C_C1 code coverage of Enable_A and Enable_S reached during black-box tests increased using test sequences generated by the ET system.

Only a few of the test goals not reached are a sequence test-specific problem. Some of them refer to the problem of unreachable code created by the code generator and some to the problem of performing an evolutionary test with flag conditions. In the next paragraphs we will give a short overview of the reasons why certain test goals were not reached.

Code coverage analysis for test object 'Ctrl_S'. All the program's conditional statements use flags that have been assigned previously. This is an area of research [3]. Some conditions access flags that have been assigned in previous calls of the function. No test data could be found that executes a special statement and the corresponding two branches. The problem of finding a solution has been traced to a search which performs poorly in the case of the corresponding flag condition.

Code coverage analysis for test object 'Enable_S'. The automatic testing of this module demonstrates the potential of the evolutionary testing approach. The test object consists of 800 lines of code leading to 135 control flow nodes, leading to 196 branches.

During the statement coverage test only one statement was not executed. The non-executable statements and branches appear when using current versions of code generators. A constant condition leads to a non executable statement.

The performance of the branch coverage test was more challenging, resulting in coverage of 98%. Only five branches were not traversed. Upon checking the code we found that two branches belong to the non-executable statement. One branch is not executable because the associated condition cannot be evaluated as *false*. The ET system was not able to find an input for two conditions which depend on sequential calls of the function. This is due to the high nesting level of the conditions to be fulfilled. At the moment it is not possible to guide the search to a solution without more data flow information. This would require further research.

Code coverage analysis for test object 'Enable_A'. This module is not particularly complex with regard to metrics, however, it contains some program structures that are difficult to test. First of all, the code is state oriented and many conditions depend on the settings of state variables. The function has a high nesting level of *if-then-else* statements and employs a state encoded using a set of flags. Again, the test object contained unfeasible code because of the use of library templates (three statements). Two other statements were not covered because of the flags used in the conditions. The results of branch coverage are similar to those of statement coverage. The test case set found did not cover 4 branches starting at flag conditions and 6 branches are placed at unreachable code. One branch could not be traversed because a precondition had not been satisfied. This branch requires an input sequence assigning two variables in previous calls of the function. The approach does not find a solution for this.

4 Analysis of the experimental results

4.1 Model and code coverage attained with black-box tests

Although information can be found on code coverage attained with black-box tests in the relevant literature (see [9] and [5]), we are not aware of corresponding data for attainable model coverage.

The C_C1 code coverage (branch coverage) degrees of 58, 89 or 100% (Table 2) determined in the experiments, accumulated during the black-box test, lie within or above the bandwidth (40-60%) specified in [15], but, apart from Ctrl_S, below the coverage required by [5] (90%). The model coverage reached with black-box tests (Table 2) fluctuates between 54% and 100% for M_D1 (decision coverage) and between 66% and 85% for M_C1 model coverage (condition coverage) and, thus, lies in a comparable order of magnitude. The coverage reached by test vector / test sequence generators, both on model and code level, is higher than that reached by black-box tests and almost satisfies the 90% coverage required by [5]. As Table 3 shows, the model coverage reached with the test sequences generated by Reactis Tester lies between 87% and 100% for M_D1 and between 80% and 91% for M_C1 model coverage. According to Table 6 the code coverage reached with the test sequences generated by the ET system varies between 91% and 98% for C_C1 code coverage.

4.2 Assessment of coverage measurements at model level

Whilst the code coverage criteria generally used are defined tool-independently and are applicable to various imperative programming languages of the 3rd generation, we consider that the coverage

measurements determinable with the Model Coverage Tool and Reactis Tester should rather be interpreted tool-specifically. The arguments for this are twofold: Firstly, the completely different coverage metrics provided by the Model Coverage Tool and Reactis Tester should be mentioned. Secondly, we observed that the set of blocks considered for the determination of coverage metrics expands during the version change from Matlab R12 to R12.1 and R13 as well as from Reactis Tester V2002 to V2003 ([16], [20]). This can lead to different, version-dependent, coverage figures for the aforementioned coverage metrics at model level – which is not a satisfactory situation. The existence of coverage criteria of various kinds in the Model Coverage Tool allows a sophisticated consideration of the model coverage: The decision coverage metric (M_D1) makes a first impression of the test quality with respect to model coverage, possible. The further metrics may then contribute to the ascertainment of the causes of missing coverage. The different coverage criteria used by Reactis Tester have a common conceptual basis, namely the decision or branching points in Simulink and Stateflow. The three Reactis Tester coverage criteria used in the experiments might together be compared with the decision and condition coverage of Model Coverage Tool, however the set of model elements covered by Reactis Tester is smaller than that covered by M_D1 and M_C1 of the Model Coverage Tool.

4.3 Correlations between model and code coverage

Due to structural similarities between the model and the code generated from it, certain interrelations between attained model and code coverage can be expected. We restricted the correlation analysis to the Model Coverage Tool. The coverage criteria used by Reactis Tester were not included in this analysis.

During the experiments a strong correlation between decision coverage (M_D1) at model level and branch coverage (C_C1) at code level, which is also sometimes termed decision coverage in the relevant literature (see e.g. [15]), was established. This correlation can be explained by considering the structure, since code branches emerge on the decision points of the model. For this reason, the branch coverage values of the code are close to the decision coverage values of the model.

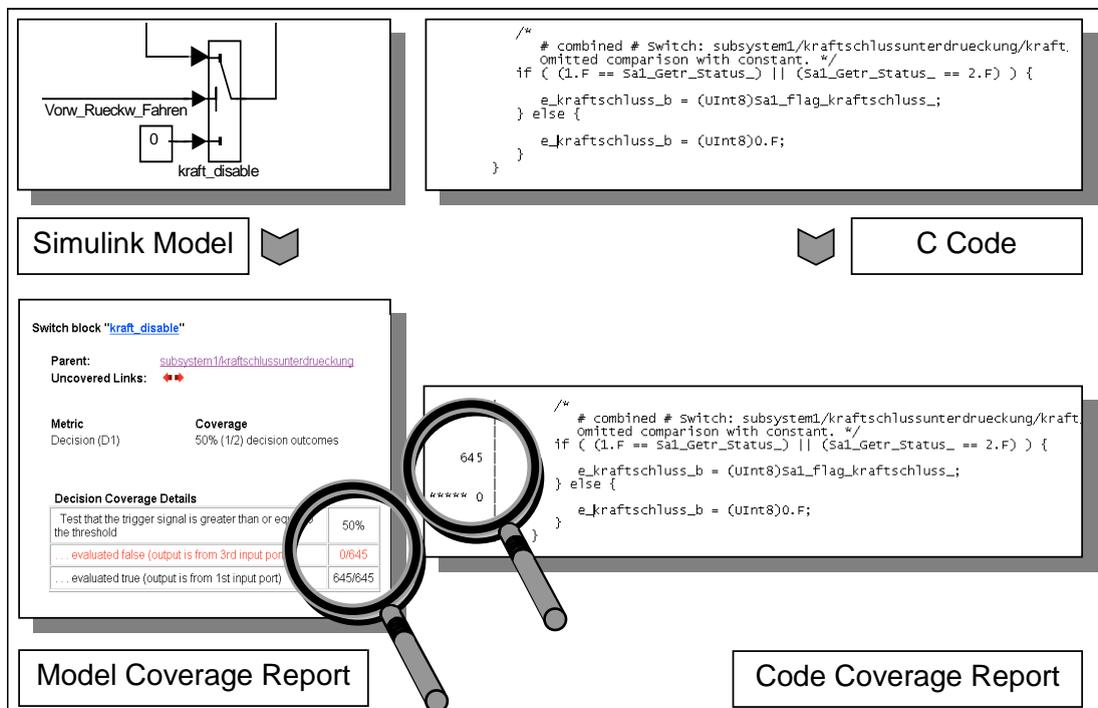


Figure 4. Comparison of M_D1 and C_C1 coverage

Figure 4 illustrates the interrelation between M_D1 and C_C1 coverage at model and code level using the example of a switch block. The switch block was realized in the course of code generation with TargetLink by means of an if-statement, i.e. a program branch, in the code. Thus, the Model Coverage Tool and TESSY both calculate the same number of executions (645 times) for one of both branches at this decision point.

Figure 5 graphs the correlation between the model coverage metric M_D1 and the code coverage metrics C_C0 and C_C1. The M_D1 model coverage degrees from Table 2 (black-box tests) have been entered on the x axis and the corresponding C_C0 and C_C1 code coverage degrees on the y axis. Taking the individual scenarios into account, the calculated correlation between M_D1 and C_C1 is 0.983 and the correlation between M_D1 and C_C0 0.973. The nearness of the correlation figures to 1 indicates the close statistical interrelation between M_D1 coverage at model level and C_C1 or C_C0 coverage at code level.

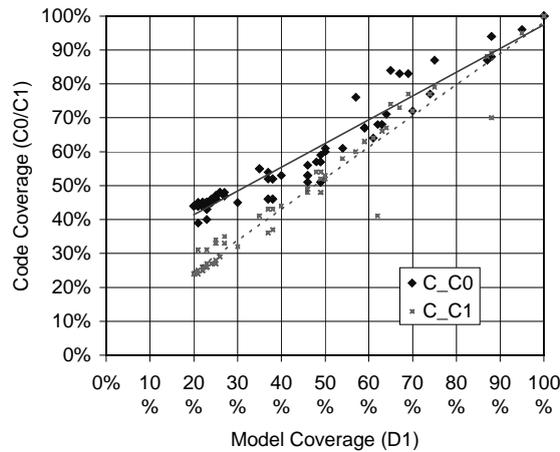


Figure 5. Correlation between model and code coverage

Under the given constraints, the C_C1 code coverage can, then, be approximated with the M_D1 model coverage. Here, the M_D1 model coverage seems to permit an estimation of the code coverage to be expected.

The exact mathematical correlation varies depending on the transformation algorithms used during the code generation. Thus, different code generators or different configurations of the code generator can lead to different realizations of one and the same model part (e.g. of a nested conditional statement) and, consequently, to potentially different coverage degrees. In any case, the correlation we gained in the experiments is stronger than the 'tenuous agreement' stated in [1].

4.4 Benefits of automatic test vector / test sequence generation

By deploying automatic test vector / test sequence generation tools we were able to optimize the structural coverage reached during black-box testing on both model and code level. While Reactis Tester works on model level and generates test sequences according to a number of tool-specific coverage metrics, the ET system generates test sequences to reach high coverage on code level according to the general and well-known code coverage metrics. Both tools are also different concerning the algorithms they use for generating test sequences: While Reactis Tester exploits the information on the structure of the test object and accomplishes a kind of 'guided simulation', the ET system employs evolutionary algorithms to reach high 'fitness values' for the generated data. Despite these differences Reactis Tester and the ET system have the same objective: The generation of test data to reach high structural coverage. The experiments described in this paper showed that both of the tools are able to achieve this objective. The automatic test data generation minimizes the effort of selecting test data covering the uncovered structural elements, which is indeed a

very complex task. Consequently, the tester can concentrate on evaluating the results of the tests executed using the generated test inputs.

The test vector / test sequence generators have been developed to optimize the test coverage concerning their own coverage metrics, i.e. metrics provided by and implemented in these tools. However, our experiments showed that these tools can also be deployed to optimize coverage metrics which are different from but structurally related to the metrics defined in the tools. The results are certainly not as good as the results achieved for the ‘native’ metrics.

As shown in [10], random testing is not capable of reaching high structural coverage as an individual test approach. Only as a complementary approach to black-box testing could it improve the coverage reached. Our experiments showed, however, that it is possible to reach coverage degrees higher than 90% with test vector / test sequence generators.

5 Conclusion

As part of an experiment, three functional modules of an automotive body control system modeled with Simulink/Stateflow were subjected to a model test, and the C code generated from these models was subjected to a code test. An analysis of the model and code coverage achieved during the test showed that comparable model and code coverage measurements exist. The correlation between these measurements depends on the way (manual or automatic) that the model was transformed into code. Furthermore, the deployment of automatic test vector / test sequence generation tools led to optimized coverage on both model and code level. This reduces the test effort needed for searching for appropriate test data to reach high coverage.

Model-based development makes it possible, besides the prevalent structural coverage measurements at code level, to also determine structural test criteria at model level and integrate these into the control and evaluation of the test process. Model coverage measurements can already be determined early in the development and test process, i.e. before the program code is available. The advantage of this procedure is that the relevant test activities are brought forward resulting in early error detection and low-cost error correction.

Model coverage can be measured (as a background activity) during the execution of functional model tests. By means of a rough analysis of the coverage reports created, the tester can establish which functional parts have not yet been checked. New test scenarios must be created manually or, as shown in the experiments, automatically in order to cover this functionality.

Model coverage degrees to be reached can (besides other criteria) be used to control the test depth and as a (necessary) test criterion for the functional model test termination. Under certain conditions they permit an estimation of the code coverage to be expected and can, in future, possibly replace or complement certain structural test criteria at code level for the program parts created using code generators.

A test strategy is effective if the tests included by that strategy are likely to reveal bugs in the test object [7]. Effective test strategies should therefore combine different test approaches. According to an effective test strategy, known from the classic software test [11], specification-based tests are first carried out in order to check, whether the requirements have been implemented correctly. The coverage reached by the specification-based tests is then measured, and additional white-box-tests are defined and executed in order to cover the yet uncovered program elements. Based on the experiments described in this paper and the availability of tools for measuring model coverage as well as tools for automatic test vector / test sequence generation we suggest an adaptation of this strategy to model-based testing as follows:

1. Specification-based tests are carried out at model level in order to check whether the requirements have been modeled correctly.
2. The model coverage reached by the specification-based tests is measured.
3. Additional white-box-tests are manually defined or automatically generated in order to cover the yet uncovered model elements. These tests are executed and evaluated.

4. The program code generated from the model is tested with the tests defined in step 1 in order to check whether the requirements have been implemented correctly.

If requirements on high code coverage exist, the model-based effective test strategy proceeds as follows:

5. The code coverage reached by the specification-based tests is measured.
6. Additional white-box-tests are manually defined or automatically generated in order to cover the as yet uncovered program elements. These tests are executed and evaluated.

Acknowledgements

The work described was performed within the SysTest project. The SysTest project is funded by the European Community under the 5th Framework Programme (GROWTH), project reference G1RD-CT-2002-00683.

References

- [1] Aldrich, W.J.: Using Model Coverage Analysis to Improve the Controls Development Process. *Proc. of AIAA Modeling and Simulation Technologies Conference and Exhibition*, Monterey, USA, 2002.
- [2] Baresel, A., Pohlheim, H., Sadeghipour, S.: Structural and Functional Sequence Testing of Dynamic and State-Based Software with Evolutionary Algorithms. *Proc. of Genetic and Evolutionary Computation Conference (GECCO 2003)*, Chicago, Illinois, USA, 2003.
- [3] Baresel, A., Sthamer, H. and Schmidt, M.: Fitness Function Design to improve Evolutionary Structural Testing. *Proc. of Genetic and Evolutionary Computation Conference (GECCO 2002)*, New York, USA, 2002.
- [4] Beacon Tester: Applied Dynamics International Inc., www.adi.com/products_be_bss_te.htm, 2003.
- [5] Bender, B.: How Do You Know When You Are Done Testing? *Proc. of 2nd International Conference on Software Testing (ICS TEST '01)*, Bonn, 2001.
- [6] Benjamin, M., Geist, D., Hartman A., Wolfsthal, A., Mas, A., Smeets, A.: A Study in Coverage-Driven Test Generation. *Proc. of 36th Design Automation Conference (DAC'99)*, New Orleans, LA, 1999.
- [7] Beizer, B.: Black-Box Testing. New York, John Wiley & Sons, 1995.
- [8] Beizer, B.: Software Testing Techniques. New York, Van Nostrand Reinhold, 1983.
- [9] Burr, K., Young, W.: Combinatorial Test Techniques: Table-Based Automation, Test Generation, and Code Coverage. *Proc. of International Conference on Software Testing, Analysis, and Review (STAR '98)*, 1998.
- [10] Conrad, M., Sadeghipour, S.: Einsatz von Überdeckungskriterien auf Modellebene – Erfahrungsbericht und experimentelle Ergebnisse (in German). *Softwaretechnik-Trends 22:2*, pp. 1-6, May 2002.
- [11] Grimm, K.: Systematisches Testen von Software – Eine neue Methode und eine effektive Teststrategie (in German). PhD Thesis, GMD Bericht Nr. 251, 1995.
- [12] Hoskote, Y., Kam, T., Ho, P.-H., Zhao, X.: Coverage Estimation for Symbolic Model Checking. *Proc. of 36th Design Automation Conference (DAC'99)*, New Orleans, LA, 1999.
- [13] ISO/TR 15497:2000 Road vehicles - Development guidelines for vehicle based software. International Organization for Standardization (ISO), 2000.
- [14] Korel, B.: Automated Test Data Generation. *IEEE Transactions on Software Engineering*, vol. 16 no. 8, pp.870-879, August 1990.
- [15] Malaiya, Y. K. et. al.: The Relationship between Test Coverage and Reliability. *Proc. of the 5th International Symposium on Software Reliability Engineering*, 1994.
- [16] Model Coverage Tool, The MathWorks Inc., www.mathworks.com/products/slperftools/, 2001.
- [17] Myers, G. J. The Art of Software Testing, Wiley, 1979.
- [18] Pitschinetz, R.: Das Testsystem TESSY Version 2.5 (in German). Technical Report FT3/S-1999-001. DaimlerChrysler AG, Berlin, 1999.
- [19] Pohlheim, H.: GEATbx - Genetic and Evolutionary Algorithm Toolbox for Matlab. <http://www.geatbx.com/>, 1994-2003.
- [20] Reactis User's Guide, Reactive Systems Inc., www.reactive-systems.com, 2003.
- [21] Sthamer, H.-H.: The Automatic Generation of Software Test Data Using Genetic Algorithms. PhD Thesis, University of Glamorgan, Pontyprid, Wales, Great Britain, 1996.

- [22] Stateflow User's Guide (Version 4). The MathWorks Inc., Natick (US), 2000.
- [23] Structural Coverage Metrics – Their Strengths and Limitations. White Paper, IPL Information Processing Ltd., Bath (UK), www.iplbath.com/pdf/p0823.pdf, 1997.
- [24] TargetLink Production Code Generation Guide, dSpace Inc., www.dspaceinc.com , 2001.
- [25] Using Simulink (Version 4.1). The MathWorks Inc., Natick (US), 2000.
- [26] Wegener, J., Sthamer, H., Baresel, A.: Evolutionary Test Environment for Automatic Structural Testing. *Special Issue of Information and Software Technology*, vol. 43, pp. 851–854, 2001.
- [27] Zhu, H. , Hall, P. A. V., May, J. H. R.: Software unit test coverage and adequacy. *ACM Computing Surveys*, Vol. 29, Issue 4 (December 1997), pp. 366 – 427, 1997.