

A Systematic Approach to Testing Automotive Control Software

Mirko Conrad

DaimlerChrysler AG, Research and Technology

Copyright © 2004 Convergence Transportation Electronics Association

ABSTRACT

Usually, the testing of today's ECU software follows a gut feeling approach, leading to test gaps and test redundancies. This paper presents a new, more systematic way of testing automotive control software. The central element of the approach is the Classification-Tree Method for EMBEDDED SYSTEMS (CTM/ES). Using an interface description, which can be based on the specification and / or an executable model of the software, test scenarios can be derived systematically and described in a graphical way so as to provide the user with visual information about test coverage. The CTM/ES can be integrated into an overall test strategy for automotive control software developed in a model-based way. The approach opens up a new way of assuring quality for embedded control software which is especially designed for automotive software developers.

INTRODUCTION

The rapid increase in the software complexity of today's ECUs makes testing a central and significant task within automotive control software development. *Dynamic Testing*, meaning the execution of a test object with selected (sequences of) test data for the analysis of behavior in these cases, is the most important and most widespread measure for software quality assurance [Lig92].

Since exhaustive testing is impossible in practice, dynamic testing is always a sampling procedure. A subset of test scenarios¹ which is as small as possible and which is able to reveal as many errors in the test object as possible, needs to be selected from the complete set of possible test scenarios. The selection of adequate sample elements (test scenarios) decides on the extent and quality of the whole test (cf. Fig 1).

¹ General description of a test case which specifies a set of (detailed) test data or test data sequences.

The work described was partially supported by the IMMOS project funded by the German Federal Ministry of Education and Research (project ref. 01ISC31D)
www.immos-project.de

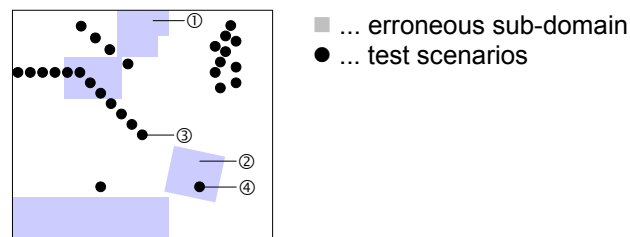


Figure 1: Input domain, errors and test scenarios

In Fig 1 the input domain of the test object is depicted as a black framed rectangle, the areas marked gray (e.g. ①, ②) illustrate erroneous sub-domains. Those parts of the input domain which are covered by test scenarios are presented as circles. If a test scenario lies within a gray area (e.g. ④) this means that it is an error-revealing test scenario, otherwise (e.g. ③) it is non error-revealing.

Because of the exceptional significance of the selection process there is a need for *systematic* techniques which support the tester when designing adequate test scenarios. The application of these *test design techniques*² systematizes the selection of test scenarios and makes it both comprehensible and reproducible [Sim97].

In addition to the test design techniques, there is a need for suitable *notations* for the documentation of the *test scenarios* determined. The automatic creation of test scenarios of functional tests is desirable but only possible to a limited extent. As a result, test design must largely be carried out manually. It requires a test (scenario) notation which caters for the human user [Con01].

When testing automotive control software, an *ad-hoc selection of test scenarios* is typical, i.e. the test scenarios are determined without (explicitly) defined procedures. This selection depends on the experience and expertise of the tester and can only be reproduced with difficulty, if at all. An ad-hoc selection of test scenarios leads, on the one hand, to test scenarios which are more or less redundant (cf. upper right area in Fig. 1) and, on

² Standardized procedures for selecting test scenarios on the basis of certain sources of information (e.g. functional specification, executable model, program code).

the other hand, to test gaps (cf. lower left area in Fig. 1). Moreover, one factor at a time testing is typical, i.e. only variations in single influencing variables take place (cf. test scenarios arranged like beads on a string in Fig 1). As a rule, it is impossible to make statements as to the quality or completeness of the ad-hoc test scenarios. With regard to the number of test scenarios, their level of error detection is often inadequate.

The test notations which are often used when developing automotive control software, such as the direct description of the test scenarios in the form of time-dependent value courses or the use of test scripts, lead to a description of test scenarios on a very low level of abstraction, making maintainability and reuse difficult. In addition, the methodical support for the use of the notations is insufficient.

Those software test design techniques which are methodically well-founded and which exist in the area of general software engineering, are often tailored towards administrative or scientific software. They are not sufficiently adapted to the specifics of embedded automotive control systems and can therefore not be transferred just like that. The temporal aspects of the test scenarios, in particular, are not suitably taken into account.

In order to effectively minimize these deficits, the Classification-Tree Method for Embedded Systems (CTM/ES), an efficient procedure for the selection and description of test scenarios for embedded automotive software, was developed.

THE CLASSIFICATION-TREE METHOD FOR EMBEDDED SYSTEMS (CTM/ES)

The CTM/ES [CDF+99, Con04, LBE+04], an extension of the Classification-Tree Method [GG93], is a black-box test design technique which allows systematic test design for embedded systems and their control software as well as a comprehensive graphical description of time-dependent test scenarios by means of abstract signal waveforms that are defined stepwise for each input. The basic concept is to split up the input domain of the test object according to different aspects usually corresponding to different input data sources. The different partitions, called classifications, are subdivided into (input data) equivalence classes. Finally, different combinations of input data classes are selected and arranged into test sequences.

The starting point for the proposed systematic test design approach is an interface description of the test object. Based on this, and by using the Classification-Tree Method for Embedded Systems, the tester can derive test scenarios systematically and describe them graphically. The graphical representation provides the user with visual information about test coverage. Test coverage indicates how well the test scenarios cover the possible test input combinations and is therefore an important test metric.

Example PedInt

For illustration purposes throughout this paper, a simplified component is used to interpret the pedal positions. This subsystem can be employed as a preprocessing component for various vehicle control systems. The pedal interpretation (short: PedInt) subsystem interprets the current, normalized positions of accelerator and brake pedal (ϕ_{Acc} , ϕ_{Brake}) by using the actual vehicle speed (v_{act}) as desired torques for drivetrain and brake (T_{des_Drive} , T_{des_Brake}), see [Höt97] for details. Furthermore, two flags (AccPedal, BrakePedal) are calculated which indicate whether or not the pedals are considered to be depressed.

The software requirements on the PedInt subsystem are summarized in Tab 1. For the following considerations, those sub-functions of PedInt should be tested which both flags calculate (SR-PI-01.x). □

Table 1: Software requirements specification of PedInt (excerpt)

ID	Description	
SR-PI-01	Recognition of pedal activation If the accelerator or brake pedal is depressed more than a certain threshold value, this is indicated with a pedal-specific binary signal.	
SR-PI-01.1	Recognition of brake pedal activation If the brake pedal is depressed more than a threshold value ped_min , the BrakePedal flag should be set to the value 1, otherwise to 0.	▶
SR-PI-01.2	Hysteretic behavior of brake pedal activation No hysteresis is to be expected during brake pedal activation recognition.	▶
SR-PI-01.3	Recognition of accelerator pedal activation If the accelerator pedal is depressed more than a threshold value ped_min , the AccPedal flag should be set to the value 1, otherwise to 0.	▶
SR-PI-01.4	Hysteretic behavior of accelerator pedal activation No hysteresis is to be expected during accelerator pedal activation recognition.	▶
SR-PI-02	Interpretation of pedal positions Normalized pedal positions for the accelerator and brake pedal should be interpreted as desired torques. This should take both comfort and consumption aspects into account.	
SR-PI-02.1	Interpretation of brake pedal position [...]	▶
SR-PI-02.2	Interpretation of accelerator pedal position [...]	▶

TEST INTERFACE DETERMINATION

In the first step, the interface which is relevant for the test has to be determined.

Let the basis of further considerations be a test object with n inputs and m outputs (Fig 2). The input interface I' of the test object is represented via the input variables i'_i ($i=1,\dots,n$), the output interface O' via the output variables o'_j ($j=1,\dots,m$).

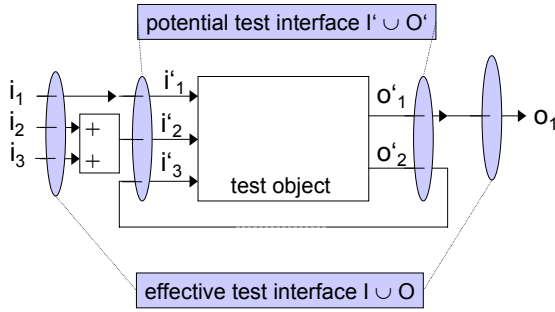


Figure 2: Potential and effective test interface

The input variables of the test object form the potential inputs for the test. They are consequently called *potential test input interface*. It is not necessary, however, to use the potential test interface for test object stimulation on a one-to-one basis: values which are fed back, for example, do not need to be predetermined as they are generated by the system environment (e.g. i'_3 in Fig. 2). On the other hand, it is often easier to describe a complex input signal by means of the (additive or multiplicative) superposition of two sub-signals. In this case, the two sub-signals would be described instead of the potential interface signal (e.g. $i'_2 = i_2 + i_3$ in Fig. 2). The variables actually used for the test are referred to as *effective test interface* $I \cup O$. If there are differences between the potential and the effective interface for a certain test object these have to be mapped onto each other.

In order to test the test object, each input variable of the effective test interface has to be stimulated with a time-variant signal. The output variables have to be recorded. Thus, the effective test interface determines the *signature of the test scenarios* belonging to it.³

$$\begin{aligned}
 I: \quad & i_1^*: \quad \text{Time} \rightarrow I_1 \\
 & \dots \\
 & i_n^*: \quad \text{Time} \rightarrow I_n \\
 O: \quad & o_1^*: \quad \text{Time} \rightarrow O_1 \\
 & \dots \\
 & o_m^*: \quad \text{Time} \rightarrow O_m
 \end{aligned}$$

Example PedInt

Table 2 describes the potential test interface $I' \cup O'$ of the PedInt component. For some fundamental test scenarios it is identical with the effective test interface $I \cup O$.

Table 2: Interface description of PedInt

variable	\leftrightarrow	unit	domain	data type
v_{act}	\blacktriangleright	m/s	[-10,70]	real
ϕ_{Brake}	\blacktriangleright	%	[0,100]	real
ϕ_{Acc}	\blacktriangleright	%	[0,100]	real
T_{des_Drive}	\blacktriangleleft	Nm		real
T_{des_Brake}	\blacktriangleleft	Nm		real
AccPedal	\blacktriangleleft	—	{0,1}	bool
BrakePedal	\blacktriangleleft	—	{0,1}	bool

Thus, the signature for the fundamental functional test scenarios is as follows⁴:

$$\begin{aligned}
 I: \quad & \phi_{Acc}^*, \phi_{Brake}^*: \quad \text{Time} \rightarrow R_{[0,100]} \\
 & v_{act}^*: \quad \text{Time} \rightarrow R_{[-10,70]} \\
 O: \quad & T_{des_Drive}^*, T_{des_Brake}^*: \quad \text{Time} \rightarrow R \\
 & AccPedal^*, BrakePedal^*: \quad \text{Time} \rightarrow \{0,1\}
 \end{aligned}$$

TEST INPUT PARTITIONING

In a second step, the admissible values of the signals which form the effective input interface must be disjointedly and completely partitioned into (equivalence) classes which are suitable abstractions of individual input values for testing purposes. The partitioning is graphically represented by means of a *classification-tree*.

The partitioning aims to achieve a selection of the individual *equivalence classes* in such a way that they behave homogeneously with respect to the detection of potential errors. That is, the test object behaves either correctly or erroneously for all the values of one class (*uniformity hypothesis*).

A heuristic procedure has proved successful in approaching this ideal partitioning as much as possible in practice. The inputs' data types and value ranges provide the first valuable clues to partitioning: where real-valued data types with minimum and maximum values established are concerned, it is possible, for example, to create a standard class each for the boundary values, for the value of zero and for those intervals in between. Similar standard classifications which are data type-

⁴ If, however, during a later robustness test, inaccuracies in the recording and A/D conversion of the brake pedal sensor are to be reproduced, it would be possible to additively superpose the corresponding input signal with a noise signal. In order to do this, one would extend the effective test interface with an additional input signal

$$\phi_{Bra_noise}^*: \quad \text{Time} \rightarrow R_{[0,1]}$$

The mapping between the potential and the effective test input interface would then be as follows:

$$\begin{aligned}
 \phi_{Acc}(t) &= \phi_{Acc}^*(t) \\
 \phi_{Bra}(t) &= \phi_{Bra}^*(t) + \phi_{Bra_noise}(t) \cdot \text{rnd}(t) \\
 v_{act}(t) &= v_{act}^*(t)
 \end{aligned}$$

³ I_i and O_j are the sets of admissible values for the variables i_i and o_j respectively.

specific can also be utilized for different data types [CDS+02, Con04].

In general, the data type-specific, standard classifications are not detailed enough for a systematic test. They have to be refined or modified manually in order to approach partitioning according to the uniformity hypothesis. The quality of the specification and the tester's experience are crucial in this respect.

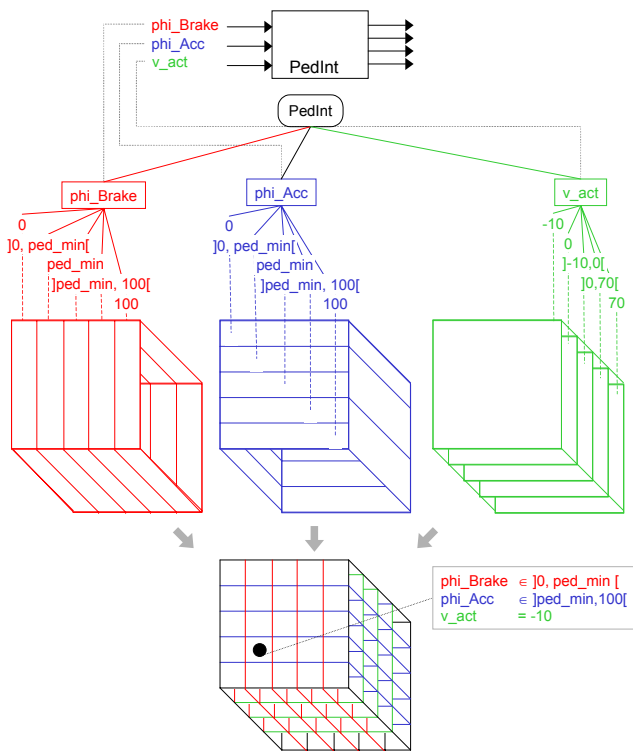


Figure 3: From test interfaces to classification-trees

Example PedInt

The classification-tree for the PedInt example is derived from standard partitioning for real-valued signals for the three effective interface input signals.

According to standard partitioning, the pedal positions which can take values from the range of 0% to 100%, would be partitioned into 3 classes 0,]0, 100[, and 100. For the vehicle speed the five partitions -10,]-10,0[, 0,]0,70[, and 70 are obtained⁵.

As v_act is only of secondary importance for testing those sub-functions of PedInt which both flags calculate, the standard classification has not been modified there. On the other hand, the evaluation of the pedal positions recognizes a pedal as depressed only if it is activated above a certain threshold value ped_min. Therefore, the

⁵ [x, y] denotes an interval, closed on both sides,]x, y[an interval, open on both sides, with the boundary values x and y.

pedal values above and below the threshold should be considered separately because behavior is expected to differ. A class has also been added for the exact threshold value. In order to keep the classification-tree flexible, parameter names were partly used as class boundaries rather than fixed values. The result is a final partitioning of the pedal positions into the classes 0,]0, ped_min[, ped_min,]ped_min, 100[, and 100.

The partitioning into classes is visualized by means of a classification-tree (Fig. 3, middle part). The lower part of Fig. 3 shows how the equivalence classes for the individual input variables partition the input domain of the entire test object. □

TEST SCENARIO DETERMINATION

In the third step, test scenarios are determined based on the test input partitioning. The test scenarios describe the course of these inputs over time in a comprehensive, abstract manner. Each scenario captures a data abstraction of the test object's inputs and thus describes – largely independent of detailed test data – what is to be tested.

The classification-tree is used to define the columns of a *combination table*. In order to represent test scenarios in an abstract way, they are decomposed into individual *test steps*. These compose the rows of the combination table according to their temporal order (Fig. 4, upper and middle part).

The input situation for each test step is defined by combining classes of different classifications from the classification-tree. This is done by marking the appropriate tree elements in the main column of the combination table. This leads to a sequence of input situations. The duration of each input situation can be captured by the annotation of time tags in the rightmost column of the combination table (Fig. 4, middle part).

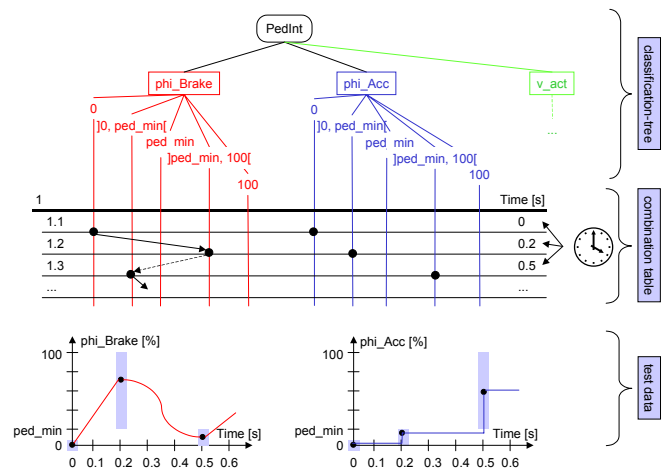


Figure 4: Defining test sequences

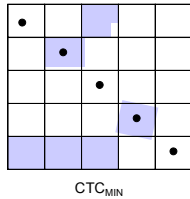
Time-dependent behavior, i.e. changing the values of an input over time in successive test steps can be modeled

After the determination of test sequences has been completed, it is necessary to check if they ensure a sufficient *test coverage*. At this early stage of the testing process, the CTM/ES already allows the determination of different abstract coverage criteria on the basis of the classification-tree and the test sequences:

A *requirements coverage analysis* can verify whether all requirements of the requirements specification which concern the test object are covered sufficiently by the test scenarios. In general, an n:m-relationship exists between requirements and test scenarios. In the course of the analysis, it is necessary to prove that every requirement is being checked by at least one test scenario and that the existing test scenarios are adequate to test the respective requirements. If necessary, the model-based black-box testing (MB³T) methodology [CFS04] provides the tester with the equipment for a more sophisticated requirements coverage analysis.

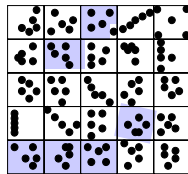
Furthermore, the CTM/ES supports a *range coverage analysis*. This analysis checks the sufficient consideration of all equivalence classes defined in the classification-tree in the test scenarios. This check can be executed, according to the respective application case, by using different, so-called *classification-tree coverage criteria* (CTC) (cf. [GG93, LBE+04]):⁶

- CTC_{MIN}: The *minimum criterion* requires every single class in the tree to be selected in at least one test step. The minimum criterion is normally accomplishable with a few test sequences. The error detection rate, however, is rather low.



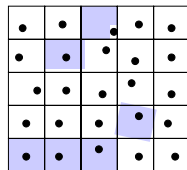
CTC_{MIN}

- CTC_{MAX}: The *maximum criterion* requires every possible class combination to be selected in at least one test step. The fulfillment of the maximum criterion should result in a high error detection rate. However, the problem with this criterion is a possible combinatorial explosion. This makes it impracticable when a large number of classes is involved.



CTC_{MAX}

- CTC_n: The *n-wise combination criterion* presents a compromise. Here, it is necessary to ensure that every possible combination of n classes is selected in at least one test step. For example, a pair-wise combination of classes (CTC₂) is practicable.



CTC₂

⁶ The illustrations on the right-hand side show 2-D projections of the input domain of the PedInt example with test scenarios fulfilling the criteria described on the left-hand side.

The selection of appropriate criteria has to take place in a problem-specific way during test planning. If the criteria defined beforehand have not been sufficiently fulfilled, additional test steps / test sequences need to be added until the required criteria are reached.

Example PedInt

Table 4 presents the coverage of requirements by means of those test scenarios that have been determined up to this point. The necessary requirements coverage has thus been fulfilled for the requirements SR-PI-01.1 to SR-PI-01.4. The higher-level requirement SR-PI-01 does not have to be tested separately, as it has already been tested implicitly by all of the derived requirements being tested.

Table 4: Traceability matrix PedInt

	SR-PI-01	SR-PI-01.1	SR-PI-01.2	SR-PI-01.3	SR-PI-01.4
test scenario #1	(✓)	✓		✓	
test scenario #2	(✓)		✓		✓
test scenario #3					

The determined test scenarios fulfill the minimum criterion CTC_{MIN}, but they do not fulfill any higher criteria, such as CTC₂ or even CTC_{MAX}. In practice, this means that further test scenarios would have to be added until the attainable classification-tree coverage criterion is also fulfilled. □

TEST DATA REFINEMENT

In a fourth step, the abstract test scenarios will be refined into signal waveforms.

The test scenarios gained so far contain abstracted stimulus information because only equivalent classes, but no specific data have been used. Thus, in a further step, the test data is instantiated by the use of specific numbers.

The test scenarios defined in the combination table form signal corridors for the individual input signals, within which the actual courses of input values must be located. The borders of the equivalent classes form the constraints of the value ranges at the respective test steps. This step is illustrated in the lower part of Fig. 4.

Example PedInt

On the basis of the uniformity hypothesis any value within the marked interval can be selected when determining the values at the base points.

Within the scope of this example the principle of mean value testing is being used, i.e. in each case the mean values of the equivalence classes selected are used as test data (ped_min = 5 %, Fig.7). □

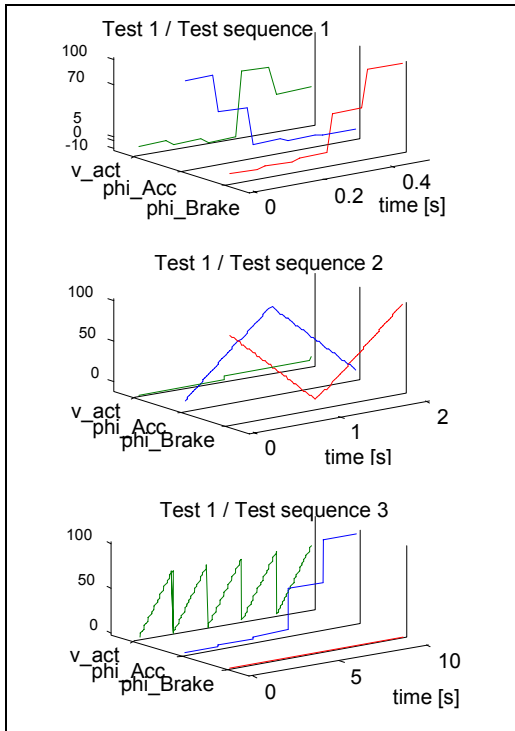


Figure 7: Test data curves PedInt

What has been presented so far, is how to systematically determine and compactly and adequately describe test scenarios for the black-box test of automotive control software with the help of CTM/ES. Requirements coverage and input data range coverage (classification-tree coverage) allow for the quality assessment of the determined test scenarios.

CTM/ES can be deployed as a black-box test design

technique, independently from an actual development process. The only pre-requisites are the description of the test object's behaviour and interface.

In the following chapter the deployment of CTE/ES within the framework of model-based development of control software will be described exemplarily as part of being embedded in a specific development procedure.

A TEST STRATEGY FOR AUTOMOTIVE CONTROL SOFTWARE

This chapter introduces a test strategy for the model-based test, which focuses on executable models as part of analytical quality assurance. Furthermore, it explains the role of CTM/ES within this strategy.

MODEL-BASED DEVELOPMENT

Model-based development can be regarded as an answer of numerous automobile suppliers and manufacturers to the increased demands on the development of embedded software [KCF+04, SZ03, Bro03, Rau02]. This innovative development approach is characterized by the integrated deployment of executable graphical models for specification, design and implementation, using commercial modeling and simulation environments such as Matlab/Simulink/Stateflow [Matlab] or ASCET-SD [ASCED-SD]. These tools use block diagrams and extended state machines as modeling notations.

Very early in this development procedure an executable model of the control software (functional model) is created, which can be simulated as well as tested. This executable model is used throughout the downstream development process and forms the 'blueprint' for the automatic or manual coding of the embedded software and its integration into the electronic control unit (ECU). The seamless utilization of models facilitates highly consistent and efficient development.

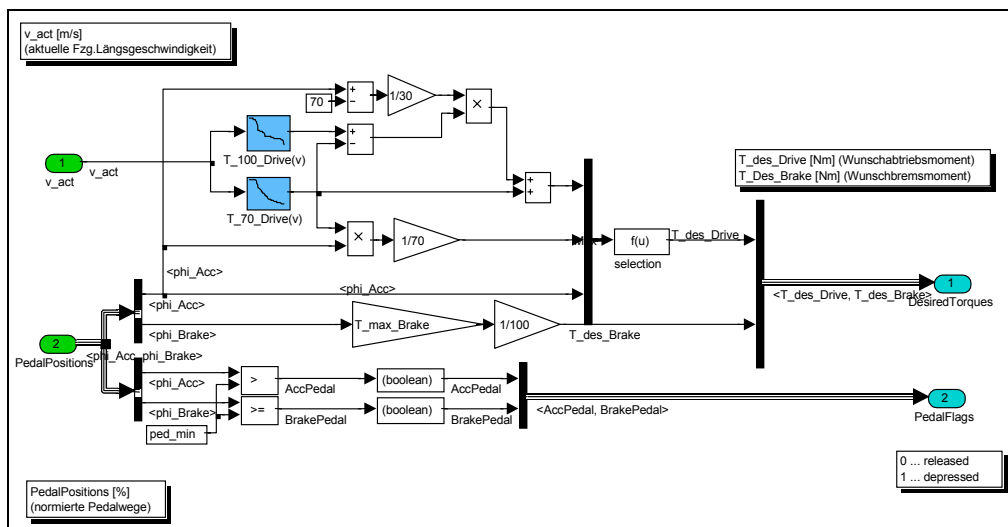


Figure 8: Simulink model of PedInt component

Example PedInt

Fig. 8 shows the functional Simulink model for the PedInt component which was developed based on the requirements and interface specification described earlier. □

MODEL-BASED TEST STRATEGY

A singular testing technique does not generally lead to sufficient test coverage. Therefore, in practice, the aim is for complimentary test design techniques to be combined in the most adequate way to form a test strategy. Here, effective test strategies include combinations of functional and structural testing techniques [Bal98, Gri95].

The aim of an effective test strategy is to guarantee a high probability of error detection by combining appropriate testing techniques. An effective model-based test strategy must take into account the specifics of model-based development and especially the existence of an executable model in an adequate way.

The systematic selection of test scenarios from the functional specification, the interfaces, as well as the executable model of the embedded software forms the focal point of such a model-based test strategy. In addition, an adequate structural test criterion is defined on model level, with the help of which the coverage of the tests thus determined can be evaluated and the definition of complementary tests can be controlled. If sufficient test coverage has thus been achieved on model level, the functional as well as the structural scenarios can be re-used for testing the control software generated from the model and the control unit within the framework of back-to-back tests. In this way, the functional equivalence between executable model and derived forms of representation can be verified (see [BCS+03, CFS04]). In detail, the procedure will be as follows:

① Systematic functional testing on model level

First, at an early stage of development, functional test scenarios are derived systematically from the functional specification, the interfaces, and / or the executable model.

CTM/ES is an advisable test procedure for a systematic functional test on model level. Alternatively, different functional testing procedures can be used. Test scenarios which are determined in this way can be described with the CTM/ES-inherent notation in order to be able to document tests derived from different sources clearly and consistently.

The test scenario determination is to be continued until sufficient requirements and range coverage has been achieved. The test scenarios determined are then to be executed within the frame of a model test. The model reaction to the test scenarios is to be documented.

Example PedInt

The test scenarios which have been defined using the Classification-Tree Method for Embedded Systems can be applied to the different representation forms of the test object.

If the model represented in Fig. 8 is being stimulated with the defined test scenarios, the system will reply as shown in Fig. 9.

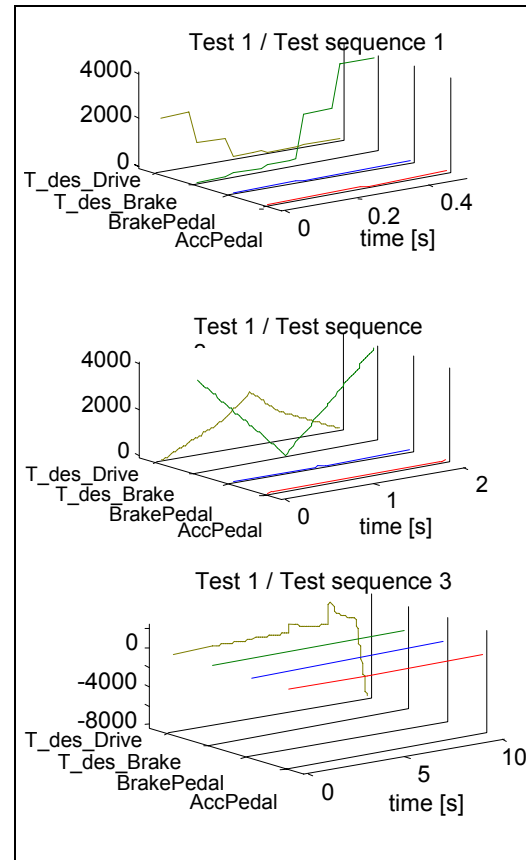


Figure 9: System reaction PedInt

An actual evaluation based on the system's reactions shall be illustrated in an exemplary way with test scenario #1.

The top plot in Fig. 10 shows the brake-pedal travel over time. Areas in which it is greater than the threshold value ped_min are gray. The subjacent plot shows the expected (black line) and the actual (dotted line) value of the brake-pedal flag. As they are not identical in the area between 0.2 and 0.3, the brake-pedal recognition is not in accordance with the specification. In analogy, the two lower plots are concerned with the accelerator pedal activation recognition. Here, it is functioning correctly. Actual and expected signal response are in agreement.

An analysis of the erroneous behavior shows that the comparative block, which compares the brake pedal travel phi_Brake with the respective threshold value

ped_min, incorrectly checks for \geq instead of $>$ (cp. Fig 8, lower left-hand corner).

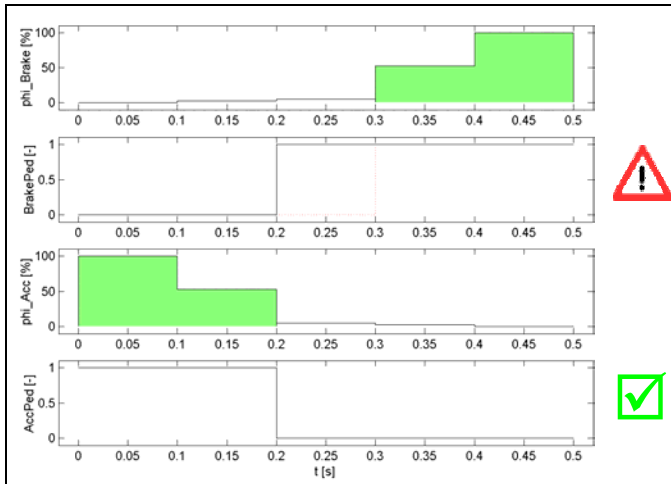


Figure 10: Test evaluation PedInt (test scenario #1)

A regression test executed after the comparative block has been corrected then asserts a behaviour in accordance with the specification.

② Monitoring model coverage

In order to be able to make statements about the coverage of the test object's structure as early as possible, it is advisable to use structure-oriented coverage criteria on model level, as their performance can be gauged even before the actual software is available.

When using the modeling and simulation environment Matlab / Simulink / Stateflow the decision coverage criterion on model level (D1 model coverage) [Ald02, SLPerf] lends itself to this procedure, since a good correlation with the subsequent code coverage can be expected [BCS+03].

The degree of model coverage that has been achieved with the test scenarios determined in step ① (and, as the case may be, in step ③) has to be monitored. If sufficient model coverage is achieved, step ④ can be initiated. If this is not the case, the next step will have to be step ③.

③ Structural testing on model level

If model coverage has not been sufficient up to this point, the model parts which have not been covered need to be identified. Test scenarios for the coverage of these model parts have to be created systematically and to be added to the existing test suite. Then, step ② has to be executed again. This procedure is to be continued until sufficient model coverage has been achieved.

For the description of these supplemental test scenarios for structural testing, the CTM/ES-inherent means of representation can be used.

These test scenarios are to be executed in the context of a model test. The model reaction to the test scenarios is to be documented.

The supplemental structural testing scenarios can be created either manually or in an automated way. Experiences with this procedure are laid down in [BCS+03, Ran03, Ald02], amongst others.

④ Executing back-to-back tests with software and ECU

If, in the further course of development, the software derived from the model, or the control unit respectively, is available, the test scenarios (created in step ① and ③) are to be repeated for the software or the embedded system. Again, the system reaction is to be documented and to be compared with the model reaction. Appropriate comparative methods are described, for example, in [CFP03].

In the case of sufficient similarity (functional equivalence) it can be assumed that the transformation of the model into C code and its embedding in the control unit has been achieved without error.

The model-based test strategy, which results from steps ① to ④, is schematically depicted in Fig. 11. The selection of the testing technique in step ①, the structural testing criterion in step ②, as well as the comparative method in step ④ are to be adapted in a project-specific way if required.

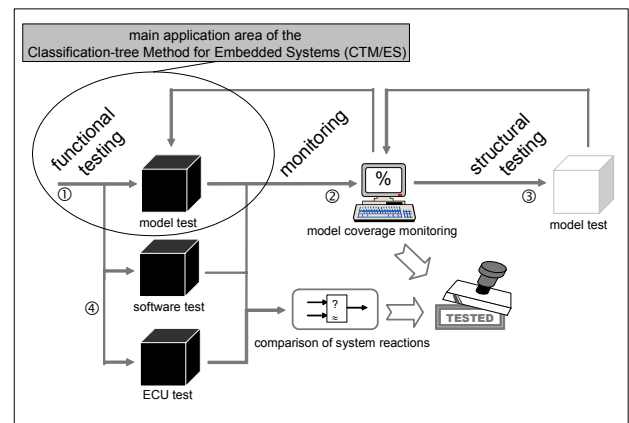


Figure 11: Model-based test strategy

TOOL-SUPPORT AND RELATED WORK

The systematic definition of functional test scenarios with CTM/ES is supported by the different versions of the classification-tree editor CTE/ES [Mis04, CTE/ES] and CTE/XL [LW00, CTE/XL]. Alternative or supplemental test design approaches for embedded control software, especially for reactive components, are described in [HPP+03, Leh02, BPS03].

Comprehensive automation of functional testing on model level (step ①) is possible with the model-based

test environment MTest [CDF+99, LBE+04, MTest], which utilizes the CTE/ES for the test design. Basic functional tests on model level can be created and executed with the Matlab Automated Testing Tool MATT [MATT]. The I/O oriented test design approach within the STEP-X test framework [HSM+04] utilizes a simplified version of CTM/ES.

Monitoring of model coverage (step ②) can be carried out by the model coverage tool [SLPerf] or the monitoring component by Reactis Tester [Reactis].

An automation of structural test design on model level (step ③) is possible with Reactis Tester [Reactis, Ran03] or with an extended version of the evolutionary testing system ET [BCS+03, WSB01].

The execution of back-to-back tests between model and the control software derived from it (step ④) can again be carried out with MTest. In order to also check the embedding of the software in the control unit, test scenarios can be exported from MTest to AutomationDesk [ADesk, LRR03]. There, it can be reused for the (Hardware-in-the-Loop) test of the control unit.

The results of back-to-back tests can be analyzed with the help of MEval [CFP03, MEval]. A modified version of the algorithms used therein is also being employed by the STEP-X environment. For further information on the execution of back-to-back test see [CFP03, Rau02].

SUMMARY AND CONCLUSION

There are currently no universally recognized measurement procedures for software quality. Measuring is therefore often replaced with testing [HV98].

The selection of suitable, i.e. error-sensitive, test scenarios (test design) is the most crucial activity for a trustworthy software test. It ultimately determines the scope and quality of the test. Moreover, an appropriate description of the test scenarios used is essential for the human tester, particularly when developing automotive control software.

A systematic approach for the selection of test scenarios and a notation for their appropriate description, must therefore form the core elements of a testing approach for automotive control software. In order to allow testing to begin at an early stage, test design should build upon development artifacts available early on, such as the specification or an executable model.

Current practice, however, is characterized by test scenario design according to a gut feel approach, leading to test gaps and test redundancies. As a rule, test design begins late in the development process when the software is already available. The test scenarios are described on a low level of abstraction, making them difficult to understand and reuse.

In this paper a new, more systematic way of testing automotive control software has been presented. The core element of the approach is the Classification-tree Method for Embedded Systems (CTM/ES).

Based on the data-oriented partitioning of the input domain into equivalence classes, time-variable test scenarios can be derived systematically and described graphically.

A compact, problem-oriented representation, suitable for a human tester and containing a high potential for reusability, is transformed into a technical representation which is suited to the test objects' stimulation.

The abstract graphical notation provides the tester with visual information about data range coverage. Data range coverage indicates how well the test scenarios cover the range of possible test input combinations and is therefore one of the most important test metrics. In addition, requirements coverage analysis is supported.

The CTM/ES also allows the uniform representation of test scenarios which result from other (functional and structural) test procedures. It can thus be employed as a universal means of description for almost all the test scenarios which arise as part of automotive control software testing. Another distinct advantage of the CTM/ES is that it can even be applied in situations in which the specification is incomplete or out of date.

The CTM/ES has recently been successfully employed in different control software development projects. One of the main application areas is the testing of automotive software developed in a model-based way.

To seamlessly support this application area, functional (black-box) testing with the CTM/ES was combined with structural (white-box) testing on model level. This combination, together with the subsequent reuse of the test scenarios for software and ECU testing, leads up to an integrated model-based test strategy.

Automotive control software which has been developed in a model-based way can be tested systematically and efficiently by means of the CTM/ES and the tools based thereon, such as the classification-tree editor CTE, as well as the model-based test environment MTest.

The systematic determination of test scenarios according to functional as well as structural aspects facilitates an effective detection of software-based error types. Sufficient test coverage is ensured within the frame of the effective test strategy by using a combination of requirements-oriented, data-range-oriented, as well as structure-oriented coverage criteria.

REFERENCES

- [Ald02] Aldrich, W. J.: Using Model Coverage Analysis to Improve the Controls Development Process. AIAA Modeling and Simulation Technologies Conference and Exhibition, Monterey, US, 2002.
- [ASCED-SD] ASCET-SD (product information). ETAS GmbH, de.etasgroup.com/products/ascet_sd/.
- [ADesk] AutomationDesk (product information). dSPACE GmbH, www.dspace.de/ww/en/pub/products/prodover/expsoft/automdesk.htm
- [Bal98] Balzert, H.: Lehrbuch der Software-Technik. Band 2, Spektrum Verlag, 1998
- [BCS+03] Baresel, A., Conrad, M., Sadeghipour, S., Wegener, J.: The Interplay between Model Coverage and Code Coverage, 11. Europ. Int. Conf. on Software Testing, Analysis and Review (EuroSTAR 03), Amsterdam, NL, 2003.
- [BN03] Broekman, B., Notenboom, E.: Testing Embedded Software. Addison-Wesley, 2003
- [BPS03] Baresel, A., Pohlheim, H., Sadeghipour, S.: Structural and Functional Sequence Testing of Dynamic and State-Based Software with Evolutionary Algorithms. Genetic and Evolutionary Computation Conference (GECCO 2003), Chicago, US, 2003.
- [Bro03] Broy, M.: Automotive Software Engineering. 25th Intl. Conference on Software Engineering (ICSE 03), Portland, US, 2003, pp. 719-720.
- [CDF+99] Conrad, M., Dörr, H., Fey, I., Yap, A.: Model-based Generation and Structured Representation of Test Scenarios. Workshop on Software-Embedded Systems Testing (WSEST), Gaithersburg, US, 1999.
- [CDS+02] Conrad, M., Dörr, H., Stürmer, I., Schürr, A.: Graph Transformations for Model-based Testing. Lecture Notes in Informatics (LNI), Vol. P-12, Köllen Verlag, 2002, pp. 39-50
- [CFP03] Conrad, M., Fey, I., Pohlheim, H.: Automatisierung der Testauswertung für Steuergerätesoftware. VDI-Berichte, Vol. 1789, VDI Verlag, 2003, pp. 299-315.
- [CFS04] Conrad, M., Fey, I., Sadeghipour S.: Systematic Model-Based Testing of Embedded Automotive Software - The MB³T Approach. ICSE 2004 workshop on Software Engineering for Automotive Systems (SEAS '04), Edinburgh, UK, 2004
- [Con01] Conrad, M.: Beschreibung von Testszenerarien für Steuergerätesoftware - Vergleichskriterien und deren Anwendung. VDI-Berichte, Vol. 1646, VDI Verlag, 2001, pp. 381-398.
- [Con04] Conrad, M.: Auswahl und Beschreibung von Testszenerarien für den Modell-basierten Test eingebetteter Software im Automobil. PhD thesis, Technical University Berlin, 2004 (to appear)
- [CTE/ES] CTE for Embedded Systems (product information). Razorcat Development GmbH, www.razorcat.com.
- [CTE/XL] CTE with Extended Logics (product information). DaimlerChrysler AG, www.systematic-testing.com/ctm_cte.htm
- [GG93] Grochtmann, M.; Grimm, K.: Classification Trees for Partition Testing. Software Testing, Verification and Reliability, 3, 63-82, 1993.
- [Gri95] Grimm, K.: Systematisches Testen von Software - Eine neue Methode und eine effektive Teststrategie. PhD Thesis, GMD report 251, Oldenbourg Verlag, 1995
- [HSM+04] Horstmann, M., Schnieder, E., Mäder, P., Nienaber, S., Schulz H.-M.: A framework for interlacing Test and/with Design“, ICSE 2004 workshop on Software Engineering for Automotive Systems (SEAS '04), Edinburgh, UK, 2004
- [Höt97] Hötzer, D.: Schaltstrategieentwurf mit Statemate unter Einbindung kontinuierlicher Modelle zur Software-verifikation. 5. Statemate Anwenderforum, München, Germany, 1997.
- [HPP+03] Hahn, G., Philipps, J., Pretschner, A., Stauner, T.: Prototype-Based Tests for Hybrid Reactive Systems. 14. IEEE Intl. Workshop on Rapid System Prototyping, San Diego, US, 2003.
- [HV98] Hohler, B., Villinger, U.: Normen und Richtlinien zur Qualitätssicherung von Steuerungssoftware. Informatik-Spektrum, 21, 63-72 (1998)
- [KCF+04] Klein, T., Conrad, M., Fey, I., Grochtmann, M.: Modellbasierte Entwicklung eingebetteter Fahrzeugsoftware bei DaimlerChrysler. Lecture Notes in Informatics (LNI), Vol. P-45, Köllen Verlag, 2004, pp. 31-41
- [LBE+04] Lamberg, K., Beine, M., Eschmann, M., Otterbach, R., Conrad, M., Fey, I.: Model-based testing of embedded automotive software using MTest. SAE World Congress, Detroit, US, 2004.
- [Leh00] Lehmann, E.: Time Partition Testing - A Method for Testing Dynamic Functional Behaviour. TEST2000, London, UK, 2000.
- [LRR03] Lamberg, K., Richert, J.; Rasche, R.: A New Environment for Integrated Development and Management of ECU Tests. SAE World Congress, Detroit, US, 2003.
- [Lig92] Liggesmeyer, P.: Testen, Analysieren und Verifizieren von Software - eine klassifizierende Übersicht der Verfahren. In: Liggesmeyer, P. et al. (Ed.), Testen, Ana-

lysieren und Verifizieren von Software. Informatik aktuell, Springer, 1992, pp. 1-25.

[LW00] Lehmann, E., Wegener, J.: Test Case Design by Means of the CTE XL. 8. European Int. Conf. on Software Testing, Analysis and Review (EuroSTAR '00), Copenhagen, DK, 2000.

[Matlab] Matlab/Simulink/Stateflow (product information). The MathWorks Inc., www.mathworks.com/products.

[MATT] MATT (product information). The University of Montana, www.cs.umt.edu/RTSL/matt.

[MEval] MEval (product information). IT Power Consultants, www.itpower.de/meval.html

[Mis04] Mischke, J.: Classification-tree Editor for Embedded Systems CTE/ES Version 2.3 User Manual. BoD GmbH, 2004

[MTest] MTest (product information). dSPACE GmbH, www.dspace.de/ww/en/pub/products/prodover/expsoft/mtest.htm

[Ran03] Ranville, S.: MCDC Unit Test Vectors From Matlab Models – Automatically. Embedded Systems Conference, San Francisco, US, 2003.

[Rau02] Rau, A.: Model-Based Development of Embedded Automotive Control Systems”, PhD Thesis, dissertation.de, 2002.

[Reactis] Reactis Tester (product information), Reactive Systems Inc., www.reactive-systems.com.

[Sim97] Simmes, D.: Entwicklungsbegleitender Systemtest für elektronische Fahrzeugsteuergeräte. PhD Thesis, Herbert Utz Verlag Wissenschaft, 1997.

[SLPerf] Simulink Performance tools (product information). The MathWorks Inc., www.mathworks.com/products/slperftools/

[SZ03] Schäuffele, J., Zurawka, T.: Automotive Software Engineering. Vieweg Verlag, 2003.

[WSB01] 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.

CONTACT

Mirko Conrad is research scientist and project manager at the Software Technology Lab of DaimlerChrysler Research & Technology. His nine years industrial experience includes model-based development and testing of embedded automotive software. He is member of the Special Interest Group for Testing, Analysis and Verification of Software in the German Computer Society (GI) and the MathWorks Automotive Advisory Board (MAAB).

E-mail: Mirko.Conrad@DaimlerChrysler.com