Specification of Model-based Test Case Generation & Execution Methods and Tools

D_WP2.3_2

<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Specification of Model-based Test Case Generation &amp; Execution Methods and Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality</td>
<td>Public</td>
</tr>
<tr>
<td>Project</td>
<td>MBAT</td>
</tr>
<tr>
<td>Deliverable File Name</td>
<td>D_WP2.3_2</td>
</tr>
<tr>
<td>Status</td>
<td>Final, submitted to ARTEMIS-JU</td>
</tr>
<tr>
<td>Contact Person</td>
<td>Sven Sieverding</td>
</tr>
<tr>
<td>Phone</td>
<td>+49 441 9722-452</td>
</tr>
</tbody>
</table>
### Authors

<table>
<thead>
<tr>
<th>Name</th>
<th>Company</th>
<th>E-Mail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sven Sieverding</td>
<td>OFFIS</td>
<td><a href="mailto:sven.sieverding@offis.de">sven.sieverding@offis.de</a></td>
</tr>
<tr>
<td>Christian Ellen</td>
<td>OFFIS</td>
<td><a href="mailto:christian.ellen@offis.de">christian.ellen@offis.de</a></td>
</tr>
<tr>
<td>Stefan Häusler</td>
<td>BTC Embedded Systems</td>
<td><a href="mailto:Stefan.Haeusler@btc-es.de">Stefan.Haeusler@btc-es.de</a></td>
</tr>
<tr>
<td>Stefanie Götzfried</td>
<td>MBtech Group</td>
<td><a href="mailto:Stefanie.Goetzfried@mbtech-group.com">Stefanie.Goetzfried@mbtech-group.com</a></td>
</tr>
<tr>
<td>Nicky Williams</td>
<td>CEA LSL</td>
<td><a href="mailto:nicky.williams@cea.fr">nicky.williams@cea.fr</a></td>
</tr>
<tr>
<td>Frank Elberzhager</td>
<td>Iese</td>
<td><a href="mailto:Frank.Elberzhager@iese.fraunhofer.de">Frank.Elberzhager@iese.fraunhofer.de</a></td>
</tr>
<tr>
<td>Andres Kull</td>
<td>Elvior</td>
<td><a href="mailto:andres.kull@elvior.com">andres.kull@elvior.com</a></td>
</tr>
<tr>
<td>Florian Lorber</td>
<td>TUG</td>
<td><a href="mailto:Florian.Lorber@student.tugraz.at">Florian.Lorber@student.tugraz.at</a></td>
</tr>
<tr>
<td>Cristina Seceleanu</td>
<td>MDU</td>
<td><a href="mailto:cristina.seceleanu@mdh.se">cristina.seceleanu@mdh.se</a></td>
</tr>
</tbody>
</table>

### Reviewers

<table>
<thead>
<tr>
<th>Name</th>
<th>Company</th>
<th>E-Mail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mehrdad Saadatmand</td>
<td>Enea</td>
<td><a href="mailto:Mehrdad.saadatmand@xdin.com">Mehrdad.saadatmand@xdin.com</a></td>
</tr>
<tr>
<td>Darren Sexton</td>
<td>Ric</td>
<td><a href="mailto:Darren.Sexton@ricardo.com">Darren.Sexton@ricardo.com</a></td>
</tr>
<tr>
<td>Michael Dierkes</td>
<td>RCF</td>
<td><a href="mailto:mdierkes@rockwellcollins.com">mdierkes@rockwellcollins.com</a></td>
</tr>
<tr>
<td>Andreas Dereani</td>
<td>DAI</td>
<td><a href="mailto:andreas.dereani@daimler.com">andreas.dereani@daimler.com</a></td>
</tr>
</tbody>
</table>

### Distribution

<table>
<thead>
<tr>
<th>Name / Role</th>
<th>Company</th>
<th>Level of confidentiality</th>
<th>Type of deliverable</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBAT Consortium</td>
<td>RE</td>
<td>RE</td>
<td>R</td>
</tr>
<tr>
<td>Antonio Vecchio, Project Officer</td>
<td>ARTEMIS-JU</td>
<td>RE</td>
<td>R</td>
</tr>
</tbody>
</table>
## CHANGE HISTORY

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason for Change</th>
<th>Pages Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>19.04.2012</td>
<td>Initial version of the document</td>
<td>All</td>
</tr>
<tr>
<td>0.02</td>
<td>09.10.2012</td>
<td>Added partner contributions</td>
<td>All</td>
</tr>
<tr>
<td>0.03</td>
<td>19.10.2012</td>
<td>Include first level review</td>
<td>All</td>
</tr>
<tr>
<td>1.00</td>
<td>29.10.2012</td>
<td>Include second level review</td>
<td>All</td>
</tr>
</tbody>
</table>
12  TOOLS/METHODS PROVIDED BY ALES  ................................................................. 33
  12.1  FORMAL SPECS VERIFIER ................................................................. 33
  12.1.1  General Tool description ............................................................ 33
  12.1.2  Development within MBAT ............................................................. 34
Content of Figures

Figure 1-1: SP2 work package timeline .............................................................................................................. 7
Figure 3-1: Tools & Equipment is divided into three major sections: products, solutions, services & support. 10
Figure 3-2: Example for visualization in PROVEtech:TA .............................................................................................................. 11
Figure 3-3: Scripts are written in WinWrap Basic with a specific test language extension 12
Figure 3-4: MBtech’s vision – OTX as standardized sequence format for V&V activities 13
Figure 3-5: Prototype of TCC+: PROVEtech:TA plug-in for graphical test case composing 14
Figure 11-1: Interface of an example SUT (Simulink model) 31
Figure 11-2: The simplest test model in TPT .............................................................................................................. 31
Figure 11-3: Two parallel state machines .............................................................................................................. 31
Figure 12-1: FSV-ATG flow .............................................................................................................................. 33
Figure 12-2: Example of BCL contract specification in Simulink ........................................................................ 34

Content of Tables

Table 1 ViTAL TestGen functionalities ............................................................................................................... 8
Table 2 PROVEtech:TA functionalities ............................................................................................................. 10
Table 3 PathCrawler functionalities .................................................................................................................. 15
Table 4 BTC EmbeddedTester functionalities .................................................................................................. 18
Table 5 MaTeLo functionalities .......................................................................................................................... 20
Table 6 TestCast MBT functionalities ............................................................................................................... 22
Table 7 TUG Tool functionalities ...................................................................................................................... 23
Table 8 AAU Test Tool functionalities .............................................................................................................. 25
Table 9 LBTest functionalities ........................................................................................................................... 27
Table 10 TPT functionalities ............................................................................................................................. 30
Table 11 FormalSpecs Verifier functionalities .................................................................................................. 33
1 Introduction

This work package deals with the problem how to efficiently generate appropriate test patterns covering specified test goals and test coverage criteria as well as test execution procedures, based-on the T&A models defined and generated in the work packages WP2.1 and WP2.2. The generation of test cases depends on the one hand on the sources (requirements, system models, design (control) models, implementation models ...) and on the other hand on the test objectives (requirements coverage, structural coverage ...) as well as its usage for simulation (MIL, SIL, HIL).

This deliverable provides specifications of methods and tools for model-based test case generation and execution to be implemented within WP2.3. Furthermore, it will specify the functionalities of the tool versions available at major project milestones M2, M3, resp. M4. Since the project has started in November 2012, the milestones are in May 2013 (M2), May 2014 (M3), and at the end of the project in November 2014 (M4), see Figure 1-1.

The document is structured as followed: There is one chapter for each partner of WP2.3. The chapters are structured such that first there is an overview about the availability of tool functionalities (mapped to major project milestones). Afterwards the tools and the development of the tools within the MBAT project are described.
2 Tools/Methods Provided by MDU

ViTAL (A Verification Tool for EAST-ADL Models using UPPAAL PORT) integrates the architectural description language EAST-ADL with verification techniques in order to provide model checking of EAST-ADL models with respect to timing and functional behavioral requirements. ViTAL provides an automatic model transformation to UPPAAL PORT model checker, which enables UPPAAL PORT to handle EAST-ADL models as inputs and model-check functional and timing behavior of functional blocks enriched with timed automata behavior.

The ViTAL TestGen will extend ViTAL with test suite generation capabilities. The test case generation will exploit trace information resulted from the formal analysis of EAST-ADL enriched with Timed Automata behavior models, by prioritizing the test suite generation based on the counter-example produced by UPPAAL PORT. The test suites will be generated from UPPAAL models for different coverage criteria.

<table>
<thead>
<tr>
<th></th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ViTAL TestGen</td>
<td>Test-suite generation for EAST-ADL models from UPPAAL PORT models for a given coverage criterion.</td>
<td>Test-suite generation for EAST-ADL models from UPPAAL PORT models for various coverage criteria.</td>
<td>Test suite generation + test suite optimization.</td>
</tr>
</tbody>
</table>

Table 1 ViTAL TestGen functionalities

2.1 ViTAL TestGen

2.1.1 General Tool description

We propose a technique for generating test cases and test suites for embedded real-time systems. We will use model-based testing, a black-box testing technique, to derive test cases from a formal model that specifies the expected behavior of the system. ViTAL TestGen will exploit the precise semantic of EAST-ADL models, which have been enriched with functional and timing behavior of timed automata semantics. In the beginning, we will manually derive test cases. However, for later versions of the tool, we intend to develop algorithms for the automatic test case generation from our formal methods.

Our approach benefits from the effectiveness of testing based on test generation from an abstract formal model of the system under test (SUT). Besides the SUT, we will also model the environment of the system (Env), and the communication between the SUT and Env will be done via a well-defined set of observable actions. The controlled environment provides the possibility to measure different characteristics of the system, in particular extra-functional behaviors, like e.g. performance or resource usage.

The UPPAAL model checker has a state-space exploration engines used in model-based testing. The model checker can formally verify temporal properties of the system, e.g. reachability properties. UPPAAL PORT, integrated in ViTAL TestGen, will help us to transform the test case generation problem in a reachability problem, using a reachability property to determine if a test purpose can be fulfilled.

The diagnostic trace facility of the UPPAAL PORT model checking tool is used to generate test sequences. The test cases can either be generated using manually formulated test purposes or automatically, from several kinds of coverage criteria of the timed automata model.

ViTAL TestGen tool work flow:

- **Pre-Condition: Requirements on input models**
  The input model is a simple timed automata model file to show definition-use coverage, in XML format (“Model.xml”). The property (file “property.q”) is specified with coverage in the observer automata language (file “Observer.obs”).

- **Execution: TCG work flow. configuration of the tool, selection of test objective**
  Once the models above are specified, one can create the trace by running the tool.

- **Post-Condition: Format of generated test cases**
ViTAL TestGen extension with regard to MBAT goals:
  o Interoperability
    Our plans include developing ViTAL TestGen starting from our formal analysis tool, ViTAL, such that IOS compliant interfaces will be incorporated. There is ongoing work in making ViTAL OSCL compliant, so the ViTAL TestGen will follow.
  o Combination of test and analysis
    The test case generation will exploit trace information resulted from the formal analysis of EAST-ADL enriched with Timed Automata behavior models, by prioritizing the test suite generation based on the counter-example produced by UPPAAL PORT

Currently, ViTAL TestGen is under development at Mälardalen University.

2.1.2 Development within MBAT

M2: We will develop algorithms that use traces generated by UPPAAL PORT to derive test-suites, rather than simple test-cases, for UPPAAL models of EAST-ADL functions. In this version, ViTAL TestGen will generate test-suites for a fixed coverage criterion, which will be decided later.
M3: The result from M2 will be extended towards adding several coverage criteria.
M4: Since the test-suites could be redundant with respect to different test-goals, we will incorporate optimization techniques to reduce redundancy.
3 Tools/Methods Provided by MBtech Group

MBtech group’s department Tools & Equipment provides various off-the-shelf products and solutions for validation and verification along the E/E development chain.

Within MBAT we will provide mainly PROVEtech:TA as our test case execution solution. We will work on a new feature, called TCC+ (Test Case Composer) to offer a standardized exchange format (OTX) for test cases, to support a user-centred way to compose test cases graphically and to choose and visualize automatically generated test cases.

<table>
<thead>
<tr>
<th></th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROVEtech:TA – TCC+</td>
<td>PROVEtech:TA – Esterel Scade Connection</td>
<td>Version 1.0 available</td>
<td>Version 2.0 available</td>
</tr>
<tr>
<td>PROVEtech:TA</td>
<td></td>
<td></td>
<td>--</td>
</tr>
</tbody>
</table>

Table 2 PROVEtech:TA functionalities

3.1 PROVEtech:TA (MBtech Group)

3.1.1 General Tool description

PROVEtech:TA is a COTS tool for test automation across the domains (chassis, body, power-train) and across the development stages (SiL, component HiL, Integration HiL for several networked ECUs, Vehicle testing) and supports standard HiL (Hardware in the Loop) platforms (dSPACE, PROVEtech:RE, Opal-RT, ETAS, etc.). Some of the key features are listed below:

- **Visualization**: PROVEtech:TA enables control and visualization of all test system signals and messages for interactive testing. A wide range of display options is available for the user to tailor the work page accurately to his/her needs. Further, comfortable measurement features with intuitive filter options and a measurement analysis synchronized with automated test execution, offer the tester several possibilities for accurate test & analysis.
Test Management & Automation: An integrated IDE for test scripting, management & automation with extensive user/project management & versioning possibilities is a feature that results in several advantages for the customer. Creation & execution of test suites, with generation of test protocols and saving all associated results, measurements and data together with the test scripts in a database provide for comfortable test automation. Further, the test language extension of PROVEtech:TA allows the tester to automate almost everything that he would do manually: for example set a signal, create an electrical fault, read the reaction via diagnostics, all automated via scripting.
Figure 3-3: Scripts are written in WrinWrap Basic with a specific test language extension

- **Integrated Diagnostics & Fault Simulation**: An interactive tab to execute diagnostic services, access to ECU error storage, and support of flashing and coding of ECUs via dedicated diagnostics HW makes diagnostics comfortable. To test the ECU robustness and reliability, one needs to inject electrical faults to the ECU. This is also made comfortable with the integrated fault simulation unit of PROVEtech:TA.

- **The real time automation engine (RTAE)** allows for real time testing of the ECU. This means actions (stimulus) could be performed and reactions can be measured in real time. This feature is available on all supported HIL platforms and the real time granularity is fixed by the model time step defined on the platform. A typical example would be: perform parallel actions, e.g., press the brake for 0.4 sec and simultaneously press the Set+ button of the cruise control. Further exchange of data between real-and non-real-time parts of the code and built-in verification functions enable analysis and generation of protocols to happen as part of the script execution.

- Accessing the ECU internal variables via XCP or CCP (CAN Calibration Protocol) is possible with PROVEtech:TA: this means, that test scripts can use XCP/CCP variables like any other variable be it in test scripts, or work page or RTAE or for measurements.

- Standard Automotive bus systems such as CAN, LIN, MOST, FlexRay are supported both at messaging & signal levels.

- **PROVEtech:TA management features**: Generation of data for the management to control project status or other performance measures, is available as a module in PROVEtech:TA.

For over 10 years, PROVEtech:TA has been established as a standard tool for test automation in several OEMs and Tier-1s worldwide. With a fixed release cycle twice a year, a feature planning based on user inputs and industry trends (ASAM, AUTOSAR etc.), the user is assured of a high quality product.

### 3.1.2 Development within MBAT: TCC+ (PROVEtech:TA Plug-In)

Within MBAT MBtech develops a Plug-In for the test automation tool PROVEtech:TA supporting the Open Test sequence eXchange format (OTX, ISO 13209). This plug-in is called TCC+ (Test Case Composer) and will provide an open interface to model-based-testing-technologies within the environment of its widely-used and industry proven host application by supporting a standardized exchange format (OTX). Furthermore TCC+ offers graphical composing and viewing of test sequences and their execution and therefore helps to overcome existing problems with calibration and selection of test cases from model-based test case generation approaches.

MBtech does not offer its own model-based-testing technology but interact with several model-based-testing-tools and techniques (one outcome of MBAT should be to standardize those connections). We try to stay flexible on this side as our core competence is test execution.

Our vision is the use of OTX (open test sequence exchange/ISO 13209) as the “Model of Test Case” for storage and exchange of test cases (coming from model-based-testing technology/automatic test case generation tools).

We think that transforming result-vectors (coming from static analysis) into OTX could be one efficient solution for the combination of Analysis and Test. Results from Analysis and Test could be stored in one database and an overall V&V evaluation report could be created. Furthermore result-vectors from static analysis could become new test cases (see figure Figure 2-3).
Furthermore the tester of the future will face new tasks as following:

- **Away from hard coding towards test model development**: Tomorrow’s testers must be capable of creating test models for automatic test case generation. Over the short or long-term, model-based test approaches will replace script-based procedures.
- **Away from the direct translation of the test specifications into test code and towards the intelligent selection of the test cases to be generated**: The challenge no longer consists of transferring natural text language into test code, but rather of logically selecting the test cases utilized in the testing process. This approach focuses on quality rather than quantity!

The testers of the future will value the ability to rapidly gain an overview of the automatically generated test cases. They will no longer ask: „How do I program this?“, but rather: „What is being tested?“ Their tasks are more of a selective and supervisory nature and less of an operative nature. The graphical representation also simplifies the communication such as the day-to-day interfaces with the function developers. The graphical procedural representation also provides a more efficient answer to the question „What was tested?“ than source code. That is why we believe in our graphical approach to test case creation.

**More Advantages at a glance:**
- Rapid overview of a test procedure
- Support visual information input by color and shape codes
- Drag & drop enables a test procedure to be rapidly assembled
- Switching between the code and the graphical interface accelerates the learning process.
- Simplified information transfer
- Designed for visually-oriented human perception
Figure 3-5: Prototype of TCC+: PROVEtech:TA plug-in for graphical test case composing
4 Tools/Methods Provided by CEA LSL

<table>
<thead>
<tr>
<th>PathCrawler</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Generation of complementary tests to increase the coverage of the implementation</td>
<td>Generation of an oracle for the complementary tests</td>
<td>--</td>
</tr>
</tbody>
</table>

Table 3 PathCrawler functionalities

4.1 PathCrawler

4.1.1 General Tool description

White-box test generation
The PathCrawler test-generation tool generates tests to explore all behaviors of a C function. Let us suppose that the behavior of component M of a UML statechart model S of a system is implemented by the C function F. PathCrawler will explore all actions of F, i.e. all state variable assignments and all ways in which events are dispatched and event parameters calculated and/or other input/output actions are performed. Because PathCrawler explores the implementation and not the model, PathCrawler may generate tests to distinguish different behaviors in the implementation which do not appear in the test model, because the model is more abstract than the code. This may be because certain details are deliberately not included in the model, in order to keep it simple, or because the modelled component calls external functions or standard blocks whose detailed specification is not known.

White-box testing as a complement to model-based testing
An example of a gap between the model and the implementation is the model of the software in a drinks vending machine. Let us suppose that the model contains one event which corresponds to the user entering a sum of money, another event which corresponds to the user selecting a drink, and a transition corresponding to the delivery of the drink which is guarded by a comparison between the sum entered and the price of the selected drink. However, the coins making up the sum of money and the order in which they are entered are not modelled. The implementation, however, contains code to decide the worth of each coin entered and add it to the sum of money entered so far. Suppose that there is an error in this code in the case of a 20 cent piece.

Using the same example, suppose that the requirements for this software include the following: “if the user enters the exact price of the drink and the drink is in stock then the drink is delivered”. If tests are selected so as to ensure one test per requirement then the test which covers this requirement may not use a 20 cent piece and so it may not reveal the bug. This is why tests which cover the source code of the implementation are more likely to reveal certain bugs than tests which cover a model which is not as detailed as the implementation.

The importance of the reachable extended states
PathCrawler constructs test cases by selecting values for all effective inputs of F. These represent the event received by the component M and its parameters but also the extended state (i.e. the state in the UML state machine, also called the structural state, and values of all state variables) of M at this point. The extended state includes the state of M in the UML state machine, also called the structural state, and the values of all state variables of M. The extended state is a result of the successive state transitions and assignments of state variables performed in previous calls to F. Note that the extended state, as defined above, is a concrete state, i.e. the state variables have constant values. By default, PathCrawler will construct a test case to cover a particular behavior by selecting any combination of inputs which activate the behavior. However, the combination of state variable values selected by PathCrawler may not correspond to an extended state of M which is effectively reachable from the initial state of the system model S.

As an example, suppose that M performs a certain action when it is in a certain state, U, a certain event is received and the sum of state variables x, y, and z is equal to 2 and x, y, and z each have values 0 or 1. PathCrawler may construct a test in which x=1, y=1, z=0. However, if x is different to y in the initial state of M and no event received by M causes x and y to be set to the same values while in state U, then this test does
not represent an extended state which can really exist. If the test created by PathCrawler provokes some unexpected behavior then the user will conclude that there is a bug in the implementation but if this behavior is really due to the fact that x and y are equal then this bug does not really exist. This is why PathCrawler must only construct tests - cases by selecting values which correspond to a reachable extended state. However, the user often does not know which states are reachable.

4.1.2 Development within MBAT
4.1.2.1 Generation of complementary tests to increase the coverage of the implementation
The Diversity model-based test-generation tool developed by the CEA LISE laboratory can construct tests to satisfy a certain criterion. This may be coverage of all fireable transitions in S, coverage of all structural states, or, when it is not possible to achieve complete coverage with a reasonable number of tests, coverage up to a certain test case length and total number of test cases. These tests are concrete instances of the abstract tests generated by Diversity. The set of abstract tests generated by Diversity can be used to define a set of reachable extended states of the component M. Indeed, each abstract test generated by Diversity is composed of:

1) a sequence, E, of
- external events received by S;
- assignments of state variables of the different components of S;
- state transitions of the different components of S;
- and internal and external events dispatched,

2) a and consistent set of constraints, C, on the parameters and arguments of E.

Some abstract test sequences have one or more prefixes, E', which end in the reception by M of an event and the resulting assignment of one or more state variables of M and possible dispatching of one or more new events. Indeed, each such prefix E' of an abstract test generated by Diversity defines a symbolic state, Pre, of M, i.e. a structural state of M and a set of constraints on the concrete values of the state variables of M. Pre is the symbolic state before the final action, A, in E'. The abstract test also defines the symbolic state, Post, after A is executed. Pre is the same as Post if A just consists in dispatching events. Pre is different to Post if A includes a state transition or assignments of new values to the state variables of M. Note that Pre is a symbolic state in which M can react to an event whereas Post may be a final symbolic state of M. Each symbolic state defines a set of extended concrete states. This means that each abstract test prefix, E', along with the corresponding constraints, C', defines the sub-sets Pre and Post of reachable extended states of M.

If M has a finite set of reachable extended states and if Diversity can achieve coverage of all symbolic states of M then the set of reachable extended states of M is defined by the union of the initial extended state and all sub-sets Pre and Post defined by abstract test prefixes E'. This explains the potential synergy between Diversity and PathCrawler. Diversity can be used for model-based test generation and PathCrawler can be used to generate complementary tests to improve the coverage of the source code and increase the chances of detecting bugs. In order to ensure that the tests generated by PathCrawler represent reachable extended system states, PathCrawler can select the test case values from the set of reachable extended states defined by Diversity.

4.1.2.2 Generation of an oracle for the complementary tests
An oracle is needed for the tests generated by PathCrawler and one possibility is to use information extracted from the model S for this purpose. Indeed, an oracle for all tests generated by PathCrawler from extended states in an abstract test prefix E';C' can be obtained by extracting from E';C' the state variable assignments and events dispatched by M as a consequence of the final action, A, performed by M in E'; and the constraints on their parameters and arguments.
These events, assignments and constraints must just be mapped to the inputs and outputs of $F$ and then automatically translated into a C function with the appropriate signature so that PathCrawler recognizes it as the oracle.
5 Tools/Methods Provided by BTC Embedded Systems AG

The main contribution from BTC Embedded Systems within this workpackage is the enhancement of BTC EmbeddedTester regarding model level terminology and regarding floating point models. The two main functionalities are listed within the table below.

A third enhancement around “Simulation based formal Verification” is strongly coupled with contributions within BTC EmbeddedSpecifier. For more details refer to WP2.4.

<table>
<thead>
<tr>
<th>BTC Embedded Tester</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTC Embedded Tester</td>
<td>--</td>
<td>Model based Test Case Generation</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Test Case Generation for Floating Point Models</td>
<td></td>
</tr>
</tbody>
</table>

Table 4 BTC EmbeddedTester functionalities

5.1 BTC EmbeddedTester

5.1.1 General Tool description

BTC EmbeddedTester from BTC Embedded Systems is a test generation, execution and test management environment especially made for TargetLink® and Simulink®. For the application in the automotive domain, the tool has been pre-qualified with respect to ISO-26262-8. It supports unit and module testing for Simulink®-Models, TargetLink®-Models and Target Code as part of a model based development process. Automatic test generation and code verification capabilities are supported additionally.

A test vector manager is an integral part of EmbeddedTester allowing to organize, visualize, export and debug test vectors. Groups of either stimuli vectors or/and test vectors, including reference values, can be managed. Complete test sequences can be defined to allow batch test execution.

BTC EmbeddedTester offers execution of test vectors on different levels, such as Model-in-the-loop, Software-in-the-loop or Processor-in-the-loop in a back-to-back-testing fashion. Furthermore, vectors can be exported for further reuse, for instance Hardware-in-the-loop Testing. It supports easy reuse of new and existing test sets from various sources. After importing existing test cases BTC EmbeddedTester shows achieved code coverage.

Test models for automatic test generation can be constructed using existing modeling tools such as MATLAB Simulink/Stateflow and dSPACE TargetLink. Please refer to these tools for more information on how to model with them. BTC EmbeddedTester can also be applied directly for test vector generation on code.

5.1.2 Development within MBAT

Within this MBAT workpackage BTC EmbeddedTester will be enhanced in two different directions.

Model level view:

The tool will be enhanced such that it understands both model level and code level view. This will enable then both model-based test generation for implementation level and also code level generation of test cases. Today, only code-based test vector generation is available. The following steps will be possible:

1. Creation of model based testing profiles based on Simulink and or TargetLink models. In advance to our existing technologies, we will also be able to import TargetLink models that are not fully implemented for final code generation.

2. Automatic test vector execution. It will be possible to execute any test vector talking about the model interface within Embedded Tester. Test vectors can come from automatic test vector generation or from an external test vector specification format.

3. Test vector generation for Simulink/TargetLink models. For this purpose, we will implement internally the following functionalities:
a. The tool will internally create an annotated TargetLink still holding the semantics of the SUT but can also be used for code generation using TargetLink Auto Coder.
b. The C-Code representation is then applicable to our existing test vector generation engines.
c. The resulting test vectors will be transferred back to the model level and will be appropriately reported.

ATG for Floating point models:

The built-in techniques to generate test cases for fixed-point resp. integer models and code work well. With the previous mentioned functionality, it will also work for Simulink and TargetLink models. But for floating-point models (what is very often the case when talking about Simulink/TargetLink models only) and code some severe limitations exist. The tool will be enhanced such that it can generate test cases both for model level and code level even if floating point signals and variables are involved.

Within MBAT, it will be possible to apply also the formal analysis engine called “CV Engine” on models containing floating point signals and variables. This functionality will be integrated within BTC Embedded Tester. From a user point of view the only effect here is that the formal test vector generation engine CV will be available on a broader class of models. Note that by using the formal analysis technique from the CV Engine to generate test cases a combination of analysis and test is obtained, which is one major goal of MBAT.

For this purpose, we will enhance BTC EmbeddedTester with a fully automated refinement mechanism to derive an integer approximation for a floating point implementation with the aim to achieve high structural coverage rates on the original floating point model. The algorithm will start with rough approximations to reduce model checking complexity. Refinement steps will increase the computation precision to cover previously uncovered test goals.
6 Tools/Methods Provided by All4tec

All4tec contributes to this WP with MaTeLo. MaTeLo is a test case generator. The test cases in MaTeLo can be generated using different generation algorithms. The existing generation algorithms can be improved in the MBAT context, or new algorithms can be added. The main functionalities planned to be done in the major MBAT milestones are summarized in the Table 5.

<table>
<thead>
<tr>
<th></th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaTeLo</td>
<td>&quot;TG based on analysis results&quot; (V1)</td>
<td>&quot;TG based on test results&quot;</td>
<td>&quot;TG based on analysis results&quot; (V3)</td>
</tr>
<tr>
<td></td>
<td>&quot;TG based on equivalent test cases identification&quot;</td>
<td>&quot;TG based on analysis results&quot; (V2)</td>
<td>&quot;TG based on test cases coverage&quot; (V3)</td>
</tr>
<tr>
<td></td>
<td>&quot;TG based on coverage criteria&quot; (V1)</td>
<td>&quot;TG based on test cases coverage&quot; (V2)</td>
<td></td>
</tr>
</tbody>
</table>

Table 5 MaTeLo functionalities

TG: Test Generation

6.1 MaTeLo

6.1.1 General Tool description

The **MaTeLo Usage Model Editor** allows test engineers to design a functional testing model of the SUT. The model is based on a finite state machine. A transition corresponds to a test step. A test step consists of a stimulus and a set of expected results. Each part of the model can be linked to the requirements and weighted (using probabilities) to their usages.

**MaTeLo Testor** contains several test generation algorithms:

- **User-oriented algorithm**: automatically generates the needed number of test scenarios (nominal, intuitive, unintuitive) in accordance with the usage of the system. These test sequences, executed with any test platform, triggers bugs by importance or criticality order due to the most probable usage of the system.
- **User-oriented with bounds values algorithm**: the path of the test case is calculated in the same way than the first algorithm. The main difference is that stimulation data are triggered in a deterministic way (lower bound, upper bound, or randomly upper or lower)
- **Arc coverage algorithm**: generation of a test suite which covers all test model transitions. The number of test steps is optimized.
- **Most probable algorithm**: generates the most probable test cases. According to a profile, the next transition chosen during the generation is the transition which is associated to the highest probability.
- **Tagged-path algorithm**: used to re-generate an existing test case. The principle is based on tags associated to some model elements. These tags should be set manually or, as an automation way, according to an existing test case. During generation, the algorithm decides the next taken transition according to these paths.

**MaTeLo Test Campaign Analysis** (TCA) makes available reports with quality and reliability indicators. With its SUT version management, TCA shows the system quality progression during the whole test life cycle. This tool is commercialized and used mainly in the transportation domain.
6.1.2 Development within MBAT

"Test cases generation based on test results": Using the test results from other levels or from the previous tests executions to guide the test generation. This can be done by adding flags automatically in the test model paths that have led to the discovery of errors in the system. This will help focusing on new parts of the test model instead of covering again these transitions.

"Test cases generation based on analysis results": Uses analysis results to generate test cases. Currently we have the following ideas:

- Using the invariant notion to reduce tests. When some properties are proved by analysis, then it is possible to avoid testing them.
- Using the Astrée analysis results to focus testing effort on detected system parts as faulted.
- Using Safety Architect tool results (by importing critical traces) to generate test cases.

"Test cases generation based on equivalent test cases identification": Generation of new test suites by removing redundant or equivalent test cases.

"Test cases generation based on coverage criteria":

- Generating a new test suite by filtering test cases in order to maximize the transitions coverage in a minimum of test cases.
- Generating test cases that cover all stimulations equivalent classes.
- Generating test cases that cover all model paths.
- Implementing an algorithm that covers a sub-set of transitions. This sub-set of transitions is deducted from a sub-set of requirements (to be covered or not).
7 TestCast MBT Provided by ELV

7.1 TestCast MBT

7.1.1 General Tool description
TestCast MBT is a model based test design, test execution and analysis tool. SUT modeling notation is UML state machine and the executable test case format is TTCN-3. The commercial version of the tool is available. The tool is not domain specific. It can be used widely in many different domains like automotive, avionics, industrial automation, telecom, etc.

TestCast MBT comes with two different test generation engines – Conformiq and Elvior Test Generator. More thorough information about test generation features of Conformiq test generator can be found at www.conformiq.com.

Elvior Test Generator features are the following:
- Hierarchical UML state machine is the input for the test generator.
- Any UML CASE tool that exports the models in XMI format can be used to prepare the model.
- Uppaal model checker is used for generating abstract test sequences.
- Test model has to be deterministic.
- Abstract test sequences are rendered to TTCN-3 test scripts.
- The following test coverage criteria can be used – requirements coverage, model structural coverage like all transitions, selected states/transition. Model structural test coverage criteria (selected states/transition) can be also ordered to test particular scenarios over test coverage items.

7.1.2 Development within MBAT

1) Support for OSLC
This is the key feature that allows connecting TestCast MBT to the MBAT RTP. This will enable TestCast MBT to participate in many different RTP instances. TestCast MBT will then be able to work with variety of requirement management tools and test management tools that the manufacturers of different software intensive products manufacturers have in place.

2) Attaching requirements to the model
This feature will allow introducing requirements driven model based testing. Currently this feature is available implicitly and it is not integrated with any 3rd party requirement management tools. Introduction of the feature will make the requirements driven testing more intuitive and user friendly.

3) Requirements coverage
Introducing the requirements coverage will allow to generate tests based on the individual requirements and to measure the overall test coverage in terms of requirements.

4) Requirements tracing through entire MBT workflow
This feature will make the requirements tracing on every phase of the testing workflow possible. Requirements will be possible to trace from requirements management system to the SUT model, to the generated abstract test cases to the concrete test cases and further to the test logs and back on the model level.

Table 6 TestCast MBT functionalities

<table>
<thead>
<tr>
<th></th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestCast MBT</td>
<td>Support for OSLC</td>
<td>Attaching requirements to the model</td>
<td>Requirements tracing through entire MBT workflow</td>
</tr>
</tbody>
</table>

© MBAT Consortium
8 Tools/Methods Provided by TUG

TUG supplies two tools for this work package, both using model based mutation testing for test case generation: the first tool, Ulysses, is part of a tool chain developed in cooperation with AIT. It allows test case generation from action systems. The frontend by AIT can process UML state charts and allows mutating the models. Within MBAT it will be enhanced by adding new input formats.

The second tool allows model-based mutation testing of timed automata in UPPAAL’s XML format and aims on the testing of real time behavior. It has been developed within MBAT and several new features are planned within the project.

<table>
<thead>
<tr>
<th>M2</th>
<th>M3</th>
<th>M4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ulysses and Frontends</strong></td>
<td><strong>SCADE frontend</strong></td>
<td>--</td>
</tr>
<tr>
<td>Test Case Generation from Action System models:</td>
<td>Mutation of SCADE models</td>
<td></td>
</tr>
<tr>
<td>• Model-based mutation testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Random testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutation analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stepwise Execution of models</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processing of:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• partial models</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• non-deterministic models</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML frontend</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutation of UML models</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Model Based Mutation Testing for Timed Automata (TA)</strong></td>
<td><strong>Parallel composition of TA</strong></td>
<td><strong>Adaptive test cases</strong></td>
</tr>
<tr>
<td>Test Case Generation from TA:</td>
<td>Processing of non-deterministic models (internal transitions)</td>
<td></td>
</tr>
<tr>
<td>• Model-based mutation testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Random testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stepwise execution of models</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processing of:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• partial models</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• under-specified models</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameterized transitions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutation of TA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time adaptive test cases</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7 TUG Tool functionalities

8.1 Ulysses

8.1.1 General Tool description

Ulysses is part of a combination of tools for test case generation from UML models with behavior expressed in state machine diagrams. The tool chain can also be used for mutation coverage analysis and to apply the test model as an oracle to test cases from other sources.

The test case generator uses altered test models (mutants) to derive test cases that are able to reveal whether a modeled fault has been implemented. This requires conformance checking between the original and the mutated model. The actual test case generator Ulysses works on an adaption of Back’s Action Systems (AS), interpreted as labelled transition systems. These action systems can be derived from UML state charts via object oriented action systems (OOAS). The model mutation and translation from UML to OOAS is done with the tool UMMU, the translation to AS with the command line tool Argos. Given a
specification and a set of mutated models, Ulysses produces a set of test cases. Each test case corresponds to a trace of one mutant to a transition that violates input / output conformance.

8.1.2 Development within MBAT
No changes to Ulysses itself have been planned, but the tool chain will be improved and Ulysses might be adapted to the changes.

8.2 Model Based Mutation Testing of Timed Automata

8.2.1 General Tool description
This is a mutation testing framework for real-time applications, where the model of the SUT and its mutants are expressed as a variant of timed automata. It implements an algorithm for mutation-based real-time test case generation that uses symbolic bounded model checking techniques and incremental solving. Mutation can be done via several different mutation operators. It can process models stored in UPPAAL’s XML files and stores the resulting test cases in the same format.

8.2.2 Development within MBAT
The tool itself has been developed during MBAT and several further improvements are planned for MBAT. The most important changes planned are the introduction of non-determinism and allowing parallel composition of timed automata.
9 Tools/Methods Provided by AAU

UPPAAL is a tool suite for modelling, simulating, verifying and testing real-time systems modelled as network of timed automata communicating via channel synchronization and extended with discrete variables. Yggdrasil and TRON targets model-based testing for such models, but in two different ways, a conventional offline test generation (Yggdrasil), and online testing (TRON). We plan a further integration of these now separate components and will therefore treat them as one tool in the following.

<table>
<thead>
<tr>
<th>Tools/Methods Provided by AAU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>9.1 Uppaal Testing Components</strong></td>
</tr>
<tr>
<td><strong>9.1.1 General Tool description</strong></td>
</tr>
<tr>
<td><strong>UPPAAL-TRON</strong> is a tool for black-box conformance testing of real-time systems at a system (or sub-system) level against timed automata models using observable timed I/O conformance relation as correctness criterion. The model consists of environment assumptions (one or more timed automata processes) composed in parallel with requirements for implementation under test (IUT) (again one or more timed automata processes). The requirements may contain non-determinism in time and functionality. The user then provides an adapter (via an API) which translates inputs from a model to physical inputs to IUT and recognizes physical outputs to outputs in the model. TRON then generates randomized timed input sequences based on environment assumptions, executes them via the adapter and monitors any incoming timed outputs and checks these against IUT requirements at the same time (online). The benefits of performing tests online are deep testing, stressful test cases, and adaptation to observation uncertainty; real-time specifications are highly non-deterministic (due to jitter in communication, scheduling, measurement imprecision and desired flexibility). The environment assumptions can be very specific (e.g. regression test), modelling use case, or very abstract, allowing any behavior, thus stressful for IUT.</td>
</tr>
<tr>
<td><strong>UPPAAL-Yggdrasil</strong> is an offline test case generator. The tool takes models created in UPPAAL and creates a suite of test cases that cover all required test purposes and all transitions in the model (edge or location coverage), executing a three phase test generation procedure, first targeting mandatory test purposes, then optimizing coverage, and finally targeting uncovered items. By using a special syntax within UPPAAL, the test suite that is the output of the tool can take the format of a test script, in principle in any desired language that can be used as input to test execution engines such as e.g. Selenium (<a href="http://seleniumhq.org">http://seleniumhq.org</a>).</td>
</tr>
<tr>
<td><strong>9.1.2 Development within MBAT</strong></td>
</tr>
<tr>
<td>Besides general tool improvements, the following developments of targeting MBAT goals is planned to be developed:</td>
</tr>
<tr>
<td><strong>TRON Emulator-Monitor Architecture:</strong> We propose a new tool architecture where the online testing execution divided into separate parts (processes): environment emulation (generating input stimuli) and monitoring processes (performing the oracle function of evaluating observed timed input/output sequences). This involves refactoring the testing algorithm and test adapter API. This</td>
</tr>
</tbody>
</table>
enables a more practical and effective setup for emulation and monitoring of timed properties. First, emulation is simpler than monitoring and has hard real-time requirements, and may thus benefit from a separate execution process. In contrast, monitoring is often computationally more involved, but may be delayed and performed asynchronously with emulation. Secondly, the increased flexibility also enables easier interaction with other (external) test, monitoring or analysis tools.

- **Hybrid and stochastic emulation and monitoring:** The current environment emulation algorithm in TRON supports timed automata. We plan to generalize this to include support for stochastic and hybrid systems based on timed automata extended with differential equations and stochastic transitions. The stochastic part is in particular useful for performance testing or operational profile based testing. Further, we intend to develop and experiment with probabilistic property monitoring using ideas from statistical model-checking where probabilities of satisfying a property (based on system executions) are estimated and reported. This is for example necessary for performance testing or testing of soft- or firm real-time system.

- **LSC property monitoring:** The Live Sequence Charts (LSC) is often a more intuitive way to specify properties than explicit automata or TCTL (Timed Computation Tree Logic) because they are graphical and scenario based. We plan to support monitoring of (a subset of) LSC properties via a translation into timed automata acceptable by TRON.

- **Interface and API for external emulators:** The new architecture of TRON will be capable of co-simulation with other external simulators. In particular we plan to develop Matlab/Simulink integration via co-simulation. This will enable environment emulation performed in part by TRON and in part by Matlab/Simulink. It also facilitates (approximate) refinement checking by means of testing of low level models against (timed) automata, by treating the low-level models, which are potentially large and rich model and thus not suitable for model-checking as the implementation under test.

- **Service enabling and RTP integration:** We intend to implement OSLC Adaptor for the Uppaal testing environment.

- **Test-GUI and simulator:** To make the Uppaal test generators TRON and Yggdasil more usable, we will develop a graphical user interface to these and integrate them via plug-ins in the main Uppaal environment. This will support model-partitioning (environment and SUI model), test setup and configuration, and diagnostics. Also we plan to extend the Uppaal simulator to visualize test coverage and support simulation and animation of a test case and test traces. Remark that coverage of non-deterministic and timed models is not trivial to obtain/measure, as a distinction between possibly and definitely covered (respectively not covered) items is necessary.

- **Analysis Links:** Export of test traces in an (open) XML format for analysis tools, re-execution of traces (specifically preset timed input sequences), and import of execution trace (for simulation, re-execution, and visualization of coverage) produced by other means, e.g. analysis tools.

- **Yggdrasil:** The current maturity of the tool is a (relative mature) prototype. The tool and procedure behind it has been successfully demonstrated effective on an industrial case (outside MBAT), but requires expert users and specifically trained engineers. Further, its user interface and engine are isolated from other Uppaal components. We will further develop Yggdrasil by integrating the agent based search and test generation features into the main engine, and developing a common test generation front-end with features for configuration, measuring and visualising coverage, traceability, simulation of test cases and execution traces. We will investigate how to support, and possibly implement, another feature that is required by MBAT: after a model modification, existing test cases (from a previous test generation iteration) should be re-used, provided that these test cases are still valid for the modified model. Finally, we plan to develop exporting of test cases into a standard format (e.g., TTCN3 or UML-TP), or XML format.

- **MBAT use-case specific test adaptors.** We will develop the test adaptors to support the specific use cases that AAU is involved in.
10 Tools/Methods Provided by KTH Stockholm

The Software Reliability Group (Meinke, Niu, de Oliveira, Sindhu) at KTH Stockholm carries out research and prototype tool development in the area of automated software testing. In recent years the group has focused on the area of learning-based testing (LBT) as a new technology for requirements-based black box testing. We have specifically focused on testing of reactive systems, which include client-server and embedded controller systems of the kinds found in MBAT.

LBT is a promising new technology for black box testing that combines methods from computational learning with model checking technologies to perform requirements testing from formal system requirements. Such system requirements are usually expressed in temporal logic. The aims of this technology are:

(i) to achieve a high volume of automatically generated test cases
(ii) to generate high quality test cases that are effective at finding bugs quickly
(iii) to require a minimum of human interaction with the system, in particular the oracle step should be completely automated by model checking,
(iv) to derive a self-optimising test case generation technique that systematically improves the quality of test cases.

A distinctive feature of LBT is that it can perform model-based testing even without a model. The computational learning technology that is used is able to extract a model from the outcome of current and previous test cases. This model is then used to derive counterexamples to correctness by the process of model checking. The derived counterexamples will then be used as new test cases. Therefore this technology has been termed "model based testing without a model".

Prior to MBAT we have implemented a variety of experimental LBT systems. These have convincingly demonstrated the soundness of our approach. Our published results in the literature show that LBT can outperform random testing by several orders of magnitude in the speed with which it finds errors. We have developed a test platform LBTest, which integrates the most viable technologies derived from our experimentation. This platform will be made available to the MBAT consortium as background technology.

<table>
<thead>
<tr>
<th></th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBTest</td>
<td>Fully functional workflow (as defined in 2.1)</td>
<td>OSLC/RTP integration v2</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>with Use Case Specific Extensions</td>
<td>Black box coverage models</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use Case specific test adaptors</td>
<td>Hybrid automaton learning techniques.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OSLC/RTP integration v1</td>
<td>Design and implementation of space-bounded model checker</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Design for a full RTPv2 integration</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9 LBTest functionalities

10.1 LBTest

10.1.1 General Tool description

LBTest has been developed as a platform to integrate different computational learning technologies and model checking technologies that together can support learning-based testing. This platform also solves
practical issues such as how to integrate a system under test (SUT) with the testing environment. Our data exchange protocol ensures smooth communication at all times between the test case generator, the SUT and the test oracle. Furthermore, when testing is complete, LBTest must convey the practical outcome of testing in a variety of ways, including black box coverage.

A noteworthy issue is the problem of coverage, since true black box testing cannot make reference to code coverage models such as node/path coverage, decision coverage, MC/DC etc. Instead we aim to support black box coverage models. In addition to test results and coverage, LBTest is able to export the learned model.

The inputs to LBTest are:

(i) A SUT as an executable program e.g. Java Byte code
(ii) A data exchange protocol, which identifies the interface of the SUT in terms of variables, types and values.
(iii) A set of requirements formulas expressed in propositional linear temporal logic (PLTL). We use the concrete syntax that is supported by the NuSMV model checker to achieve compatibility with other popular analysis tools.
(iv) A termination condition, that can either be a period of elapsed time (e.g. 5 hours), a minimum number of test cases (e.g. 10,000) or a convergence criterion (e.g. 10 identical learned models). This set of conditions will be added to as appropriate in MBAT.

The main output from LBTest is, for each requirements formula, a list of all test cases generated for that formula and for each test case a verdict (pass/fail/warning). Additional outputs include the learned model of the SUT.

A distinctive feature of LBTest is that it can test both safety requirements (nothing bad should happen) and liveness requirements (something good should happen). Note that liveness requirements include use case scenarios. Note that there exist liveness properties that cannot be refuted in finite time (termination is a good example, i.e. the absence of any infinite loop). In such cases, LBTest issues a warning that a liveness requirement has never been seen to be passed after some finite time. This time is user definable. Testing liveness requirements seems to be an important and unique feature of LBTest that we have found to be useful in practise, perhaps precisely because other tools cannot support this analysis.

Currently our model checking technologies come from other academic research groups outside KTH. These provide model checkers under public licenses such as GNU. The main externally integrated model checker currently supported is NuSMV.

**Workflow.**

LBTest is aimed at full 100% black box test automation. It will construct, execute and judge test cases automatically and on the fly until the termination criterion is met. Since the system tries to avoid interactive testing as much as possible, the workflow is fairly simple:

1. A data exchange interface between the LBTest and the SUT must be defined by the tool user. This specifies the SUT input and output variables to be written and read, their types and specific values for testing.
2. A set of behavioral requirements must be expressed in PLTL. These will be based on the data exchange specification of step 1, i.e. they will refer to specific input/output variables and specific values for these.
3. Just before an automatic testing session can be initiated, a black box termination criterion must be specified. LBTest supports several types of black box criteria, and this will grow with future releases.
4. When steps 1, 2 and 3 are complete, a testing session can be started. On a typical case study, a testing session will last several hours and automatically generate and execute 10,000-50,000 test cases, until the termination criterion is met. This step is typically performed overnight.
5. The outcome of the test session is reported, and various test artifacts may be exported in XML

Note that the learning algorithm currently used in LBTest can only learn a finite state machine. Therefore, if infinite data types are involved (such as floating point, integer etc.) then these must be sampled or
approximated by a finite sample set. This typically involves discretizing the infinite domain, and usually requires some domain knowledge on the part of the tool end user. This discretization problem is not dissimilar to a boundary testing problem, which is a known problem in the literature. This discretization will be formally defined in the data exchange interface. KTH-CSC collaborates mainly in the Volvo brake-by-wire (BBW) use case. We will study approaches using LBTest together with different discretization strategies for BBW.

10.1.2 Development within MBAT

(1) RTP Compatibility Model. Collaboration with MBAT must solve the problem of integrating LBTest with the MBAT RTP. For this KTH-CSC will actively participate in the definition of RTP standards for data interchange and sharing of testing and analysis artifacts.

(2) RTP Compatibility Implementation. Based on the agreed RTP standards we will produce a design for full RTP integration, and incrementally implement this design by bringing current data interchange standards in line with MBAT RTP recommendations. Within the constraints of the time and financial resources (man-months) available to KTH, we will fully implement these data exchange facilities.

(3) Discretization Strategies and Methods. KTH-CSC collaborates mainly on the Volvo brake-by-wire (BBW) use case. Here we will study the application of LBTest within various workflows and scenarios suggested by the MBAT RTP. Furthermore, we will study the efficiency of using LBTest to find errors using different data discretization strategies. The outcomes of all these studies will be used to improve LBTest within the context of MBAT, including methodologies to solve this problem in future use cases.

(4) Black box coverage models. Since LBTest is a black box testing tool, we will study existing and new black box coverage models, including requirements coverage models. These will be implemented and added to LBTest and evaluated in the context of Volvo BBW.

(5) Hybrid Systems Learning. The Volvo BBW use case can be viewed as a kind of hybrid system. However, hybrid systems represent a class of infinite state systems for which no learning algorithms currently exist. We will investigate new learning algorithms for hybrid systems which can be used for learning based testing. We will consider if, and how such learning algorithms can be used within LBTest for hybrid systems, for example by combining with existing model checkers for hybrid systems.

(6) Resource-bounded model checkers for infinite state systems. We will consider the development of resource bounded model checkers that can be used to generate test cases even for infinite state systems. Currently model checkers for infinite state systems often exhaust memory resources before a problem can be solved. We will apply new results in space bounded theorem proving to improve the quality of model checking in this respect.
11 Tools/Methods Provided by PikeTec

The main contribution from PikeTec to this work package is the enhancement of TPT (Time Partition Testing). TPT is a test tool-suite for model-based test. TPT is specialized and tailored for testing embedded control and feedback control systems on a model, software, or controller level.

<table>
<thead>
<tr>
<th>M2</th>
<th>M3</th>
<th>M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPT</td>
<td>Concept and Feasibility Study regarding TCG for TPT test models</td>
<td>TPT TCG v1 prototype extension of TPT for service oriented API testing</td>
</tr>
<tr>
<td>Concept for the support of service oriented API testing for control systems</td>
<td>TPT model improvements</td>
<td></td>
</tr>
</tbody>
</table>

Table 10 TPT functionalities

11.1 TPT

11.1.1 General Tool description

TPT is a model-based testing tool for testing embedded systems, especially the testing of control systems. TPT supports test modeling, test execution, test assessment, and test documentation for such systems – fully automated, even in real-time contexts. Test models described with TPT implement deterministic test behavior with real-time semantics. The test models are purely functional oriented (black-box) and consist of hierarchical state machines.

For the testing of complex systems TPT offers a powerful approach for systematic test-case-generation which guarantees easy interaction and readability even with a larger number of tests. Test cases are modeled graphically based on hybrid test models that are derived from a functional specification. These test models comprise all test relevant aspects that belong to the system under test (SUT) and represent all data and timing aspects of the tests. Test execution with TPT is done fully automatically in the test environment. The automatic test evaluation can be done online (i.e. during the test execution) or offline (at the end of the test execution). It can formulate complex quality criteria made up from powerful operations such as comparisons with reference data, signal filters, state sequences and timing conditions. TPT produces a test report of the test execution which contains the most important information regarding the execution and results of the test case. The information in the report can be configured such that it remains a readable summary of the execution even for highly complex tests.

Up to now the general procedure to construct a TPT test model is to derive it manually from the functional requirements specification. For that purpose there is the ability to link requirements to test cases and to track the functional test coverage together with the test model.

11.1.1.1 TPT test models

The approach for test modeling used within TPT is based on a specialized notation of state machines. The models are usually created manually based on the functional requirements specification. The test model represents an enclosing model of all tests and constitutes semantically the superset of all paths of usage scenarios that are considered as test cases. The prerequisite is a SUT (system under test) with a well-defined interface, for example a Simulink model:
Figure 11-1: Interface of an example SUT (Simulink model)

With TPT the simplest test model is a state sequence as follows:

In this sequence the overall course of the test is divided into three states: “switch is OFF”, “switch is ON” and “switch is OFF again”. The transitions between those states represent the conditions at which the state machine switches from one state to another.

The modeling language of TPT allows to use labels with human language annotations within the state machine. Nonetheless there is a formal language behind the graphical representation.

The TPT state machines support parallel and hierarchical state machines as well. Parallelism is deterministic (synchronous execution of parallel state machines). In the example in Figure 11-3, there are two parallel state machine separated graphically by a horizontal line.

The state “ambient light” in the figure above is the only state in the second (parallel) state machine. It is responsible to specify the stimulation for SUT input “sensor” in this case.

The TPT test modelling language also supports a concept called variation points. This means that states and transitions can have multiple variants of informal and formal behavior. This means that, for example, a state “turn switch” can define two different variants: one for turning ON a switch and one to turn into some AUTO state, as depicted in the figure below. The same flexibility exists for transitions.

Test cases within a test model that contains variation points are constituted by a projection that selects one variant per variation point. So, in the example above there are 2*2=4 alternative options for the constitution of test cases. This concept ensures that the overall test model remains coherent even for bigger test problems.
11.1.1.2 Test model and test case construction

Up to now the general procedure to construct a TPT test model is to derive it manually from the functional requirements specification. For that purpose there is the ability to link requirements to test cases and to track the functional test coverage together with the test model.

11.1.1.3 Tools for modeling and construction

The tool that supports both, modeling and execution of TPT test models is called “TPT” too. It is a commercial tool developed by PikeTec.

11.1.2 Development within MBAT

1) **TPT modeling for service oriented control systems.** Since control systems are getting more and more complex the need for service API oriented testing in the domain of control systems is increasing. PikeTec will define and implement extensions to the TPT test modeling notation that support hybrid test modeling containing signal and service oriented tests in a consistent flavor. The purpose is to achieve a higher degree of expressive power for test models for complex control systems.

2) **TPT interface for RTP integration.** PikeTec will develop an API that allows access to data elements and functions of TPT for external tool components. This API will leverage the RTP integration.

3) **Concept and Feasibility Study regarding TPT TCG extension.** PikeTec will extend TPT in order to integrate test case generation techniques in close co-operation with Fraunhofer IESE. The idea is to create test models based on functional requirements and statistical information that will be used to fine-tune the test case generation procedure. The purpose of this enhancement is to achieve higher efficiency and improved overall coverage during development of test cases.

4) **Application and Workflow Evaluation in MBAT use cases.** The prototypes developed in (3) above will be evaluated based on PikeTec internal and MBAT use cases (e.g. UC4, Daimler Blinker).
12 Tools/Methods Provided by ALES

Advanced Laboratory on Embedded Systems (ALES) develops methods and tools for the validation and verification of complex embedded systems. ALES contributes to the WP with the enhancement of the Automatic Test Generation plug-in of the FormalSpecs Verifier framework (FSV-ATG) as summarized in Table 11.

<table>
<thead>
<tr>
<th>M2</th>
<th>M3</th>
<th>M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSV-ATG</td>
<td>Contract-based test generation</td>
<td>Contract-based test generation optimization using formal verification (V1)</td>
</tr>
<tr>
<td></td>
<td>RTP (v1) Integration</td>
<td>RTP (v2) Integration</td>
</tr>
</tbody>
</table>

Table 11 FormalSpecs Verifier functionalities

12.1 FormalSpecs Verifier

12.1.1 General Tool description

The FormalSpecs Verifier (FSV) is a model based framework for the verification of complex embedded systems providing a MATLAB Simulink based environment for the description and verification of requirements modeled as contracts. A contract is a formal representation of a requirement made of two parts: the *promises*, which are properties that must be guaranteed, and the *assumptions*, which define under what conditions the promises must hold. The FSV tool provides a plug-in for contract-based test case generation called FSV-ATG.

The ATG engine is used for requirement-based test generation based on the workflow based on Figure 12-1.

![Figure 12-1: FSV-ATG flow](image)

Requirements formalization

Requirements are modeled using the BCL (Block-based Contract Language) that allows for the formalization of requirements as contracts using the concept of assertion that represents a formalized sentence. Assertions can be composed using specific operators to build complex formal sentences and contracts are specified using this mechanism identifying for each of them the assumption and promise assertions. Figure 12-2 represents an example of contract specification using BCL. Atomic patterns are used to assert...
conditions that must hold for the assumptions or promises or to compose them. Finally the contract block identifies the assumption and the promise.

![Diagram of BCL contract specification in Simulink](image)

**Figure 12-2: Example of BCL contract specification in Simulink**

*Test generation execution*

After requirements are formalized, the FSV-ATG tool produces a test suite for each contract using an offline test case generation technique and storing the different test cases as xml artifacts. The generation process can be customized using specific generation options. The FSV-ATG engine uses the NuSMV\(^1\) model checker as backend for the test generation process.

*Model-in-the-loop test case simulation*

The FSV-ATG provides an automatic flow to import an existing MATLAB Simulink model and exercising it using the generated test case as inputs.

*Tool status and application domains*

The tool is a robust prototype but further development is needed to be applied in an industrial context. The target application domains are avionics and transportation.

### 12.1.2 Development within MBAT

In the context of the MBAT project ALES will develop the FSV-ATG tool in several directions as summarized in this sub-section. In addition to the specific enhancements listed below, effort will be dedicated to fully support the use case providers to adapt and optimize the application of the tool and of the techniques to the use cases.

*Contract based test generation*

The tool will be enhanced to provide a fully automatic support of contract-based test generation that currently relies on a manual step to guide the engine. This will be performed by adapting the BCL language Simulink implementation and improving the test generation flow. In addition contracts coverage objectives will be identified and taken into account during the test generation process.

*Contract-based test generation optimization using formal verification*

To fully exploit the synergies between static analysis and testing we plan to use formal verification techniques to optimize the test case generation process by applying formal analysis on contracts in order to obtain formal proof that can be used to drive the ATG process. The goal of this process is to reduce the total number of generated tests improving the coverage of each case. This step will be addressed enhancing the interoperability between the FormalSpecs Verifier Property Verification and the ATG engine as well as implementing novel ATG techniques.

\(^1\) [http://nusmv.fbk.eu/](http://nusmv.fbk.eu/)
RTP Integration
The FSV-ATG will be integrated in the MBAT RTP (v1) platform to enable full interoperability with the MBAT platform developing the needed adapters.