TESSY and ISO 26262
How TESSY helps to achieve ISO 26262 compliance

TESSY White Paper
Author: Frank Büchner
Preface
My White Papers detail specific topics related to methods, tools, and other aspects of embedded software quality.


Abstract: ISO 26262 recommends methods for unit and integration testing of automotive software. This paper discusses how the unit testing tool TESSY helps to apply these methods and other requirements of ISO 26262.

Version: 2017-09-08 - 001

Author:
Frank Büchner
Dipl.-Inform.
Principal Engineer Software Quality
Hitex GmbH
Greschbachstr. 12
D-76229 Karlsruhe
Tel.: +49-721-9628-125
frank.buechner@hitex.de
## Contents

1. Introduction 5

1.1. ISO 26262 5
   1.1.1. Part 6 5
   1.1.2. Part 8 5
   1.1.3. Tables in ISO 26262 5
   1.1.4. Disclaimer 5

1.2. TESSY 5

2. Basics 5

2.1. What Is a Software Unit? 5

2.2. To Stub or Not To Stub? 6

3. Requirements for Software Unit Testing 7

3.1. Methods for Software Unit Testing 7
   3.1.1. Requirements-based test 7
   3.1.2. Interface test 8
   3.1.3. Fault Injection Test 10
   3.1.4. Resource Usage Test 17
   3.1.5. Back-to-back Comparison Test 18

3.2. Methods for Deriving Test Cases for Software Unit Testing 19
   3.2.1. Analysis of Requirements 19
   3.2.2. Generation and Analysis of Equivalence Classes 19
   3.2.3. Analysis of Boundary Values 21
   3.2.4. Error Guessing 21
   3.2.5. TESSY and the Methods for Deriving Test Cases for Software Unit Testing 22

3.3. Structural Coverage Metrics at the Software Unit Level 24
   3.3.1. Statement Coverage 24
   3.3.2. Branch Coverage 25
   3.3.3. Modified Condition/Decision Coverage (MC/DC) 25

3.4. Test Environment Close to Target Environment 27

4. Requirements for Software Integration Testing 28

4.1. Methods for Software Integration Testing 28
   4.1.1. Requirements-based test 28
   4.1.2. Interface test 28
4.1.3. Fault Injection Test
4.1.4. Resource Usage Test
4.1.5. Back-to-back Comparison Test
4.2. Methods for Deriving Test Cases for Software Integration Testing
4.3. Structural Coverage Metrics at the Software Architectural Level
   4.3.1. The Difference Between Function Coverage and Call Coverage
   4.3.2. Function Coverage
   4.3.3. Call Coverage
5. Tool Qualification
   5.1. Qualification of Software Tools Classified TCL 3
      5.1.1. Increased Confidence from Use
      5.1.2. Evaluation of the Tool Development Process
      5.1.3. Validation of the Software Tool
      5.1.4. Development In Accordance with a Safety Standard
6. Versions
7. The Author
8. References
1. Introduction

1.1. ISO 26262

1.1.1. Part 6
In part 6 of ISO 26262:2011, section 9 discusses software unit testing and section 10 discusses software integration and testing, resulting in requirements for testing.

1.1.2. Part 8
In part 8 of ISO 26262:2011, section 11 discusses tool qualification.

1.1.3. Tables in ISO 26262
Several tables in these sections of the standard recommend methods that should be applied for testing. The recommendation depends on the Automotive Safety Integrity Level (ASIL). ASIL ranges from A to D, where D is the highest level (i.e. the level requiring the most effort to reduce risk). Methods that are “highly recommended” are marked by a double plus sign (“++”); methods that are “recommended” are marked by a single plus sign (“+”). Methods numbered 1a, 1b, 1c, … are alternative methods; methods numbered 1, 2, 3, … are unrelated methods.

1.1.4. Disclaimer
A big deal of this white paper interprets ISO 26262. It is common knowledge that different people interpret the standard differently on certain topics. Please keep in mind that the interpretations given in this white paper are the interpretations of me, the author, and --- besides made to the best of my knowledge and belief --- might be disputable. Errors excepted. If you have comments, corrections, different interpretations I would be glad if you would let me know.

1.2. TESSY
TESSY is a tool to automate the unit and integration testing of embedded software written in C or C++. A part of TESSY is the Classification Tree Editor (CTE) for test case specification according to the Classification Tree Method [4]. Furthermore TESSY measures code coverage. TESSY features requirement management. A Tool Qualification Package (TQP) for TESSY is available.

2. Basics

2.1. What Is a Software Unit?
Section 9.1 of part 6 of ISO 26262 states: “The objective of this sub phase is to demonstrate that the software units fulfill the software unit specifications ...”.

This leads us to the question what a software unit is. Part 1 (Vocabulary), section 1.125 of ISO 26262 defines a software unit as “atomic level software component of the software architecture that can be subjected to stand-alone testing”. If we look for the definition of “software
component”, ISO 26262 defines a software component in part 1, section 1.123 as “one or more software units”.

From this I conclude that a software unit shall not be devisable in smaller parts (because that would violate “atomic”) that can be tested stand-alone. Stand-alone means for me that a test can be performed in isolation from the rest of the software, i.e. the other software units. Literature [2] designates units as the smallest part of the software that can be sensibly tested isolated respectively independent from the rest of the software. Please note: The emphasis here is on sensible.

However, these definitions do not help us along until we relate them to a certain programming language. Because automotive software is mostly written in C or in C++, we can apply the definitions to these languages and conclude:

- A unit is a function of the C programming language
- A unit is a method of the C++ programming language

In praxis there might be units that consist of more than one function, e.g. if the function under test calls another function with the only purpose to provide a simple service to the function under test.

\[\text{Fig. 1: A single unit or two separate units?}\]

In the figure above the function is\_value\_in\_range() calls the function absolute(). In the definition strictest sense, each function forms a unit of its own. This is backed by the fact that a unit consisting of both functions is clearly not atomic. In my opinion, it is sensible to perform unit testing of absolute().

### 2.2. To Stub or Not To Stub?

However, does it make sense to test is\_value\_in\_range() in isolation from absolute()? This would require having a stub function (or dummy function) for absolute() during the test. Such a dummy function can be generated automated by TESSY, i.e. it is not the additional effort preventing us from testing is\_value\_in\_range() in isolation from absolute(). But does it make sense? Testing of is\_value\_in\_range() requires that absolute() performs correctly, whether it is a stub function or the real implementation. That might be different if the called function can provide wrong or unexpected values, which shall be handled by the calling function. E.g. a function is specified to return either 0 or 1. What happens in the calling function if e.g. the value 2 is returned? This can
be comfortably tested if the called function is replaced by a stub function, but it is hard to test if the actual implementation is used. An example for this is presented below, in the section Fault Injection from Stub Functions.

3. Requirements for Software Unit Testing
This section discusses which requirements of section 9 of ISO 26262 can be fulfilled using TESSY and how TESSY supports the methods recommended in the tables of section 9 of part 6.

3.1. Methods for Software Unit Testing
Table 10 of part 6 of ISO 26262:2011 lists the methods for software unit testing.

<table>
<thead>
<tr>
<th>Methods for software unit testing</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Requirements-based test</td>
<td>++</td>
</tr>
<tr>
<td>1b Interface test</td>
<td>++</td>
</tr>
<tr>
<td>1c Fault injection test</td>
<td>+</td>
</tr>
<tr>
<td>1d Resource usage test</td>
<td>+</td>
</tr>
<tr>
<td>1e Back-to-back comparison bet. model and code</td>
<td>+</td>
</tr>
</tbody>
</table>

*Fig. 2: ISO 26262, part 6, table 10: methods for software unit testing*

3.1.1. Requirements-based test
Method 1a of table 10 requires that the test cases are derived from the software requirements at the unit level. This is highly recommended for all ASILs. Requirement-based test is not so much a feature of a testing tool, but the approach how test cases are specified / created. Methods for test case derivation are listed in more detail in table 11 of part 6 of ISO 26262 (see below). However, TESSY provides means to support requirements-based test: (1) TESSY includes a component for test case specification according to the Classification Tree Method (CTM). This method starts by analysis of the requirements. Hence, if this method is used for test case specification, it is guaranteed that requirements-based testing is performed. (2) TESSY includes a component for requirement management that allows to link software requirements at the unit level to unit test cases. This feature allows comfortably to verify that each requirements has at least a test case assigned to it, and that this test case was executed, and that this test case has passed.
3.1.2. Interface test

Method 1b of table 10 requires interface test. This is highly recommended for all ASILs. Unfortunately, I do not know any further explanation or description in ISO 2626 for interface test. In my opinion, interface test means testing at the interface of the software unit under test. This is supported by the fact that unit testing is black-box-testing, i.e. testing without internal knowledge of the test object. The interface consists of the input variables and the output variables of the unit under test. TESSY analyses the source code of the unit under test and determines the interface.

![Interface of a unit under test in TESSY](image)

Fig. 3: Interface of a unit under test in TESSY

In the figure above the interface of a unit under test is depicted. The interface consists of a global variable \( m \) which is output from the unit under test; furthermore the interface has a parameter \( k \) which is input to the unit under test; and eventually the unit under test has a return value which is naturally output.

The interface of the unit under test determines the structure of the test cases for this unit. The input variables need to have values prior to the test and the output variables need to have expected values which will be compared to the actual values of these variables after the test. This comparison results in the pass/fail verdict of the test.
In the figure above, the input value for the parameter k is 1; the value of the output variable m is 2 (what by the way is the expected value, because the value has a green background); and the return value is 3 (what also is the expected value). Because both output variables have their expected values after the test, the test is passed (green).

Hence it is clear that unit testing as done with TESSY tests at the interface of the unit under test.

Excursus: If the unit under test calls other units, and these units have parameters and return values, these variables also belong to the interface of the unit under test. Values of variables passed from the unit under test to the called unit are output (result) of the unit under test; return values of the called function are input to the unit under test.
3.1.3. Fault Injection Test

Method 1c of table 10 requires fault injection test. This is recommended for ASIL A to C, and only highly recommended for ASIL D. ISO 26262 states that fault injection includes the injection of arbitrary faults, e.g. by corrupting values of variables in memory. If this is done arbitrary, in my opinion, fault injection testing is nothing else than robustness testing, because the outcome of such a test only shows if the test object crashes resp. hangs or ends in a safe state. For me, fault injection testing is more: it requires an exactly defined fault (that is injected) and an exactly defined result that can be evaluated for a test verdict (pass/fail). Sticking to the example of corrupted values for variables in memory: A fault injection test should change the value of a specified variable in a specified way (Change one or more bits? Set? Reset?) and should have an expected result (e.g. a RAM test algorithm detects the change and reports it correctly). That brings me to the point that fault injection testing only is sensible if the test object is prepared to detect and handle the fault. E.g. if a software unit receives a pointer as a parameter, passing a NULL pointer to the software only is reasonable if we can expect that the software unit checks the passed pointer for NULL prior to dereferencing it and reacts on a NULL pointer in a predefined way. If the software unit does not check the pointer and hence dereferences a NULL pointer, the software unit will crash and this is the only thing we can find out during unit testing (because to find out if it comes to a safe state is beyond unit testing). The fact that we expect the software unit to react in a predefined way to an injected fault (e.g. a NULL pointer) let us conclude that there need to be a requirement, which describes the intended reaction (e.g. setting an error number and returning false).

In the following we present as examples for fault injections that TESSY can perform:

- A simple one, the passing of a NULL pointer,
- A normal one, using stub functions, and
- A slightly complicated one, the changing of a memory cell.

Please note: Future versions of TESSY (> V4.0) might probably feature more advanced mechanism for even better possibilities for fault injection.

**Passing a NULL pointer**

If an input variable to the test object is a pointer, TESSY lets us quite comfortable set this input variable to NULL. This is done in the context menu of the Test data view. This allows us to see how the test object reacts on this input.
Fig. 6: The context menu allows setting the value of a pointer to NULL comfortably

Fault Injection from Stub Functions

typedef enum {negative, positive, zero, error} ret_type;

ret_type func(int value)
{
    switch (signum(value))
    {
    case 0:
        return zero;
    case 1:
        return positive;
    case -1:
        return negative;
    default:
        return error;
    }
}

int signum(int val)
{
    if (val > 0)
        return 1;
    else if (val < 0)
        return -1;
    else
        return 0;
}

Fig. 7: Source code of test object func() and the function signum(), called by func()

In the figure above, the function func(), our test object for unit testing, calls the function signum(). The return value of signum() selects a label of the switch-statement and in turn decides which value is returned by func(). Because the actual implementation of signum() can only return the values -1, 0, 1, it is obvious that the value error cannot be returned. This means in consequence that you do not get 100% coverage for func() and that you cannot test if the correct value is returned. To remedy the problem, the (advanced) stub functions of TESSY can be used.
Fig. 8: TESSY is directed to create an advanced stub function for signum() in the interface editor

Fig. 9: Advanced stub functions might extend the interface by additional variables

By creating an advanced stub function for signum() the interface is extended by two variables: The return value of signum() to func() with the passing direction IN and the parameter val which is passed from func() to signum() and therefore has the passing direction OUT.
Fig. 10: The parameter val will not show up in a test case if its passing direction is set to IRRELEVANT.

We are only interested in the return value but not in the parameter, therefore we set the passing direction of the parameter to IRRELEVANT.

Fig. 11: Test case with (advanced) stub function for signum()

In the figure above a test case for func() is depicted. The test case consists of three values. The value 100 is passed as parameter to func() which in turn passes this value as parameter to signum(). Because an advanced stub function for signum() is used, and we have set the passing direction for the parameter of signum() to IRRELEVANT, the value 100 is no longer relevant. It is simply discarded. The return value of the advanced stub function for signum() is arbitrarily set to 2 (any value besides -1, 0, 1 would fulfill our purpose.) This is the fault injection. Because there is no case label for 2, the default label is used. This results in the return value error for func(), what is the expected result. Therefore the test case passes.
```c
ret_type func(int value) {
    switch (signum(value)) {
    case 0:
        return zero;
    case 1:
        return positive;
    case -1:
        return negative;
    default:
        return error;
    }
}
```

**Fig. 12: Branch coverage for the test case above**

In the figure above the coverage measurement results for branch coverage are shown. On the left hand side the source code of the test object is shown; on the right hand side the flow chart of the test object is shown. Green depicts executed; red depicts not executed parts of the test object. It is clearly visible that the branch of the default label was executed.

**Fig. 13: Three test cases using the actual implementation of signum() (no stub involved)**

The figure above shows three test cases for `func()`, where no stub function for `signum()` is used. Hence the actual implementation of `signum()` is used. (This is actually integration testing between `func()` and `signum()`) Because the actual implementation of `signum()` can only return -1, 0, 1, the return value `error` for `func()` is not possible.
Changing of Memory

```c
int WriteRead(char cPar)
{
    char volatile cVal;

    cVal = cPar;
    if (cVal == cPar)
        return 1;
    else
        return -1;
}
```

Fig. 14: To get a return value of -1, a variable in memory need to be changed

The function WriteRead() in the figure above can only return -1 if the value of a variable (either cVal or cPar) is changed in memory between the assignment of cPar to cVal and the check for equality in the if-instruction in the following line. We take the task of forcing a return value of -1 as an example for a fault injection. (We can consider WriteRead() as part of a simple RAM test algorithm which checks if memory location is working correctly by checking if a value written to the memory location is read back correctly, i.e. uncorrupted. The normal case is that the memory location is working correctly, i.e. the value written and the value read back are identical and the value 1 is returned by WriteRead(); the faulty case, which shall be detected, is that the memory location does not work correctly, i.e. the value read back is different from the value written. The detection is not very subtle, i.e. we do not know if one or more bits are different or at what bit position in the memory location the difference manifests.)

```c
#endif TESSY
#define TS_FAULT_INJECTION_FUNC(x) _TS_FAULT_INJECTION_FUNC((x))
#endif TESSY
```

```c
extern void __TS_FAULT_INJECTION_FUNC(char *TS_var);
#endif TESSY
```

```c
int WriteRead(char cPar)
{
    char volatile cVal;

    cVal = cPar;
    TS_FAULT_INJECTION_FUNC(&cVal);
    if (cVal == cPar)
        return 1;
    else
        return -1;
}
```

Fig. 15: Preparations for fault detection in the source code
In the figure above, a very simple fault injection mechanism is shown. If TESSY is defined, `_TS_FAULT_INJECTION_FUNC()` is called. TESSY is directed to create a stub function for `_TS_FAULT_INJECTION_FUNC()`.

**Fig. 16: The stub function code for _TS_FAULT_INJECTION_FUNC()**

Here the same stub function with user code is used for all test cases. The body of the stub function sets (more or less arbitrarily) the least significant bit of the value the parameter pointer points to (i.e. the value of the variable cVal) to 0. This means depending on the initial value of cVal, the value of cVal is changed by the stub function (simulating memory not working correctly) or not (simulating correctly working memory). Different approaches for corrupting data are possible (e.g. different stub function bodies depending on the test case number). You may even use advanced stub functions.

**Fig. 17: Test report for two test cases for WriteRead()**
We run two test cases to demonstrate the different behavior either if the fault is injected or not. In the first test case, where the initial value of cVal is 0x41, the value of cVAL is set to 0x40 in the stub function (this is the injection of the fault simulating memory not working correctly) and therefore the return value of WriteRead() is -1. In the second test case, where the initial value of cVal is 0x42, the value of cVal is not changed in the stub function, hence no fault is injected and therefore 1 is returned by WriteRead().

The example above was exercised using TESSY V4.0. Later versions of TESSY are expected to feature an improved fault injection mechanism.

3.1.4. Resource Usage Test

Method 1d of table 10 requires fault injection test. This is recommended for ASIL A to C, and only highly recommended for ASIL D.

Unfortunately, I cannot find a place in ISO 26262 where ISO 26262 does detail the usage of which kind of resources shall be tested. It could be (execution) time, or (memory) space, or (consumption of) current, or any other resource. Furthermore, in my opinion, the basic objective for software unit testing is to test if certain input values result in the expected output values and not resource usage testing. I consider software unit testing not appropriate for resource usage testing. In my opinion, this is more a task of static source code analysis. However, for some special environments TESSY is able to measure execution time.

**Execution Time Measurement**

For selected environments (microcontroller, software debugger) and under some technical prerequisites TESSY is able to measure the execution time of test cases.

Please note: This feature is not generally available.

Please note: There is a little (constant) overhead in the measurement. If the test case execution time is very short, this overhead may distort the result.

---

![Fig. 18: This application note describes execution time measurement with TESSY](image-url)
If time measurement is enabled, TESSY measures the execution times of the test cases for a certain test object by starting a timer prior to the call to the test object and stopping the timer after the return of the test object. TESSY displays all times in the test report and determines the longest and the shortest execution time.

<table>
<thead>
<tr>
<th>Project:</th>
<th>Selectionsort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module:</td>
<td>Tasking167</td>
</tr>
<tr>
<td>Testobject:</td>
<td>Selectionsort</td>
</tr>
</tbody>
</table>

![Teststep Time Summary](image)
- **Longest (3.1):** 18575.20 ys
- **Shortest (1.1):** 50.80 ys

![Time Measurements](image)

<table>
<thead>
<tr>
<th>Teststep</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>50.80 ys</td>
</tr>
<tr>
<td>2.1</td>
<td>835.20 ys</td>
</tr>
<tr>
<td>3.1</td>
<td>18575.20 ys</td>
</tr>
<tr>
<td>4.1</td>
<td>50.80 ys</td>
</tr>
</tbody>
</table>

**Fig. 19: Excerpt from a test report with execution time measurement**

Please note: This execution time measurement does not necessarily determine the Worst Case Execution Time (WCET). This is because only the execution times of the actually executed test cases are measured, and if none of these test executions need the WCET, the WCET is not detected. Don’t be fooled by execution time measurement.

### 3.1.5. Back-to-back Comparison Test

Method 1e of table 10 requires back-to-back comparison test between model and code, if applicable. This is recommended for ASIL A and B, and highly recommended for ASIL C and D.

Obviously, to apply this method, a model is needed which simulates the behavior of the software unit. The idea is to compare the behavior of the model and the behavior of the software unit when both are stimulated the same way.

Actually quite often such models are generated e.g. in Simulink, and then the model is tested, generating test output data for certain inputs. If the model works correctly, source code is generated from the model. Often the generated code uses fixed-point arithmetic, whereas the model might use floating point arithmetic. So it makes sense to compare the output data generated during the testing of the model with the output data generated by the source code of the model, naturally for the same input data (stimulus).

**Data Transfer by Files**

TESSY users do this usually by using the possibility to import test data for unit testing from files, especially from Excel workbooks. I.e. the data from the simulation are transferred into Excel workbooks and the data from the Excel workbook is imported into TESSY.
3.2. Methods for Deriving Test Cases for Software Unit Testing

Table 11 of part 6 of ISO 26262:2011 lists the methods for deriving test cases for software unit testing.

<table>
<thead>
<tr>
<th>Methods for deriving test cases for software unit testing</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Analysis of requirements</td>
<td>++</td>
</tr>
<tr>
<td>1b Generation and analysis of equivalence classes</td>
<td>+</td>
</tr>
<tr>
<td>1c Analysis of boundary values</td>
<td>+</td>
</tr>
<tr>
<td>1d Error guessing</td>
<td>+</td>
</tr>
</tbody>
</table>

Fig. 20: ISO 26262, part 6, table 11: Methods for deriving test cases for software unit testing

These methods are black-box test case specification methods, i.e. they can (and should) be applied without knowledge of the internals of the test object. In the case of software unit testing this means without knowledge of the source code.

3.2.1. Analysis of Requirements

Method 1a of table 11 requires that the test cases for software unit testing are derived from the requirements. This is highly recommended for all ASILs. Method 1a of table 11 stands in close relationship to method 1a of table 10. Both methods put the requirements in the center of test case derivation and testing. This is the naïve approach: If something (a piece of software, for instance) needs to fulfill some requirements, the test cases naturally shall test if the requirements are fulfilled. This seems so obvious that you might think it is not worth to mention it. However, I have seen that the source code of a test object was used to derive test cases. This is dangerous because of two reasons: (1) You will not detect omissions in the code. E.g. if a requirement specifies a certain check (e.g. “if both parameters are equal, return false”) but the software does not implement the check, this will never be detected if only the source code forms the basis of test case derivation. (2) You suppose the code to be correct. E.g. if a statement in the software is “if (a>5)”, how will you detect that this is wrong and the correct statement is “if (a>=5)” from (1) and (2) you can conclude that you are on thin ice if you derive test cases from the code. And, furthermore, deriving test cases from the code is not included in table 11, i.e. it is not conformant to ISO 26262.

3.2.2. Generation and Analysis of Equivalence Classes

Method 1b of table 11 requires that generation and analysis of equivalence classes is used to derive test cases for software unit testing. This is recommended for ASIL A and highly recommended for ASIL B to D.
Generation and analysis of equivalence classes is also called “equivalence partitioning”. Equivalent partitioning solves the problem that an input variable can take on too many values, and it is not possible / not efficient to use all values in test cases, especially in combination with many values of other input variables (“combinatorial explosion”). Equivalence partitioning divides all input values into classes. Values are assigned to the same class, if the values are considered equivalent for the test. Equivalent for the test means that if one value out of a certain class causes a test to fail and hence reveals an error, every other value out of this class will also cause the same test to fail and will reveal the same error.

In other words: It is not relevant for testing which value out of a class is used for testing, because they all are considered to be equivalent. Therefore, you may take an arbitrary value out of a class for testing, even the same value for all tests, without decreasing the relevance of the tests. However, the prerequisite for this is that the equivalence partitioning was done correctly, what is in the responsibility of the person applying equivalent partitioning.

Fig. 21: Example for equivalence partitioning (according to shape)

The figure above shows an example for equivalent partitioning. The upper part of the figure above shows the input space, consisting of geometric objects of different size, different color, and different shape. The lower part of the figure above shows the partitioning in three classes according to the shape.

Remarks:

- Equivalent for the test does not necessarily mean that the result of the test (e.g. a calculated value) is the same for all values in a class.
- Equivalence partitioning must be complete in the mathematical sense: Every possible value of an input variable must be assigned to a class.
- Equivalence partitioning must be unique in the mathematical sense: A value of an input variable must be assigned to a single class, and not to several classes.
• Equivalence partitioning can also be applied if only few input values exist, e.g. a switch having on and off as possible values. For this example, equivalence partitioning would result in two classes (“switch on” and “switch off”). In this case each class would have one value only.

3.2.3. Analysis of Boundary Values

Method 1c of table 11 requires analysis of boundary values is used to derive test cases for software unit testing. This is recommended for ASIL A and highly recommended for ASIL B to D.

The naive message of the method is to use boundary values for test cases. However, this requires that the possible values are ordered in some way.

**Fig. 22: Naïve interpretation of “boundary value”: The values in red are boundary values**

In the figure above a range of values (on a number line) is specified by the sign ‘[‘. If one wants to test if a given value is inside this range, the values marked in red are boundary values. Using the boundary values result in “better” test cases than using the values in the middle of the range (in green). This is because the boundary values have the potential to detect defects of “<” vs. “≤” and the like, whereas values in the middle have not. (Test cases using boundary values are considered to be more “error-sensitive” than those using normal values.)

However, in my opinion, the term “boundary value” is too restrictive. (Apart from this, a boundary requires an order of the values. And what order do a handful of colors have, for instance?) In my opinion, test cases should also use invalid / unexpected values (if those values exist) and “extreme” values. E.g. if an input value for a test object is supposed to have only the two values 0 and 1, what happens if the value 2 is passed to the test object? The value 2 is invalid and might be unexpected by the test object. Consider the input to a sorting function: If all values to sort are equal, this would be an extreme input. Those tests are more error-sensitive than tests using “normal” inputs.

3.2.4. Error Guessing

Method 1d of table 11 requires error guessing is used to derive test cases for software unit testing. This is recommended for all ASILs.

Error guessing usually requires an experienced tester who is able to find error-sensitive test cases from experience. Hence, it is usually an unsystematic method (opposed to the first three methods). [I admit, you could use checklists or failure reports of previous systems or something similar as basis for guessing.] Error guessing is related to the section above, because thinking about possible invalid / unexpected / extreme test cases actually is error guessing. If a system under test has two buttons, and it is supposed that only one of these buttons is pressed at a time:
What happens if the two buttons are pushed simultaneously? Can a button be pushed too fast / too often / too long? These are examples for error guessing.

3.2.5. TESSY and the Methods for Deriving Test Cases for Software Unit Testing

TESSY includes the Classification Tree Editor (CTE), a tool supporting the Classification Tree Method (CTM). The CTM is a method for test case specification.

The CTM starts by analyzing the requirements. This is conformant to method 1a of table 11. The requirements determine which inputs are relevant (i.e. which inputs should be variated, i.e. which inputs should have different values during the test). In the next step the possible input values are divided into classes according to the equivalence partition method. This is conformant to method 1b of table 11. The third step is to consider boundary / extreme / invalid input values. This is conformant to method 1c of table 11, and, to some extent, also to method 1d. These three steps result in the classification tree. The classification tree forms the upper part of the (graphical) representation of the test case specification according to the CTM. The root of the tree is at the top; the tree grows from top to bottom; classifications have frames; classes are without frames; the leaf classes form the head of the combination table. The combination table consists of lines, each line specifying a test case. The purpose of a test case is determined by markers on the respective line, each marker selecting an equivalence class, from which the value for the test is taken. A human draws the tree, giving names to classifications and classes, and sets the markers on the test case line, i.e. test case specification is a human activity (subject to human error, unfortunately).

![Classification Tree](image)

*Fig. 23: Test case specification according to the Classification Tree Method (CTM)*
In the figure above an example for the test case specification according to the classification tree method is given. The root of the tree is labelled “Suspension”, i.e. the test object obviously is a suspension. Also quite obviously, two inputs are relevant for the test: “Speed” and “Steering Angle”. “Speed” and “Steering Angle” are classifications (in frames), at the topmost level also called “test relevant aspects”. Both classifications are divided into equivalence classes (not in a frame). For “Steering Angle” there are three equivalence classes: “left”, “central”, and “right”. From the classification tree we cannot conclude which values are inside a certain class, e.g. “left”, and how the values are represented. This is implementation dependent, and the CTM being a black-box test specification method, not relevant for the CTM. (The test case specification is abstract.) If one don’t takes “central” as an extreme steering angle position, no boundary / invalid / extreme values are forced for the steering angle. This is different for “speed”. The classification “speed” is divided into the two equivalence classes “valid” and “invalid”. The latter class guarantees that the invalid values for speed will be used during testing, because in a valid specification according to the CTM, all classes that were present in the tree need to be used in one test case specification at least. The class “invalid” is divided again using the classification “Too low or too high?”. This results in additional classes “negative” and “> v_max”. Test cases using values from these two classes will find out what happens if the unexpected hits the (software) test object. The valid speeds are divided into “normal” speeds and “extreme” speeds. We can assume that the class “zero” for a valid speed contains only one value (probably the value 0), as the class “v_max” which probably contains the maximum speed as specified in the requirements.

The combination table (the lower part of the figure above) consists of seven lines and, hence, specifies seven test cases. The test case specifications are given names and the markers set on each line indicate, which values from which classes shall be combined. In our case this is also indicated by the name of the test case specification, but this does not need to be the case always.

From the test case specification it is clearly visible that there are only three “normal” test cases (the first three test cases). For instance, a test case specification that requires testing e.g. low speed with the steering angle right does not exists. If you feel that three normal test case specifications are not enough, you might opt to add an additional one. However, the question is not if three is enough; the point is that it is obvious for everyone that there are only three. This is an important advantage of the CTM.

What is the difference between a test case specification and a (concrete) test case? The outcome of the CTM are test case specifications, aka abstract test cases. We only know that e.g. a “left” steering angle shall be used in a test case. We do not know how “left” is represented (e.g. in the source code). Is the steering angle given in degrees? Or in radian measure? Can it have a sign? If we assign a concrete value (e.g. -45 degrees) that represents left steering angle to the test case, we have made a concrete test case out of an abstract one.
3.3. Structural Coverage Metrics at the Software Unit Level

Table 12 of part 6 of ISO 26262:2011 lists the structural coverage metrics at the software unit level.

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Statement coverage</td>
<td>++</td>
</tr>
<tr>
<td>1b Branch coverage</td>
<td>+</td>
</tr>
<tr>
<td>1c MC/DC (Modified Condition/Decision Coverage)</td>
<td>+</td>
</tr>
</tbody>
</table>

*Fig. 24: ISO 26262, part 6, table 12: Structural coverage metrics at the software unit level*

3.3.1. Statement Coverage

Method 1a of table 12 lists statement coverage as structural coverage metric at the software unit level. Statement coverage is highly recommended for ASIL A and B; statement coverage is recommended for ASIL C and D.

If all statements of a software unit are executed you reach 100% statement coverage.

TESSY measures statement coverage.

*Fig. 25: Statement coverage measurement results in TESSY (two excerpts from a test report)*
3.3.2. Branch Coverage

Method 1b of table 12 lists branch coverage as structural coverage metric at the software unit level. Branch coverage is highly recommended for ASIL B and C; branch coverage is recommended for ASIL A and D.

If all branches of a software unit are executed you reach 100% branch coverage. An if-instruction, for instance, always has two branches, even if the else-branch is not implemented.

TESSY measures branch coverage based on primitive (aka essential) branches.

![Diagram of a function with branch coverage measurement](image)

*Fig. 26: Branch coverage measurement in TESSY: Source code, flowchart, result*

In the figure above, the source code of the test object is displayed. One test case was executed, and for this test case obviously the value of the variable decision was true. Highlighting in red and green shows which branches were executed and which not. Please note the ‘}’ in red. This means that the else-branch of the if-instruction was not executed. The flowchart on the right hand side displays the else-branch in red for the same reason. And eventually 50% is the numeric result of branch coverage measurement.

3.3.3. Modified Condition/Decision Coverage (MC/DC)

Method 1c of table 12 lists MC/DC as structural coverage metric at the software unit level. MC/DC coverage is recommended for ASIL A to C and highly recommended for ASIL D.
MC/DC takes into account if a decision consists of conditions, separated by logical operators.

Fig. 27: Modified Condition/Decision Coverage (MC/DC) takes conditions into account

If all conditions in a decision influence the overall outcome of the decision sufficiently, you achieve 100% MC/DC. For a decision with n conditions you need n+1 test cases to get 100% MD/DC.

TESSY measures MC/DC based on the executed test cases.

```c
if ((v1 < r1.range_start) || (v1 >= (r1.range_start + r1.range_len)))
    return no;
```

Fig. 28: Decision

Fig. 29: Two test cases are executed
Fig. 30: Modified Condition/Decision Coverage (MC/DC) result

For a given set of test cases TESSY measures MCDC. In the example in the figures above the decision consist of two conditions. Therefore, we need three test case at least to reach 100% MC/DC. Only two test cases were executed, resulting in 66% MC/DC. For a given set of test cases TESSY does not only calculate MC/DC, but also displays the test cases that would extend this set so that 100% MC/DC would be reached. This is depicted in the figure above: We need an additional test case where the first condition is false and the second condition is true to reach 100% MC/DC.

3.4. Test Environment Close to Target Environment

Section 9.4.6 of part 6 of ISO 26262:2011 states:

The test environment for software unit testing shall correspond as closely as possible to the target environment. If the software unit testing is not carried out in the target environment, the differences in the source and object code, and the differences between the test environment and the target environment, shall be analysed in order to specify additional tests on the target environment during the subsequent test phases.

TESSY can execute tests in the target environment. For these kind of tests the actual cross-compiler for the embedded system in question is used (i.e. version number of the compiler, optimization level, etc. are identical). These kind of tests can also execute on the actual target microcontroller in the actual embedded system. This means there is no need to rely on the correctness of a simulation environment for the target hardware and there is also no need to analyze the differences in the source and object code, and the differences between the test environment and the target environment. And of course, no additional tests on the target environment during the subsequent test phases are needed.
### 4. Requirements for Software Integration Testing

This section discusses which requirements of section 9 of ISO 26262 can be fulfilled using TESSY and how TESSY supports the methods recommended in the tables of section 9 of part 6.

#### 4.1. Methods for Software Integration Testing

Table 13 of part 6 of ISO 26262:2011 lists the methods for software integration testing.

<table>
<thead>
<tr>
<th>Methods for software integration testing</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Requirements-based test</td>
<td>++</td>
</tr>
<tr>
<td>1b Interface test</td>
<td>++</td>
</tr>
<tr>
<td>1c Fault injection test</td>
<td>+</td>
</tr>
<tr>
<td>1d Resource usage test</td>
<td>+</td>
</tr>
<tr>
<td>1e Back-to-back comparison betw. model and code</td>
<td>+</td>
</tr>
</tbody>
</table>

**Fig. 31: ISO 26262, part 6, table 13: Methods for software integration testing**

Table 13 is very similar to table 10 with the exception of the recommendation for method 1c (Fault injection test) at ASIL C.

#### 4.1.1. Requirements-based test

Method 1a of table 13 requires that the test cases are derived from the software requirements at the architectural level. This is highly recommended for all ASILs.

With respect to TESSY, the comments to section 3.1.1 apply also to method 1a of table 13.

#### 4.1.2. Interface test

Method 1b of table 13 requires interface test. This is highly recommended for all ASILs.

In TESSY, the test objects of the integration test are called components. The interfaces of components are displayed in two different forms in TESSY.
Fig. 32: Interface of a component in the Interface Editor perspective

The interface of a component has sections like “Component Functions” and “Static Functions” which are not present in the interface of a unit.

Fig. 33: Interface of a component in the Scenario perspective (Component Functions view)

Fig. 34: Interface of a component in the Scenario perspective (External Function Calls view)
4.1.3. Fault Injection Test

Method 1c of table 13 requires fault injection test. This is recommended for ASIL A and B; and highly recommended for ASIL C and D. (This is different to table 10.)

For the discussion of fault injection with TESSY, please see section 3.1.3.

4.1.4. Resource Usage Test

Method 1d of table 13 requires resource usage test. This is recommended for ASIL A to C; and only highly recommended for ASIL D.

To my knowledge, TESSY does not support resource usage test for integration testing.

4.1.5. Back-to-back Comparison Test

Method 1e of table 10 requires back-to-back comparison test between model and code, if applicable. This is recommended for ASIL A and B; and highly recommended for ASIL C and D.

To my knowledge, TESSY does not support the import of test data from files for integration testing and therefore does not support back-to-back comparison test for integration testing.

4.2. Methods for Deriving Test Cases for Software Integration Testing

Table 14 of part 6 of ISO 26262:2011 lists the methods for deriving test cases for software integration testing.
### Methods for deriving test cases for software integration testing

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Analysis of requirements</td>
<td>++</td>
</tr>
<tr>
<td>1b Generation and analysis of equivalence classes</td>
<td>+</td>
</tr>
<tr>
<td>1c Analysis of boundary values</td>
<td>+</td>
</tr>
<tr>
<td>1d Error guessing</td>
<td>+</td>
</tr>
</tbody>
</table>

*Fig. 36: ISO 26262, part 6, table 14: Methods for deriving test cases for software integration testing*

The methods listed in table 14 are identical with the methods for unit testing in table 11 and also the recommendations with respect to the ASIL are identical.

With respect to TESSY: All said about the methods with respect to unit testing also applies to integration testing. Especially you can use the CTE to specify test cases for integration testing.

### 4.3. Structural Coverage Metrics at the Software Architectural Level

Table 15 of part 6 of ISO 26262:2011 lists the structural coverage metrics at the software architectural level.

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Function coverage</td>
<td>++</td>
</tr>
<tr>
<td>1b Call coverage</td>
<td>+</td>
</tr>
</tbody>
</table>

*Fig. 37: ISO 26262, part 6, table 15: Structural coverage metrics at the software architectural level*

#### 4.3.1. The Difference Between Function Coverage and Call Coverage

```c
int f0(void) {
    return(divide(7,3));
}
int f1(void) {
    return(divide(7,0));
}
```

```c
int divide(int a, int b) {
    return(a/b);
}
```

*Fig. 38: Difference between function coverage and call coverage*
The figure above shows the difference between function coverage and call coverage. For function coverage a function (unit) is covered if it is called at least once. I.e. for the function divide() in the figure above it is sufficient for 100% function coverage that it is called only from function f0(). For call coverage a function (unit) is covered if it is called from all locations from which this is possible. I.e. the function divide() in the figure above needs to be called (at least) both from function f0() and f1() to reach 100% call coverage. It is obvious that the call from f1() will cause a division by zero and therefore detects a problem in the code, whereas a call from f0() does not. Hence, it is obvious that call coverage needs more effort but has a higher potential to detect defects in the code.

Call coverage is also called “call pair coverage”.

4.3.2. Function Coverage

Method 1a of table 15 lists function coverage as structural coverage metric at the software architectural level. Function coverage is recommended for ASIL A and B; and highly recommended for ASIL C and D.

TESSY can measure function coverage.

4.3.3. Call Coverage

Method 1b of table 15 lists call coverage as structural coverage metric at the software architectural level. Call coverage is recommended for ASIL A and B; and highly recommended for ASIL C and D.

TESSY V4.0 cannot measure call coverage.
5. Tool Qualification
In part 8 of ISO 26262:2011, section 11 discusses tool qualification.

A certificate from the TÜV SÜD in Munich for TESSY exists.

Fig. 40: Certificate from the TÜV SÜD for TESSY

This certificate qualifies TESSY as a tool with Tool Confidence Level (TCL) 3 according to ISO 26262.

5.1. Qualification of Software Tools Classified TCL 3

<table>
<thead>
<tr>
<th>Methods / TCL 3</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Increased confidence from use</td>
<td>++</td>
</tr>
<tr>
<td>1b Evaluation of the tool development process</td>
<td>++</td>
</tr>
<tr>
<td>1c Validation of the software tool</td>
<td>+</td>
</tr>
<tr>
<td>1d Development in accordance with a safety standard</td>
<td>+</td>
</tr>
</tbody>
</table>

Fig. 41: ISO 26262, part 8, table 4: Qualification of software tools classified TCL3

5.1.1. Increased Confidence from Use

It is possible to apply method 1a from table 4 to qualify TESSY. The certificate from the TÜV and other documents serve as a base for this. The certificate from the TÜV is freely available from the Hitex web. Other documents are available on request.

However, whether or not you can apply method 1a for your project is dependent on the project and other circumstances. Especially, it might not be possible for ASIL C and D.
5.1.2. Evaluation of the Tool Development Process
It is generally not possible to apply method 1b from table 4 to qualify TESSY.

5.1.3. Validation of the Software Tool
It is possible to apply method 1c from table 4 to qualify TESSY. A Tool Qualification Package (TQP) for TESSY exists and this package is very helpful if you need to validate TESSY. The TQP is a separate product and not included in TESSY.

5.1.4. Development In Accordance with a Safety Standard
It is not possible to apply method 1d from table 4 to qualify TESSY, because TESSY is not developed in accordance with a safety standard.

6. Versions
The initial version of this white paper was written when TESSY V4.0.17 was current and ISO 26262:2011 was valid.

7. The Author
Frank Büchner studied Computer Science at the Technical University of Karlsruhe, today the Karlsruhe Institute of Technology (KIT). Since graduating, he has spent almost thirty years working in different positions in the area of embedded systems. Over the years, he specialised in testing and software quality of embedded software and passes on his expertise regularly on congresses and seminars. He works currently at Hitex GmbH, Karlsruhe, as Principal Engineer Software Quality.

8. References
[1] www.hitex.com/tessy More on TESSY and the Classification Tree Method
<table>
<thead>
<tr>
<th><strong>Hitex Head Office, Germany</strong></th>
<th><strong>Hitex UK</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hitex GmbH</strong></td>
<td><strong>Hitex (UK) Ltd</strong></td>
</tr>
<tr>
<td>Greschbachstr. 12</td>
<td>Millburn Hill Road</td>
</tr>
<tr>
<td>76227 Karlsruhe</td>
<td>University of Warwick Science Park</td>
</tr>
<tr>
<td>Germany</td>
<td>Coventry CV4 7 HS</td>
</tr>
<tr>
<td></td>
<td>United Kingdom</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Phone:</strong></td>
<td><strong>Phone:</strong></td>
</tr>
<tr>
<td>+49-721-9628-0</td>
<td>+49-721-9628-0</td>
</tr>
<tr>
<td><strong>Fax:</strong></td>
<td><strong>Fax:</strong></td>
</tr>
<tr>
<td>+49-721-9628-149</td>
<td>+49-721-9628-149</td>
</tr>
<tr>
<td><strong>Email:</strong></td>
<td><strong>Email:</strong></td>
</tr>
<tr>
<td><a href="mailto:info@hitex.de">info@hitex.de</a></td>
<td><a href="mailto:info@hitex.de">info@hitex.de</a></td>
</tr>
</tbody>
</table>