Abstract – Functional testing of software dedicated for hybrid embedded systems should start at the early development phase and requires analysis of discrete and continuous signals, where timing constraints play a significant role. Test data generation is done manually nowadays, though it should be automated to the highest possible extent. Hereby, a concept for testing at the model level is introduced. Firstly, the test evaluation design based on the requirements is specified. Then, an automatic and systematic derivation of test data structures appears and finally, generation of test data variants in combinations occurs. The application of signals’ features is proposed for both – test data generation and test evaluation. Furthermore, patterns of generators for such features are built, concrete test stimuli are produced and default sequencing algorithms are created.

Keywords: automatic test data generation, functional test, test evaluation, hybrid behavior

1 Introduction

Testing costs at least 50% [3] of the effort while software development. To reduce this cost and guarantee software quality, testing should emerge earlier in the software development cycle. Numerous works have contributed to making testing more efficient and effortless (i.e., automatic). However, a systematic and automated test data generation is still a research problem as its progress depends on a lot of factors (e.g., the testing method applied, interfaces and type of the system under test (SUT) or evaluation algorithm). In this paper our goal is to support dynamic black-box testing of hybrid embedded software. We support testing on the Model-in-the-Loop Level [5], when neither code, nor hardware exists yet so as to reduce the resources and cost of the entire software development.

On the basis of our preliminary work [23] regarding the test specification and test evaluation design, this paper introduces a test data generation approach. The first concepts on the generation of test data sets and structures have been already introduced in [24]. The test data patterns can be retrieved from the test evaluation design automatically. Hence, we focus on the test data variants generation, their management and combination of the retrieved signals within a test case. We investigate different types of signals’ features, their dependencies and influences on each other in terms of the test data retrieval. Finally, synchronization of the obtained test data is considered and test control is discussed.

The paper is structured as follows. In Subsection 1.1 a short characteristic of hybrid embedded systems is given. Subsection 1.2 introduces our previous work being the basis for this research. Section 2 provides the details on the structure of the generated test data sets. Also, an overview on the transformation targeting this structure is depicted. Further on, in Section 3 the concrete variants of the test data for every single test case are given. In Section 4 a case study from the automotive domain is provided. In Section 5 related work is discussed. Section 6 evaluates the proposed test approach. Conclusions complete the paper.

1.1 Hybrid embedded systems

Embedded control systems are usually reactive with continuous-time dynamics, discrete events, and discrete mode changes. They must meet specific timing constraints and they use continuous signals for monitoring and controlling their environment via sensors and actuators and discrete signals for communication and coordination between system components [19]. There are tens of electronic control units in an embedded system (e.g., a vehicle) and software has a major influence on their performance and safety. To handle the growing complexity of this software, model-based development is nowadays applied. It allows for models specification, code generation and subsequently for their primary execution [5], [6]. Such models and their requirements serve as a basis for testing, which is a central and significant task within software production [18].

1.2 Proposed test methodology

The support of functional, dynamic testing requires an analysis of the SUT specification, test data selection, test evaluation algorithms and an execution or simulation environment. Figure 1 presents a simplified view on the test harness.
The SUT requirements specify its functionality and indicate the test objectives. Together with the interfaces of the executable SUT model they drive both – test data generation and test evaluation.

Our objective is to handle both discrete and continuous signals and the timing constraints between them. Similarly to [9] we aim to describe the test of an SUT by signal’s features. Under feature we understand an identifiable, descriptive property of a signal. Giving examples – increase, step response characteristics, step, maximum etc. are considerable signal’s features. We compare the features extracted from a signal with the expected values given in the requirements. We distinguish different feature types considering the feature availability on the one hand and the identification delay on the other hand [13]. However, their classification is out of the scope of this paper.

The test evaluation process happens by applying the so called validation functions (VFs). A VF is created out of a single requirement following a generic conditional rule:

\[
IF \ \text{preconditions set} \ THEN \ \text{assertions set} \quad (1)
\]

**Preconditions set** consists of a number of SUT signals’ feature extractors related to each other by temporal, logical or quantitative dependencies (e.g., \( \text{FeatureA AND after(time1)FeatureB OR NOT(FeatureC)} \)); a comparator for every single feature or dependency; and a unit for preconditions synchronization (PS).

**Assertions set** looks similar, however includes a unit for preconditions and assertions synchronization (PAS), respectively. The details concerning synchronization problems are described in [13].

We define patterns of VFs that are able to continuously update the verdicts for a test case already during SUT execution. They are defined to be independent of the currently applied test data and can set the verdict for all possible test data vectors and activate themselves (i.e., their assertions) only if the predefined conditions are fulfilled. Hence, a pattern\(^1\) for a validation function (shown in Figure 2) consists of a preconditions block which activates the assertions block, where the comparison of actual and expected signal values happens. The activation and by that the actual evaluation proceeds only if the preconditions are fulfilled.

\[\text{Pattern} 1\]

The easiest assertions block is built following the schema shown in Figure 3.

\[
IF \ \text{preconditions set} \ THEN \ \text{generators set} \quad (2)
\]

It includes a signal’s feature extraction part, a block comparing the actual values with the expected ones and PAS synchronizer. Optionally, some signal deviations within a permitted tolerance range are allowed. We distinguish further schemas of preconditions and assertions blocks for triggered features discussed by [13] in detail; however they are beyond the scope of this paper.

## 2 Test Data Structure Preparation

### 2.1 Systematic test data structure retrieval

In Figure 4 a pattern for the test model providing test data sets is presented. As discussed in [24] for each data set determined by a single preconditions set, a block is provided, where features generation happens. The test data can be generated following a conditional rule in the form:

\[
IF \ \text{preconditions set} \ THEN \ \text{generators set} \quad (2)
\]

Additionally, SUT output signals may be checked, if necessary. The features generation is activated by a Stateflow diagram (Std) sequencing the features in time according to the default temporal constraints (e.g., \( \text{after(time1)} \)). A switch is needed for each SUT input to handle the dependencies between generated signals. 

**Initialization & Stabilization** block enables to reset the obtained signal so that there are no influences of one test case on another.

\[\text{Pattern} 2\]

### 2.2 Transformation rules

The transformation from VFs to the generators of test signal’s feature is implemented following a set of rules. These are summarized in Table 1 for illustration purpose. The general principle is that if a given fact in the source is detected, then the action to generate the target is performed.

---

\(^1\) The patterns are designed in MATLAB/Simulink/Stateflow® [14]
Together with this analysis, equivalence partitioning [2] and signals according to the requirements or engineer’s assumption for test cases are designed to cover each partition at least once. In principle test cases are designed to execute representatives from equivalence partitions.

3.1 Test data selection

The test data are understood as the stimuli for the SUT. These are the test signals which stimulate the SUT to invoke a given behavioral scenario. In our approach a particular set of feature generators producing selected signals corresponds to the stimuli used within a particular test case.

Let us consider the generate increase in terms of a real world signal, i.e., vehicle velocity. Its value range is between <-10, 70> [7]. Additionally, {0} is taken into account as the car changes its driving direction from backwards to forwards at this point. We choose the third generation option from Table 2. Our algorithm computes 10% of the current range around all boundaries and partition points. Hence, the following increases are obtained as representatives: <-10,-9>, <-1,0>, <0,7> and <63,70>. The duration of those increases can be either derived from the VFs, or set on default values, or changed manually. The

3 Concrete test data generation

Table 1: Transformation Rules for Test Data Sets Retrieval

<table>
<thead>
<tr>
<th>Validation Functions</th>
<th>Test Data Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘Test info’ interface</td>
<td>Generate a requirements set pattern</td>
</tr>
<tr>
<td>‘Activate’ interface</td>
<td>Generate a preconditions set pattern and a preconditional state in SUT</td>
</tr>
<tr>
<td>SUT input signal in Bus Selector, ‘MATLAB Fcn’ signal comparison block</td>
<td>Generate a ‘MATLAB Fcn’ connected to an output port labeled with SUT input’s name</td>
</tr>
<tr>
<td>SUT input signal in Bus Selector, ‘Logical expression’ signal comparison block</td>
<td>Omit ‘Logical expression’ and detect other connected feature extractor</td>
</tr>
<tr>
<td>SUT input signal in Bus Selector, ‘Complete step’ signal comparison block</td>
<td>Generate ‘Complete step’ connected to an output port labeled with SUT input’s name</td>
</tr>
<tr>
<td>SUT input signal in Bus Selector, ‘Detect constant’ signal comparison block</td>
<td>Generate a subsystem labeled ‘Constant’ and an output port labeled SUT input signal</td>
</tr>
<tr>
<td>SUT input signal in Bus Selector, ‘Detect increase’ signal comparison block</td>
<td>Generate a subsystem labeled ‘Increase’ and an output port labeled SUT input signal</td>
</tr>
<tr>
<td>SUT input signal in Bus Selector, ‘Detect decrease’ signal comparison block</td>
<td>Generate a subsystem labeled ‘Decrease’ and an output port labeled SUT input signal</td>
</tr>
<tr>
<td>SUT input signal in Bus Selector, ‘Detect step’ signal comparison block</td>
<td>Generate a ‘Step’ and an output port labeled SUT input signal</td>
</tr>
<tr>
<td>SUT output signal in Bus Selector, signal comparison block, and reference block</td>
<td>Generate a Bus Selector (with SUT output selected inside), signal comparison block, reference block, Memory block, and an output port labeled “Activate”</td>
</tr>
<tr>
<td>SUT input signal in Bus Selector and signal comparison block parameterized by ‘&gt;=’</td>
<td>Generate a subsystem labeled ‘Constant’ connected to an output port labeled with SUT input’s name</td>
</tr>
<tr>
<td>SUT input signal in Bus Selector and signal comparison parameterized by ‘&lt;’ or ‘&lt;=’</td>
<td>Generate a subsystem labeled ‘Decrease’ connected to an output port labeled with SUT input’s name</td>
</tr>
</tbody>
</table>

Table 2: Options for Generation of Increase

<table>
<thead>
<tr>
<th>no.</th>
<th>criteria</th>
<th>signal (f)</th>
<th>signal (f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>time</td>
<td>&lt;10, 70</td>
<td>&lt;10, 70</td>
</tr>
<tr>
<td>2</td>
<td>signal range</td>
<td>min</td>
<td>min</td>
</tr>
<tr>
<td>3</td>
<td>signal range</td>
<td>min</td>
<td>min</td>
</tr>
<tr>
<td>4</td>
<td>tangent of angle</td>
<td>min</td>
<td>min</td>
</tr>
<tr>
<td>5</td>
<td>tangent of angle</td>
<td>min</td>
<td>min</td>
</tr>
<tr>
<td>6</td>
<td>tangent of angle</td>
<td>min</td>
<td>min</td>
</tr>
</tbody>
</table>
steps of computing the representatives on the value axis are illustrated in Figure 5.

A)    B)    C)

Figure 5: Steps of Computing the Representatives

Firstly, the boundaries of the range or the physical limits in the environment are identified (step A). Then, all risk-based partition points are included (step B). They result from the specification or the test engineer’s experience. The increase ranges are calculated and the variants are generated (step C). Finally, durations of the features are added on the time axis.

3.2 Variants sequencing

When a test involves multiple signals, the combination of different signals and their variants should be established. Several combination strategies are known to construct the test cases – minimal combination, one factor at a time, and n-wise combination [10]. Combination strategies are the selection methods where test cases are identified by combining different values of test data parameters according to some predefined criteria. We implemented the first two strategies.

If the test data variants are available and the combination strategy has been applied, the test cases can be established. Every signal’s feature generators set constitutes a test purpose for a test case. All the sets together form a test suite and they are sequenced in the test control. Before we go into the details of the test control, the sequencing of the test data within the test data sets will be explained. Hence, every single generated signal is activated for a given predefined period of time. The signal’s features within one set should be synchronized so that the timing is the same for all of them. Then, the Stateflow diagram (Std) on the test data level, called ‘Sequencing of test data in time due to Preconditions’ controls the activation of a given variants combination applying a predefined time too. This activation must be synchronized with the timing given in the test control algorithm.

Thus, in Std a time-related parameter needs to be added. It results from the test control specification. It enables to synchronize the starting point of a selected test data set forming a test case on the test data level with the starting point of a test case within the test control. The first test case in the test control starts without any delay, so the parameter should be equal to 0. The duration of this first test case specified on the test control level determines the starting point of the next test case. Hence, the starting of the following test case appears after the former finishes, which means after a specified period of time. The same applies to the activation of the test data sets on the test data level. It starts after the same period of time as the test case specified on the test control level. Further on, the next following test case starts after all the previous finish. Thus, the same applies to the activation of the further test data set. Since we do not include the duration of all the previous test cases on the test data level (i.e., within the Std), we need to calculate this parameter (called pre_time) following the formula:

\[ \text{pre_time} = \text{sum of durations of all previous test cases} \]  

Let us consider an example for this algorithm. Assume that a test control depicted on the left hand side of Figure 6 is given the parameters are calculated as follows.

Figure 6: Test Control and its Implication on the Test Data Sequencing

This algorithm applies only to the first iteration within the test control. If more iterations are needed (i.e., if more variant sets are applied for a particular test execution loop over the test control), a more complex algorithm should be used depending on the current number of the iteration.

The test cases are executed one after another according to the sequence specified in the test control. After execution of one set of variants for such a sequence the next set of variants is chosen and the sequence of new test stimuli repeats.

4 Case study

This section demonstrates the application of the presented concepts for a selected case study. A simplified component – Pedal Interpretation of an Adaptive Cruise Controller developed by Daimler AG is used. This subsystem can be employed as pre-processing component for various vehicle control systems. It interprets the current, normalized positions of acceleration and brake pedal (phi_Acc, phi_Brake) by using the actual vehicle speed (v_act) as desired torques for driving and brake (T_des_Drive, T_des_Brake) [5]. Furthermore, two flags (AccPedal, BrakePedal) are calculated, which indicate whether the pedals are pressed or not. Some excerpts of these functional requirements are given in Table 3, whereas the SUT interfaces are presented in Table 4.

We selected the Matlab/Simulink/Stateflow [14] environment to show the feasibility of our solution. It provides a simulation engine, which allows for the execution of tests, thus their dynamic analysis. It supports
hybrid systems development and allows for using the same development language for both – system design and test design – so as to integrate testing and system modeling [16].

Table 3: Requirements for Pedal Interpretation [5] (excerpt)

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Interpretation of pedal positions</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>1.1</td>
<td>Interpretation of brake pedal position</td>
</tr>
<tr>
<td></td>
<td>Normalized brake pedal position should be interpreted as desired brake torque $T_{des_Brake}$ [Nm]. The desired brake torque is determined when actual pedal position is set to maximal brake torque $T_{max_Brake}$.</td>
</tr>
<tr>
<td>1.2</td>
<td>Interpretation of accelerator pedal position</td>
</tr>
<tr>
<td></td>
<td>Normalized accelerator pedal position should be interpreted as desired driving torque $T_{des_Drive}$ [Nm]. The desired driving torque is scaled in the non-negative range in such a way that the higher the velocity is given, the lower driving torque is obtained.</td>
</tr>
</tbody>
</table>

Table 4: SUT Inputs of Pedal Interpretation Component

<table>
<thead>
<tr>
<th>Velocity (v)</th>
<th>Acceleration pedal (phi_Acc)</th>
<th>Brake pedal (phi_Brake)</th>
<th>SUT Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;-10, 70&gt;</td>
<td>&lt;0, 100&gt;</td>
<td>&lt;0, 100&gt;</td>
<td>Permitted values range</td>
</tr>
<tr>
<td>m/s</td>
<td>%</td>
<td>%</td>
<td>Unit</td>
</tr>
</tbody>
</table>

Let us analyze requirement 1.2 for illustration purpose. It is interpreted as the following conditional rules:

- IF $v$ is constant AND phi_Acc increases AND $T_{des\_Drive}$ is non-negative THEN $T_{des\_Drive}$ increases.
- IF $v$ increases AND phi_Acc is constant AND $T_{des\_Drive}$ is non-negative THEN $T_{des\_Drive}$ does not increase.
- IF $v$ is constant AND phi_Acc decreases AND $T_{des\_Drive}$ is non-negative THEN $T_{des\_Drive}$ decreases.
- IF $v$ is constant AND phi_Acc decreases AND $T_{des\_Drive}$ is negative THEN $T_{des\_Drive}$ decreases.
- IF $v$ is constant AND phi_Acc increases AND $T_{des\_Drive}$ is negative THEN $T_{des\_Drive}$ increases.
- IF $v$ is constant AND phi_Acc is constant THEN $T_{des\_Drive}$ is constant.

Let us analyze requirement 1.2 for illustration purpose. It is interpreted as the following conditional rules:

The number of preconditions blocks in Figure 8 suits the number of VFs appearing in Figure 7. Sequencing of the features generation is performed in the Stateflow diagram. Signal switches are used for connecting different features with each other according to their dependencies as well as for completing the rest of unconstrained SUT inputs with user-defined, deterministic data, when necessary (e.g., phi_Brake). Considering the first VF and the first test data set from Figures 7 and 8 respectively, the following applies. If the velocity is constant and an increase in the acceleration pedal position is detected then the assertion is activated. Thus, as shown in Figure 9 (middle part) – a constant signal for velocity is generated; its value is constrained by the velocity limits <-10, 70>-. The partition point is 0 as discussed.
in Subsection 3.1. The TDG produces five variants out of this specification. These belong to the set: {-10, 5, 0, 35, 70}.

5 Related work

In the following, some of the available methodologies for testing the dynamic functional behavior of embedded systems are reviewed. The evaluation is based on the test design availability, test data selection algorithms and the automation grade of the method.

Model-checking approaches used for checking temporal safety critical requirements are available in Safety Checker Blockset [17] and EmbeddedValidator [1]. Together with the validation support given in [15] and watchdogs in [6] they contribute to the concept of signals’ features in our technique. The fact is especially valuable in the context of the test oracle design and test evaluation. We aim to cover complex properties and dependencies between continuous signals, although we aim to structure and systematize our solution. Reactis Validator [4] provides a test framework to graphically express assertions that check an SUT for potential errors, and user-defined targets that monitor system behavior in order to detect the presence of certain desirable test cases. If a failure occurs, a test execution sequence is delivered and it leads to the place where it happens. However, no predefined patterns enabling a scalable and systematic test design are available.

With the help of Classification Tree Editor for Embedded Systems (CTE/ES) [5], [7], it is possible to set up value ranges for the individual ports and to combine these ranges in sequences. To represent test scenarios in an abstract way, they are decomposed into individual test steps [7]. Each test step defines the input situation at a certain time. A sequence of such test steps is called test sequence.

The Time Partition Testing (TPT) [11], [12] supports the manual selection of test data on the semantic basis of so-called testlets and the syntactic techniques Direct Definition, Time Partitioning and Data Partitioning which are used to build testlets. Testlets facilitate an exact description of test data. TPT is platform-independent and can be used at several embedded software development stages, which we do not directly support with our solution. The verdict and arbitration concepts known from such standards like UML 2.0 Testing Profile (U2TP) [16] or Testing and Test Control Notation – version 3 (TTCN-3) [8] are used in our method too.

Our approach differs from the related work as we start with designing the test evaluation model. Further on, we retrieve the test data sets following the structure of the test evaluation. The TDG concretizes the data. Its functionality has some similarities to the method applied in CTE/ES [5]. We use the SUT input partitions and boundaries to find the meaningful representatives. Additionally, we consider the SUT outputs too. The related work on model-based testing is provided in the surveys by [2], [21].

6 Evaluation

Evaluating the discussed test approach, we propose to use some black-box test criteria related to our method. With this practice, different types of test coverage based on the
functional relevance are supported. Hence, we define and apply such test quality metrics like e.g.: $SUT input/output range coverage$, test data variants coverage, input range consistency coverage, variants related-precondition activation coverage, variants related-assertion activation coverage. Due to the lack of space we consider them to be beyond the scope of this paper and we show only a few as a matter of examples.

The assumption is that the generated test data variants should provide representatives to cover all the input/output partitions and boundaries. These should be selected on the base of the predefined criteria for every single type of signal’s feature. The simplest metric – $SUT input/output coverage$ achieves 100% if all possible variants of a given signal, based on the values analysis, are generated and applied during the test execution.

The range of a given signal must be consistent to the constraints given in the VFs’ preconditions (i.e., such that the constrained values do not cross beyond the allowed range). $Input range consistency coverage$ checks this consistency in our approach.

Evaluating our test method further on, it provides a set of patterns which support the systematic test design and enable to reuse a number of structures for modeling the test. Additionally, some of the steps are automatic, which reduces the time of the test development.

7 Summary

In this paper a method for a systematic test data variants generation based on our preliminary work has been introduced. A special attention was drawn to: the automatic generation of the test data sets using transformation functions, production of the test data variants and their sequencing. The test control in respect to the obtained test data has been also mentioned. Finally, a case study was provided. Since, its test system was rather easy to implement we achieved 100% of $input and output coverage$. However, we expect that additional manual refinements are needed, especially when integration level test is considered. This part of our research is still under investigation with some results described in [22].

The scalability of the proposed test design is supported by a library/patterns extension mechanism. Systems may be analyzed either using the predefined elements or by creating new ones using the given patterns.

We support only positive tests by now, thus the reciprocals of the generated data would be suitable candidates for negative testing. However, the VFs would have to be specified for them too.

The transformation of requirements into conditional rules may be complex for some cases, especially when timing constraints are included. Thus, we still want to explore the specification driven tests and their relation to VFs modeling and test data generation deeper, based on more case studies.

8 References


