# Overall T&A Methodology

**D_WP2.1_2_1**

<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Overall T&amp;A Methodology</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_WP2.1_2_1</td>
</tr>
<tr>
<td>Contract Number</td>
<td>269335</td>
</tr>
<tr>
<td>Date</td>
<td>2013-03-26</td>
</tr>
<tr>
<td>Status</td>
<td>Final, submitted to ARTEMIS-JU</td>
</tr>
<tr>
<td>Contact Person</td>
<td>Brian Nielsen</td>
</tr>
<tr>
<td>Organisation</td>
<td>AAU</td>
</tr>
<tr>
<td>Phone</td>
<td>+45 99408883</td>
</tr>
<tr>
<td>E-Mail</td>
<td><a href="mailto:bnielsen@cs.aau.dk">bnielsen@cs.aau.dk</a></td>
</tr>
</tbody>
</table>
## Authors

<table>
<thead>
<tr>
<th>Name</th>
<th>Company</th>
<th>E-Mail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brian Nielsen</td>
<td>AAU</td>
<td><a href="mailto:bnielsen@cs.aau.dk">bnielsen@cs.aau.dk</a></td>
</tr>
<tr>
<td>Michael Siegel</td>
<td>OFFIS</td>
<td><a href="mailto:Michael.Siegel@offis.de">Michael.Siegel@offis.de</a></td>
</tr>
<tr>
<td>Udo Brockmeyer</td>
<td>BTC</td>
<td><a href="mailto:Udo.Brockmeyer@btc-es.de">Udo.Brockmeyer@btc-es.de</a></td>
</tr>
<tr>
<td>Wolfgang Herzner</td>
<td>AIT</td>
<td><a href="mailto:Wolfgang.Herzner@ait.ac.at">Wolfgang.Herzner@ait.ac.at</a></td>
</tr>
<tr>
<td>Eckard Böde</td>
<td>OFFIS</td>
<td><a href="mailto:eckard.boede@offis.de">eckard.boede@offis.de</a></td>
</tr>
<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>Eckard Bringmann</td>
<td>PikeTec</td>
<td><a href="mailto:Eckard.Bringmann@piketec.com">Eckard.Bringmann@piketec.com</a></td>
</tr>
<tr>
<td>Reny Grönberg</td>
<td>PikeTec</td>
<td><a href="mailto:Reny.Groenberg@piketec.com">Reny.Groenberg@piketec.com</a></td>
</tr>
<tr>
<td>Markus Pister</td>
<td>AbsInt</td>
<td><a href="mailto:pister@absint.com">pister@absint.com</a></td>
</tr>
<tr>
<td>Thomas Bauer</td>
<td>IEESE</td>
<td><a href="mailto:thomas.bauer@iese.fraunhofer.de">thomas.bauer@iese.fraunhofer.de</a></td>
</tr>
<tr>
<td>Frank Elberzhager</td>
<td>IEESE</td>
<td><a href="mailto:frank.elberzhager@iese.fraunhofer.de">frank.elberzhager@iese.fraunhofer.de</a></td>
</tr>
<tr>
<td>Jens Hermann</td>
<td>DAI</td>
<td><a href="mailto:jens.herrmann@daimler.com">jens.herrmann@daimler.com</a></td>
</tr>
</tbody>
</table>

## Reviewers

<table>
<thead>
<tr>
<th>Name</th>
<th>Company</th>
<th>E-Mail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harald Brandl</td>
<td>AVL</td>
<td><a href="mailto:harald.brandl@avl.com">harald.brandl@avl.com</a></td>
</tr>
<tr>
<td>Michael Dierkes</td>
<td>Rockwell Collins France</td>
<td><a href="mailto:mdierkes@rockwellcollins.com">mdierkes@rockwellcollins.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>WP2.1</td>
<td>MBAT</td>
<td>PU</td>
<td>R</td>
</tr>
<tr>
<td>MBAT Partners</td>
<td></td>
<td>PU</td>
<td>R</td>
</tr>
<tr>
<td>Antonio Vecchio, Project Officer</td>
<td>ARTEMIS-JU</td>
<td>PU</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>0.1</td>
<td>16-06-2012</td>
<td>Initial draft</td>
<td></td>
</tr>
<tr>
<td>0.2</td>
<td>28-08-2012</td>
<td>Included early comments by BTC, AIT (Herzner), M Siegel OFFIS, AAU, looked at D_WP1.2_1_1 &quot;current processes&quot;, applied template, reordered sections, toc</td>
<td>*</td>
</tr>
<tr>
<td>0.2</td>
<td>05-02-2013</td>
<td>Integrated methodology presented at Y1 Review</td>
<td>*</td>
</tr>
<tr>
<td>0.5</td>
<td>14-02-2013</td>
<td>Workflow descriptions, Tools, Intro Revised by Herzner</td>
<td>*</td>
</tr>
<tr>
<td>0.6</td>
<td>21-02-2013</td>
<td>Added Levels from BTC, Added more on overall method, Restructured</td>
<td>*</td>
</tr>
<tr>
<td>0.7</td>
<td>05-03-2013</td>
<td>WH added contributions by AIT, Piketec, Offis, BN added: ABS, TUG, MDU, RWC, IESE</td>
<td>*</td>
</tr>
<tr>
<td>0.8</td>
<td>06-03-2013</td>
<td>Addressed some comments, Added contribution by Jens (sect 3.1) Moved &quot;generic step&quot; to appear earlier</td>
<td>*</td>
</tr>
<tr>
<td>0.9</td>
<td>12-03-2012</td>
<td>Addressing comments from Telco (11/3), and other inputs, Microguides for MBT, Model-checking</td>
<td>*</td>
</tr>
<tr>
<td>0.92</td>
<td>14-03-2012</td>
<td>Finalizing document for review</td>
<td>*</td>
</tr>
<tr>
<td>1.0</td>
<td>26-03-2012</td>
<td>Addressed comments by reviewers, and ENEA, and inputs on example from Absint.</td>
<td>*</td>
</tr>
</tbody>
</table>
# CONTENT

## 1 INTRODUCTION

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Scope and Purpose</td>
<td>7</td>
</tr>
<tr>
<td>1.2 Relation to MBAT RTP</td>
<td>8</td>
</tr>
<tr>
<td>1.3 Preconditions</td>
<td>9</td>
</tr>
<tr>
<td>1.4 Outline</td>
<td>10</td>
</tr>
</tbody>
</table>

## 2 BACKGROUND

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Verification and Validation</td>
<td>11</td>
</tr>
<tr>
<td>2.2 MBAT V-reference-model and V&amp;V activities</td>
<td>12</td>
</tr>
<tr>
<td>2.3 Levels/types of “combined use”</td>
<td>15</td>
</tr>
<tr>
<td>2.3.1 Overview</td>
<td>15</td>
</tr>
<tr>
<td>2.3.2 Model-based Testing – Overview</td>
<td>16</td>
</tr>
<tr>
<td>2.3.3 Static program analysis – Overview and Applications</td>
<td>16</td>
</tr>
<tr>
<td>2.3.4 Combining model-based Testing and Static Analysis</td>
<td>17</td>
</tr>
</tbody>
</table>

## 3 OVERALL METHOD

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Vision</td>
<td>18</td>
</tr>
<tr>
<td>3.1.1 The motivating idea behind the combination</td>
<td>18</td>
</tr>
<tr>
<td>3.1.2 The combination picture</td>
<td>18</td>
</tr>
<tr>
<td>3.1.3 Objective “Reducing Costs”</td>
<td>19</td>
</tr>
<tr>
<td>3.1.4 Objective “Detecting more defects”</td>
<td>19</td>
</tr>
<tr>
<td>3.1.5 High level perspective on the MBAT method</td>
<td>19</td>
</tr>
<tr>
<td>3.2 Main V&amp;V flow</td>
<td>21</td>
</tr>
<tr>
<td>3.3 A generic A&amp;T step</td>
<td>22</td>
</tr>
<tr>
<td>3.4 Exploiting combined use</td>
<td>25</td>
</tr>
<tr>
<td>3.4.1 Exploiting results of Model analysis</td>
<td>27</td>
</tr>
<tr>
<td>3.4.2 Exploiting results of Testing</td>
<td>27</td>
</tr>
<tr>
<td>3.4.3 Exploiting results of code analysis</td>
<td>28</td>
</tr>
<tr>
<td>3.4.4 The in2Test approach and general concepts</td>
<td>31</td>
</tr>
<tr>
<td>3.5 Integration with the RTP</td>
<td>31</td>
</tr>
</tbody>
</table>

## 4 GUIDELINES FOR CHOOSING AND USING TECHNIQUES

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Guidelines about techniques</td>
<td>33</td>
</tr>
<tr>
<td>4.1.1 Classification of techniques</td>
<td>33</td>
</tr>
<tr>
<td>4.1.2 Model-checking</td>
<td>34</td>
</tr>
<tr>
<td>4.1.3 Abstract interpretation</td>
<td>35</td>
</tr>
<tr>
<td>4.1.4 Theorem proving</td>
<td>36</td>
</tr>
<tr>
<td>4.1.5 Review and inspection</td>
<td>36</td>
</tr>
<tr>
<td>4.1.6 Model-based testing</td>
<td>36</td>
</tr>
<tr>
<td>4.1.7 Monitoring (passive testing)</td>
<td>38</td>
</tr>
<tr>
<td>4.1.8 Simulation</td>
<td>39</td>
</tr>
<tr>
<td>4.1.9 Debugging</td>
<td>39</td>
</tr>
<tr>
<td>4.1.10 Risk analysis methods</td>
<td>39</td>
</tr>
<tr>
<td>4.1.11 Overall comparison</td>
<td>41</td>
</tr>
<tr>
<td>4.2 Best Practices</td>
<td>43</td>
</tr>
<tr>
<td>4.2.1 Model and analyze early</td>
<td>43</td>
</tr>
<tr>
<td>4.2.2 Design for verification and testability</td>
<td>44</td>
</tr>
<tr>
<td>4.2.3 Joint formal input constraints and requirements</td>
<td>44</td>
</tr>
<tr>
<td>4.2.4 Make joint V&amp;V planning</td>
<td>44</td>
</tr>
<tr>
<td>4.2.5 Perform (early) static code analysis</td>
<td>45</td>
</tr>
<tr>
<td>4.2.6 Analysis first, then test (test what cannot be analyzed)</td>
<td>45</td>
</tr>
<tr>
<td>4.2.7 Prioritize effort</td>
<td>45</td>
</tr>
</tbody>
</table>

## 5 EXAMPLES

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Rockwell Collins “Flight Guidance System Mode Logic”</td>
<td>46</td>
</tr>
</tbody>
</table>

© MBAT Consortium
Content of Figures

Figure 1: Relationship of overall methodology (green) to MBAT RTP ingredients ............................................ 9
Figure 2: Conventional V-Model in the Transportation Domain ........................................................................... 13
Figure 3: Consolidated V&V Activities in Automotive (from D_WP1_1.2.1) .................................................... 14
Figure 4: MBAT Combined Approach ........................................................................................................... 18
Figure 5: Combined A&T for the Major Levels ................................................................................................. 20
Figure 6: Main V&V Flow: Relations and means to check them ........................................................................ 21
Figure 7: A Generic Analysis and Test Step ................................................................................................... 23
Figure 8: Meta-model Concepts for V&V ....................................................................................................... 23
Figure 9: Propagating Results among techniques ........................................................................................... 26
Figure 10: The In² Test approach ................................................................................................................... 29
Figure 11: In² Test Process ................................................................................................................................ 29
Figure 12: General concepts of a combined usage of V&V techniques ............................................................ 31
Figure 13: Organization of A&T Steps and status propagation via RTP ........................................................... 32
Figure 14: Classification of Main (automated) V&V techniques ...................................................................... 33
Figure 15: Illustration of Important MBAT Techniques ..................................................................................... 34
Figure 16: Comparison of main A&T Techniques ............................................................................................... 43
Figure 17: A&T workflow for the flight mode control system proposed by Rockwell Collins. ........................... 47
Figure 18: Identified A&T Steps (green) and Potential new steps (orange) .......................................................... 47
Figure 19: Example of a model-checking step (hypothetical) ......................................................................... 49
Figure 20: Example of a static code analysis step (hypothetical) ................................................................... 50
Figure 21: Quality assurance activities applied in UCA4 ................................................................................. 51
Figure 22: Decomposition of Requirements (based on [Engel, 2010]) ............................................................. 53
Figure 23: Example of a Requirements Verification Matrix (from [Engel, 2010]) ............................................. 54
Figure 24: Another example of an RVM [DOD, 2006] ..................................................................................... 54
Figure 25: Modeling workflow (with analysis) ................................................................................................. 58
Figure 26: Main Model-based testing workflow ............................................................................................... 60
Figure 27: Test selection process ..................................................................................................................... 61
Figure 28: Model-based mutation testing workflow ......................................................................................... 61
Figure 29: Counter-example driven abstraction refinement .............................................................................. 69
Figure 30: Workflow with Model-checking ..................................................................................................... 70
Figure 31: Analysis Tool Landscape ................................................................................................................ 72
Figure 32: Test Tool Landscape (G=Test Generation, E=Test Execution) .......................................................... 73

Content of Tables

Table 1: Overview of risk analysis methods ................................................................................................. 40
Table 2: Comparison of Testing and Static Analysis ...................................................................................... 42
1 Introduction

1.1 Scope and Purpose

The main goal of MBAT is to meet the industrial needs within EU, in particular in its transportation domains, for improving verification and validation (V&V) of embedded software, in order to assure high product quality despite facing increasing system complexity, to reduce costs of defects and of V&V efforts, and to reduce the time-to-market.

It is a main thesis of the project that significant progress towards these goals can be reached by the use and smart combination of advanced existing model-based test and analysis techniques and tools. However, a simple collection of individual tools is insufficient, even if they are mature and applicable in an industrial context, because their isolated application can increase quality of and trust in the target systems, as current experiences show, but at (too) high costs. Hence, MBAT aims at smart combinations of analysis and test techniques that allow reducing the overall V&V costs without compromising the system quality (or even with its increase). For that purpose, it is insufficient that the tools are interoperable and systematically applicable, they have also to be used in an optimized manner, or even in a way yielding a new V&V quality that cannot be achieved by isolated tools alone.

Therefore, the objective of this deliverable is an overall methodology that will support effective application and combination of the MBAT techniques for model-based analysis, test case generation and test execution, with focus on (at least):

- Workflow adaptation / optimization
- Management of complexity
- Exploitation of MBA & MBT synergies.

This deliverable presents the preliminary results towards such a methodology. A final, second, version will be produced towards the end of the project (Month 30). Other related deliverables are:

- D_WP2_1_3 MBA/MBT Synergy Exploitation (“a set of recommendations and strategies for mutual improving MBA and MBT by means of exploiting results of the one domain in the other”).
- D_WP2.2_1_1-3 Report and tool prototypes on MBAT models and their construction approaches, needed for Fast Track RTP
- D_WP2.3_2 Specification of Model-based Test Case Generation & Execution Methods and Tools
- D_WP2.4_2 Specification of Model-based Analysis Methods and Tools
- D_WP1.2_1 Current State of Practice of V&V processes
- D_WP1.1_3 Collection of validated Work Package Requirements
- D_WP3.1_1 Meta Models for RTP
- D_WP3.2_2 Specification for RTP Interoperability

The reason for the two-step development of this deliverable is that it not only considers background knowledge, i.e., information that is available at begin of the project, e.g.

- State of the art/practice of MBA&T uses
- Partner experiences
- Theoretical foundations

but it shall also regard foreground knowledge finally, i.e., information generated in the course of the project, e.g.

- Use case experiences
- New tool combinations.

The output of this task will be a specification of the overall model-based analysis and test methodology to be developed in SP2, including a set of recommendations and guidelines for how to apply model-based analysis...
and testing, their combined use, and the principal process outline of how to integrate the new techniques and their application within the MBAT RTP into the V&V process.

Since this is an interactive process – first aids should be given to the use cases in the first project half, while new insights gained during the project should be finally included – several versions of the deliverable are foreseen, as mentioned above.

1.2 Relation to MBAT RTP

MBAT contains a set of other activities, which serve the same goal as the overall methodology addressed in this deliverable. Central to these activities is the reference technology platform (RTP), the primary outcome of MBAT. The MBAT RTP itself is composed of several ingredients, which together enable smart and flexible composition of model-based T&A tools, namely

- **MBAT meta-model** (MBAT-MM): defines all terms and their relations needed by MBAT RTP such that artifacts and semantics can be unambiguously shared by RTP components and A&T tools populating the RTP. See [MBAT3, 2012] for more details on the MBAT-MM.
- **MBAT interoperability specification** (MBAT-IOS): means and specification allowing diverse MBA&T tools to exchange information and to interact. It exploits an interoperability model that allows to associate artifacts such as analysis objectives or test cases on syntactic and semantic level. It is itself closely related to MBAT-MM. See [MBAT4, 2013] for more details on MBAT-IOS.
- **RTP basic services**: tools provided by RTP the provide services such as logging or requirements tracking that can be used by other tools in a generic and application independent manner. See [MBAT2, 2012] for examples of RTP basic services.

In addition to these core RTP elements, all modeling languages, concrete T&A models as well as model-based analysis and test (case generation) tools, which use the RTP services in order to cooperate, complete the RTP. All these means (specifications, models, tools) allow that V&V tools can interoperate; e.g., they describe what kind of information needs to be exchanged, and how it can be done. But they do rather not address in which way and order interoperation steps should be carried in order to optimize their efficiency; i.e., they do not address the interoperation process itself. This process is the topic of this deliverable, and hence complements the RTP ingredients listed above (which are all managed by subproject 3). In this sense the RTP is the technical framework for the realization of the concepts of the methodology.

It cannot be assumed, however, that one single process or work-flow would suffice all potential applications. It appears even unlikely that for a given use case a rigid process would be feasible. Instead, it appears more appropriate to identify a set of rules, guidelines, or assumptions which, given a certain situation (requirements, application, constraints on available MBA&T tools/methods), guide the user by choosing and combining the appropriate techniques. As these advices will in most cases settle on experience and best practice, results from their application should be used for their assessment. The latter can be used to improve the overall methodology, including modification and update of the set of rules and guidelines.

These relationships are illustrated in Figure 1. Edges outgoing from the meta-model (MBAT-MM) shall indicate that concepts, terms, and relations among these defined by the MM are used both by other RTP components as well as by the methodology for combined T&A addressed here. Blue arrows indicate potential interchange paths of artifacts (models, test cases) between T&A tools; for explanation of other symbols refer to [MBAT4, 2013].
1.3 Preconditions

Some fundamental pre-conditions the methodology works under are:

- It is developed from the perspective of validation and verification, less than on development, as carried out by (sometimes independent) verification and validation teams/experts.
- It must support different abstraction levels ranging from specification, design to implementation, and it must support architectural and behavioral aspects, and functional as well as extra-functional properties like timing, reliability, performance. Tools are supplied by different tool vendors, and are not interoperable a priori and with semantic variations. MBAT will enable interoperability and exchange of artifacts and results via the Reference Tool Platform (RTP) being developed in MBAT.
- The preferences of modeling notation are not the same across all domains and industries (e.g., Simulink in the automotive domain and Scade in the rail and avionic domains).
- There is not one single modeling/requirements notation that captures all these aspects, and that is supported by all vendors (at least not with an accurate semantic basis). Hence test and analysis models will be a heterogeneous collection of model languages and notations.

The methodology must address several issues:

- **Workflow**: The new and advanced techniques proposed by MBAT will significantly affect the current V&V workflow. The overall methodology will deal with all steps from turning requirements into models (including interfacing with model-based engineering), handling transformations and variations of these, choosing the specific MBA and MBT techniques, applying these, to interpreting and combining the results, and performing impact analysis.
- **Management of complexity**: As previously stated, systems are so large and complex that their correctness cannot be established through a single modeling, analysis or testing step. This requires integration of two orthogonal views: the techniques for modeling, analysis and test generation, and the system level (including a product line view) at which they are applied. The overall methodology should therefore address the challenge of planning what system properties and quality aspects
should be modeled, verified and tested and at what levels, how to make best use of each technique, and how to combine the results of each activity to make statements about system correctness and quality. It is also desirable that the method has capability to take verification credit from pre-tested / pre-certified components which get part of a composed system.

- **Exploit MBA and MBT synergies:** MBA and MBT can benefit from each other resulting in lower V&V effort and higher system quality. An example of one highly beneficial aspect of this is to formally analyze test models to check that these satisfy important consistency and correctness requirements, in part because it is “easy” access to the benefits of exhaustive formal analysis, and in part because, using MBT, the test model is the specification of the test cases, and using a test model that fails satisfying important system requirements will result in the testing of wrong properties and erroneous test cases. Another aspect is that due to bug clustering, an error found in formal analysis identifies a troublesome area that should be further investigated using testing (or vice versa). A further example is that algorithms for formal analysis and state exploration may help generating sound test cases, and test suites with a-priory model-coverage. There are numerous other examples; it is not the aim of the methodology to provide an exhaustive catalogue of these, but provide a common framework that can explain most of these.

- **Exploit risk and safety analysis:** The proposed formal risk and safety analysis techniques can be further exploited by directing other MBT&A efforts to areas of higher risk.

- **Support for traceability and impact analysis:** The method should support traceability among requirements, models, and analysis results. Similarly, the goal of impact analysis is to identify the parts of the artifacts (requirements, model, test case, system component) models or systems that are affected by an identified error. Thus, enabling traceability is essential for reducing the V&V complexity. The mechanisms for traceability and impact analysis are to be provided by the RTP.

### 1.4 Outline

The remainder of this deliverable is organized as follows. Chapter 2 introduces high level background information, summarizing currently used development processes, and identification of three levels of integration of combined model-based testing and analysis.

Chapter 3 introduces the overall method. This includes the visions behind it and potential gains to be achieved from its applications, the idea of using requirements and verification results and coverage to guide analysis and testing, the definition of a flow of verification tasks from requirements to final system such that a chain of arguments of correctness are established, the concept of a generic A&T step that systematically transforms a set of requirements into verification results and additional verification obligations, and finally a set of guidelines for how to exploit combined analysis and rest results. The RTP contains the underlying mechanisms needed to support working according to the proposed methodology.

Chapter 4 provides guidelines and best practices for putting the MBAT techniques into practice. It presents a classification of the main V&V techniques, and a discussion of their inherent advantages and disadvantages.

Chapter 5 exemplifies important elements of the method.

Chapter 7 is intended to give engineers specific advice on how to work with the MBAT model-based testing and analysis techniques in a professional manner by outlining good practices and work flows for modeling and analysis, static code analysis based on abstract interpretation, model-based testing, and model-checking.

Chapter 8 summarizes the current MBAT tool landscape, and Chapter 9 concludes and outlines future work.

The overall method is deliberately introduced early in the document, leaving more in-depth discussions and descriptions till later, but guiding the reader with forward references where appropriate.
2 Background
This chapter collects general terms and conceptual background that will be considered in the elaboration of the overall MBAT methodology.

2.1 Verification and Validation
Software V&V processes determines whether the development products of a given activity conform to the requirements of that activity and whether the software satisfies its intended use and user needs. Specifically, the “1012-2004 IEEE Standard for Software Verification and Validation” [IEEE-1012. 2004] defines Verification and Validation as

The verification process provides objective evidence whether the software and its associated products and processes
a) Conform to requirements (e.g., for correctness, completeness, consistency, accuracy) for all life cycle activities during each life cycle process (acquisition, supply, development, operation, and maintenance)
b) Satisfy standards, practices, and conventions during life cycle processes
c) Successfully complete each life cycle activity and satisfy all the criteria for initiating succeeding life cycle activities (e.g., building the software correctly)

The validation process provides evidence whether the software and its associated products and processes
a) Satisfy system requirements allocated to software at the end of each life cycle activity
b) Solve the right problem (e.g., correctly model physical laws, implement business rules, use the proper system assumptions)
c) Satisfy intended use and user needs

Similarly, in ISO 9000:2008 it is defined as [Boulanger(a), 2012]:
Verification: Confirmation by tangible proof that the specified requirements have been met at each stage of the production process.

Validation: Confirmation by tangible proof that the anticipated requirements for a specific use or application have been met.

Overall speaking, V&V can be performed using static techniques or dynamic techniques. Static techniques analyses the system (or artifacts describing it) without actually executing it. Dynamic techniques are based on observing sample executions of the system (or artifacts describing it). In the following, analysis refers to particular static techniques, whereas testing refers to particular dynamic techniques. Generally a model is “a semantically closed abstraction of a software or hardware system, i.e. a simplification of reality that gives a complete description a system from a particular perspective.”, e.g., reliability growth models, usage models such as operational profiles, or behavioral models such as decision table or state transition diagrams. Model-based testing is “testing based on a model of the component or system under test (SUT)” [ISTQB, 2012]. Model-based analysis is the algorithmic checking of properties of models or relationships between models. More detailed definitions, classifications, and descriptions of the techniques relevant for MBAT are provided in Section 4.1.

During development, developers and engineers may make mistakes in understanding specifications, wrong use of a programming language that may result in a fault in the produced artifact. A fault may lead to an erroneous system state that can manifest itself in the observable failure of the system where the system does not deliver the expected result, reaction or performance. In other words

Mistakes lead to faults lead to errors \(^1\) lead to failures

\(^1\) The notion of error is used differently in different standards. Following the standard IEC 1508: “Functional Safety: Safety-Related Systems”, an error is a discrepancy between a computed, observed, or measured...
Hence following the safety standard IEC 1508:

Mistake: A human action that produces an incorrect result. [After IEEE 610]. Examples are the use of x coordinates where y coordinates was intended, the use of a wrong array index, locking a wrong mutex-lock (or its omission), etc.

Fault: An incorrect step, process, or data definition in a computer program which causes the program to perform in an unintended or unanticipated manner. For instance $a = \mathcal{F}(y, x)$ rather than $a = \mathcal{F}(x, y)$, or $a = A[n]$ rather than $a = A[n-1]$.

Error: A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. For instance, the variable $a$ become -1 rather than the expected, say 10.

Failure: Deviation of the component or system from its expected delivery, service or result. For instance, a cruise control system pulls back on the throttle where an increase was expected.

We will use the term **defect** to denote any undesired system property, anomaly, fault, bug, error, and flaw in a component or system that can cause the component or system to fail to perform its required function, e.g., an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system. One aim of V&V is to check that sufficiently few and serious defects exist for the targeted level of quality.

### 2.2 MBAT V-reference-model and V&V activities

In MBAT the V-model is used as reference model for structuring A&T activities, i.e., it serves as a vehicle for explaining the decomposition of V&V activities at various phases and levels, and functions as a common well-understood abstract model. It is not proposed as the model for a MBAT Combined A&T process: Companies have their own tailored and well established processes that consequently are difficult to change. However, they are often sufficiently similar to the V-model to allow new concepts explained in context of the V-model to be transferred to the current practices with relative ease. Therefore, the combined A&T method will take outset in the V-model, rather than revolutionize the company processes.
Figure 2: Conventional V-Model in the Transportation Domain
### Main Activities (Automotive)

<table>
<thead>
<tr>
<th>1 - User's needs</th>
<th>Company specific activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Requirements elicitation</td>
<td></td>
</tr>
<tr>
<td>System specification</td>
<td></td>
</tr>
<tr>
<td>System Requirements specification</td>
<td></td>
</tr>
<tr>
<td>System architecture definition</td>
<td></td>
</tr>
<tr>
<td>System Architecture Validation</td>
<td></td>
</tr>
<tr>
<td>Safety analysis</td>
<td></td>
</tr>
<tr>
<td>System Requirements Validation</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2 - System Design</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Component requirement specification</td>
<td></td>
</tr>
<tr>
<td>SW Test Specification</td>
<td>Black-Box Testing</td>
</tr>
<tr>
<td>SW component development</td>
<td>White-Box Testing</td>
</tr>
<tr>
<td>Standards Compliance</td>
<td>Analysis</td>
</tr>
<tr>
<td>SW components verification</td>
<td></td>
</tr>
<tr>
<td>HW Test Specification</td>
<td></td>
</tr>
<tr>
<td>HW component development</td>
<td></td>
</tr>
<tr>
<td>Test Reports</td>
<td></td>
</tr>
<tr>
<td>HW component verification</td>
<td></td>
</tr>
<tr>
<td>HW/SW integration</td>
<td></td>
</tr>
<tr>
<td>HW/SW integration test</td>
<td></td>
</tr>
<tr>
<td>System integration</td>
<td></td>
</tr>
<tr>
<td>Real System Test</td>
<td></td>
</tr>
<tr>
<td>System Validation &amp; Verification</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3 - Implementation &amp; Unit Test</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional safety assessment</td>
<td></td>
</tr>
<tr>
<td>Requirements Validation</td>
<td>Acceptance Test / Tool Qualification</td>
</tr>
<tr>
<td>Robustness Test</td>
<td></td>
</tr>
<tr>
<td>Assemble Vehicle</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3:** Consolidated V&V Activities in Automotive (from D_WP1_1.2.1).

Figure 2 shows the current V-model typically found in the automotive domain, and Figure 3 lists the V&V activities in the V-model in more detail. Remark that this is the set of current activities, and is not necessarily considered an ideal set of activities or appears in an ideal ordering.

In contrast to Figure 2, in MBAT V&V activities will to a significant extent be based on models. Thus, different models will exist at the identified levels, e.g., system level specification models, subsystem level architectural and behavioral models, and detailed component and implementation level models, each supporting V&V at that level, and between levels. Further, MBAT not only uses testing, but various analysis techniques, and combined application of testing and analysis.

It should be stressed that application of models and combined model-based analysis and testing is *not* limited to the V-model or processes built around it, but can equally well be applied in incremental and agile methods. A key observation is that models are also grown incrementally by gradually adding behavior aspects of the systems, and gradually supporting analysis or testing of an increasing set of properties or test purposes. Automated model-based tools have the advantage that artifacts (analysis results, test scripts, code) can be automatically re-generated with each increment, with potentially less manual rework. Yet there is less experience with model-based approaches in agile methods in industrial practice, and furthermore, it is a challenge for modeling notations and tools to make easy to incrementally add aspects to models without significant rework. Thus, agile model-based development remains an area where practitioners and...
researchers must collaborate to find best practices. For further information to model-based testing and testing strategies in agile projects, see [Utting, 2007], [Claesson, 2012].

2.3 Levels/Types of “Combined Use”

The following provides a high level description of the benefits industry may achieve from an application of model-based analysis and testing, and their use in combination. An important point is that benefits of combination can be gained at different levels of integration of the techniques, also identified in the following.

2.3.1 Overview

Safety-related software has to satisfy stringent quality requirements. The complexity of software-implemented functionality grows at a fast pace, but at the same time the development teams have to meet tight budget constraints while facing pressure to reduce time-to-market. To meet these conflicting goals the development process has to be sound and efficient. A leap in development efficiency can be reached by a holistic model-centric approach to software development, testing and verification.

In model-based development the software is graphically developed at a high abstraction level, typically by hierarchical finite state machines and data flow diagrams which represent specification and model at the same time. From this high-level model the implementation is automatically generated by configurable code generators, typically in the form of C code. Model-based testing aims at automating testing activities and integrating the development of both design artifacts and test artifacts in a unified framework. This makes it possible to automatically create test architectures, and generate and execute the test cases. Furthermore, the traceability from requirements to test cases checking them against the generated software implementation can be automatically established. Alternatively, model checking can be applied to formally verify properties of the model.

The functional safety for road vehicles is regulated by the norm ISO-26262 [ISO26266, 2011] published in 2011. It requires to identify functional and non-functional hazards and to demonstrate that the software does not violate the relevant safety goals. Depending on the criticality level of the software the absence of safety hazards has to be demonstrated by formal methods or testing with sufficient coverage. Functional properties can be efficiently addressed by automatic model-based testing.

The critical non-functional safety-relevant software characteristics are essentially implementation-level properties, e.g., whether real-time requirements can be met, whether stack overflows can occur, and whether there can be runtime errors like invalid pointer accesses, or divisions by zero. The classification as non-functional properties is a historic terminology; actually, they are functional in that sense that they are crucial for a correct functioning of the system. Because of the high abstraction level of model-based development these properties are largely hidden from the developers. Moreover they are very hard to check experimentally, i.e., by testing and measurements. Identifying safe end-of-test criteria for program properties like timing, stack size, and runtime errors is an unsolved problem. In consequence the required test effort is high, the tests require access to the physical hardware and the results are not complete. Formal verification methods provide an alternative, in particular for safety-critical applications. One such method is abstract interpretation, which allows properties to be proven for all program runs with all inputs. Nowadays, abstract interpretation-based static analyzers that can detect stack overflows and violations of timing constraints [Souyris, 2005], and that can prove the absence of run-time errors, are applicable by industry [Delmas, 2007] (cf. Section 4.1.3 and 7.3). The advantage of abstract interpretation is that it enables full control and data coverage, but can be easily automatized and can reduce the testing effort.

Hence, from a workflow perspective the verification process is split into two parts: model-based testing and model checking of models is used for showing functional program properties, and static analysis to prove the absence of non-functional program errors. For the time being these methods are implemented in separate tools. First that means that additional effort is required to create the needed V&V harnesses for the model/code under test and static analysis, respectively. Furthermore, abstract interpretation based static analysis can profit from a coupling to the modeling level, since often not all model-level information is accessible from the generated code. To compute safe results imprecise information is handled conservatively which can lead to an increased number of false alarms. Lastly, neglecting the non-functional
software properties during model-based design and testing entails the risk of detecting severe errors only late in the development stage.

2.3.2 Model-based Testing – Overview

For safety-related software, it is mandatory to have full traceability from requirements to software architecture to code. Additionally, traceability from requirements to test cases that check the correctness of requirements against the developed software is required (in particular important when the code is handcrafted and not auto-generated by certified code-generators). Implementing elements such as test cases as model-level concepts allows bidirectional traceability to be realized directly at the model level. This enables automated analysis of both requirements coverage and structural coverage of model and code. Additionally, when test cases are specified and executed on the model level verification and problem analysis is much easier and more efficient than it is for traditional, code-centric test cases. A model-based approach allows the development of both design artifacts and test artifacts in a unified framework. Therefore, it adds agility to the development and test process, and that improves efficiency and lowers cost in comparison to processes with separate development and traditional code-centric test phases. Model-based testing approaches automate many of the testing activities, including creation of test architectures, generation and execution of test cases. Therefore, testers can merely focus on the correctness and completeness of their test cases rather than at the same time spending time and mental effort on tedious and error-prone tasks, such as creating test harnesses. Finally, path synthesis tools can help generate tests to achieve the required code coverage, see also Section 7.2.

2.3.3 Static Program Analysis – Overview and Applications

During the last years static analyzers based on model-checkers and abstract interpretation have evolved to be the state of the art for verifying functional and non-functional software properties. A static analyzer is a software tool which computes information about the software under analysis without executing it. Abstract interpretation is a semantics-based method for program analysis which belongs to the formal verification methods [ISO26266, 2011]. Its results are sound, i.e., they are valid for all program runs with all inputs and achieve full data and control coverage. The soundness of the analysis can be formally proven. Examples for abstract interpretation based static analyzers are tools to compute safe upper bounds on the worst-case execution time and the maximal stack usage of tasks [Ferdinand, 2007] and to prove the absence of runtime errors [Kästner, 2011].

Stack Usage Analysis: One possible cause of catastrophic failure is a stack overflow which might cause the program to behave in a wrong way or to crash altogether. When they occur, stack overflows can be hard to diagnose and hard to reproduce. The problem is that the memory area for the stack usually must be reserved by the programmer. Underestimation of the maximum stack usage leads to stack overflow, while overestimation means wasting memory resources. Measuring the maximum stack usage with a debugger is no solution since one only obtains a result for a single program run with fixed input. Even repeated measurements with various inputs cannot guarantee that the maximum stack usage is ever observed.

Worst-Case Execution Time Prediction Many tasks in safety-critical embedded systems have hard real-time characteristics. Meeting their deadlines is necessary to maintain a correct system behavior. Due the characteristics of modern software and hardware [Kästner, 2012] determining the Worst-Case Execution Time (WCET) of a task is a difficult problem. A survey of methods for WCET analysis and of WCET tools is given in [Wilhelm, 2008].

Runtime Error Analysis: A further class of critical programming errors is the so-called runtime errors, e.g., arithmetic overflows, array bound violations, division by zero, and invalid pointer accesses. They can destroy the data integrity of a program, causing the program to behave erroneously, or to crash altogether. A well-known example for the possible effects of runtime errors is the explosion of the Ariane 5 rocket in 1996.

An alternative way to perform formal model and software verification is to apply model checking. Model checking can be used to formally verify that the model fulfills a set of formally specified requirements. Also,
model checking can be applied to check robustness properties of models. Once a model has been formally proven against a set of requirements it can be considered to be a thorough test of the model due to its complete nature. There is a trade-off in applying model checking versus model based testing. While model checking is a complete method due to its mathematical nature, this technique has some severe limitations regarding complexity of the system under test. In contrast, model based testing if often able to deal with larger sized models, but testing is always incomplete of course.

2.3.4 Combining Model-based Testing and Static Analysis

As discussed above state-of-the-art solutions use model-based testing including model checking for showing functional program properties, and static analysis to prove invariants and the absence of non-functional program errors. For the time being these methods are implemented in separate tools. Such tools are not yet integrated into holistic V&V environments, and hence have to be applied individually. This requires additional effort to create the needed V&V harnesses for the model/code under test and static analysis, respectively. In other words, V&V cost is much higher than it could be due to the missing integration. In the sequel we present an integrated approach for model-based testing and analysis addressing both aspects seamlessly. Model-level information like execution model or environment specifications are automatically taken into account, reducing setup for test and analysis efforts and improving analysis precision. Tests and analyses can be launched seamlessly and produce result reports. This allows to comprehensively address the safety requirements of the ISO 26262, and to significantly reduce V&V efforts. Combination of analysis and testing techniques can be implemented with various degrees of integration, leading to more or less deep couplings of the methods and tools. In the sequel of this discussion we distinguish three kinds of integration, moving from a loosely coupling into a real deep coupling of methods and tools:

1. **Workflow integration**: with workflow integration we associate that two or more test and analysis tools are loosely coupled within a customer V&V process. The main purpose is the integrated application of such analysis and test tools. Workflow optimization is the main benefit, hence saving time and money. The tools are still visible as individual tools, but some interfaces and loosely couplings have been implemented between such tools.

2. **Methodological integration**: with methodological integration we associate that two or more test and analysis tools are closely coupled within a customer V&V methodology inside a given customer process. The main purposes are the integrated application of such analysis and test tools including feedback loops between the tools. Workflow optimization and increased test and analysis precision are the main benefits, hence saving time and money as well as improving customer product quality. The tools might still be visible as individual tools, or hidden underneath a unified user interface layer to ease usage. Obviously, some interfaces and close couplings have been implemented between such tools.

3. **Technical Tool integration**: with technical tool integration we associate that two or more test and analysis techniques are deeply coupled within one common tool environment. Thus, the individual analysis and test tools and techniques are hidden underneath a unified user interface layer to ease usage. The main purposes are the integrated application of such analysis and test tools including strong feedback loops between the tools. Workflow optimization, increased test and analysis precision, and convenience in using the tools are the main benefits. Hence saving time and money, improving customer product quality, as well as bringing fun into the model based test and analysis work. Obviously, some interfaces and deep couplings have been implemented between such tools and techniques.

The main thesis of MBAT is that a significant improvement can be achieved by the efficient coupling of existing tools, and hence MBAT mainly targets the first two levels. The third level requires new research and significant development effort on the individual tools, and even then cannot span all the tools needed in systems development. Hence, an effective solution for the first two levels is required.
3 Overall Method

3.1 Vision

3.1.1 The motivating Idea behind the Combination

As already introduced, the basic idea motivating the combination of static analysis and dynamic testing is to
- reduce costs for V&V and/or,
- detect more defects in the system under V&V.

For both points, several scenarios could be described. E.g.,
- reducing overall V&V costs by uncovering defects applying those V&V techniques that are the
  cheapest to uncover specific kind of defects,
- detecting more defects by applying the most powerful techniques that are applicable in a special
  situation.

Basically, it is all about a smart combination, i.e., to decide when to use which V&V techniques for a given
purpose.

3.1.2 The Combination Picture

The MBAT combined approach is depicted in Figure 4 (compare the numbers in the figure):
1. Based on embedded system descriptions, an A&T model is created. This A&T model is reflecting the
   point of view of V&V people having in mind to uncover defects in the system under V&V and hence
   containing data to support exactly this purpose.
2. Ideally, analysis and/or test cases could be generated automatically out of this A&T model.
3. Outcomes of the V&V activities executed will be fed back into the A&T model.
4. A decision is made which step to take next (one of the both numbered with 2) or to terminate the
   V&V activities.

![Figure 4: MBAT Combined Approach](image-url)
3.1.3 Objective “Reducing Costs”

Basically, the reduction of costs is possible when each of the two V&V categories of techniques (analysis and test) is applied for the case where uncovering defects with a technique from one category is cheaper than with a technique from the other category. E.g., structural defects as data or control flow anomalies (e.g., unused variable values or dead code), boundary violations or division by zero can basically be found by static analysis. The statically detectable defects shall be aimed at and removed first. In addition, all kind of metrics as e.g., cyclomatic complexity could be calculated.

Once the static analysis has found all defects that it can basically find, testing could start, taking the results of the static analysis into account for steering testing activities. E.g., if a there are regions with high cyclomatic complexity then it may make sense to increase the testing intensity for these regions.

Basically, as testing is usually much more expensive compared to static analysis there should be as much static analysis as reasonable as first step and testing afterwards heading for defects that are probably remaining in the system under V&V.

3.1.4 Objective “Detecting more Defects”

It is an experience gained in industrial environment and also confirmed by case studies that in regions where faults are in the system under V&V there are most probably more. One explanation that is psychologically motivated is that defects are made because the developer has misinterpreted requirements or systems descriptions and has not only produced weaknesses in a certain spot of the system, but a few more in the vicinity of that spot.

For the combined approach in MBAT this means that if one V&V technique discover a defect in the system under V&V it can find some more in the vicinity of this defect and that also the other V&V technique can support uncovering an defect in this region, also because the other technique has another V&V point of view, heading for other kind of defects.

3.1.5 High Level Perspective on the MBAT Method

The MBAT method must support verification at all major levels (System Level, Subsystem Level, and Component Level) and encourage reuse within each level. Also it must support verification between levels, and encourage reuse of results established at one level to verify the next). The use of different verification techniques and integration of results must be coordinated via a V&V plan that is developed under guidelines for the different techniques considering guidelines for reuse of results. The RTP serve as the vehicle underlying the methodology keeping a repository of the requirements, verification objectives and status, and for commencing the required tools, see also Figure 13.
The general approach for combined analysis and testing can be applied at the major levels and stages in the V-model as depicted in Figure 5. Thus, the main approach is:

1. From the requirements, make a V&V plan that considers both (model-based) analysis and testing techniques.
2. Respectively, define and execute analysis and test cases, supported by T&A model(s)
3. Evaluate results and coverage
4. Revise verification plan and models, and iterate until sufficient confidence in the V&V task has been achieved.

A T&A model is a system description (abstract behavioral model) that is developed to enable V&V of a selected set of requirements of the system under V&V by analysis, testing, or as emphasized in MBAT, a clever combination of the two. As the Figure 5 suggest there may be made several T&A models during the development cycle at different levels of abstraction and also focusing on different subsets of requirements (e.g., different system characteristics), see also Section 7.1.

In addition, the method should, at each level, provide guidelines when to do what (analysis/testing), guidelines for revision at the next level, and guidelines for design for verification. The method aims at supporting reuse in several ways:

- Reuse (intra-level): A main point of the MBAT method is that different requirements can be verified using different techniques using the most feasible and effective one for the problem at hand. Similarly, the feedback loop from established results to the test and analysis tasks enable focusing and improving each technique, see Section 3.4.
- Reuse (horizontal): Models and verification results established during design and implementation can form the basis for and provide inputs to models and verification obligations at the V&V. However, this should be balanced against the need for independent critical V&V. For V&V, the models and results must be revised with critical eyes.

---

2 The system under V&V may itself be a model at a lower level of abstraction.
• Reuse (vertically): Results established by a technique at a lower level, can be used as an assumption when considering models at a higher level. Similarly, simplifications, abstractions made at a higher level must be justified through modeling, testing, and verification at lower levels. For example, WCET analysis of the code at the implementation is needed to perform schedulability analysis and task allocation optimization at the sub system level, which again is necessary to establish system level end-to-end timing properties.

3.2 Main V&V flow

The MBAT method must address the flow from requirements till implemented system and establish a verification chain of arguments. The suggested primary steps in the MBAT method are depicted in Figure 6. Also it illustrates which technique is most dominating at each transition.

Figure 6: Main V&V Flow: Relations and means to check them

1. From the requirements, a (formal) specification model is constructed that reflects the main aspects of specified behavior. There is much evidence suggesting that such a high-level model both greatly helps in understanding the implications of the requirements, and in obtaining a complete and consistent set of requirements. The consistency of the model can be checked by
   • Animation and visualization
   • Logical satisfiability checks
   • Other sanity checks (deadlock, basic reachability, etc.)

2. From this requirement model, the design and V&V flow can start; analysis, test, and design models can be derived. Ideally, the dedicated analysis and test models are derived through automatic semantic preserving transformations. However, this is not always possible, but still the manual construction use information from design models and documents. Also V&V (independent of developers) may require a degree of independently constructed models.

The A&T models must satisfy (or refine) the requirements:

---

3 The figure emphasize the V&V, although the links 2, 3, 4 also exist in design and implementation, in addition to others. It is not a goal of MBAT to provide a complete design methodology.
• Given that both (requirement and A&T models) are formalized, this can be done by analysis or model check (provided language compatibility).
• Alternatively, the A&T model is too large or complex, or if analysis tool support is inadequate, simulation/testing/monitoring (model-in-the-loop) of the A&T model can be used as an approximation.
• The two techniques can be used in conjunction such that unverified requirements can be tested or simulated.

3. The low level design models are typically richer models and elaborate in detail how (as opposed to what is required) each component is going to function. Logically the relation between the high level and low level design is a refinement.
• In principle it is a relation between two models, so it can be checked by analysis.
• However due to richness and size, it may be necessary to do MiL simulation and testing, (or combines as indicated above in item 2). Similarly, for legacy components where no model is available testing may be necessary.
• Due to size it is often necessary to perform the refinement check component-wise.
• (Model-learning is a technique for extracting models of observed behavior and thus an approximate automated abstraction technique. Currently it is not ready for industrial use.)

4. The relation between the produced (manually and in some cases also synthesized) code for a component and its design model is in principle a refinement, that can be checked by
• Static code analysis: verifies satisfaction of component properties like absence of certain runtime errors, and satisfaction of invariants and assertions (typically derived from design or component level requirements). Similarly (with additional platform assumptions) worst-case execution time and stack consumption can be computed.
• Functionality is checked by testing (Software in the Loop).
• Directly model-checking source code (a.k.a software model checking) is not readily an available MBAT technique.

5. The integrated system is validated by means of testing (Hardware/Processor in the Loop).

With WP2.2 on model-construction, the individual types and notations for models for the different levels and application domains will be identified.

### 3.3 A Generic A&T Step

To systematically capture the (subset of) requirements checked by one verification technique, the results produced, and identified new verification cases, and feedback of results, we propose the concept of a generic analysis or testing step.

The main idea is that system correctness can be verified by verifying each requirement, i.e., that correctness follows by a conjunction of the results of verifying each requirement. This is not a trivial assumption as 1) each requirement may influence the behavior of several sub systems, and thus induce several sub-requirements/properties to be checked, 2) initial requirements are informal, and it is not trivial to deduce the right (formal) properties to check, and 3) capturing, understanding and validating the assumptions implicitly or explicitly made by a verification step.

To reduce these problems we suggest that requirements (potentially decomposed), derived properties, and the underlying assumptions, and verification results, are systematically and explicitly traced for each verification step and result.

Because each verification step may be based on different analysis and testing techniques (test, abstract Interpretation, model-checking, simulation, etc.) we try to capture the relationship between requirement and verification result independently of the choice of specific technique and tool. A generic analysis and testing step is depicted in Figure 7:
Below a number of important concepts are recalled from the MBAT Meta-model [MBAT3, 2012]:

**Requirement:** A measurable statement of intent about something that the product must do, or a property that the product must have, or a constraint on the system.

**V&V Objective:** A V&V Objective is a description of the capability to be analyzed or tested in a VV Case. The objective is typically derived from a requirement or related to an obligation for verification or validation given by a trace-link. The evaluation of such an objective is performed in a referenced VV Case. V&V objectives are situations which are of interest for V&V. They are largely determined on the basis of the V&V object’s status, its system descriptions, and the intended quality targets. For each user requirement there is at least one V&V Objective and a related V&V case should be generated which examines if the developed embedded software system meets the requirement (or not). In doing so, it is important to determine and test the quality criteria linked with the requirements. Determination of V&V objectives is an essential activity with the help of which the quality and scope, and thereby also the effectiveness and the costs of an analysis can be assessed. Additionally, it is a very creative phase in which the imagination, intuition, and experience of the V&V team play a vital role. V&V case determination often aims at detecting most possible errors and realizing the highest possible V&V object coverage.

**V&V Case:** A V&V Case is a specification of one case to test or to analyze. One V&V Case is a complete technical specification of how a subject component should be tested or analyzed for a given V&V Objective. It is a behavioral feature or behavior specifying test. A V&V Case specifies how a set of environment components interacts with a subject component to realize a V&V Objective and returns a verdict value.
**V&V Suite:** A V&V Suite represents a verification or validation effort describing a combination of multiple analyses or tests in the form of V&V Cases. The V&V Suite defines a set of V&V Cases.

**Test Vector:** Test Vector is a concept that denotes a set of values to be used as stimuli for the system under test and an intended outcome for these stimuli in a VV Case.

**V&V results log:** The V&V Results Log organizes and presents the test results and key measures of analysis for review and assessment. In addition, the Summary contains an evaluation, by the V&V team, of this information and recommendations for future efforts.

**V&V object:** All kinds of (intermediate) development products to be checked by V&V (i.e., inputs for V&V). Examples are Requirements, Design, Models, Component code, Subsystem code, System code, and Interfaces.

A generic step is preparation and execution of a specific **verification procedure** (tool or integrated tool set) on a selected set of requirements (or verification obligations). It has the following main ingredients:

- It has a well-defined purpose defining the intended knowledge to be gained and expectances about the results. Typically the purpose is linked to one or more requirements (or aspects thereof) to be checked.

- Based on the purpose and included requirements, a number of V&V objectives are defined that should be checked by this step. In a next step these will be turned into the exact procedure to be executed. A case corresponds to a precisely described observation the engineer would like to make on the object under analysis and test.

- Development and V&V engineers make a number of (implicit and explicit) assumptions during their tasks. Typically these are related to the behavior of the operational environment of the system (the “plant”), the underlying hardware, or external components. Assumptions may be about (expected) capabilities and compliance of developed or bought components to standards or requirements. However, assumptions may also be related to simplifications or abstractions made during verification, e.g., that a given function is implemented correctly, bounds on response time from a component, that one component has no (or ignorable) influence on the (sub-) system under study.

It is important to make these assumptions explicit and to systematically justify their validity. If no convincing informal argument can be made about the validity of an assumption, it may spawn of another V&V step whose purpose is to validate this assumption, or the risk assessment should be updated with the knowledge about this precarious assumption.

- The object under A&T consists of the main artifacts to be studied, e.g., a model to be analyzed, code to be statically checked, or, for MBT, a pair of system under test and test model.

- The results include a verdict for each analysis or test case together with logfiles, computed metrics, traces etc. The MBAT Meta-Model [MBAT3, 2012] defines Verdicht as a predefined enumeration specifying the set of possible evaluations of an analysis or test:
  - Pass: The analysis or test result adheres to the V&V Objective. That is, the observation requirement defined by the objective and A&T case was successfully obtained without any evidence to the contrary.
  - Fail: The analysis or test result does not adhere to the V&V Objective. The observed behavior contradicts the required and allowed behavior. Hence, evidence of non-compliance with the A&T procedure has been identified.
  - Inconclusive: The evaluation cannot be evaluated to be pass or fail.
  - Error: An error has occurred within the analysis or testing environment.
  - Suspect: The V&V Case is new or the last analysis or test result is suspicious.

The Inconclusive verdict may be issued for a number of reasons:

- Unobserved (Inconc): The desired observation defined by the A&T case could not be obtained, but also no evidence of non-compliance was detected.
Maybe (Maybe Not): The defined observation can possibly (possibly not) be made. These verdicts typically arise in static analysis tools that compute over- or under-approximations of system behavior to make analysis possible/decidable or efficient.

None: Result could not be obtained. This may occur if the implemented procedure for some reason is not executable e.g., because a static analysis tool runs out of memory or takes too long, or because the test harness does not function as expected.

When the results have first been obtained, or if the verdict is not as expected, the V&V engineer is expected to scrutinize the results and logs, to ensure that the verdicts are really valid and not resulting from errors in the procedure (test cases, specified properties, models, etc.), and if necessary make the local corrections.

If verdicts are unexpected this may spawn off new A&T cases to be checked in another A&T step, possibly using another technique, e.g., “an error found by a test case warrant static analysis of the infected area due to bug clustering” assumptions to be verified by other means. Similarly, any new assumption made during the A&T step may require execution of another step to check its validity. Additional metrics can be collected as part conducting a step that is useful and important when evaluating the outcome of a step and its consequences on other verification activities. Examples include obtained coverage, complexity of code and models, warning density, defects, See Section 3.4.4.

3.4 Exploiting Combined use

As indicated in the previous section, it is necessary to use different techniques during the V&V cycle (analysis of models for critical requirements and components, static analysis of code, simulation for large and rich models where analysis tools fail, and testing for physical systems), each with different strengths and weaknesses. However, further benefits can be obtained by switching between analysis and testing and using results produced by one to focus the other. This feedback loop is depicted in Figure 9.
Consider a given level of abstraction (system, sub-system or component), the V&V in MBAT will be conducted using analysis (of models), testing, and analysis (of code).

1. Partition the requirements, assumed to be refined to fit the given level of abstraction, into 3 sets of requirements that should be checked by model-analysis, testing, and static code analysis using the most suitable technique for that requirement.

   The main hypothesis is that a requirement proven by one technique (e.g., analysis) need not be checked again by another (e.g., testing). However, such a hypothesis is not trivially valid to make because there rarely is a formalized production/engineering flow that guarantees preservation of an established property. Therefore it is also difficult to give universal advice on when this is a reasonable hypothesis, but requires scrutiny and insights by the V&V engineer. For this reasons it is likely that the sets of requirements will intersect, i.e., there are requirements (or aspects thereof) that need results from application of more than one technique. It also depends on the criticality of the system\(^4\) and qualification level of the used tool suite.

   It is normally recommended (See Section 4.2 on best practices) to start with analysis and (formally) verify as many requirements as possible at the model level. But there are often functional and extra-functional requirements that cannot be checked on the model level (because the model is not rich or detailed enough - e.g., one cannot verify timing on a model that is purely functional.

2. From the requirements and other engineering artifacts available a suitable model is constructed for the V&V task at hand (See Modeling Guide in Section 7.1)

\(^4\) the higher the required quality of the target system, the more evidence for it is required (e.g. if a safety case has to be established)
3. After suitable preparation, the analysis or testing step is executed to obtain results (See Section 3.3 on possible outcomes)

4. The evaluation of each result may roughly fall into one of the following categories:
   a. The requirement is verified and sufficient evidence is at hand to reasonably conclude that the requirement is satisfied and no further V&V for that is necessary.
   b. The requirement is not satisfied, and there is evidence to its contrary. Correcting actions are needed.
   c. The result is inconclusive. There is no result available. The requirement needs to be checked by alternative techniques or alternative tools (e.g., simulation, testing, or manual test.), or the requirement should be refined into simpler sub-requirements.
   d. A need for additional checks has been identified, either because suspect behavior has been identified or for uncovered verification obligations and items. A complementary technique may be attempted.

5. The V&V plan must be updated with the new verification status, and a revised plan for the changed items must be made. Iterate.

Next we consider the outcomes and actions to be taken one technique at a time. The degree to which these feedback steps can be automated depends on the capabilities of the tools involved and compatibility of the used modeling notations.

3.4.1 Exploiting Results of Model analysis

   a. Success (Verified): By assumption the requirement is satisfied. If the assumption is precarious, see case d, below.
   b. Failed: The model, and engineering artifacts that it depends on, code, documents etc. must be corrected, and re-verification of the (impacted) system parts and requirements is necessary. It may be necessary to update the plan with (requirements for) analysis and (additional) test cases for regression testing. In particular, the failed test case and associated trace should be fed to the model analyzer to determine if similar errors can be found on the model level.
   c. Inconclusive: Depending on the reason for the missing results (here assuming that over- or under approximate verification options of the tool did not yield useful results) two actions can be taken
      • Use (under) approximate methods like simulation and testing (MiL)
      • Strengthen assumptions behind model, simplify model, refine requirements into simpler objectives, or replace some components in model by more abstract versions. Verify by additional V&V requirements/cases that the introduced abstractions are valid.
   d. Suspected: During verification it may turn out that certain requirements have been difficult to formulate, or a certain model-element (component model) has needed repeated correction to enable verification, or simply that “tricky” scenarios has been observed. In these cases it is recommended to formulate additional checks for (analysis or testing). It may be particularly interesting to execute a (lower level) SUT or simulation under supervision of the “interesting” high level trace provided by the model-checker.

3.4.2 Exploiting Results of Testing

   a. Success (Passed): By assumption the requirement is satisfied. However, since testing cannot show the absence of errors (only few positive traces can be observed) care must be taken. If this assumption is considered too fragile, see case d, below.
   b. Failed: The model, and engineering artifacts that it depends on, code, documents etc. must be corrected, and re-verification of the (impacted) system parts and requirements is necessary. It may be necessary to update the plan with (requirements for) analysis and test cases for regression testing. The generated counter examples can serve as a starting point.
   c. Inconclusive: The used tool failed to complete the tests, or observed results that do not allow deciding whether the tests performed successful or failed. Counter measures are (a) to refine the test cases, (b) additional instrumentation of the SUT to enable the required observation and control,
(c) refine (decompose) requirement and try to verify refined requirements, (d) to replace the tool. If no other technical explanations can be found, it should be treated as a suspected case (if the engineer has defined an observation to be made on the SUT, it is suspicious, if after repeated execution, the SUT never reveals this behavior).

d. **Suspected:**
   - Inspection of log files may reveal suspicious behavior that warrant further testing, model analysis, or as check of code invariants
   - A test run/simulation run revealing an interesting situation should be further analysed by importing this scenario to the model-checker to “warm-up” the model.
   - Obtained (model) coverage is too low. Target missing coverage items with model-analyzer (either to generate a dedicated test) or to check (by formulating reachability properties) behavior “around” missing items.
   - Obtained (code) coverage is too low. Use path synthesis tool to generate missing input sequences.
   - Experiences of frequent defects

### 3.4.3 Exploiting Results of Code Analysis

a. **Success (Verified):** By assumption the requirement is satisfied. It is normally a reasonable assumption that the compiler/platform will not generate the type of runtime errors proved absent by abstract interpretation and similarly proven invariants will not be violated - provided the assumptions in input variables hold. Focus testing/model analysis on the assumptions, and less on the error classed proved absent.

b. **Failed:** The code must be corrected. Regression test must be conducted. If an invariant need to be changed, this must be propagated to the model-level, and analysis must be repeated.

c. **Inconclusive:** A static code analyzer will normally provide results within a reasonable time for the error classes it is defined to handle, unless there is an exceptional tool failure. Dedicated tests must be devised.

d. **Suspected:** Generate (additional) analysis or test cases for
   - Modules of high code complexity
   - Modules with high warning density
   - Experiences of typical defects
   - Dead-code: track origin (eliminate from model(s)). Recheck these.
   - Warnings: if a warning cannot be discharged by local inspection of the code, devise test cases that exemplifies that the indicated error does not occur (the indicated path may be infeasible, so it is not always possible).
   - It has been suggested to apply model-checking of a (narrow) model automatically constructed from the code to eliminate warnings. However, this is a very local technique and which requires tight interaction between the abstract interpreter and model-checker. The end result is a more precise analysis tool.

### 3.4.4 The In²Test approach and general concepts

The integrated inspection and testing approach In²Test [Elb, 2012] uses inspection or review results (i.e., certain metrics such as defect content or defect density) in order to focus testing activities on those parts of a system that are expected to be highly defect-prone, or on those defect-types that are expected to show up. This means, a combination of an analysis techniques (namely a “static” inspection) and testing techniques is conducted in a way where testing uses input from an earlier conducted inspection, i.e., an integration of these techniques is achieved. The prioritization is done based on (context-specific) knowledge between the inspection and testing techniques, or, if not available, based on assumptions. Figure 10 gives a rough overview of the approach.
In general, no prerequisites with respect to concrete inspection or testing techniques are made. The defect data from an inspection should be presented in a way it is suitable for conducting predictions of defect-prone parts during testing. For instance, the number of defects per module or the defect type per defect can be analyzed. If, for example, during testing equivalence partitioning should be conducted on the code level, the code is the input for deriving test cases for those parts that are prioritized based on the inspection results.

Figure 11 gives a more detailed view on this process. The defect data that is derived from the inspection is sometimes also called a defect profile. In such a profile, different representations of inspection defect data are stored, e.g., number of defects per module, defect density per module, defect severity or defect types. This information can also be considered as a model that is the input for the subsequent prioritizations. The model can also be improved by considering additional product and historical data.

For selecting / defining test cases, established techniques can be applied, i.e., the In²Test approach does not make any prerequisites here. On a unit level, this could be equivalence partitioning or boundary value analysis. All data and knowledge gathered during the application of the approach should be stored in a database and reused for future quality assurance runs.

The main remaining question is how the prioritization is done concretely. As mentioned before, such a prioritization is usually based on knowledge between the quality assurance techniques that are applied. For example, if one knows that after defects are found with the inspection in certain parts of the system under
test, testing finds more defects in these parts, the results from the inspection can be used to focus testing to those parts that were most defect-prone during the inspection (i.e., we follow a Pareto distribution of defects). Unfortunately, such knowledge is often not available for a new context. Consequently, such knowledge has to be gathered during, for example, a retrospective analysis considering historical defect data of the quality assurance techniques that should be combined. In addition, product and process data or knowledge from experts can be additional input for this step. Based on such available data, assumptions are defined and evaluated. Assumptions thereby can be defined in an analytical or an empirical manner. An example for an empirical assumption was already given be the aforementioned Pareto example, which was confirmed in many different environments. Analytically derived assumptions are defined based on logical conclusions. If, for example, certain parts are heavily checked by an inspection, the inspectors had a lot of knowledge and experience, and a certain threshold of the identified inspection defects is achieved, it might be possible to skip these parts during testing and to concentrate on different parts. However, it is important to check each assumption! Furthermore, while assumptions are often too coarse-grained to conduct the evaluation, selection rules can be defined to make them more operational. A concrete selection rule for the above mentioned Pareto assumption might be: “Focus testing on those parts where the inspection found more than 25 defects per 1000 lines of code.” Again, such a concrete selection rule has to be evaluated in the given context, and might be further adjusted.

The combination of inspection and testing in the described manner is one way how static analysis (in this case inspections) and testing can be integrated to exploit synergies, such as finding more defects, being more efficient, or increasing the coverage during the quality assurance activities. Until now, the approach was applied and evaluated on the code level, and first positive trends could be observed. However, the In2Test approach was developed to be easy applicable and to be generic. This means, we expect that it is easy to adapt certain concepts of the approach in the MBAT project, for example:

- Different quality assurance techniques might be used. For example, instead of an inspection, another static analysis technique could create a defect profile that is used for the prioritization during testing.
- The inspection or another static analysis technique does not necessarily be performed first, i.e., testing results could also drive the analysis technique.
- The approach can be applied on different levels, e.g., on an integration of system level.
- It might be possible to draw conclusions from one level to another, i.e., results from the unit level might give a support to focus quality assurance activities on, for example, an integration level.

Figure 12 presents an overview of a more generalized approach, starting with the definition of the objective of a combined application of V&V techniques and the context. Afterwards, the calibration of the approach is conducted first, i.e., the relationships between the V&V techniques are elaborated in a retrospective manner based on the available data. Finally, the V&V techniques are applied in a combined manner to exploit synergy effects.
3.5 Integration with the RTP

The main idea is to continuously trace verification status via a requirements verification matrix (RVM) that is updated as each step is executed and results are produced (section 6.2 discusses RVM in more detail). An initial RVM is made during V&V planning. However this is often quite high level and reflects only an initial estimate. As mentioned, a step checks a requirement (or collection thereof) using some tool/technique and model under given explicit assumptions and produces verification results of the requirements. However in this process new verification obligations may be generated, new A&T cases may be identified, and the model might be updated. Information about these must be maintained via the RVM maintained in the RTP. Similarly, the identification of a defect may also spawn off a new set of A&T cases that targets a suspicious
code/model block, functionality etc., which are then added to the information base. This is illustrated in Figure 13.

Figure 13: Organization of A&T Steps and status propagation via RTP
4 Guidelines for choosing and using techniques

The purpose of this chapter is to give overall guidelines about the most important automated analysis and testing techniques available and how to put these into good practical use.

4.1 Guidelines about techniques

All of the automated V&V techniques have different basic capabilities and purposes, and have complementary strengths and weaknesses and it is essential to understand these to be able to professionally use them, and to select the most appropriate one.

4.1.1 Classification of Techniques

Figure 14 provides a classification of the most important techniques for MBAT. **Static analysis** broadly covers a wide range of algorithmic analysis techniques for evaluating computer software and artifacts without executing them. **Dynamic analysis** aims at evaluating the behavior of computer software and systems by executing and observing them.

![Figure 14: Classification of Main (automated) V&V techniques](image)

Testing, Monitoring, and Simulation are dynamic execution based techniques, whereas Model-checking, Refinement-checking, Abstract interpretation, and Theorem-proving are static techniques that through algorithmic analysis checks the requested properties. The lowest level in Figure 14 depicts some common hybrid of the indicated basic techniques.
Figure 15: Illustration of Important MBAT Techniques

Figure 15 graphically depicts the main purpose of important basic techniques:

**Model-checking:** Algorithmically checks logical properties of a (formal) behavioral model.

**Simulation:** Based on sample executions of the model, selected properties can be checked, or parameters/parameter dependencies be computed. Simulation is included as a separate main technique because of its importance for executing models, and as an under-approximation mode for model-checkers; further testing techniques are often applied to model simulations (model-in-the-loop testing).

**Testing:** In model-based testing, the aim is to check by execution that the behavior of implementation conforms to that prescribed by the specification.

**Abstract interpretation:** Algorithmically computes an abstraction of the input artifact (normally program code) and checks (typically) generic or invariant properties of that abstraction.

To help choose the right technique (and tool based on this technique), it is necessary to have a fundamental understanding of the capabilities and limitation of these. Therefore, in the next sections we provide a more in-depth characterization of the techniques according to the following elements:

- Definition of the technique
- Position in development cycle under typical use.
- Main supported abstraction level (system, control, code)
- Support for quantitative/qualitative aspects
- Characteristics, completeness/approximation, advantages and disadvantages.

### 4.1.2 Model-Checking

**Definition:** Model-checking is an automated technique that, given a finite state model (or one that can be reduced to finite state via abstraction) of a system and a formal property, exhaustively checks whether this property holds for (a given state in) that model [JPK, 2008]. Normally, the property is given in a formal logic like CTL (computation tree logic) or LTL (linear temporal logic), but may also be served in other specification languages, like (live) sequence charts. Model checking has been successfully used in the context of reactive systems, and in particular hardware design. In addition, model checking techniques have been also developed for more complex software systems, real-time and probabilistic embedded systems.

**Exactness:** Model-checking aims at giving an exact yes/no answer about the property by (potentially) exhaustively searching the state-space, but it may also allow over- or under-approximation techniques to
help analysis of larger models, when the state-space is too big for exact analysis. Such approximative analysis may sometimes result in "property may be satisfied" answers.

**Position:** Model-checking can be applied early in the development cycle where models are built prior to system implementation. Here it is typically used to validate the consistency of requirements, check properties of an analysis model, explore properties of proposed designs (design-space exploration), or check behavioral aspects of a concrete design. Model-checking also has significant applications in later stages. During system integration it can be used to check that a component (or set of components) that has an interaction pattern that was found more complex or difficult to implement than expected functions correctly. In particular, model-checking has proven effective as a defect localization and correction technique: a model is made to capture the suspected behavior and then all corner cases are explored, and when a defect has been identified to check that a proposed correction works in all situations.

Model-checking is normally applied to the system or sub-system levels, and typically to explore whether the interaction of a set of components satisfy functional and quantitative properties. Software model-checkers operating at the component/implementation level also exists.

**Properties:** Model-checking considers behavioral models of finite state systems (or systems that can be reduced to finite state via abstraction), often given as extended finite state machines in a suitable notation (e.g. UML state charts). Such models capture functional behavior, but tools supporting extended formalisms and notations capturing resource usage, timing, and stochastic phenomena exist as well.

**Discussion:** A pre-requisite for model-checking is a model – manually created at the right level of abstraction. Making good (formal) models that can be effectively analyzed by a model-checker is not trivial, and requires both expertise in the problem domain and modeling notation and tool insights. It should be remarked that model-checking is only complete with respect to the queries posed to it – it does not ensure that the right questions are asked or that the queries are formulated to capture the intended requirement. Also the validity of the produced results depends on appropriateness of the model (as is true for any prediction made from models).

**Refinement checking.** Refinement checking can be viewed as a special case of model-checking where the objective is to check automatically whether (the behavior of) one finite state model (the "implementation") refines the behavior of a more abstract finite state model ("the specification"). Applied repeatedly ("step-wise refinement"), this procedure may result in models that are sufficiently concrete and easy to map to code, where each step is formally proven correct.

### 4.1.3 Abstract Interpretation.

**Definition:** Abstract interpretation is a method and theory for (automatically) creating mathematically sound abstractions of (the semantics of) a program, and use this abstraction to infer properties about the dynamic behavior of the program. A text book example of an abstraction over an integer variable (concrete domain) is the two valued abstract domain of positive and negative natural numbers; this information could be used to show that a C-function could possibly return a negative value. The mapping of the concrete domain to the abstract domains is called an abstraction function. Another example of an abstraction is the worst case execution time of a C-function; here only information about its worst case execution time is preserved. Abstract interpretation use (over-) approximate algorithms to create effective tools for automatic verification of complex (or otherwise undecidable) properties of critical embedded systems.

**Position:** Whilst abstract interpretation in principle can be applied to models at various levels of abstraction and design stages, it is normally applied to developed source code, i.e. medium to late in the development process during component programming or component integration. Slightly earlier application is also possible when model-based source-code generators are used to generate code from (implementation) models. Abstract interpretation techniques implemented by compilers or external analysis tools has numerous applications ranging from program optimization to defect detection (e.g., uninitialized data variables, out-of-bounds array access, null pointer dereference, divide by zero, race conditions, illegal type conversion, memory leaks, unintentional non-terminating loops), synthesis of program invariants and WCET analysis.
**Properties:** Functional properties like race conditions, deadlock, floating point stability, loop bounds and invariants/assertions. Quantitative properties like WCET, maximum stack usage.

**Exactness:** A main challenge of program analysis is that algorithmic computation of many interesting abstractions is undecidable (e.g., the halting problem). Therefore the techniques typically use over-approximations of the program semantics. In effect, the techniques may report false positives (problems that are not really present). The more precise the approximation is required to be the more costly in terms of computation time and space it typically is.

**Discussion:** Abstract interpretation tools can more easily be integrated into existing development processes as it doesn’t introduce new artifacts, and few new concepts that require training. In practice some training and insights are necessary to interpret the warnings produced by these tools, and especially to investigate false positives. Most often abstract interpretation based tools targets generic properties like absence of certain classes of runtime errors. Some tools also allow invariants and post conditions to be checked, but these features seem less frequently used. Hence, correct functionality and reactions must be checked by other techniques.

Abstract interpretation is usually applied to the code level, but the technique can in principle equally well be applied to models.

**Symbolic execution.** Symbolic execution is a kind of abstract interpretation where the variables and numerical quantities in the program/model are represented by symbols, and where expressions are manipulated symbolically, typically by defining and solving constraint systems.

**Software model-checking:** The source code (given a formal semantics) is viewed as a finite state model that can be explored state-by-state. However, such a direct interpretation usually gives a very large state-space, and abstract interpretation, program slicing, and symbolic execution techniques are applied to achieve a feasible finite state abstraction.

**4.1.4 Theorem proving**

Theorem proving is a technique that can be used to formally prove correctness of programs in a way similar to making a mathematical proof of the correctness of a theorem. It is mainly applied manually, but may be supported through theorem proving assistant tools.

**4.1.5 Review and Inspection**


**4.1.6 Model-based Testing**

**Definition:** Testing is the execution of a (software) system under well-defined conditions (predefined environment/input sequences) and checking whether the observed behavior deviates from the specified behavior. In model-based testing tools assists the generation of test cases (inputs plus oracles) - normally from high-level behavioral models.

**Position:** Testing is normally applied late in the development process. At least test execution and getting verdicts requires an executable artifact to be tested. This artifact may in be a (executable) model, thus allowing testing to be used as an approximate verification technique (like simulation), but the main usage is application to actual implementations. Testing is applicable to all levels: unit/component, subsystem integration, and system level.
Properties: Model-based testing mostly supports functional properties in terms of correct i/o reactions. Some techniques support real-time properties and stochastic properties like use-profile testing.

Exactness: Applied systematically, testing gives indispensable insights into the behavior of the actual implemented (software) system under test. The strength of testing is thus that it checks the actual executing system in a realistic environment. On the other hand it must be stressed that a fundamental limitation of testing is that only a very small sample of the possible system behaviors can be evaluated. The required number of test cases needed for exhaustive (in the sense that a passing system is guaranteed to be correct) testing is practically infinite (“testing can only show the presence of defects, not their absence”\(^5\)). Thus given that only extremely few executions (compared to the full behavior) can be checked, testing results in a gross under-approximation of the total system behavior. Coverage criteria are heuristics used for test-selection.

Discussion: Model-based testing assumes a high-level behavioral model of the system to be tested. Since test cases (inputs and oracles) are generated from this model, the resulting test cases are only as good as the source model. Hence it is essentially that a test model captures the requirements to be tested and that this model is made at the right level of abstraction, that includes details that are necessary for test generation and the oracle, but does not include too many irrelevant implementation details (i.e., it should specify what behavior is to be tested, not how this is to be realized).

Model-checking can be employed to help ensuring that a test model captures the intended and correct behavior.

Certification: since it is the actual system that must be certified, testing is an essential part of any certification procedure.

4.1.6.1 MiL, SiL, PiL

One major benefit of model-based development is the possibility to “do things as early as possible”. In terms of testing this means to test the functionality in the model before the software is implemented and integrated into the final ECU (electronic control unit). Between the initial modeling and the integration into the ECU there is several intermediate integration levels described below. Since the functionality of the system should remain constant and independent of the integration level, relevant test cases should also be constant throughout the integration and implementation. In order to maximize reuse, a test procedure for model-based developments should support the portability or reuse of test cases between the various platforms. On the one hand this reduces the effort of test case design tremendously and, on the other hand, allows for easy comparison of test results between the different integration levels which may be tested on different systems. In addition sharing test cases between different levels also means that test cases can be expressed in a common notation. Test cases can be corrected or extended in a central test model without the need to adjust a lot of different test implementations for the different integration levels. Although this requirement sounds trivial from a theoretical point of view it is a weak point in today’s testing practice for model-based development because test procedures and test languages are usually specialized for one particular test platform and are very difficult to share with other test platforms. In general, the following integration levels are distinguished:

Model-in-the-Loop (MiL): The first integration level is based on the model of the system itself. Testing an embedded system on MiL level means that the model and its environment are simulated (interpreted) in the modeling framework without any physical hardware components. This allows testing at early stages of the development cycle. MiL is an inexpensive way to test embedded systems.

In most automotive model-based development projects there are different kinds of models. Functional models are rather abstract and do not consider all aspects such as robustness and performance. In the course of the development functional models are transformed into implementation models. Implementation models are often used together with a code generator to automatically derive production code from the system models. Development and simulation environments for model-based development are, for example, MATLAB / Simulink or ASCET. Both functional and implementation models can be tested. Functional models are tested on a system level (system test). When testing implementation models for complex systems it

\(^5\) Static analysis techniques only prove the absence of the properties they are explicitly asked to check, but are then exhaustive in the answers.
makes sense to distinguish between module tests (testing subsystems of the model that handle particular functional areas) and system tests.

In subsequent stages of development of software then in the Loop (SIL), Processor in the Loop (PIL) is spoken.

Software-in-the-Loop (SiL): Testing an embedded system on SiL level means that the embedded software is tested within a simulated environment model but without any hardware (i.e. no mechanical or hydraulic components, no sensors, actuators). Usually the embedded software and the simulated environment model run on the same machine. Since the environment is virtual, a real-time environment is not necessary. Usually SiL tests are performed on Windows- or Linux-based desktop machines.

Processor-in-the-Loop (PiL): Embedded controllers are integrated in embedded devices with proprietary hardware (ECU). Testing on PiL level is similar to SiL tests, but the embedded software runs on a target board with the target processor or on a target processor emulator. Tests on PiL level are important because they can reveal faults that are caused by the target compiler or by the processor architecture. It is the last integration level which allows debugging during tests in a cheap and manageable way. Therefore the effort spent by PiL testing is worthwhile in almost all cases.

### 4.1.6.2 HiL “Simulation”

Hardware-in-the-loop (HIL) simulation has the same principle as MiL, SiL, PiL: The embedded system is simulated together with a model that depicts the environment of the system.

Hardware-in-the-loop (HIL) simulation is a technique that is used in the development and test of complex real-time embedded systems. When testing the embedded system on HiL level the software runs on the final ECU. However the environment around the ECU is still a simulated one. ECU and environment interact via the digital and analog electrical connectors of the ECU. HiL simulation provides an effective platform by adding the complexity of the plant under control to the test platform. The complexity of the plant under control is included in test and development by adding a mathematical representation of all related dynamic systems. These mathematical representations are referred to as the “plant simulation”. The embedded system to be tested interacts with this plant simulation.

### 4.1.7 Monitoring (passive testing)

**Definition:** Monitoring is (normally) an automated technique that consists of observing and evaluating the execution of a system in its operational environment. The oracle used for evaluating the observed behavior may be specified more or less specifically and more or less formally. It is often referred to as “passive testing” because the tester does not actively control or stimulate the system under test. See also Runtime Verification.

**Position:** Late to very late: at integration level, system level testing, and even after deployment.

**Properties:** Both functional and non-functional, but only if observable.

**Exactness:** Only observable behavior can be checked – either observed in a test environment or in its actual operating environment.

**Discussion:** Monitoring as several important and easy to use applications, e.g., requirements can be turned into observer automata that can be monitored at several levels of abstraction, and deployed at various development stages.

**Runtime Verification:** Runtime verification is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property [Leucker, 2009]. An example is checking whether an observed execution of a system is formally accepted (or rejected) by an automaton (may be generated from a logical property). This check is often made by instrumenting the system to detect and generate observations/events at runtime and to feed these to the generated monitor. It is thus related to both testing and formal property checking.
4.1.8 Simulation

**Definition:** Simulation is the execution of models of (software) systems under pre-defined with the intent of checking its dynamic behavior. Simulation is therefore similar to testing, but applied to models.

**Position:** Typically applied at the design level to explore different proposed designs. Thus it is typically applied relatively early and before coding starts. But it can also be used to fine tune parameters of existing implementations.

**Properties:** Most functional and several quantitative properties (response times, resource usage, reliability).

**Exactness:** Execution may be deterministic or randomized, dependent on the model/simulator tool. Given that relatively few executions (compared to the full behavior) can be checked, simulation results in a gross under-approximation of the total system behavior. Simulation is not well suited for rare capturing corner cases where ugly bugs may lie.

**Discussion:** Simulation can (and is) be applied to formal models and in a rigid manner, and can be useful for checking properties of complex, large, and feature-wise rich models that cannot be fully formally analyzed. Unfortunately simulations has a bad reputation because it is often applied in a highly ad-hoc manner; the models are created ad-hoc, implemented directly in code or informal simulation languages, no accurate (or formal) semantics of the models or their interpretation are given, scenarios/conditions not systematically explored, and results are derived by manual mental inspection of the executions.

**Statistical Model-Checking.** Statistical Model-Checking uses a precise stochastic semantics of a finite state model to perform numerous simulation-runs in order to obtain a statistically accurate sampling of the model to settle formal (probabilistic) properties to a given (desired) level of significance. Statistical model-checking thus constitute a bridge between simulation and model-checking techniques. The main applications are under-approximate analysis of larger models, and especially performance analysis. Given that it is simulation based does not require the state-space to be exhaustively explored and stored, it allows larger model instances with more concurrent components, and feature-wise richer (hybrid, stochastic, priced, liberal types/expressions) models than can normally be fully analyzed through model-checking. Given the approximate and stochastic nature it is particularly suited for performance analysis.

4.1.9 Debugging

Debugging is an activity carried out by developers to determine the cause of a deviation of actual system behavior from its specified behavior.

4.1.10 Risk Analysis Methods

Risk analysis aims at assessing the potential of a system for causing severe hazard to its environment. Developed initially for assessing large industrial units such as chemical plants, they also try to identify existing risk reduction measures. Later, several have been extended to deal with electronic systems and software, and some methods can also be applied to models. Due to their main goal, they can be used for identifying critical and weak points in a system or design and therefore can also help to focus other analysis and test methods.

The following Table 1 summarizes main properties for the most commonly used techniques. See [AIChE_CCPS, 1992] for a general introduction; for HAZOP see [EC-61882, 2001]; for FMECA see [MIL-STD-1629A, 1980] and [ECSS-Q-30-02B]; and for FTA see [IEC 61025, 2006] and [ECSS-Q-40-12A, 1997].
Table 1: Overview of risk analysis methods

<table>
<thead>
<tr>
<th>Definition</th>
<th>Purpose</th>
<th>Position in Dev.proc.</th>
<th>Quant./Qual. aspects</th>
<th>Techniques used</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHA – Preliminary Hazard Analysis</td>
<td>• Identification of (all) top level hazards • their potential reasons • possible counter measures</td>
<td>Early stages</td>
<td>semi-quantitative</td>
<td>Team with experts fills out table with columns like • Hazard type (e.g. fire) • Accidental event (e.g. leaking of inflammable liquid) • Probability causes (e.g. damaged valve) • Preventive actions • Probability / Severity</td>
<td>Views system as black box Human experts work Little automation possible?</td>
</tr>
<tr>
<td>HAZOP – Hazard and Operability Analysis</td>
<td>• Identification of hazards caused by deviation from normal operation • their potential reasons • possible counter measures</td>
<td>Concept … late design (Needs least some details of design need to be known)</td>
<td>qualitative</td>
<td>Table-based • Identify components • Identify their i/o and control parameters • Identify potential causes • Identify means to detect it • Specify safe guards, if necessary</td>
<td>Very systematic Within limits, can be automated (though expertise knowledge is important) Variants: – Function – Process – Software Distinction: mainly the guide words</td>
</tr>
<tr>
<td>FMEA – Failure Modes and Effects Analysis</td>
<td>• Identification of potential failure modes, • determine their effect on the operations of the product, • identify actions to mitigate the failures</td>
<td>Usually concept … late design (If it cannot use experiences from early developments, the system architecture should be known at rather detailed level)</td>
<td>quantitative – if known component reliabilities are used qualitative – if no reliability figures used</td>
<td>Table-based • Identify component (and operating modes) • Identify their potential failures (in operating modes) – together with reason and detection means • Identify consequences on subsystem where it is part of the system • If quantitative: use probabilities for computing resulting system failure probability • of course, identify risk reducing measures</td>
<td>Mainly Bottom-up Prerequisites: system structure, failure probabilities of bottom level components Top-down: if only required functions known, but not system structure: decompose function recursively down to elementary items Presumably hard to automate</td>
</tr>
<tr>
<td>FMECA – Failure Modes, Effects, and Criticality Analysis</td>
<td>Like FMEA</td>
<td>Like FMEA</td>
<td>quantitative</td>
<td>Like FMEA</td>
<td>Difference: charts the probability of failure modes against the severity of their consequences. The result highlights failure modes with relatively high probability and severity of consequences, allowing remedial actions.</td>
</tr>
</tbody>
</table>
FTA - Fault Tree Analysis

Find all potential causes for one particular accident or main system failure ("top event"). It uses Boolean logic to combine a series of lower-level events which, in combination, cause the top event.

Should be used in concept and design phase, but can also be used after an accident to analyze potential causes qualitatively, but can be used to quantitatively determine the probability of a safety hazard.

- Define top event as root of a tree.
- Add leaves in Boolean NOT|AND|OR|XOR... combination that can lead to the top node.
- Apply this process iteratively until "(atomic) bottom events" encountered.
- If probabilities added: probability of top event can be calculated

"Minimal cut sets MCS": sets of bottom events that each individually can cause the top event.

- Size: number of events in MCS (1 – a single event, 2 – a pair of events etc.)

Top down
Can be used to scrutinize a hazard identified with some of the above techniques.
Can be – and is already – automated, also on models.

The table addresses the most common risk analysis methods, more examples are

- ETA (Event Tree Analysis)
- check list analysis
- what-if analysis
- double-failure analysis
- HRA = Human Reliability Analysis

Besides FTA, all listed techniques can be carried out with simple table-supporting tools like Excel or even Word. But several commercial tool(s) exist that provide more method-specific support, in particular for FTA. Examples are the ITEM toolkit von Itemsoft, Byteworx FMEA, Isograph Reliability Workbench (Isograph), Reliasoft XFMEA.

Level: In general, all risk analysis methods are applied at system level, but could also be applied to components. Special variants, such as Software HAZOP, can be applied to implementation level.

4.1.11 Overall Comparison

First we make a high-level overall comparison of test automation with static analysis according to their cost, coverage, applicability, and debugging. The comparison with advantages and disadvantages are summarized in Table 2.
## Table 2: Comparison of Testing and Static Analysis

<table>
<thead>
<tr>
<th>Test Automation</th>
<th>Static Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cost</strong></td>
<td></td>
</tr>
<tr>
<td>Expensive to set up and run</td>
<td>Low cost set up and run</td>
</tr>
<tr>
<td>• Substantial effort to define test cases and implement test benches</td>
<td>• No construction of test bench necessary</td>
</tr>
<tr>
<td>• Static code analysis may start as soon as code fragments are available</td>
<td>• Model-analysis: requires a model, but normally cheaper than test scripting due to less script maintenance</td>
</tr>
<tr>
<td><strong>Coverage</strong></td>
<td></td>
</tr>
<tr>
<td>Inherent coverage holes:</td>
<td>100% Coverage</td>
</tr>
<tr>
<td>• Even with test automation, only a modest number of test case / input vectors can be selected and executed</td>
<td>• Exhaustive verification of all possible cases, vectors, and system behaviors</td>
</tr>
<tr>
<td>• Coverage criteria weakly linked to behavior and requirements</td>
<td>• Corner cases covered without extra effort</td>
</tr>
<tr>
<td>• High effort to cover corner cases</td>
<td></td>
</tr>
<tr>
<td><strong>Applicability</strong></td>
<td></td>
</tr>
<tr>
<td>General applicability:</td>
<td>More restricted</td>
</tr>
<tr>
<td>• Almost no complexity/scalability issues regarding size of DUT (MBT have limits)</td>
<td>• Complexity and scalability problems</td>
</tr>
<tr>
<td>• Also applicable on system level</td>
<td>• Mostly applicable to component / sub-system level</td>
</tr>
<tr>
<td>• Every observed trace shows an actual system under test behavior</td>
<td>• Trace may be spurious (false-positive, unreachable counter example)</td>
</tr>
<tr>
<td>• Blackbox test less sensitive to code change</td>
<td>• Code analysis is sensitive to code change</td>
</tr>
<tr>
<td><strong>Debugging</strong></td>
<td></td>
</tr>
<tr>
<td>High debug effort</td>
<td>Lower debug overhead</td>
</tr>
<tr>
<td>• Possibly long trace/log files need inspection</td>
<td>• Short counter example</td>
</tr>
<tr>
<td>• Animation of counter examples typically supported</td>
<td>• But effort may be needed to check warnings for possible spurious traces</td>
</tr>
</tbody>
</table>

Next we provide a more detailed comparison and discussion of advantages and disadvantages, see Figure 16. Some remarks are

- The need for models can be considered both a disadvantage and an advantage. The disadvantage is that they have to be constructed, often manually. Engineers (and project managers) are typically reluctant to make them unless they give direct benefits and do not appear as ad additional work items. Making good models require expertise and training in the same way as writing good and efficient programs. The advantage of constructing modes is that a lot of unclear points and interesting questions about the problem become evident during the modeling process enabling early resolution of these. When the model exists automated design and verification tools may exploit it.

- The results computed from models are only as good as the models and questions asked about them. This applies both to model analysis and model-based test case generation.

- In contrast to model-checking, simulation allows checking of very large and rich models, but at a cost of precision. Simulation also has a bad reputation, because it is often ad-hoc, both in terms of understanding the model (what is simulated) and the behavior of the simulator (no formal semantics) "the semantics of the model is what a specific simulator tool does with the model". The ad-hoc use is not an inherent feature of simulation, and careful application can produce good results.

- When abstract interpretation is used to detect generic runtime errors, the actual functionality is not checked. Hence, it cannot stand alone and must be complemented by testing and model-checking.

- Testing and simulation are both dynamic techniques. We keep them distinct, as testing can target a real physical integrated system under test (available very late in the development process);
simulation can only deal with models (typically used earlier - to help produce the final system). Testing may also reveal emerging behavior; models can only show the behavior built into them.

Figure 16: Comparison of main A&T Techniques

Details of how to apply the individual techniques may be found in Chapter 7.

4.2 Best Practices

Guiding Principles and Best Practices

1. Model and analyze early
2. Design for verification and testability
3. Joint Input Constraints and Requirements
4. Make Joint Verification Planning
5. Perform (early) static code analysis using static code analyzers
6. Analysis first, then test (test what cannot be analyzed)
7. Prioritize effort

The following best practices are formulated on the basis of the following references: [Boulanger(a), 2012], [Foster, 2006], [Perry, 2005], [Lam, 2005], and [Drechsler, 2004].

4.2.1 Model and analyze early

It is well-known that identifying and correcting defects is much cheaper in the earlier stages than in the later stages of systems development. In particular, when found during system and acceptance testing, the costs are extremely high due to the amount of rework and risks for delayed time-to-market increase. Using models early is a great vehicle for checking that requirements are consistent and understood correctly, for asking
nasty questions about possible (mis-)interpretations of the requirements (thus also a good source for test and verification cases). Critical or complex functionalities and behaviors can also be modeled and (potentially exhaustively to cover the critical corner cases) analyzed before design and coding.

4.2.2 Design for verification and testability

A key to successful development as well as completion of the corresponding V&V tasks is that the system is comprehensible. With a large monolithic and complex design it becomes impossible to isolate components/sub-systems for testing or analysis (because their boundaries and intended function cannot be identified), and verification can only be done on large complex subsystems, and coverage and thoroughness becomes low. With simpler independent components, verifying them and their interaction also becomes simpler and has a higher likelihood of being performed thoroughly and correctly. Therefore the design should be made with verification tasks in mind. The design and implementation should respect the following guidelines (that also enhance maintainability):

- Enforce low complexity,
- Use small interfaces
- Use encapsulation, and components with (high) cohesion and (low) coupling
- Consider adding test and verification support e.g., by extra functions in the API for controlling and observing the component.
- Use well-structured code (and respect coding guidelines needed by static code analyzers?).
- Use guidelines and best practices for making analyzable models

4.2.3 Joint Formal Input Constraints and Requirements

Formal input constraints and formal requirements are a prerequisite for both model-based analysis and testing.

Input constraints capture how the system or component (at any level) under V&V can be stimulated by its environment (other hardware and software components, physical plant, users, external devices etc.). This includes signal types, ranges of input variables and parameters, behaviors and scenarios, load and load patterns. It is normally not reasonable to test or verify system behavior with arbitrary input behavior, but only those that legally (or under reasonable assumptions) can occur. Some care is needed not to rule out behavior that under unexpected conditions can occur.

These joint input constraints should be formulated in a formal language (e.g., as common environment models), and preferable the same language for both testing and analysis.

Similarly, joint formalized requirements are needed for combined model-based analysis and testing, and should preferably be specified in a formalism that is compatible or interoperable with the input constraints (environment models). There exist many approaches for systematic development of (formal) requirements:

- Designer requirements (white box, implementation-level, simple, early)
- Protocol requirements (between modules)
- Functional requirements (module or system-level)
- Integration requirements (sub-system and system-level)

4.2.4 Make Joint V&V Planning

A joint V&V plan considers that both analysis and testing activities are being applied. The joint plan should comprise input constraints (environment assumptions), formal requirements, and verification status (results of analysis or testing activities). In general, use (according to existing guidelines) whatever analysis or
testing technique delivers faster/better results for requirement at hand (general guideline: “formal first”) – see also the integration levels identified in clause 2.3.4.

The testing and analysis results should be recorded in the V&V plan to track and understand (requirements) coverage of the verification plan.

Use coverage of the items in the verification plan as sign off criterion for verification (in combination with code coverage information from dynamic verification and verification plan reviews for completeness of requirements)

- Cf. recommendation in DO178c for combination of static and dynamic techniques
- Verification planning tools: Rational Quality Manager, HP: Quality Center
- Note: combined coverage metrics not solved yet (cf. activities in Accellera “Unified Coverage Interoperability Standard”, v1 released June 2012)

4.2.5 Perform (early) static code analysis

Static code analyzers are able to perform a number of advanced “Push-button” checks of source code with very little set-up overhead, and do so as soon as code (or fragments) is available. Finding certain errors in the code and improvement of the code is therefore possible early without waiting for test cases and test benches. It is a reasonable hypothesis that less testing for these error classes is required. Some effort is required to analyze the results of the analyzer (identify “false warnings”). The Push-button checks that can be made without formulating (hard to learn) properties include the following generic error classes:

- Array boundary checks
- Value range violation
- Race conditions
- Division by zero
- Value under/overflow
- Etc.

4.2.6 Analysis first, then test (test what cannot be analyzed)

Analysis tools aims by exhaustive analysis at proving the absence of certain error classes, or prove satisfaction of user defined properties, is therefore more reliable/trustworthy than testing with respect to the formulated properties. Analysis results are typically available earlier, and analysis in principle possible as soon as a model or code is available. It is therefore a reasonable hypothesis that less effort needs to be spent on testing those properties. Testing can thus focus on what cannot be verified

- Special extra-functional behaviors
- Requirements that could not be checked by analysis
- Validation / qualification
- System integration

4.2.7 Prioritize effort

It is clear that not every requirement or system property can be verified/validated equally thoroughly due to time and resource constraints. In general, the prioritization can be based on experiences (Elberzhager), complexity, or risk analysis. Since some analysis techniques require special expertise and effort it should not necessarily be applied to very verification problem. However, it is in particular recommended to use analysis for problems with a high degree of (control) complexity, or complex interactions (e.g., due to high degree of concurrency) or protocols.
5 Examples

The purpose of this chapter is to illustrate central concepts of the MBAT combined method. The examples are not intended as full case studies applying the proposed methodology. A systematic evaluation of the method is a future task. The examples were chosen because they at the time of writing were advanced in their proposals for combined analysis and testing. In the second version of this deliverable D2.1_2_2, we try to gather more and further detailed examples.

5.1 Rockwell Collins “Flight Guidance System Mode Logic”

The following example is based on the Rockwell Collins “Flight Guidance System Mode Logic” use case UC AE7 description for the MBAT RTPv0 integration scenarios, and as such neither represents actual production software nor how actual software is actually developed at that company. But it does represent a problem typical for that domain. This piece of software determines which control law is applied at any moment of the flight, and is therefore highly safety critical [MBAT1, MBAT2]:

The Flight Guidance System (FGS) is a component of the overall Flight Control System (FCS) used in regional jet aircrafts. It compares the measured state of the aircraft to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. The FGS has two physical sides (one on the left side and one on the right side of the aircraft) that communicate over a bus. Each side can be further broken down into the mode logic and the flight control laws. The latter compute the pitch and roll guidance commands, while the former determines which flight control laws are active and armed by selecting lateral and vertical flight modes.

To ensure that meaningful guidance is provided to the flight director and autopilot, only one lateral and one vertical mode can be active at any time. For the same reason, if the autopilot is engaged or the flight director is turned on, at least one lateral and one vertical mode must be active. Other constraints enforce sequencing of modes that are dictated by the characteristics of the aircraft and the airspace.

5.1.1 Proposed Workflow

A main objective is to use static analysis techniques and avoid redundant testing using the following tools: Simulink Real-Time Workbench (RTW) for code generation, an internal model-checker for analysis of Simulink Stateflow models, MaTeLo for model-based testing, and Astrée for static code analysis wrt. runtime errors.

The workflow envisioned by Rockwell Collins in [MBAT2] is (also reproduced in Figure 17):

It starts from the system requirements specification given in the Requirement Specification Language (RSML). From this document, a set of formal properties is derived, as well as a Simulink model, and a test model. The formal properties are checked on the Simulink model, which can have two outcomes:

- either a formal property is proven on the Simulink model, in which case it does not need to be tested on the model (however it might need to be tested on the code),
- or it cannot be proven, in which case tests need to be generated to be run on the Simulink model and later on the code.

Then, tests are generated from the test model, and applied to the Simulink model (where verification of the requirements are covered by proven properties is omitted). As a last step, implementation code is generated from the Simulink model, and the generated tests are applied on the code. Finally, the absence of runtime errors is proven on the code.
From this description it is easy to identify the individual A&T steps as shown in green in Figure 18:

Step 1 is model-checking of the formalized properties (derived from the requirements) on the design model. Remark that this is a low-level model in that it is sufficiently concrete and rich to enable executable code to be generated.
Step 2 is model-based testing threatening the design-model as system-under-test focusing at the properties/requirements that could not be model-checked.

Step 3 is application of the static code analyser to check for absence of certain generic runtime error classes (array indexing, division by zero, indirection of NULL pointers,..). It this case this step is necessary because these error classes cannot be detected at the model-level, nor checked/ensured by the RTW compiler.

Step 4 is the application of model-based testing of the generated C-code as system under test aimed at checking that the code has the correct functionality and correctly implements the (safety) requirements. At the surface, this step appears redundant because the functionality has already been checked on the design model using model-checking or MBT. However, Rockwell Collins may have reasons for executing this quite resource consuming step. Firstly, it is safety critical software, and development standards of this domain require that the implemented system is qualified for flight by testing (validation). Secondly, the RTW compiler is not trusted to generate correct code either due to errors in the compiler, used libraries, or due to the imprecise semantics of the Stateflow language which may cause a discrepancy gap between the interpretation used for analysis and that used for code-generation. Finally, the target system may use legacy or 3rd party components or libraries.

By using the idea of establishing a chain of arguments, and inspecting the assumptions behind each step it becomes evident that the proposed workflow is not optimal. One observation is that the model used for MBT is handcrafted, and is not linked to the requirements and checked against these. Using MBT this has the particular negative effect that it is not known whether the generated tests really check the right behaviour. Using MBT, “what-you-model-is-what-you-test”, and the quality of the resulting test cases is only as good as the basis from which it is (automatically) derived. Therefore, it is well warranted to analyse the test-model, as suggested using orange color in Figure 18.

Another alternative step could be to add a requirements model derived from the requirements that is then analysed. There is much evidence suggesting that such a high-level model both greatly helps in understanding the implications of the requirements, and in obtaining a complete and consistent set of requirements. From this requirement model a test-model and more detailed design model can be derived (perhaps using automated refinement checks).

### 5.1.2 A Model-checking Step

An example of a (hypothetical) model-checking step is illustrated in Figure 19.

**Purpose:** The purpose of this step is to analyze by model-checking that the FCS chooses the right control mode, i.e., the requirements that it always uses the right mode under the right condition, and that no illegal jumps between certain modes can take place. The purpose influences both the properties to be checked, but also the model to be analyzed: it must capture the right aspects, and may possibly require some annotations to make the check technically work.

**Verification Cases:** To really verify this requirement require a number of situations to be analyzed. These must be specified so accurately that they can be turned into properties that can be formally analyzed by the model checker. Example cases are:

- It must be possible to engage each specified mode,
- It always uses a valid mode
- Whenever mode “x” is engaged the conditions for “x” are true,
- Whenever the conditions for “x” are satisfied, mode “x” is selected,
- Mode switch time takes place within a given time bound.
- Etc.

**The A&T technique:** An internal model checking tool for Simulink Stateflow models.
Figure 19: Example of a model-checking step (hypothetical).

Make V&V Object: The object under analysis is a Simulink Stateflow design model of the FCS. If not already made, it must be constructed. If it is available it may need to be adapted for model-checking (finite domains, simplify environment model or complex components, add annotations, etc.). Moreover, the properties to be checked (derived from the set of analysis cases) must be formulated in an input language acceptable by the model-checker. In this case, these are illustrated as Uppaal-TCTL (Timed Computation Tree Logic) properties (for many industrial users it may be easier to express the properties as Live Sequence Charts or observer automata, if supported by the model-checker). See also Section 7.1.

Assumptions: For a FCS for a specific aircraft there a number of such parameters about the dynamics of the physical environment like max assumed airspeed and acceleration capabilities (it is not clear from the use case description if the specific parameters are needed at this abstraction level.) Also assumptions include over-approximations of the behavior of other components that interface to/stimulate the FCS. Similarly, the design itself makes use of components that are assumed to behave according to a given specification.

Run A&T: Running the analysis consists of executing the analysis procedure by executing the model-checker with the given model and properties to be checked. After some (potentially significant) amount of time the results are produced. Depending on the capabilities of the model-checker the properties can be batched and re-use the computed state space.

Evaluate: In the evaluation phase, the results are examined to check that the results corresponds to expectations. If not, counter-examples produced by the model-checker must be examined (through animation if supported by the model-checker). If results are inconclusive or suspected there may be several explanations:

- An error has been detected
- The modeled system or the formulated properties are inaccurate or wrong
- The modeled assumptions are too weak
- The model may be too large/complex for exhaustive analysis

In the last three cases iteration of the step may be necessary with local corrections.

Define new V&V obligations: New cases for test and analysis may be identified though a model-checking step:
• If an error is detected and corrected, the code, test-cases and other artefacts (manually or automatically) produced from this model must also be revised.
• Similarly, if other model corrections are made that are considered sufficiently significant to impact other artifacts. Also, the validity of assumptions added or strengthened must be verified (likely in another step).
• Some aspects of a requirement (identified as an analysis cases intended to be checked in this step) cannot be checked on the model because it does not capture the necessary behavior, or is insufficiently reach to capture the needed extra-functional annotations).
• Finally, if no results for a given property can be established by the model-checker, it may be necessary to simplify the model, by e.g., replacing the behavior of a complex subsystem with a simpler and more abstract version. The validity of this abstraction must be checked. Alternatively, one may attempt to verify the analysis case by simulation or testing. In any case new cases are propagated from this step.

A step may also input results produced by another step. This will then be reflected in the purpose and defined A&T cases. E.g., an “interesting” trace from a test or simulation run can be checked on the model to determine if it reflects legal behavior. Another use is to “warm-up” the model to an interesting starting state and start analysis from that.

5.1.3 A Static Code Analysis Step
5.2 Daimler Turn Indicator

The test object of the case study UCA4 is a simplified Simulink/Stateflow behavior model of a turn indicator system. The model consists of several components which are interoperating to fulfill high level functions. A textual specification of the system functions and organization-specific test objectives have been provided by Daimler. Within the UCA4 different analysis and testing techniques shall be combined. Considering different possible scenarios, the first step is to use the results from different analysis techniques to optimize the functional test case generation. One main goal is to find more defects than the existing quality assurance approaches. Therefore, the results from different analysis activities should be exploited for a more focused test generation.

Different sources for the analysis have been considered including requirements, design models, code, and test models. Tools from different partners support these analyses. Based on different metrics, such as various defect metrics and product metrics, relationships between static analysis and testing activities should be identified that can be used to optimize testing. Another goal addressed here is to assure the standard compliance of the updated quality assurance process (especially to the ISO 26262) when selecting certain analysis or testing techniques and tools. Besides the pure selection of defects, it is also of interest which kinds of defects are found with the different analysis and testing techniques in order to provide a more fine-grained focus for the different quality assurance techniques, and consequently for the overall combined quality assurance strategy.

Figure 21: Quality assurance activities applied in UCA4

Figure 21 shows the workflow of the MBAT solution proposed for the UCA4. On the left side, development artifacts (such system design models and code) are constructed based on the requirements specification. The modeling notation is Simulink which enables automated model verification and automated generation of program code. The Simulink model verification is provided by the BTC Embedded tool suite. An alternative verification approach based on extended finite state machine is provided by the Uppaal tool. Uppaal requires the transformation of the relevant behavior and time properties into the Uppaal time automata notation.

For the generation of C-code the Simulink model is transformed into a TargetLink-Model which is enriched by scaling information to enable fix-point code generation by means of the TargetLink-Code-Generator.
code is formally analyzed by the AbsInt tools which focus on the resource consumption evaluation of code artifacts (worst-case execution time and worst-case stack consumption) and the detection of runtime errors caused by code defects such as integer/floating-point division by zero, out-of-bounds array indexing, erroneous pointer manipulation and dereferencing, and unreachable code pieces. By this, these tools determine safety guarantees concerning the runtime and stack consumption budget of the application as well as proving the absence of runtime errors. Such guarantees directly address the goals of all safety standards relevant for the development of safety-critical systems in the transportation domain.

On the other side, test artifacts (test models, test cases) are created based on the same requirements specification. The solution works with an independent test model which ensures that transformation issues from requirements to system design are not reflected in the test artifacts. The test model is constructed from the functional system specification by systematic inspection (sequence-based specification (SBS)). The test model elements are annotated with information on importance, frequency, or criticality. This information is usually extracted from the system specification and expert knowledge. In MBAT, design model and code analysis results are additionally considered for assessing the importance of test model elements. The test model is transformed into TPT (model-based test generation tool developed by Piketec) test models which enable the automated generation, execution, and evaluation of test cases in different test and system environments including Simulink and compiled platform-specific code.

The use case scenarios, as described in [MBAT4, 2012], sketch ideas or process workflows for the combination of analysis and testing methods in order to reduce overall V&V costs. To achieve this goal, information exchange between static analyzers and dynamic testing tools forms the basis. On the one side, the focus for analysis activities is driven by test results, e.g., by providing information about input values of a failed test, which should be statically analyzed to examine the root cause of the failure. On the other side, specific test activities can be replaced by static analysis. For example, if Astrée proves the absence of any out-of-bounds array access, no dynamic test cases need to check array boundary accesses (following the best practices recommendations from Section 4.2).

In addition to the above outlined exploitation of synergies between analysis and testing methods, other kind of tool integrations have the potential to either reduce overall V&V costs but also to enhance quality of analysis results. Model-based testing tools have access to information which usually is not available at the implementation level. Examples are value ranges of the system’s input parameters or any other kind of information about the environment of the SUT. Integrating static analyzers into the workflow of model-based testing tools enables the transportation of such information from the model level to the static analyzer. They can then exploit the information to increase analysis precision. Furthermore, the workflow integration itself saves V&V efforts.

This static analysis precision enhancement represents information exchange “downwards”, i.e., from the model level down to the implementation level. But analysis results can also be lifted up to the model level in order to get feedback on the consequences of design decisions to the resulting code implementation with respect to certain non-functional properties like the worst-case execution time. For this, analysis results (linked to program points in the implementation) need to be imported by a tool that can map these results back to the originating components in the design model.

For the first version of the case study (RTPv0), a prototypical workflow integration [Kästner, 2013b] between the EmbeddedTester tool from BTC and the three static analyzers aiT, StackAnalyzer and Astrée (all from AbsInt) demonstrates the above described advantages. In addition to that more general tool integration, RTPv0 only considers the inspection of the informal system specification for the construction of a functional test model as analysis activity. The subsequent extension stages will also exploit code analysis and model analysis techniques for the improvement of the test model and test case generation techniques (RTPv1+2).
6 Management of Complexity

6.1 Requirements Decomposition
High level system requirements are usually decomposed into more detailed requirements and mapped to (sub) system components or modules. Such a decomposition supports hierarchical V&V where the overall task is broken down into more manageable tasks and where redundant activities can be more easily identified [Engel, 2010].

![Diagram of Requirements Decomposition](image)

Figure 22: Decomposition of Requirements (based on [Engel, 2010])

6.2 Requirements Verification Matrix
A Requirements Verification Matrix (RVM) provides an overview of the V&V strategy employed for a given project. An RVM specifies how (by what verification method and by what procedure) and when (at what point in the lifecycle) each requirement will be verified. During V&V planning an initial plan is developed, but it is maintained and iteratively refined during project development.

The rows for the RVM are defined by each individual (high-level/decomposed/atomic) requirement. The main columns are:

- Requirements ID
- Requirements traceability Information
- Verification method (e.g., Analysis, Simulation, Test, Audit/review, Similarity, …)
- Verification stage (Analysis/Definition, Design, Implementation, (component) Integration, (system) integration Acceptance/Qualification, …)
- Verification procedure
- Resources needed to execute the Verification
For MBAT the most interesting point is the choice of the verification methods: analysis and/or testing.

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Requirement Traceability</th>
<th>Verification method</th>
<th>Verification stage</th>
<th>Procedure ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL-1 A.1.1</td>
<td>X</td>
<td>None</td>
<td>Analysis</td>
<td>DD-45</td>
</tr>
<tr>
<td>SL-2 A.1.2</td>
<td>X</td>
<td>None</td>
<td>Inspection</td>
<td></td>
</tr>
<tr>
<td>SL-3 A.1.3</td>
<td>X</td>
<td>None</td>
<td>Test</td>
<td></td>
</tr>
<tr>
<td>SL-4 B.5</td>
<td>X</td>
<td>X</td>
<td>Certification</td>
<td></td>
</tr>
<tr>
<td>SL-5 B.6</td>
<td>X</td>
<td>X</td>
<td>Design</td>
<td>X</td>
</tr>
<tr>
<td>SL-6 K.22</td>
<td>X</td>
<td>X</td>
<td>Implementation</td>
<td>X</td>
</tr>
<tr>
<td>SL-7 K.23</td>
<td>X</td>
<td>X</td>
<td>Integration</td>
<td>X</td>
</tr>
<tr>
<td>SL-8 Z.1.2</td>
<td>X</td>
<td>X</td>
<td>Qualification</td>
<td></td>
</tr>
</tbody>
</table>

Figure 23: Example of a Requirements Verification Matrix (from [Engel, 2010])

```
<table>
<thead>
<tr>
<th>Rqmt</th>
<th>Ver*</th>
<th>Val**</th>
<th>DT</th>
<th>OT</th>
<th>Techniques</th>
<th>Scheduling</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assess 1 X</td>
<td></td>
<td></td>
<td>code analysis</td>
<td>start + 3 months</td>
<td>code analysis</td>
<td>software</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
<td></td>
<td>audit</td>
<td>start + 4 months</td>
<td>software engineers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assess 2 X X</td>
<td></td>
<td></td>
<td>audit</td>
<td>start + 5 months</td>
<td>software engineers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c</td>
<td></td>
<td>doctrine review</td>
<td>start + 3 months</td>
<td>doctrine, military reviewer</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d</td>
<td></td>
<td>audit</td>
<td>start + 5 months</td>
<td>software engineers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>e</td>
<td></td>
<td>system integration</td>
<td>start + 6 months</td>
<td>computer engineers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assess 3 X X</td>
<td></td>
<td></td>
<td>doctrine review</td>
<td>start + 3 months</td>
<td>doctrine, military reviewer</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>f</td>
<td></td>
<td>fleet test</td>
<td>start + 24 months</td>
<td>ship, trained crew, software</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assess 5 X</td>
<td></td>
<td></td>
<td>fleet test</td>
<td>start + 24 months</td>
<td>ship, trained crew, software</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>g</td>
<td></td>
<td>live fire test</td>
<td>start + 24 months</td>
<td>missile, homing algorithm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 24: Another example of an RVM [DOD, 2006]
6.3 A note on formal compositional reasoning

Compositional reasoning is a formal/mathematical technique for subdividing a complex verification problem into easier problems that allows one to make conclusion about the complex problem directly from results of the sub-problems. Such techniques are the holy-grail of making formal verification scale to very large and complex problems. The underlying ideas are also highly relevant for MBAT as many of the industrial scenarios asks for “avoiding testing requirements that have been verified”, using different techniques and tools to verify different aspects, and obtaining industrial scale verification.

6.3.1 Assume Guarantee Reasoning

Consider for instance a system is composed of two components $C_1$ and $C_2$ and that should satisfy some requirement formulated as a logical condition over properties $p_1$ and $p_2$. The verification rule shown below states that the composed system satisfies the composed property if $C_1$ satisfies $p_1$ and $C_2$ satisfies $p_2$:

\[
\begin{align*}
C_1 & \not\models p_1 \\
C_2 & \not\models p_2 \\
\hline
C(C_1,C_2) & \not\models L(p_1,p_2)
\end{align*}
\]

The statement under the line is the conclusion and each line above is a premise. Hence to verify the conclusion each premise must be verified. Also $C$ denotes a composition operator (e.g., sequential or parallel composition), and $L$ denote a logical composition (e.g., conjunction or disjunction). Unfortunately this rule is only correct for very special types of components, properties, and composition operators. Therefore, more specialized rules need to be developed.

Frameworks for compositional reasoning are often formulated using the assume-guarantee paradigm. The main statement is that a given system component provides a set of guarantees provided certain assumptions about its environment (or other components) holds:

\[\{\text{Assumption}\} \text{ Component } \{\text{Guarantee}\}\]

For example, many computer scientists know of Hoare-style rules for program proofs. Let $S$ and $T$ be program statements, and $P$, $Q$ and $R$ conditions over the state variables of the program, then the rule for sequential composition (the “;”-operator) states that from precondition $P$, the program $S;T$ guarantees post condition $R$, if the post condition of $S$ is $Q$, and $T$ with precondition $Q$ guarantees $R$:

\[
\begin{align*}
\{Q\} \ T \ {R} \\
\{P\} \ S \ {Q} \\
\hline
\{P\} \ S;T \ {R}
\end{align*}
\]

Similar proof rules have been proposed for verification of reactive systems that usually consists of communicating concurrent components, where the idea is to come up with a simpler assumption $A$ for one of the components and verify that the other component under this assumption provides the necessary guarantee. Alternatively, in terms of refinement checking, the composed system refines a specification $R$ if the system is correct with one component replaced by a proper\(^6\) abstraction $A$:

\[
\begin{align*}
\{P\} \ C_1 \ {A} \quad \{\text{true}\} \ C_1 \ {A} \quad C_1 \prec A \\
\{A\} \ C_2 \ {R} \quad \{A\} \ C_2 \ {R} \quad A \parallel C_2 \prec R
\end{align*}
\]

\(^6\) A specification $S_1$ refines a specification $S_2$ if it is possible to replace $S_2$ with $S_1$ in every environment and obtain an equivalent system.
Alas, such formal reasoning techniques have proven extremely difficult to develop that are easy to use, works on realistic modeling notations and properties, and have efficient tool support. A typical problem is making up or computing abstractions or simplified assumptions that allows the proof to carry through in a sound and complete manner [NT, 2010], e.g., without making having each component make assumptions about the other (such circular assume-guarantee reasoning is generally not sound).

Hence, such frameworks currently require a high degree of expertise in formal methods and proof techniques, and are not generally available for industrial adaptation, except for highly critical software. Nevertheless, it is useful to think in terms of divide-and-conquer and explicitly capture and reason about the assumptions made for every verification step.

6.3.2 Contract-based Specification and Virtual Integration Testing

Formal, contract-based component specification for functional and non-functional aspects of components enables characterizing intended system behavior in a way that allows for virtual integration testing (VIT). Contracts capture the intended context of a system using assumptions and promises that clearly define intended system context and behavior, respectively. The methodology supports breaking down contracts of top-level components and mapping them to contracts for sub-components.

Using formal specifications based on pattern specifications the analysis enables checking compatibility of components during early design phases and identifying integration problems that would otherwise only be detected much later using costly and time consuming conventional test. VIT then is used to check the correctness of design decisions early in the process using analysis methods, such as model checking. The results will either show that all sub-components can be integrated in a way that all their interdependent assumptions are satisfied while at the same the system’s guarantees are met, or clearly point out the set of component contracts is not compatible and needs to be refined.

Using formal specifications based on pattern specifications the analysis enables checking compatibility of components during early design phases and identifying integration problems that would otherwise only be detected much later using costly and time consuming conventional test.

Contract and model-based methods can also be employed to generate and validate the test cases that are needed to show that the concrete implementations of each sub-component satisfy their contracts. The structure of contracts, particularly the clear specification of intended systems context that is contained in the assumptions allows to focus the generation of test cases to contexts (of the component) for that the component has been designed.

Another scenario is to generate monitors from contracts that support the review of test results, identifying those contracts that are covered by a given collection of test/result pairs.

See [Damm, 2009] and [Damm, 2011] for details.
7 Workflow Guides

Applying advanced combinations analysis and testing presumes that the relevant analysis and testing techniques individually can be applied in an industrial context. To ease adaptation and help in professional use of the techniques, we here provide a number of guidelines for the best-practices application of each of the main techniques.

7.1 Micro Guide: Modeling (with Analysis)

A model is an abstract view of reality, in which essential properties are recorded, and other properties and details considered not important for the problem at hand, are removed.

An everyday example of a model is a road map: a road map only contains lines representing roads, and circles representing cities. The map abstracts from many other details of reality, such as buildings, forests, railways, mountains, the width and the kind of pavement of roads, et cetera. Such a map, i.e., a model, may very well help with planning your trip by car from Brussels to Paris, because all relevant details for such a trip are there. For planning a railway trip, however, or for calculating the altitude difference between Brussels and Paris, such a road map is useless. Another map, i.e., another model with other abstractions, is needed such as a railway map, or a geographic map, respectively. A good model is according to [Vaa, 2012]:

1. A good model has a clearly specified object of modeling, that is, it is clear what thing the model describes.
2. A good model has a clearly specified purpose and (ideally) contributes to the realization of that purpose.
3. A good model is traceable: each structural element of a model either (1) corresponds to an aspect of the object of modeling, or (2) encodes some implicit domain knowledge, or (3) encodes some explicit additional assumption.
4. A good model is truthful (or valid): relevant properties of the model should also carry over to (hold for) the object of modeling.
5. A good model is simple (but not too simple). Occam’s razor is a principle particularly relevant to modeling: among models with roughly equal predictive power, the simplest one is the most desirable.
6. A good model is extensible and reusable, that is, it has been designed to evolve and be used beyond its original purpose.
7. A good model has been designed and encoded for interoperability and sharing of semantics.
8. A good model has the right level of abstraction, i.e., it contains all specifications that it shall have, and as little more to make the semantics of these specifications unambiguous. See also rule 5.
Models serve several rules during development. Below are a number of typical examples.

**Specification model:** Captures the high-level requirements (for the behavior of the system). The goal is to ensure consistency and completeness of requirements, to identify ambiguities and unclear points. Behavioral models can examine the soundness and completeness of the basic/abstract functionality.

**Design model:** Design models capture the proposed structure of the system under design in terms of subsystems and components. Further, it may contain overall specifications (“what”) for the behavior of its sub-systems (or components) but without stating implementation details (i.e. “how”).

**Design exploration model:** The goal is to allow tool supported analysis of the cost and performance of different design proposals or variations. Captures and focuses on the main influencing variables.

**Implementation models (Component models):** focus on providing a behavior description of a component, sufficiently detailed that it can easily be turned into an implementation (either via manual programming, or automatic synthesis/code generation). The goal is to describe in detail how a particular task is fulfilled, not only a specification of the desired outcome.

**Analysis model:** Analysis models are made to examine specific properties of a (sub) system, e.g., interaction among system components, timing, or safety/liveness. Such analysis can be made before the system is built, or afterwards to “debug” why a system does not work as expected. Similarly, model-checking models are created with a particular purpose of objective in mind (what requirements / properties should be checked / what precise questions need answering. Given the limitation of model-checking (state-space explosion) it is even plausible that several models are needed targeted at selected subsets of the properties.

**Test model:** Test models are made with the intent of generating (abstract or concrete) test cases that can be executed against the system under test. Test models are different from design and analysis models. They often contain information about the concrete interfaces of the SUT and must reflect the number of component instances (e.g. In Daimler’s turn indicator there are 13 individual light indicators; not all are relevant from an analysis point of view.), and value-ranges provided by the SUT to make test cases realizable. Further, it may contain other test related information and annotations like strategy specifications, test-case goals, model coverage information etc. Similarly development (analysis, design or implementation) models contain details that are irrelevant from a (black-box) testing point of view, where the specified behavior is interesting. Finally, construction of test models may start right after requirements specification is available.

See more on test versus development models at [Peleska, 2008].
Performance, timing, reliability, Safety: Models focus on selected extra-functional properties because they are often at different levels of abstraction and may require specialized tool support.

Hence it is not feasible to capture all these aspects in a single model (as it would become too large, detailed, and rich, and thus defeat the purpose of having a model). Moreover, in the current state-of-the-art, there is not a single modeling notation that can capture all these aspects and is suitable to support all these needs.

7.2 Micro Guide: MBT Generation

In model-based testing, sound and complete (wrt. test selection criterion) test cases are generated by a tool from high-level (behavior) models specifying the required and allowed behavior of the system under test. If the model or requirements change the tests can be re-generated thus alleviated script maintenance. [Utting, 2012] provides a taxonomy of model-based testing.

The main workflow is:

1. The objective of the test campaign must be described. This includes an identification of the required system behavior to be tested, the exact SUT, and its interfaces (including points of control and observations).

2. A test model capturing the allowed and required behavior at a suitable level of abstraction must be developed. The observable abstract actions (or observation predicates) used in the model must match the interface provided by the test adaptor – the model and adaptor are thus inter-dependent. It is important to check the behavior of the test model, because otherwise the test cases may not test the intended behavior. The information used to develop the test model usually consists of the relevant requirements supplemented by design documents and models. A suitable level of abstraction means that it should capture the behavior to be tested (no more and no less), not reflect SUT internal behavior that is not needed (see also 7.1). On the other hand it cannot be too abstract because it must reflect the actual components and interfaces of the SUT (although many details of the actual communication can be hidden in the adaptor).

3. A test adaptor must be developed. It is a piece of software (sometimes also hardware) whose task is to map the abstract observable actions to/from concrete stimuli to, and observations of, by the SUT. Input actions (from SUT perspective) often this involves action refinement where a sequence of stimuli is issued or where the exact parameters issued determined. Outputs (from SUT perspective) must be abstracted into observable actions matching those used at the model level.

4. A test model specifies an infinite amount of behaviors and hence also an infinite number of possible test cases. Theoretically they are all needed, if the test generation is to be complete (enable all non-conforming implementations to be detected), but only a highly limited number may be executed. Hence, test selection is highly important, and test case specification of the identified cases is necessary. The amount of work depends on the criteria and tool.

5. Test execution. Test execution is normally automated. After an execution the verdicts are analyzed. Initially it is very common that the model may need to be revised due to unclear requirements or assumptions (or modeling mistakes). This is an excellent source for additional test cases (or analysis cases). Thus getting a stable model may take several iterations.
Figure 26: Main Model-based testing workflow

Test selection can be based on (combination of) several principles:

- **Requirements:** One or more test cases must be generated for each requirement. To do so the test engineer must describe a formalized test purpose (observation objective) to the test generator that can then synthesize a matching test case.

- **Structural coverage criteria** of model (or code): The generated test suite must satisfy a specified structural coverage criterion of the model (or code). Typically criteria include location, edge, def-use pairs, mc/dc, etc. The view is that each structural element captures part of the specified behavior and that as a minimum this must be tested.

- **Fault-models:** A fault model defines a set of faults one believes may occur or that one wants to target. The generated test suite will contain at least one test case that can detect/kill each fault. Mutation testing is a particular kind of fault model based test selection.

- **Random:** A (possibly large) number of random test cases are generated (usually guided by an environment model to ensure realistic test cases). Random tests can often detect defects occurring in scenarios not anticipated by short requirements and coverage based test generation.

The ensure good coverage of all these aspects we propose the procedure outline in Figure 1 where the existing set of test cases are supplemented to reach the desired next level. Thus there will be no redundant test cases.
Model-based mutation testing. Model-based mutation testing is a fault-based approach in the area of model-based testing. Via a set of different fault models (mutation operators), faults are introduced to the AT-Model producing several “mutated” models. Different sorts of refinement checks can be used to search for different behavior between the original and the mutated models. If a fault is found it produces a test case. Each test case ensures that the fault represented by the specific mutated model is not implemented. More information can be found in the Survey by Jia and Harman [JH, 2011].
7.3 Micro Guide: Static Code Analysis

The following guide explains static code analysis using a concrete tool suite. However, many general lessons can be learned from the specific guidelines.

7.3.1 Abstract Interpretation-based Static Analysers

Astrée, aiT, and StackAnalyzer are sound semantics-based static analysers based on Abstract interpretation. Such analysers compute invariants for all program points by fixed point iteration over the program structure or the control-flow graph. The theory of abstract interpretation [Cousot, 1977] offers a semantics-based methodology for static program analysis.

The semantics of a programming language (C in case of Astrée, machine code in case of aiT and StackAnalyzer) provides a formal description of the behavior of programs. The most precise semantics is the so-called concrete semantics, describing closely the actual execution of the program. Yet in general, the concrete semantics is not computable. Even under the assumption that the program terminates, it is too detailed to allow for efficient computations. The solution is to introduce an abstract semantics that approximates the concrete semantics of the program. This abstract semantics can be chosen to be computable. The static analysis is computed with respect to that abstract semantics. Compared to an analysis of the concrete semantics, the analysis result may be less precise but the computation may be significantly faster. By skillful definition of the abstract semantics, a suitable trade-off between precision and efficiency can be obtained.

Abstract interpretation supports formal correctness proofs: it can be proven that an analysis will terminate and that it computes an over-approximation of the concrete semantics, i.e., that the analysis results are sound.

- For a static runtime error analysis as in Astrée, this means that it never omits to signal an error that can appear in some execution environment. If no potential error is signaled, definitely no runtime error can occur. If a potential error is reported, this means that the analyzer cannot exclude that there is a concrete program execution triggering the error. If there is no such execution, this is a false alarm.

- For a WCET analysis as in aiT, soundness means that the reported WCET is never below the actual execution time in some execution environment. For instance, if the reported WCET is 1 ms, the actual execution definitely cannot last longer than 1 ms. On the other hand, overestimation may occur: it is possible that actual executions never take longer than 0.9 ms.

- In the same way, the stack usage reported by StackAnalyzer is never below the stack usage in any concrete execution, but overestimation may occur.

Summarizing, imprecision can occur, but only on the safe side; with Astrée it can never happen that there is an error from the error class under analysis which is not reported, and with aiT and StackAnalyzer it can never happen that the actual resource usage (time, stack) is higher than the worst-case resource usage determined by the tool.

By design, sound Abstract interpretation based static analyzers provide full control and data coverage. For approaches based on test and measurement, identifying end-of-test criteria for non-functional program properties like timing, stack size, and runtime errors is an unsolved problem. In consequence the required test effort is high, the tests require access to the physical hardware and the results are not complete. In contrast, static analyses can be run by software developers from their workstation computer, they can be integrated in the development process, e.g., in model-based code generators, and allow developers to detect runtime errors as well as timing and space bugs in early product stages. From a methodological point of view, abstract interpretation based static analyzers can be seen as equivalent to testing with full coverage. For validating non-functional program properties they define the state-of-the-art technology [Kästner, 2011].

In the following sections, the workflow of three abstract interpretation-based static analyzers is described from a user's perspective.

7.3.2 Static Runtime-Error Analysis with Astrée

7.3.2.1 Tool Overview

The objective of Astrée is to signal runtime errors and to prove their absence. The latter is possible since Astrée is sound, i.e., it signals all potential runtime errors. If no errors are signalled, this means there are no errors -- the absence of runtime errors has been proved.
The Astrée analyzer operates on C source code. The runtime errors considered by Astrée include violations of the C standard [ISO/IEC 9899:1999 (E)], implementation-specific undefined behaviours, and violations of user-specified programming guidelines:

- integer/floating-point division by zero,
- out-of-bounds array indexing,
- erroneous pointer manipulation and dereferencing (NULL, uninitialized, and dangling pointers),
- integer and floating-point arithmetic overflow,
- violation of optional user-defined assertions to prove additional runtime properties (similar to assert diagnostics),
- code it can prove to be unreachable under any circumstances (note that this is not necessarily all unreachable code),
- read access to uninitialized variables,
- write access to variables without previous read access.

Astrée is sound for floating-point computations and handles them precisely and safely. It takes all possible rounding errors into account.

### 7.3.2.2 Recommended Working Pattern

In the following the recommended working pattern with the Astrée Analyzer is summarized. For more information about the individual stages, see the Astrée User Manual [AbsInt, 2013a] or the Astrée Quick Startup Guide [AbsInt, 2013b].

#### Setup Stage
- Set analysis options: ABI, semantics, precision, verbosity
- Define an analysis entry point
- Preprocess the code (can be done with the built-in clang preprocessor or externally)

#### Parsing and Filtering Stage
- Non-C99 compliant code fragments can be filtered by the Astrée filter process.

#### Environment/Interface Modeling

External information about the environment and interfaces can be specified by Astrée directives. All directives should be specified in the AAL language to avoid source code modifications, which is an important criterion with respect to certification issues.

- Specify input ranges for external variables via the \_ASTREE\_known\_fact directive.
- Specify input ranges for volatile variables via the \_ASTREE\_volatile\_input directive.
- If necessary, declare types and variables for absolute memory addresses in the program to enable Astrée’s type-safe analysis for absolute addresses (directive \_ASTREE\_absolute\_address).
- If necessary, define an appropriate analysis wrapper, e.g., by calling the necessary initialization routines and providing an infinite loop for repetitive task executions in reactive systems.
- If desired, insert \_ASTREE\_assert directives to prove output values to be in their expected ranges.

Such kind of environmental information is available at the model level, e.g., in a TargetLink model.

#### Analysis Stage

The following steps have to be executed repetitively, until all errors have been fixed, and all false alarms are accounted for.

- Use the editor and the different views of the Astrée Output Area to proceed through alarm messages and investigate variable values.
• Use Astrée directives to tune the analysis precision, and to supply possibly missing information. The goal is to make sure that all relevant code has been analyzed and to tune the analysis precision to the software under analysis to reduce the number of false alarms.

• Alarms can be classified as true error / false alarm and can be tagged with an explanatory text directive __ASTREE_comment.

Reporting Stage

• Generate final reports.

The overall analysis result is given by the number of errors and alarms and is visualized by a traffic light symbol in the GUI. The traffic lights show

• red if there is an error, e.g., a parse error or a type A alarm which definitely occurs in some context
  o Red errors have to be fixed.

• yellow if there is no error, but there is at least one alarm of type A.
  o Alarms of type A should be eliminated by increasing analysis precision via Astrée directives.
  o If analysis precision cannot be increased enough to prevent false positives of type A, false alarms should be classified as such by the __ASTREE_comment directive and tagged with a suitable explanation.

• yellow and green if there are no errors and no Type A alarms, but at least one alarm of Type C.
  o Remaining false alarms of Type C can be classified as such alarm by the __ASTREE_comment directive and tagged with a suitable explanation.

• green if the analysis has finished without producing any errors or alarms.

The analysis stage can be performed fully automatic (batch mode) after having finished project setup as described in the first three points.

7.3.2.3 Software Life Cycle

Upon modifications of the software source code the pre-processed code has to be regenerated. Changes to the analyzed files are detected automatically by Astrée. Batch mode operation supports automatic addition/removal of files from the analysis source directories. Astrée directives externally specified by AAL directives are robust with respect to source code changes. They do not rely on line number information; instead they are based on the structure of the code inside a C function. If the implementation of a C function was modified by a source code update, the Astrée directives associated to this function should be checked for correctness. Also, alarm comments should be rechecked if the program semantics might have changed due to the source code update.

7.3.2.4 TargetLink Workflow

A tool coupling between TargetLink and Astrée is currently being developed. Current results are detailed in [Kästner, 2013a]. Astrée provides means to read the extended XML export of the TargetLink Data Directory to automatically transfer model-level information to the level of the generated code.

• Based on the TargetLink PIL frame, an Astrée analysis wrapper is automatically generated. The wrapper first calls the TargetLink restart and init functions and then enters a reactive loop from which all root step functions are invoked.

• Range information for interface variables are automatically transformed into AAL Astrée directives.

• Range information for volatile parameter values are automatically transformed into AAL Astrée directives.

• For standard TargetLink functions, e.g., interpolation routines, Astrée directives are automatically generated that enable high analysis precision.

7 Due to definite runtime errors of type A there may be non-analyzed code parts.
The Astrée-TargetLink coupling does not conceptually change the Astrée workflow described above. In the Setup Stage, and, in general every time the model and the generated code have been changed, the TargetLink DD importer has to be called as an additional step. The environment/interface modeling is simplified since a large part of the required information is automatically extracted from the data dictionary. The software life cycle is changed in that respect that the source code is regenerated upon changes in the software model. Apart from that, the considerations from Section 7.3.2.3 apply.

### 7.3.2.5 Combination with Dynamic Testing

A so-called workflow integration (as introduced in Section 2.3 of Astrée and EmbeddedTester from BTC-ES is currently being developed [Kästner, 2013b] in the context of MBAT. EmbeddedTester automatically creates test models for automatic test case generation and execution for arbitrary levels of the system design, thereby accessing the model level definitions of signals, parameters, etc. The static analysis methods and tools can leverage from this capability to gain an automatic analysis model construction and automatic application of the three AbsInt tools. This is complemented by an additional tool coupling interface between TargetLink and the AbsInt tools which allows analysing standard lookup and interpolation functions with individually optimal precision [Kästner, 2013a]. Aggregated result reports are generated providing a comprehensive and holistic view on both functional and non-functional verification results, making it convenient and efficient to assess the verification results. Hence, end users leverage even from this loosely coupled combination of model-based analysis and testing. It provides an essential workflow optimization for applying the analysis tools. This makes working with analysis tools much more convenient and flexible, and makes analysis iterations much faster, thus saving time and cost.

### 7.3.2.6 Component-Level Analysis

Component-based analysis enables a seamless integration of static analysis in the development process. Potential runtime errors can be detected and fixed early, preventing late-stage integration problems. Astrée can be applied to component-level analysis by specifying the main function of the software component as analysis entry point for Astrée. Interface information can be provided via Astrée directives as described in the Environment/Interface Modelling step in Section 7.3.2.2.

In the integration stage, the externally specified Astrée directives from all software components can be aggregated. __ASTREE_known_fact directives for specifying input ranges of software components can be transformed into static assertions that check the compliance of the integrated software with respect to the component specification. All other directives including alarm comments can be reused for the integrated software which considerably reduces the analysis effort.

### 7.3.3 Static Resource Consumption Analysis: aiT and StackAnalyzer

#### 7.3.3.1 Tool Overview

The purpose of aiT is to determine safe and precise upper bounds for the Worst-Case Execution Time (WCET) of tasks in real-time systems. These upper bounds can be used to verify that the tasks meet their deadlines, and may serve as inputs for an overall schedulability analysis [Kästner, 2008].

In the context of aiT, a task means a sequentially executed piece of code (no threads, no parallelism, no waiting for external events, and assuming no interference from the outside). aiT operates on binary executables for selected target architectures and produces results valid for all program runs with all inputs. It is intended to be used for timing verification in late stages of the system development when the system code is already available and the target processor and its configuration have been fixed. The design goal for aiT was to avoid any underestimation of the WCETs, which requires a detailed model of the underlying hardware and its timing behavior.

aiT takes as input an executable containing the task to be analyzed, a description of the hardware on which the task is running including a description of (external) memories and buses (i.e. a list of memory areas with minimal and maximal access times), and code annotations providing additional information like targets of indirect jumps, loop bounds, etc. The annotations may be written by the user, but can also be provided by model-based code generator such as SCADE from Esterel or TargetLink from dSPACE.
StackAnalyzer can determine upper bounds for the size(s) of the stack(s) in single routines and in the whole program. These upper bounds can be used to verify the absence of stack overflow for a given setting of stack parameters, or to derive a parameter setting that avoids overflow and does not reserve too much memory for the stacks.

Like aiT, StackAnalyzer operates on binary executables for various target architectures. StackAnalyzer and aiT share the same user interface called AbsInt Advanced Analyzer (a³).

7.3.3.2 Recommended Working Pattern

In the following the recommended working pattern with aiT and StackAnalyzer is summarized. For more information about the individual stages, see the a³ user manual, which covers both aiT and StackAnalyzer [AbsInt, 2013c]. Note that there are specific user manuals for all target architectures covered.

Hardware Configuration

This step does not yet depend on the specific task to be analyzed, but refers to the hardware it will run on: the exact target machine type and revision, settings of configuration registers, and cache and memory configuration. The latter includes the address ranges of all memory areas that are possibly accessed by the analyzed software, their access types and latencies. Volatile memory areas have to be declared as such so that the value analysis knows that values stored there may change. The details of what has to be specified depend on the target architecture. In some cases, parts of the configuration are predefined by the tool. Anyway, most of these settings are only relevant for timing analysis, not for stack analysis.

Compiler Declaration

Providing a compiler declaration helps the analyzer to reconstruct the control flow, e.g., by using compiler-specific patterns for loops or switch tables.

Analysis Setup

This step involves naming the executable containing the task(s) to be analyzed and the entry point(s) for the analyses (routine names or addresses). If the analyzer should write textual and/or XML reports, names for the report file(s) have to be specified.

Annotations for Decoding

The decoder cannot always find the targets of computed calls by static program analysis. Information about the targets of the unresolved calls has to be provided by annotations. In these annotations, the possible targets can be listed, or the decoder can be instructed to extract the list of targets automatically from a given array or table. It is also possible to not specify any targets, but to tell a³ to assume fixed execution times and stack properties for the potentially called routines whatever they might be.

Loop Bounds and Recursion Bounds

Loop bounds that cannot be determined automatically have to be provided by annotations in order to make a timing analysis possible at all. Stack analysis requires loop bounds only for loops with a non-zero effect on the stack level per iteration. Recursion bounds are needed for both timing and stack analysis.

Improving the Precision of the Analysis

When a timing analysis has run successfully for the first time, it often turns out that its results are higher than expected because of overestimation. There are various ways to improve the precision of the analysis, with different effects on the analysis time. Some of these methods may also be useful for stack analysis, but stack analysis results are usually quite precise right from the beginning.

- Provide address specifications for memory accesses which could not be calculated exactly or could not be determined at all by the value analysis. In case of unknown memory accesses, the timing analysis will perform a case split over all possible memory areas. If the timing properties of different
memory areas are highly heterogeneous, this can lead to huge overestimations and also to an increased analysis time.

- You can provide information about the value of a register or memory cell at some program point. This information may reflect knowledge about the range of some input variable or function result, or correspond to a specific operation mode of the analyzed system. The added information can lead to more precise address information for some memory accesses, or might imply that some conditions are always false or always true, and thus some paths might be never executed; this way, they do not contribute to the resource usage which improves the precision.

- You may also specify directly that some conditions are always false or always true.

- You may specify that certain code snippets are never executed, either because you know that this is true or because you are not interested in the contribution of these snippets, e.g., in case of error code.

- You may review and improve the results of automatic loop bound analysis.

- You may specify flow constraints that specify some relationship between the execution counts of different program points, e.g., this part is only executed if that part is executed, or the sum of the execution counts of these five parts is at most two.

- You may tune analysis parameters, e.g., increase the number of execution contexts or decide to use prediction files in path analysis at the expense of higher analysis times [Cullmann, 2011].

7.3.3.3 Batch Mode Operation
The a³ tool framework, which incorporates aiT and StackAnalyzer, supports interactive working and also a batch mode for automated analysis execution. Timing and/or stack analyses have to be configured once by using the GUI (Setup Stage). The configurations can be stored in a project file and passed as a parameter to a command line invocation of a³ which starts the configured analyses in batch mode without user interaction. The generated report files may be examined or processed at some later point in time.

7.3.3.4 TargetLink Workflow
A tool coupling between TargetLink and with aiT and StackAnalyzer is currently being developed. Current results are detailed in [Kästner, 2013a]. When TargetLink needs the WCET or the stack usage of a task, it can send a corresponding request to a³. After a successful analysis, the analysis result is communicated back to TargetLink.

Some information that is available at the source code level (e.g., constraints on input and output parameters) is not portable to the binary code level, as variable and parameter names are no longer present. Thus, value ranges for register and memory cell contents need to be recomputed by the value analysis phase of the WCET/stack analysis. A dedicated wrapper is not needed since the worst-case execution time and stack usage information is separately computed for each root function (runnable). Both analyses require knowledge on the maximal iteration count of loops in the code (stack analysis only for loops with a non-zero effect on the stack height per iteration). Whereas these loop bounds can be easily determined by the value analysis for vector iterations, this automatic determination fails for functions that are generated by look-up blocks. TargetLink allows several parameters of these blocks (e.g., axis search algorithm, parameter passing, etc.) to be modified in order to speed up table look-ups. Most of these settings directly influence the generated C code of the look-up function. The information on the values of these settings is included in the TargetLink Data Dictionary and is automatically extracted by the so-called ddconverter module of a³. As the size of the look-up tables is already fixed in the binary (although the values in the tables might be adjusted by an engineer), the extracted maximal iteration counts remain valid. The resulting annotations do not rely on a specific call stack, but dynamically evaluate actual call parameters to yield precise bounds. The annotations are written in AIS format into a separate annotation file.

The a³-TargetLink coupling does not conceptually change the workflow described in Section 7.3.3.2. The hardware configuration and the compiler have still to be specified, but these steps do not depend on the analyzed task and have only to be performed once. The effort for providing decoding annotations, loop bounds, and recursion bounds, and also for improving the analysis precision, is reduced since a large part of the required information is automatically extracted from the data dictionary.
7.3.3.5 Combination with Dynamic Testing
The workflow integration of Astrée and Embedded Tester from BTC-ES, which is described in Section 7.3.2.5, also applies to aiT and StackAnalyzer.

7.4 Micro Guide: Model-checking and Refinement Checking
Model checking consists of an automatic algorithmic procedure for exhaustive checking whether a formal model of a system meets a given specification. As a consequence, the model and its specification are at the center of applying model checking techniques. The first step in using model checking is to identify which class of systems and properties you want to formally verify and at which level of abstraction. The more expressive models are, the more computationally expensive (or even impossible) the model checking problem is. However, the landscape of model checking tools has been constantly growing, with individual tools adopting various options for improving the exploration speed, but also increasing the expressiveness of the models that can be analyzed. Nowadays, apart from model checkers supporting verification of finite state (synchronous, asynchronous etc.) models, there are a number of tools which also allow formal verification of models that provide richer (real-time, probabilistic, quantitative and even continuous) features. We give a non-exhaustive survey of most popular model checking tools for several classes of systems in the end of this section.

Model checkers are used to verify whether a model meets a high-level specification. High-level specifications are often expressed in some declarative language, such as temporal logic, and describe the expected properties of the model. Each desired property of the model, derived from the requirements, is usually specified by a temporal logic formula. Temporal logic properties are then combined by conjunction in order to specify the requirements the model shall satisfy. Model checkers can be used already at this stage, to check for consistency of temporal logic properties, thus making sure that properties are not contradicting each other.

Interpreting model checking results requires certain attention. If the model checking procedure gives a positive verdict, it means that the model satisfies its high-level specification. If the specification itself does not properly formalize the intended meaning, or is incomplete, the model may still contain unwanted features. In addition, one has to be aware that the model that is formally verified, no matter how detailed, in many cases still abstracts away many details from its actual implementation. Hence, a positive verdict by the model checker indicates that the model is correct, but this may not be sufficient to prove the correctness of the underlying actual system. Hence, it is crucial to have refinement checking done at various stages of design, from abstract models to actual implementations. In some modeling languages, such as SCADE, this problem is addressed by having a qualified code generation from the models, producing code which is correct by construction.

If the model checker shows that the model does not satisfy its high-level specification, it generates a counterexample, i.e., a trace that witnesses the reasons for violation of the property. This witness can be used to localize the error and take appropriate actions in order to update the model and make it correct. The counterexample, however, may also be caused due to over-simplification (over-approximation). (This is usually checked manually, i.e., by analyzing it.) In this case, it can be used to refine the model such that the counterexample will not occur anymore. For example, the model assumes that x > 0 and the counterexample tells that, in a given state s, for x=1 a given requirement r does not hold, but inspection displays that in s r is only violated if x>1. Then the model should be refined that for s the current condition is strengthened by adding “AND (x>1)”. This principle is called “CEDAR pattern” (Counter-Example Driven Abstraction Refinement) and is illustrated in Figure 29.
A third alternative when applying a model checker is that it stops before either a counterexample is found or the model exhaustively checked, e.g., due to insufficient memory or time. In this case, either the model or the requirements has/have to be simplified, or the model checker changed, or the computing platform. Figure 30 illustrates the resulting workflow.
The landscape of model checking tools has been constantly growing, with individual tools adopting various options for improving the exploration speed, but also increasing the expressiveness of the models that can be analyzed. In MBAT partners are offering model-checking tools, see D_WP2.4_3_1 Prototype tools for model-based analysis:

- ViTaL for model checking of EAST-ADL models with respect to timing and functional behavioral requirements.
- Uppaal for verification of real-time systems expressed as a network of timed automata extended with bounded-domain data variables. Uppaal-SMC support statistical model-checking of stochastic hybrid automata.
- FormalSpecs Verifier for checking functional properties of Simulink Stateflow models.

In addition the following is a short and non-exhaustive survey of some of other representative model checking tools for various classes of models:

<table>
<thead>
<tr>
<th>Model Class</th>
<th>Tool 1</th>
<th>Tool 2</th>
<th>Tool 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select model class</td>
<td>ViTaL</td>
<td>Uppaal</td>
<td>FormalSpecs</td>
</tr>
<tr>
<td>Develop analysis model</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Select model checker</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specify properties to be checked</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check properties for consistency</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consistent?</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Verdict</td>
<td>Conclusive</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Properties</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model Checker</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check if model satisfies properties</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check whether model refines implementation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Impl.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ok?</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Proof successful</td>
<td>Correct model or implementation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Select model or implementation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute counter-example on implementation</td>
<td>Simplify model or properties, or replace model checker</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spurious?</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Use counter-example to refine model</td>
<td>Use counter-example to correct model</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- SPIN\(^8\) is a general verification tool targeting distributed software systems, which has been continuously developed since 1991. The models to be verified are described in the PROMELA language and their correctness is checked against LTL properties, specifying the intended behavior of the model. SPIN enumerates reachable states in the model, one at a time, and uses a number of optimizations such as partial order reduction and state compression, to explore more efficiently the state space of the model.

- NuSMV\(^9\) is a symbolic model checking tool, which encodes sets of reachable states in the model with Binary Decision Diagrams (BDD), and has been successfully used for formal verification of realistic and industrial designs. NuSMV supports verification of both LTL and CTL properties.

- SLAM\(^10\) is a software verification engine developed by Microsoft and used in the Static Driver Verifier (SDV) to check that a C program correctly uses the interface to an external library. SLAM combines techniques from symbolic model checking with program analysis and theorem proving to solve this problem. It has been used in particular to support robust development of drivers in Microsoft Windows.

- PRISM\(^11\) is a probabilistic symbolic model checker that allows verification of real-time systems that exhibit probabilistic and random behaviors, such as communication and security protocols. PRISM supports verification of several classes of probabilistic models, namely discrete and continuous-time Markov chains, Markov decision processes, probabilistic automata and probabilistic timed automata. The properties of the probabilistic systems can be specified in several temporal logic-based languages, such as PCTL*, CSL and LTL.

---

\(^8\) [http://spinroot.com/spin/whatispin.html](http://spinroot.com/spin/whatispin.html)
8 MBAT Tools

MBAT is developing, extending, and making interoperable (via the RTP), a large set of tool components. Below we provide a quick overview of these components using a rough classification based on supported level of abstraction (system level, subsystem/control level, or implementation level), and their focus on system aspects (risk, functionality, or quantitative properties).

8.1 Analysis Tool Landscape

The tool landscape for MBAT analysis tools are shown in Figure 31. We refer to a detailed description of their capabilities and planned extensions available in the following deliverables:

- D_WP2.4_3_1 Prototype tools for model-based analysis
- D_WP2.4_2 Specification of Model-based Analysis Methods and Tools

<table>
<thead>
<tr>
<th>System level</th>
<th>Safety / risk</th>
<th>Functional properties</th>
<th>Quantitative properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>SafetyArchitect, FaultTree and FMEA generation</td>
<td>MaTeLo, DODT, RSL tools (RSL, Requirement Entailment and Consistency Analysis, RSL PatternEditor, Change-Impact Analysis) FormalSpecs BCL Toolbox, WEFACT</td>
<td>NFR Profile, DTFSim, UPPAAL, STSSim,</td>
<td></td>
</tr>
<tr>
<td>Subsystem/ control level</td>
<td>TestCast, EmbeddedSpecifier, EmbeddedTester, FormalSpecs Verifier</td>
<td>VITAL, ECDAR, UPPAAL-SMC,</td>
<td></td>
</tr>
<tr>
<td>Implementation / component level</td>
<td>Goblin, Fluctuat</td>
<td>hybridFluctuat, Astrée, StackAnalyzer, aiT BusScope</td>
<td></td>
</tr>
</tbody>
</table>

Figure 31: Analysis Tool Landscape

8.2 Test Tool Landscape

The tool landscape for MBAT model-based testing tools are shown in Figure 32. We refer to a detailed description of their capabilities and planned extensions available in the following deliverables:

- D_WP2.3_3_1 Prototype tools for model-based test case generation and execution
- D_WP2.3_2 Specification of Model-based Test Case Generation & Execution Methods and Tools
<table>
<thead>
<tr>
<th>Safety / risk</th>
<th>Functional properties</th>
<th>Quantitative properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System level</strong></td>
<td>FormalSpecs Verifier-ATG (G), TestCast (G), Diversity (G), UMMU/Argos/Ulysses (G) TestCast T3 (E)</td>
<td>MaTeLo(G), Uppaal-Tron (G+E), Uppaal-Yggdrasil(G), VTaL-TG (G),</td>
</tr>
<tr>
<td><strong>Subsystem/ control level</strong></td>
<td>Diversity (G)</td>
<td>PROVEtech:TA (E)</td>
</tr>
<tr>
<td><strong>Implementation / component level</strong></td>
<td>PathCrawler (G)</td>
<td>BTC-EmbeddedTester(G+E)</td>
</tr>
</tbody>
</table>

Figure 32: Test Tool Landscape (G=Test Generation, E=Test Execution)
9 Perspectives and Conclusions

9.1 Conclusions

This first version of MBAT deliverable, D_WP2.1_2_1 “Overall T&A Methodology”, collects the background knowledge and an initial set of guidelines for establishing a methodology for efficient combination of model-based analysis and test. Its content is basically structured into three major parts:

a) Collecting background information. Chapters 1 and 2 discuss motivation, the relationship of the methodology elaborated here to the MBAT RTP, general V&V terminology, and various levels and views of possible A&T combinations. Many methodological and technical challenges are identified. In addition, Chapter 8 presents the tools landscape available in MBAT.

b) Initial framework for combining model-based analysis and testing. Based on these settings, Chapter 3 proposes a methodological framework, addressing different objectives as well as how results can be integrated with MBAT RTP. In particular, it develops the main V&V flow as well as defines the activities a generic A&T step is composed of. It identifies three main categories of A&T steps – model analysis, model-based testing, and code analysis – and discusses how their results can be exploited for other steps. This is accomplished by Chapter 4, which collects guidelines, both for tool/technique selection, and for useful combinations of techniques. The latter, “best practices”, can be seen as a first set of generic patterns for A&T combination. Chapter 6 discusses how to cope with complexity, and Chapter 7 complements the guidelines by those for individual techniques, called micro guides.

c) Examples for A&T combinations as already applied in RTP 0 use case scenarios are described in Chapter 5. These examples serve as illustrations for the conceptual framework elaborated in the above listed chapters, e.g., how to find right level of abstraction / details for presenting method.

Finally, a few general observations have been done during the work on defining the methodology: MBAT aims at workflow integration and methodological integration; technical tool integration, i.e., the development of new tools by integrating existing tools is not excluded from MBAT SP2 activities, but it is not a major topic of MBAT (see 2.3.4).

Given

- the wide variety of modeling techniques and tools available and applied in MBAT, in particular by the use case providers, sometimes already for a longer time,
- the different purposes models can serve for (see Section 7.1),
- the fact that, using one model for all purposes would make it clumsy and difficult to maintain,

the A&T model for a given use case / application is usually a collection of (related) models, represented in various notations, including commercial ones such as Simulink. Their integration into a coherent model set is to be achieved by techniques provided by WP2.2 and SP3 (WP3.1 – meta model for semantic mapping between models, WP3.2 – IOS for in/exporting models into/from tools, WP3.3 – basic services for enabling linking/mapping between models).

9.2 Perspectives and Future Work

We foresee following topics to be addressed in the final version of this document:

a) Evaluation. An initial evaluation of the method will be done by re-inspecting the use case scenarios and challenges systematically, the proposed RTP integrations, and applications of the method.

b) Update of overall A&T Methodology. Based on feedback on the first version, in particular from use case providers, the (chapters on the) overall A&T methodology shall be improved.

c) “Catalogue” of Best Practices for A&T Combination. The Best Practices will be extended, presumably together with the Examples (for illustrating use of best practices), from the experiences
gathered in the project use cases. Such guidelines will address both A&T combinations and individual V&V steps.

d) Micro guides. The micro guides for best uses of individual techniques should also be completed.

e) Integration with MBAT RTP. The final version of this deliverable will describe how features and results from other SP2 and SP3 activities can be used for realizing the common A&T methodology.

f) Tailoring. It shall be investigated whether it is feasible to tailor the overall A&T combination methodology to specific domains of even use cases, e.g., to adapt the V&V workflow to the development workflow of individual companies. Also the models and tool suites used at the various levels should be spelled out to form concretized methods.

g) Related V&V methodologies. It should further be investigated whether the MBAT overall methodology can benefit from V&V methodologies (in particular, A&T combinations) well established in other disciplines, for instance, chip design and verification.

h) User Roles. It may be useful with a section about tool/RTP workflow from a user perspective, possibly together with the separation of “responsibilities” based on various expert roles (e.g., verification expert, system architecture/design expert).

It should finally be noted that deliverable D_WP2.1_3 “MBA/MBT Synergy Exploitation”, due at M30 like D_WP2.1_2_2, will also provide recommendations and strategies for mutually improving MBA and MBT. To avoid info duplication, these two deliverables will be properly aligned.
Abbreviations and Definitions

AAL  Astrée Annotation Language
ABI  Application Binary Interface
AIS  AbsInt Annotation Language for Binary Analyses
A&T  Analysis and Test
DD   Data Dictionary (TargetLink)
FMEA Failure Modes and Effects Analysis
FMECA Failure Modes, Effects, and Criticality Analysis
FTA  Fault Tree Analysis
HAZOP Hazard And Operability Study
MBAT Combined Model-based Analysis and Testing of Embedded Systems
MBT  Model-Based Testing
PHA  PHA – Preliminary Hazard Analysis
RTP  Reference Technology Platform
RVM  Requirements Verification Matrix
SUT  System Under Test
TPT  Model-based test generation tool developed by Piketec
V&V  Verification and Validation
VIT  Virtual Integration Testing
WCET Worst-Case Execution Time

Error (from MBAT glossary): Discrepancy between a computed, observed or measured value or condition (or system state) and the true, specified, or theoretically correct value or condition (or system state).

Failure (from MBAT glossary): A situation or an event where the delivered service deviates from the correct service.

Fault (from MBAT glossary): The adjudged or hypothesized cause of an error.

Model (from MBAT glossary): A semantically closed abstraction of a software or hardware system, i.e. a simplification of reality that gives a complete description a system from a particular perspective.

Model-Based Analysis: Applying static analysis methods to models of the component or system under test (SUT). Examples for analysis goals are detection of runtime-errors (e.g. memory overflow, invalid pointer dereference) or deadlocks, estimation of worst-case execution time (WCET) or worst-case memory consumption. Models can be derived from source code.

Model-Based Testing: Testing based on a model of the component or system under test (SUT), eg, reliability growth models, usage models such as operational profiles, or behavioral models such as decision table or state transition diagrams. It includes model-based test case generation, i.e. automated generation of test data from models that can be applied against the SUT or an executable model of it (simulation) [ISTQB, 2012]
Non-functional (a.k.a extra-functional): requirement / property / behavior: A requirement that does not relate to functionality, but to attributes such as reliability, efficiency, usability, maintainability and portability [ISTQB, 2012]. See Quality Characteristics.

Note: Non-functional Requirement is defined in the MBAT glossary as “Requirement that defines a system property under which the system is required to operate or exist”; which is considered as less appropriate in the context of this document.

Quantitative Requirement / Quantitative Property: Non-functional requirement that can be specified, estimated or measured numerically, typically related to timing and resource usage.

System Quality Characteristics (According to ISO/IEC 9126-1):

- **Functionality**: The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions.
  Sub–characteristics: Suitability, Accuracy, Interoperability, Security, Functionality Compliance

- **Reliability**: The capability of the software product to maintain a specified level of performance when used under specified conditions.
  Sub–characteristics: Maturity, Fault Tolerance, Recoverability, Reliability Compliance

- **Usability**: The capability of the software product to be understood learned, used and attractive to the user, when used under specified conditions.
  Sub–characteristics: Understandability, Learnability, Operability, Attractiveness, Usability Compliance

- **Efficiency**: The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions.
  Sub–characteristics: Time Behavior, Resource Utilization, Efficiency Compliance

- **Maintainability**: The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.
  Sub–characteristics: Analyzability, Changeability, Stability, Testability, Maintainability Compliance

- **Portability**: The capability of the software product to be transferred from one environment to another.
  Sub–characteristics: Adaptability, Installability, Co-Existence, Replaceability, Portability Compliance
10 References


[Boulanger(a), 2012] Jean-Louis Boulanger (Ed); Industrial Use of Formal Methods – Formal Verification; Wiley 2012

[Boulanger(b), 2012] Jean-Louis Boulanger (Ed); Industrial Use of Formal Methods – From Model to Code; Wiley 2012


EC-61882, Hazard and operability studies (HAZOP studies) - Application guide, International Electrotechnical Commission; 2001

European Cooperation for Space Standardization; ECSS-Q-30-02B Space Product Assurance, Failure modes, effects (and criticality) analysis (FMEA/FMECA); 21 July 2008

European Cooperation for Space Standardization; ECSS-Q-40-12A, Space Product Assurance; Fault Tree Analysis - Adoption Notice ECSS/IEC 61025 (14 OCT 1997)

Avner Engel; Verification, Validation, and Testing of Engineered Systems; Wiley 2010


F. Elberzhager, J. Muench, V. Tran; A Systematic Mapping Study on the Combination of Static and Dynamic Quality Assurance Techniques; Information and Software Technology, 54(1), 2012

Jeff A. Estefan; Survey of Model-based Systems Engineering (MBSE) Methodologies; INCOSSE MBSE Focus Group. 2007


Foster et al.; Guidelines for Creating a Formal Verification Testplan; in Proc. DVCon, 2006

IEEE 1012-2004; IEEE Standard for Software Verification and Validation*, 2004


IEC 61025 ed.2, Fault tree analysis (FTA), 2006


ISO 26266; Road vehicles – Functional safety; 2011.

ISTQB (International Software Testing Qualifications Board); Standard glossary of terms used in Software Testing, version 2.2; 2012


Karmeh et al.; A Design-for-verification Framework for a Configurable Performance-Critical Communication Interface; In FORMATS 2011

<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBAT1, 2012</td>
<td>MBAT Consortium; D_WP1.5.1 Aerospace use cases &amp; demonstrators specification, 2012</td>
</tr>
<tr>
<td>MBAT2, 2012</td>
<td>MBAT Consortium; D_WP3.3.1.1 RTPv0 (Fast Track Version), 2012</td>
</tr>
<tr>
<td>MBAT3, 2012</td>
<td>MBAT Consortium; D_WP3.1.1.2 Meta Models for RTP V1, 2012</td>
</tr>
<tr>
<td>MBAT4, 2012</td>
<td>MBAT Consortium; D_WP1.4.1 Automotive use cases &amp; demonstrators specification, 2012</td>
</tr>
<tr>
<td>Peleska, 2012</td>
<td>Jan Peleska; Integrated and Automated Abstract Interpretation, Verification and Testing of C/C++ Modules; Concurrency, Compositionality and Correctness, LNCS 5930.</td>
</tr>
</tbody>
</table>

