Masterarbeit

Qualification of In-House Developed Software Tools in Accordance with ISO 26262

Benjamin Archan, BSc.

Institut für Technische Informatik
Technische Universität Graz
Vorstand: Univ.-Prof. Dipl.-Inform. Dr. Kay Römer

Betreuer und Begutachter: Dipl.-Ing. Dr. techn. Christian Kreiner
Betreuer bei *Magna Powertrain*: Adam Schnellbach, MSc.

Graz, im April 2014

Kurzfassung

Die Automobilindustrie ist bestrebt die Sicherheit ihrer Produkte stetig zu verbessern. Aufgrund der hohen Komplexität moderner Fahrzeuge und Fahrzeugsysteme, wird eine Vielzahl an Software Werkzeugen zur Unterstützung der Techniker eingesetzt. Fehlfunktionen oder fehlerhafte Ergebnisse dieser Werkzeuge können jedoch zu Fehlern innerhalb des erzeugten Produktes führen. Um dieses Risiko zu minimieren müssen Software Werkzeuge gemäß dem Sicherheitsstandard *ISO 26262-8:2011(E): Road vehicles – Functional safety – Part 8: Supporting processes* qualifiziert werden. In dieser Arbeit wird die Vorgehensweise einer Qualifizierung von Software Werkzeugen nach ISO 26262-8:2011 analysiert. Basierend auf dieser Analyse wird eine Methodik zur Qualifizierung von in-house entwickelten Software Werkzeugen erarbeitet und vorgestellt. Weiters wird die empfohlene Vorgehensweise an einem Software Werkzeug angewandt. Dabei handelt es sich um ein Verifikationswerkzeug, welches von *Magna Powertrain* entwickelt wurde. Die Anwendung der Methodik soll als Leitfaden für die praktische Umsetzung der Anforderungen des Sicherheitsstandards ISO 26262-8:2011 dienen.

Abstract

Automotive industry is endeavoring to increase passenger's safety steadily. Due to the high complexity of modern automotive products, a multitude of software tools is used to provide support to automotive engineers. However, malfunctions or erroneous outputs of software tools used within the development process of automotive systems can lead to safety critical product failures. Therefore, these tools must be qualified in accordance with the safety standard *ISO 26262-8:2011(E): Road vehicles – Functional safety – Part 8: Supporting processes*, to minimize the risk of product errors caused by malfunctioning software tools. This thesis analyzes the software tool qualification process required by ISO 26262-8:2011. Based on this analysis a methodology for the qualification of in-house developed software tools is derived and presented. Furthermore, the proposed qualification approach is applied on a software verification tool, developed by *Magna Powertrain*. The application of the methodology shall give a guidance for the practical implementation of the requirements specified by ISO 26262-8:2011.

Key words: software tool qualification, ISO 26262, functional safety, automotive

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....	....
date	(signature)

Danksagung

Diese Masterarbeit wurde im Sommersemester 2014 am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

Ein herzliches Dankeschön gilt meinen beiden Betreuern Adam Schnellbach von *Magna Powertrain* und Christian Kreiner vom Institut für Technische Informatik, die mich aufgrund ihres fundierten Fachwissens und ihrer großen Hilfsbereitschaft bestmöglich unterstützten. Stellvertretend für alle weiteren Angestellten von *Magna Powertrain*, die mir jederzeit mit Rat und Tat zur Seite gestanden sind, bedanke ich mich bei Amir Tojaga und Michael Michaelis.

Ein großes Dankeschön richtet sich an meine Frau Stefanie. Ohne ihren Beistand und ihre Geduld wäre meine Masterarbeit nie in dieser Form zustande gekommen.

Abschließend bedanke ich mich bei meinen Eltern Josef und Melitta, die mir durch ihre jahrelange Unterstützung das Studium überhaupt ermöglicht haben.

Graz, im April 2014 Benjamin Archan

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and objectives

The importance of functional safety of electrical and/or electronic systems in road vehicles is continuously growing. Therefore the international standard *ISO 26262 – Road vehicles – Functional safety* was established. In *ISO 26262-8:2011(E): Road vehicles – Functional safety – Part 8: Supporting processes* ([ISO11b]) the importance is justified as follows:

> Safety is one of the key issues of future automobile development. New functionalities not only in areas such as driver assistance, propulsion, in vehicle dynamics control and active and passive safety systems increasingly touch the domain of system safety engineering. Development and integration of these functionalities will strengthen the need for safe system development processes and the need to provide evidence that all reasonable system safety objectives are satisfied.
>
> With the trend of increasing technological complexity, software content and mechatronic implementation, there are increasing risks from systematic failures and random hardware failures. ISO 26262 includes guidance to avoid these risks by providing appropriate requirements and processes.

Clause 11 of [ISO11b] deals with the confidence in the use of software tools and requires a qualification of those software tools. The objective of a qualification is, as mentioned in [ISO11b], to provide evidence that

> the user can rely on the correct functioning of a software tool for those activities required by ISO 26262.

Magna Powertrain is one of the world leading companies with focus on automotive power train products. These products come within the scope of the functional safety standard ISO 26262. Among other process steps the safety standard ISO 26262 requires multiple phases of product verification. Such a level of verification is represented by the software module testing phase. As the name suggests, at the process of software module testing a single module of the automotive embedded software is verified. That means, it is checked in a systematic way if the software module under test (MUT) complies with its specifications. At *Magna Powertrain* the process of module testing is generic. That means, for each MUT the same working steps have to be performed. Due to the high complexity of a typical MUT, the construction of the module test environment is a significant effort. Furthermore, the handling of test cases by the module tester implies an unnecessary risk of inserting an error into the developed product. To increase the efficiency of the module testing process and to reduce the risk of human error, *Magna Powertrain* developed a software tool to automate the handling of software module tests. This tool is called ATool. To provide the evidence of the correct functioning of ATool, the tool must be qualified in accordance with [ISO11b].

Due to the lack of scientific literature concerning the performance of a software tool qualification in accordance with *ISO 26262-8:2011(E): Road vehicles – Functional safety – Part 8: Supporting processes* ([ISO11b]), the general objective of this master thesis is to acquire knowledge at this subject area. The thesis focuses on the qualification of *in-house* developed software tools. One objective is to elaborate a methodology for the qualification of in-house developed software tools. Another objective of this thesis is to apply this methodology on the software verification tool ATool as a best practice for further software tool qualifications at *Magna Powertrain*.

1.2 Outline

Chapter 2 deals with software tool qualification required by different standards and is divided into three parts. The first part deals with the requirements of the automotive safety standard ISO 26262. It provides a short overview of the standard and explains the objectives, methods and work products concerning software tool qualification required by ISO 26262. The second part of chapter 2 deals with safety standards similar to the automotive safety standard ISO 26262. It is investigated if the considered standards require a software tool qualification. For standards that require software tool qualification, the qualification methods are examined. Within the third part of chapter 2 already existing methodologies for software tool qualification are investigated and presented.

In chapter 3 a guideline for software tool qualification, required by ISO 26262, is described. Furthermore a proposal for the definition of qualification methods of in-house developed software tools is given. In addition a proposed approach for the application of a qualification is explained. Finally, the correlation between the requirements of [ISO11b], clause 11 and the presented methodology is investigated.

The practical implementation of the proposed software tool qualification approach is illustrated by the software verification tool ATool in chapter 4. The chapter is divided into three parts. Within the first section of chapter 4 the tailoring of the applied safety standard for ATool development is explained. The second part of the chapter describes the specification of the ATool qualification view model. The last section of chapter 4 concerns the validation of the tool.

The first part of chapter 5 considers the results of this master thesis. Within the second part of chapter 5 the open issues are discussed.

Chapter 2

Related work

2.1 Software tool qualification within the automotive industry

Within the automotive industry the standard *ISO 26262 – Road vehicles – Functional safety* deals with issues concerning functional safety of electrical and/or electronic systems. Chapter "Introduction" in [ISO11b] describes the standard as follows:

> ISO 26262 is the adaption of IEC 61508 to comply with needs specific to the application sector of E/E systems within road vehicles. (...) ISO 26262:
>
> - provides an automotive safety lifecycle (management, development, production, operation, service, decommissioning) and supports tailoring the necessary activities during these lifecycle phases;
> - provides an automotive-specific risk-based approach to determine integrity levels [Automotive Safety Integrity Levels (ASIL)];
> - uses ASILs to specify applicable requirements of ISO 26262 so as to avoid unreasonable residual risk;
> - provides requirements for validation and confirmation measures to ensure a sufficient and acceptable level of safety being achieved;
> - provides requirements for relations with suppliers;

Clause 11 of [ISO11b] specifies the requirements for the "Confidence in the use of software tools". The objectives of this clause are to

> provide criteria to determine the required level of confidence in a software tool (...)

and to assure that

> (...) the user can rely on the correct functioning of a software tool for those activities or tasks required by ISO 26262

In accordance with [ISO11b],

> confidence is needed that the software tool effectively achieves the following goals:
>
> - the risk of systematic faults in the developed product due to malfunctions of the software tool leading to erroneous outputs is minimized, and
> - the development process is adequate with respect to compliance with ISO 26262, if activities or tasks required by ISO 26262 rely on the correct functioning of the software tool used.

For instance, this means for the verification tool in the focus of this paper, that the risk of failing to detect errors in the object under test must be minimized adequately.

To determine the level of confidence in a software tool, the standardized evaluation approach needs to be performed. Based on the results of the evaluation, an adequate set of qualification methods needs to be applied to minimize the risk of a malfunctioning software tool. [ISO11b] provides a set of specified qualification methods, that are recommended by the standard. If the software tool qualification is performed based on methods not stated by the standard a rationale has to be given. After determining the level of confidence in a software tool and applying an adequate combination of qualification methods, a "Software tool criteria evaluation report" and a "Software tool qualification report" has to be generated. These reports must be checked in a confirmation review by an external certified institution. If this review is successful, the software tool is qualified.

2.1.1 Evaluation

To apply adequate qualification methods on a software tool, the confidence in that tool has to be evaluated. Therefore [ISO11b] requires the evaluation of two criteria. The first criteria concerns the impact of a possible software tool malfunction on the product being developed. The levels of such a Tool Impact (TI) are shown in Table 2.1. The second criteria for evaluation aims at the detection and prevention probability of an erroneous SW tool output. The levels of the corresponding index number, the Tool error Detection (TD), are shown in Table 2.2. When determining the TD level, it has to be considered that measures that prevent or detect erroneous output can be external to the software tool (e.g. process steps) or internal to the software tool (e.g. rationality checks).

After evaluating the levels of TI and TD, the Tool Confidence Level (TCL) can be determined in accordance to Table 2.3. In case of TCL1, no qualification methods are needed by the standard.

Level of Tool Impact	*Possibility of impact**
TI1	An argument exists, that there is no such possibility
TI2	All other cases

Table 2.1: Criteria for determination of TI levels (*Possibility that a malfunction of a particular software tool can introduce or fail to detect errors in a safety-related item or element being developed.)

Level of Tool error Detection	*Degree of confidence**
TD1	High
TD2	Medium
TD3	Other

Table 2.2: Criteria for determination of TD levels (*Confidence in measures that prevent the software tool from malfunctioning and producing erroneous output or detect erroneous output of the software tool)

		Tool error detection		
		TD1	TD2	TD3
Tool impact	TI1	TCL1	TCL1	TCL1
	TI2	TCL1	TCL2	TCL3

Table 2.3: Determination of the Tool Confidence Level (TCL)

2.1.2 Qualification Methods

Recommendations

In requirement 11.4.6.1 in [ISO11b] appropriate sets of qualification methods, based on the determined TCL, are shown. These tables from the standard are depicted in Table 2.4 and Table 2.5. To interpret Table 2.4 and Table 2.5 in the right way, the following issues have to be considered:

- Alternative methods are indicated by a number followed by a letter (e.g. 1c)
- ”+” means a method is recommended for the correspondent ASIL
- ”++’ means a method is highly recommended for the correspondend ASIL

For instance method ”1d Development in accordance with a safety standard” is highly recommended for ASIL C, but not oblige. If this method is not applied to ASIL C, a rationale must be given. [ISO11b] even allows the application of appropriate methods that are not listed, therefore also a rationale is required. However, [ISO11b] states, that

> If methods are listed with different degrees of recommendation for an ASIL, the methods with the higher recommendation should be preferred.

Methods		*ASIL*			
		A	B	C	D
1a	Increased confidence from use	++	++	+	+
1b	Evaluation of the tool development process	++	++	+	+
1c	Validation of the software tool	+	+	++	++
1d	Development in accordance with a safety standard	+	+	++	++

Table 2.4: Qualification of software tools classified TCL3

Methods		*ASIL*			
		A	B	C	D
1a	Increased confidence from use	++	++	++	+
1b	Evaluation of the tool development process	++	++	++	+
1c	Validation of the software tool	+	+	+	++
1d	Development in accordance with a safety standard	+	+	+	++

Table 2.5: Qualification of software tools classified TCL2

Explanation

Increased confidence from use (11.4.7 in [ISO11b]): This method is suitable for software tools that have been in use for the same purpose with an unchanged specification and within an unchanged environment for an appropriate time. That means that sufficient and adequate data has been collected and

> the occurence of malfunctions and corresponding erroneous outputs of the software tool acquired during previous developments are accumulated in a systematic way.

If increased confidence from use is proven for a software tool, this proof is only valid for the considered version of the tool. That means, for later versions of the software tool the evidence of increased confidence from use has to be provided again.

Evaluation of the tool development process (11.4.8 in [ISO11b]): By means of an assessment it is proven that the development process of the software tool was complied with. An appropriate method for the assessment can be for instance [SIG10], for the development process [ISO08] can be applied, for instance.

Validation of the software tool (11.4.9 in [ISO11b]): Evidence is provided, that the software tool complies with its specification. If malfunctions of the software tool are detected during validation, these malfunctions shall be analyzed and measures to avoid or detect them shall be defined. Also the behavior of the software tool under anomalous operating conditions shall be analyzed.

Development in accordance with a safety standard: [ISO11b] annotates, that:

> No safety standard is fully applicable to the development of software tools. Instead a relevant subset of requirements of a safety standard can be selected.

Applicable safety standards for the application of this method are [ISO11a], [IEC10a], [RTC11] and so forth.

2.1.3 Required Tool Information for Software Tool Qualification

In subclause 11.5 of [ISO11b] the required work products for software tool qualification are listed. For a successful tool qualification a software tool criteria evaluation report and a software tool qualification report is needed.

Software tool criteria evaluation report: Contains a description of the applied evaluation methods, mentioned above, to the software tool. The description includes the intended purpose, the inputs and expected outputs and possible environmental and functional constraints of the software tool that has been evaluated. Also, measures for the detection of (known and potential) erroneous output of the software tool, identified during TCL determination should be listed. See requirements 11.4.1 to 11.4.5 and 11.4.10 in [ISO11b].

Software tool qualification report: This report gives a rationale for the applied subset of qualification methods to the software tool. Evidence for the application of the qualification methods is contained by this report, too (e.g. link to test results). See requirements 11.4.1 to 11.4.10 in [ISO11b].

Both reports need at least a link to an unambiguous depiction of the software tool that has been evaluated/qualified. Such a depiction has to include the following information about the considered software tool:

- Version number
- Configuration
- Use cases
- Execution environment (e.g. Windows 7, MATLAB R2010b, ...)
- Maximum ASIL that can be violated because of malfunctions of the software tool
- Features, functions and technical properties
- User manual, or similar

If applicable, the following information is also needed:

- Description of the expected behavior of the software tool under anomalous operating conditions
- Description of known software tool malfunctions and the appropriate safeguards, avoidance or workaround measures

2.2 Software tool qualification within other industrial sectors

According to [BW11], Table 2.6 lists standards that address safety relevant software-intensive systems. The following sections examine a subset of those standards concerning software tool qualification.

Contraction	*Industrial sector*
IEC 61508-3	Generic
ISO/IEC 15504	Generic
ISO/IEC 12207	Generic
IEC 62304	Medical
ISO 13849	Machinery
IEC 62061	Machinery
EN 50271	Detection and measurement of gases
EN 50402	Detection and measurement of gases
IEC 61511	Process industry
EN 50128	Railway applications
ISO DIS 26262-6/-8	Automotive
DO 178B	Aviation
IEC 60880	Nuclear power plants

Table 2.6: Standards for safety relevant software-intensive systems, according to [BW11]

2.2.1 Generic requirements for electrical/electronic/programmable electronic safety-related systems

The generic standard for electrical/electronic/programmable electronic (E/E/PE) safety-related systems is [IEC10a]. It deals with software within a safety-related context. [IEC10a] defines requirements concerning the lifecycle of safety-related software. The objective of [IEC10a] is to reduce the risk of hazards caused by the software implemented in a E/E/PE system to a tolerable threshold.

Requirements concerning software tools used for the development process of safety-related software are specified in clause 7.4.4. The standard distinguishes between two categories of software tools (see [IEC10b]). All tools that influence the safety-related software during runtime are called software online-support tools. According to [IEC10a] software online-support tools have to be considered as elements of the safety-related system. Tools that support phases of the software development lifecycle are defined as software offline-support tools. These software offline-support tools are broken down in three classes, shown in Table 2.7. The requirements concerning the software offline-support tool that have to be considered dependent on these tool classes. Each version of a software offline-support tool has to be qualified. [IEC10a] allows the qualification based on the confidence from use of previous versions of the considered software tool, if evidence can be provided that the new version of the software tool has no negative influence on the existing tool-chain and the new version of the software tool contains no essential unknown error.

Tool class	*Description*
T1	Tools that generate no output that contributes to the runnable code of a safety-related software (e.g. Configuration management tools)
T2	Tools that support the validation and/or verification of the safety-related software. Malfunctions in these tools can disguise errors in the safety-related software
T3	Tools that generate output that contributes to the runnable code of a safety-related software (e.g. optimizing compiler)

Table 2.7: Classes of software off-line support tools in IEC 61508

Although [IEC10a] is mentioning software tool qualification, no concrete qualification methods are required or recommended.

2.2.2 Railway applications

As shown in Table 2.6, [EN 11] deals with safety relevant software-intensive systems within railway applications. Clause 6.7 addresses the usage of software tools. The objective of this clause is providing the evidence that malfunctions of the software tool do not impact the product to be developed. Therefore, the software tools shall be classified. When comparing the content of [EN 11], clause 6.7 with the content of [IEC10a], it turns out, that both standards use the same software tool classes.

An application of software tools classified as T2 or T3 must be justified. The justification shall contain an identification of all possible errors that could impact the product. Measures to avoid those errors shall be derived. Furthermore, a specification or a user manual describing the behavior of these software tools shall exist.

For each software tool classified as T3, evidence must be provided that the tool's output is complied with its specification or that erroneous tool output is detected. This evidence can be provided by:

- the entries within a software tool chronicle of similar applications and environments;
- software tool validation;
- applying diverse redundant code, that enables a detection of malfunctions;
- correspondence with the measures caused by the determined safety integrity level;
- or further adequate methods

Furthermore, [EN 11] specifies requirements that address the programming language and the compiler. The programming language shall provide a detection of errors within the code and support the method of developing the software tool. However, the used compiler shall be validated and justified as well.

By means of a configuration management it shall be provided, that exclusively the justified versions of software tools classified as T2 and T3 are in use. Each new version of such a software tool must be justified before it is used.

When comparing [EN 11] with [ISO11b] it turns out, that the standards are very similar concerning their qualification methods. [EN 11] is even more severe by specifying requirements addressing the programming language and their used compiler. However, a more detailed specification of the software tool evaluation is given by [ISO11b].

2.2.3 Nuclear power plants

In [IEC06] clause 14 deals with the handling of software tools in nuclear power plants, clause 15 aims at the qualification and evaluation of software tools used in nuclear power plants. According to [IEC06] the requirements and methods mentioned in the standard are valid for software tools that are immediately used for software development.

[IEC06] requires, that the qualification shall be performed based on a qualification strategy. This strategy shall consider the impacts of errors in the software tool, the probability that the software tool causes an error in the safety related system and which methods diminish the impacts of an error in the software tool. Furthermore an analysis of the development methods of the software tool shall be performed. The tool documentation shall be checked concerning the suitability for tool verification and concerning the description of the tool behavior for its users. The tool shall be validated and then evaluated during a period of application. The experiences gained from usage of the software tool shall be considered, too.

The output of a software tool shall be verified in a systematic way. [IEC06] requires, that the software tool shall be either developed in accordance with its requirements concerning software development (clause 1 to 12 in [IEC06]) or evaluated in accordance with clause 15 in [IEC06]. This evaluation shall determine the ability and the measures for possible adaptions of the considered software. Furthermore the quality and the experiences gained from the usage of the software shall be evaluated. After evaluating the software a comprehensive analysis shall be performed to clarify whether the software can be used or not.

[IEC06] targets the qualification of software tools, but compared with [ISO11b] the definition of the required qualified methods is less specific.

2.3 Existing methodologies for software tool qualification

2.3.1 IEC 61508 Certification of a Code Generator

In 2008, when the standard ISO 26262 was drafted, [Glo08] was written. The paper describes a software tool certification in accordance with [IEC10a]. It must be mentioned, that within the following "qualification" is written instead of "certification". The considered software tool to be qualified is a code generator called "ASCET". The proposed approach can be seen as a basis for the qualification of software tools in accordance with a safety standard. Figure 2.1 shows the compilation of safety requirements. As can be seen, the requirements of IEC 61508 considering hardware and system aspects are rejected. The general requirements as well as the software requirements are filtered concerning their usage for a software tool qualification. Special function requirements are added to consider the specific behavior of a code generator. [Glo08] breaks down the tool qualification into six work packages:

- Project definition
- Preparation of qualification
- Preparation of documentation
- Inspection & audits
- Evaluation
- Project finish

[Glo08] describes the preparation of documentation as follows:

> Process, development, and product documentation needs to be collected. Before they are handed over to the certifying authority they need to be brought into a suitable form to substantiate the fulfilment of the IEC 61508 requirements. This work package requires by far the most effort.

As already mentioned, [Glo08] can be seen as a basis for the qualification in accordance with a safety standard. As a result, this methodology can be applied on the thesis.

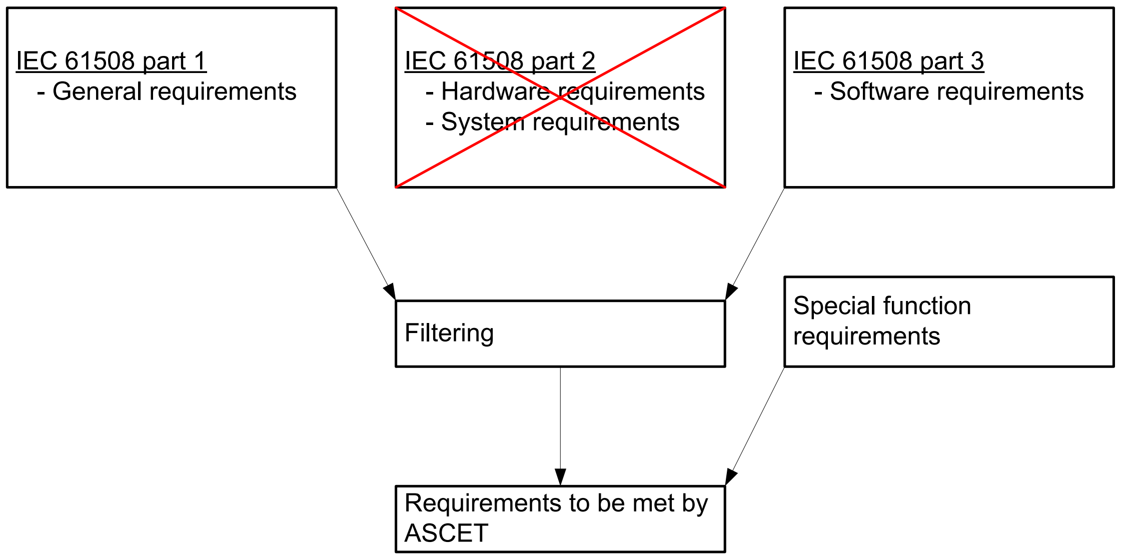

IEC 61508 part 1
- General requirements
~~IEC 61508 part 2~~
~~- Hardware requirements~~
~~- System requirements~~
IEC 61508 part 3
- Software requirements
Filtering
Special function requirements
Requirements to be met by ASCET

Figure 2.1: Preparation of qualification, according to [Glo08]

2.3.2 Qualifying Software Tools According to ISO 26262

[CMR10] describes the qualification of the MathWorks Real-Time Workshop Embedded Coder code generator in accordance with ISO 26262 by using a so called reference workflow. This reference workflow is explained by [CMR10] as follows:

> The reference workflow describes a workflow for application-specific verification and validation of models and generated code developed using the Simulink modeling environment and the Real-Time Workshop Embedded Coder C code generator. (...) The verification and validation measures described in this reference workflow form available means to detect or prevent potential malfunctions or erroneous outputs of the code generator.

As a result of establishing this process, the code generator is evaluated as TCL1 and no additional qualification methods need to be carried out. Furthermore, the qualification of the Real-Time Workshop Embedded Coder C code generator in combination with the PolySpace verifiers for C/C++ is mentioned. In case of this combination, the qualification methods "Evaluation of the development process" and "Validation of the software tool" are applied. [CMR10] states, that

> the definition of suitable verification and validation measures to be used in combination with the qualified tools provides practitioners with the necessary guidance to successfully apply Model-Based Design and advanced code verification tools in projects that need to comply with the requirements of ISO/DIS 26262.

[CMR10] was one of the first papers, that dealt with software tool qualification in accordance with ISO 26262. The paper emphasizes the key role of the qualification method "Validation of the software tool". Nevertheless the paper does not describe how the qualification methods were performed. Therefore the content of [CMR10] cannot be applied on the thesis.

2.3.3 Establishing confidence in the usage of software tools in accordance with ISO 26262

This paper presents a method to reuse and improve the processes that are related to the software tool to be qualified. As stated in [HRM+11],

> The focus of this approach is to establish a methodology that provides guidance for systematically identifying software tool malfunctions in a specific development project. (...) The methodology consists of following five phases:
>
> - Project analysis
> - Workflow analysis
> - Working step analysis
> - Use case determination
> - Identification of Tool Errors

- Analysis of Error Prevention and Detection

Each project is analyzed to obtain the workflows of a project that usually correspond with the process steps of ISO 26262. Subsequently, the working steps of each workflow are derived and the use cases of each working step are determined. After that, all possible errors of each use case need to be identified and classified. Referring to the analysis of error prevention and detection, [HRM$^+$11] states, that

> The goal of this task is to achieve a high detection probability of software tool malfunctions. The verification measures are systematically grouped into three categories:
>
> - Prevention: The error can be avoided by preventive measures due to the development process or configuration management. In an industrial context, the analysis of prevention measures must be based on existing documentation of process information.
> - Review: The error can be detected by a review of work products. In a rigorous analysis the review examines the availability of checklists for specific development steps and verifies the quality and completeness of the review protocols.
> - Test: The error can be detected by a test with another software tool within the product-specific tool chain. The analysis of tests verifies the quality of performed tests, e.g. if test cases are generated systematically.
>
> (...) In the shown example the detection probability is classified in three levels that are directly mapped to tool error detection (TD) levels, tool impact level (TI) and consequently tool confidence levels (TCL) defined in ISO 26262.

By deriving and applying adequate process measures, the software tool can be evaluated as TCL1. As a result, the application of the recommended software tool qualification methods can be eluded. The motivation of this thesis is to perform a software tool qualification. However, this method represents an approach to evaluate software tools. Therefore an application of the methodology proposed by [HRM$^+$11] on the thesis makes no sense. However, the benefit of applying this methodology for a software tool evaluation is, that in case of a subsequent qualification the correspondent development processes and tool use cases are already analyzed.

2.3.4 Test tool qualification through fault injection

A method for the qualification of software verification tools is presented by [WWI$^+$12]. The paper assumes, that the architecture of the considered software tool uses a monitor, as stated in section 7 "Case Study" within [IIW12]:

> We added a monitor to the testing tool, such that the monitor is developed to ASIL D(D) and the testing tool to QM(D). The key idea behind this decomposition is that the monitor ensures detection of testing tool failures, bringing the tool error detection to TD1. This leads to the lowest required tool confidence level TCL1, for which less qualification effort is required. While the

testing tool goes through frequent changes with re-qualification corresponding to TCL1, the monitor is not changed so often. Re-qualification to TCL3 through change management of an ASIL D Item is not required very often and there is less effort when the testing tool is changed.

By considering this monitor concept, the following is recommended for the qualification of verification tools by [WWI+12]:

> When qualifying a verification tool after a modification, using a monitor and fault injection, we iteratively apply functional stimuli that are designed to exercise the verification tool software. In each iteration we inject a known selected fault from a pre-defined fault list. If the injected fault is not detected by the monitor, one of the following actions is required.
>
> 1. Eliminate "bugs" in the verification tool, then re-start the qualification process, or
> 2. If no "bugs" can be found in the modified verification tool, analysis on conformance to the functional safety requirements will determine the next action as follows:
>   - Requirements are met and we can reduce the lists of faults to inject, or
>   - Requirements are not met and modification of the monitor is necessary to detect the "bug" undetected so far, followed by re-qualification of the monitor.
>
> If all injected faults are detected as expected in the adapted verification tool, no re-qualification is necessary.
>
> When applying the verification tool on a new system-under-test, the tool behavior is observed for each test case executed during a "golden run" (without fault injection) and then the tool is exercised with fault injection for the same test cases as in the golden run. If a mismatch is observed, the test case shall be modified.

[Izo12] explains the methodology by applying it to a verification tool, called the Universal Test Platform, testing a steer-by-wire concept classified as ASIL D. The manual justifies the usage of fault injection for the qualification of a verification tool because of the following reasons:

- Extensively used by the certification authorities in judging the system functionality with respect to safety.
- Required verification method for ASIL D development.
- Simple and easy to manage and can be efficiently optimized.
- Does not depend so much on the system complexity as the other verification methods.
- Efficiency steadily increases with the number of faults injected.
- Nondisruptive if implemented in software.

Figure 2.2, shows the fundamental architecture of a verification tool implementing the monitor and fault injection concept. The buffer is monitored to avoid data losses caused by a buffer overflow. Also the I/O port between the test procedure and the test manager is monitored. The fault injection into the "IO Manager" and the "Test Procedure" represents a white box test, because of corrupting the code of the tool. However, the fault injection considering the "Analog generator", the "PWM Capture" and the "CAN Engine" represents a black box test, because of modifying the signals.

When analyzing this qualification methodology it turns out, that it can be applied on the thesis partially. The considered method for a software tool qualification can be applied solely on tools that make use of the monitor concept. Assuming a software tool that is already qualified for ASIL B needs to be qualified for ASIL D. That means, as a first step the software tool has to be adapted by implementing a monitor. Such a fundamental change on a software tool is likely to be extremely time-consuming and therefore no option for the automotive industry. In case of a software tool that has to be developed, the monitor concept by [IIW12] and the subsequent qualification by means of fault injection as proposed by [WWI$^+$12] represents a smart way to design and qualify a software tool for the automotive industry.

However, the usage of fault injection without using a monitor is integrated in the thesis and explained in chapter 3

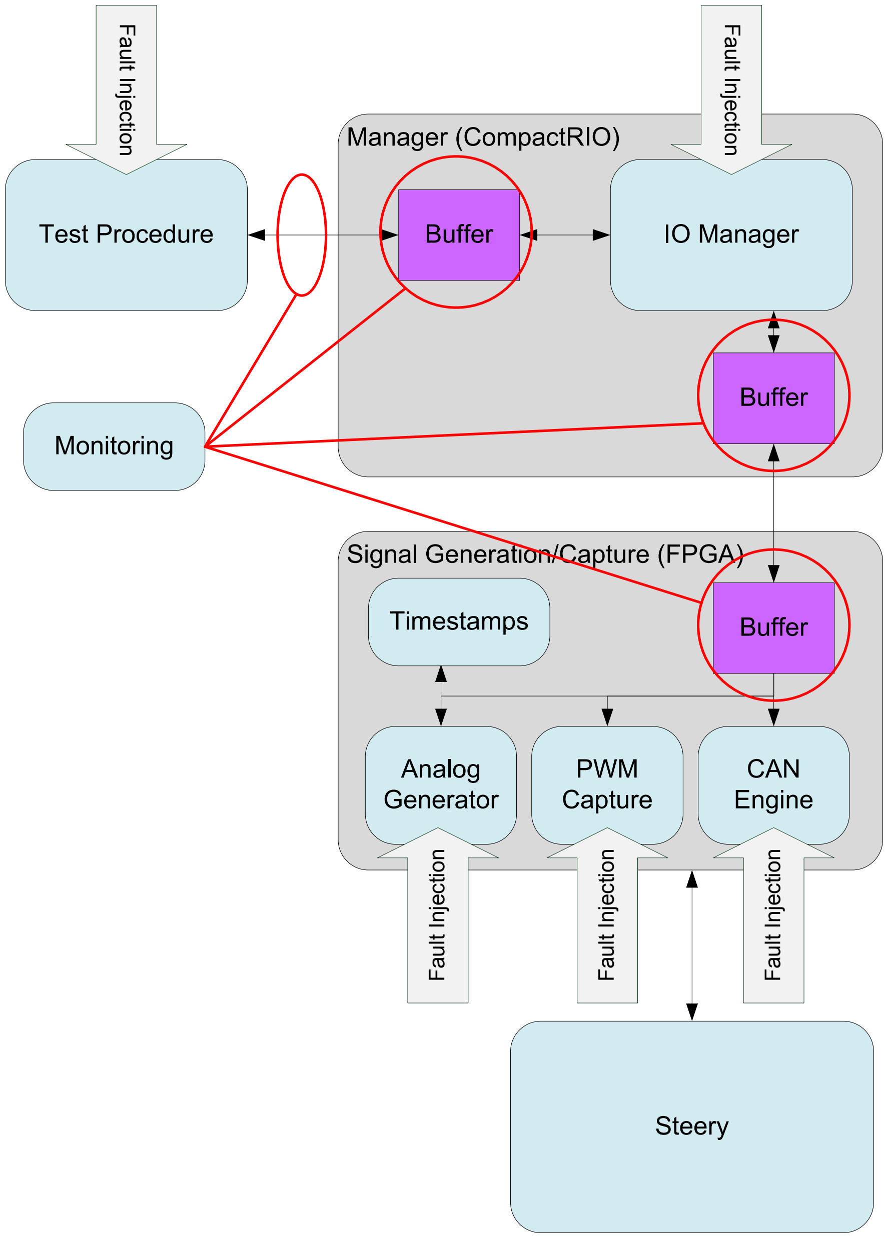


Figure 2.2: Practical Evaluation on the Universal Test Platform: Monitoring and Fault Injection, according to [Izo12]

Chapter 3

Proposed approach for the qualification of in-house developed software tools

As proposed in [Sch11], Figure 3.1 shows a guideline for the qualification of software tools according to [ISO11b].

First of all an overview of all software tools used within the development process has to be attained. That means all relevant software tools should be collected in a software tool list. For each software tool in the software tool list, its TCL has to be determined. A methodology to attain such a software tool list and to determine the TCL of all contained software tools is described in [HRM^{+}11]. It can be stated in general that tools classified with a high TCL are applied on the top of the development V-model. Considering tools on the top of the left side of the V-model it can be said, that errors or malfunctions of these tools can change the content of the requirements specified by the customer or even lead to a loss of customer requirements. However, tools applied on the top of the right side of the V-model are the last verification units of the development process. Errors that are not detected at this level end up directly in the product. The next step is to determine the target ASIL. This determination considers the maximum ASIL that can be violated by a malfunction of the considered software tool. According to [ISO11b] an assumption of the maximum ASIL can be made when the software tool shall be used generic. Due to the fact that a qualification of a software tool requires a big effort, the maximum ASIL should be chosen in order to avoid further changes in an already elaborated qualification approach caused by a changing maximum ASIL. The following considerations deal with ASIL D as the maximum ASIL. Based on the determined TCL and target ASIL [Sch11] recommends as next steps the definition of qualification methods and the application of the qualification methods. These two steps are discussed in section 3.1 and section 3.2 for the specific field of application of an in-house developed software tool. If these two steps can be performed successfully, the qualification report can be generated. In case of an already qualified software tool developed by an external vendor, the qualification reports should be reviewed. A qualification for lower ASILs shall be considered, if the definition and application of the qualification methods or the review of the existing external qualification reports yields no successful result.

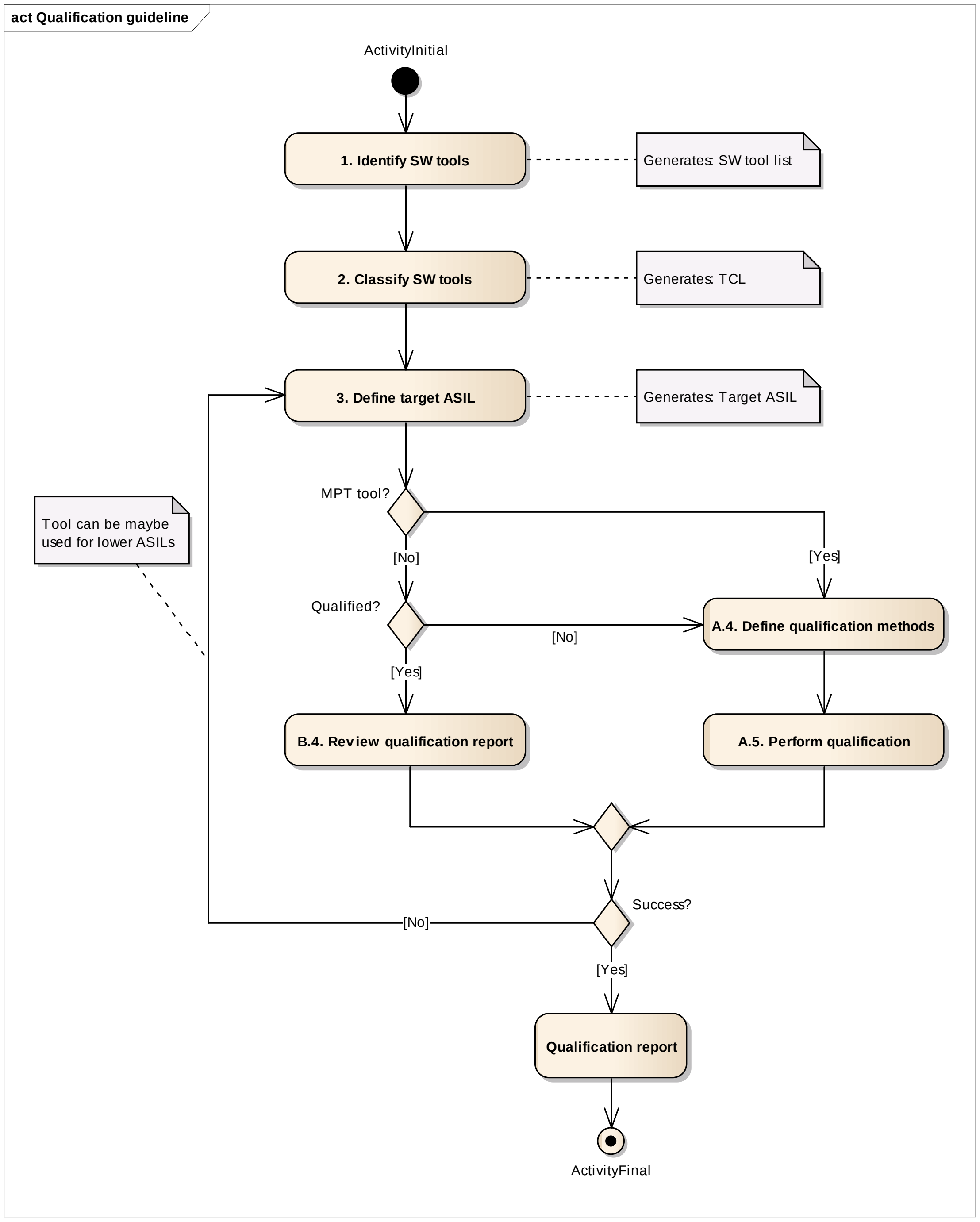


Figure 3.1: Guideline for the qualification of software tools in accordance with [Sch11]

3.1 Definition of qualification methods for in-house developed software tools

Which qualification method may be applicable or not, depends on two main points.

First, it depends on the period of time the software tool is in use. In case of an adequate long period of time without any tool malfunctions occurred, the software tool can be qualified using the method "Increased confidence from use". A prerequisite for the application of this qualification method is, that possible tool errors are recorded and remedied in a systematic way. As mentioned in [Sch11], just the systematic recording and remedy of tool malfunctions during the usage of the software tool is the difficulty by applying this method. An initial qualification based on the method "Increased confidence from use" makes no sense, except a systematic recording and remedy of the tool malfunctions has already been performed.

The second main point concerning the definition of qualification methods is, who developed the software tool. That means it makes a difference if the tool is developed in-house or if the tool is developed by an external vendor.

In case of an external developed software tool a qualification based on the method "Develop in accordance with a safety standard" is very unlikely. The method can only be applied properly, if the tool vendor offers the whole documentation of the tool development. Usually no tool vendor would do that. The application of the methods "Evaluation of the tool development process" and "Validation of the software tool" is more likely.

In case of an in-house developed software tool the key methods for qualification are "Develop in accordance with a safety standard" and "Validation of the software tool". Developing in accordance with a safety standard includes the verification of the developed product. According to [BM08], the verification of software shall answer the question:

> Did we build the product right?

However, [BM08] claims that the validation of software shall give an answer to:

> Did we build the right product?

As mentioned in section 2.1.2, only a subset of requirements specified by a safety standard can be used for software tool qualification reasonably. This means the requirements need to be analyzed focusing on their feasibility of a software tool development at first. A rationale should be given for all requirements or parts of requirements that are rejected. The content of such a rationale depends mainly on the architecture of the software tool to be qualified. Therefore, no generic statement can be made on the requirements of a safety standard that can be rejected for a software tool development. It is proposed to use a standard that deals with the special needs of the industrial sector. In the automotive industry the safety standard that should be used is *ISO 26262-6:2011(E): Road vehicles – Functional safety – Part 6: Product development at the software level* ([ISO11a]). Generally it can be said, that all requirements of [ISO11a] that concern considerations specific for embedded systems (e.g. tasking, hardware-software interfaces, error handling and so forth) can be rejected or must be modified for a software tool development.

Although [ISO11b] requires a software tool development in accordance with a safety standard for ASIL D qualifications, [ISO11b] provides no recommendations concerning the safety classification of such a software tool development. However, the safety standard *IEEE Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations* ([IEE10]) recommends, that

> If software tools are used during the life cycle process of safety-related software, one or both of the following methods shall be used to confirm outputs of that software tool are suitable for use in safety-related systems:
>
> - The output of the software tool shall be subject to the same level of verification and validation (V&V) as the safety-related software, to determine that the output of that tool meets the requirements established during the previous lifecycle phase.
> - The tool shall be developed using the same or an equivalent high quality lifecycle process as required for the software upon which the tool is being used as described in this subclause (5.3) or commercially dedicated as in 5.17, to provide confidence that the necessary features of the software tool function as required.
>
> Software tools used to support the software life cycle process of safety-related software shall be controlled under configuration management.

As a result, it is proposed to apply the target ASIL for software tool development in accordance with [ISO11a].

The proposed application of qualification methods for in-house developed software tools is shown in Figure 3.2. As can be seen, the qualification method "Develop in accordance with a safety standard" is covered in parts by the qualification method "Validation of the software tool". This shall illustrate the overlap between the software tool validation and the verification phases required by [ISO11a]. Hence, several requirements concerning the software tool verification can be performed implicitly by the software tool validation. As explained above, no generic statement can be made whether a single requirement of [ISO11a] can be applied on a software tool development or not. However, it can be said that the software tool validation covers parts of the [ISO11a] verification steps "Software integration and testing" and "Verification of software safety requirements". This is because of the stimulation of the software tool GUI and the examination of the software tool behavior at the validation phase.

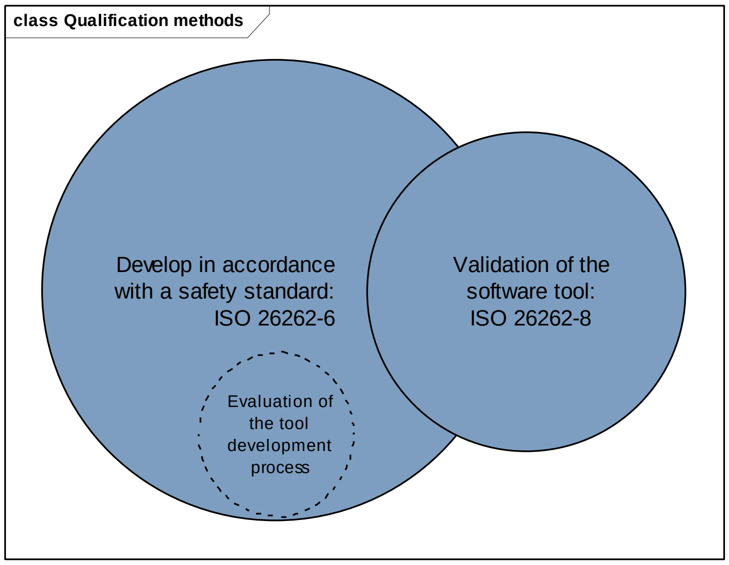

class Qualification methods
Develop in accordance with a safety standard: ISO 26262-6
Validation of the software tool: ISO 26262-8
Evaluation of the tool development process

Figure 3.2: Proposed combination of qualification methods for in-house developed software tools

To sum up, developing the software tool in accordance with [ISO11a] offers the following benefits:

1. Existing tools and processes used for automotive software development can be used or adapted for software tool development.
2. Developers and testers are familiar with those tools and processes.
3. By applying [ISO11a] a verification of the software tool is applied.[1]
4. Correspondences between the software tool validation and software tool verification exist
5. The qualification method "Evaluation of the tool development process" focuses on the correct application of [ISO11a] and can be performed as an internal assessment.

3.2 Application of the qualification methods

When qualifying an in-house developed software tool, the relevant tool information is spread over a multitude of documents. To provide an overview over all those documents a structured working method is necessary. It is proposed to structure the process of a software tool qualification based on the "4+1 View Model" of Philippe Kruchten, described in [Kru95]. This methodology is originally used to describe the architecture of software-intensive systems. The idea behind the "4+1 View Model" is to provide five different views on a software-intensive system, to satisfy the needs of each stakeholder of the system.

[1]That means, that the "inner life' of the software tool is examined.

When describing a software tool qualification with Kruchten's "4+1 View Model", the views must be adapted. Furthermore it is very important, that only the tool package that shall be qualified is described by the adapted "4+1 View Model". The adapted "4+1 View Model" shall be called "qualification view model". The qualification view model is depicted in Figure 3.3. The five views and their meaning for a software tool qualification are given below.

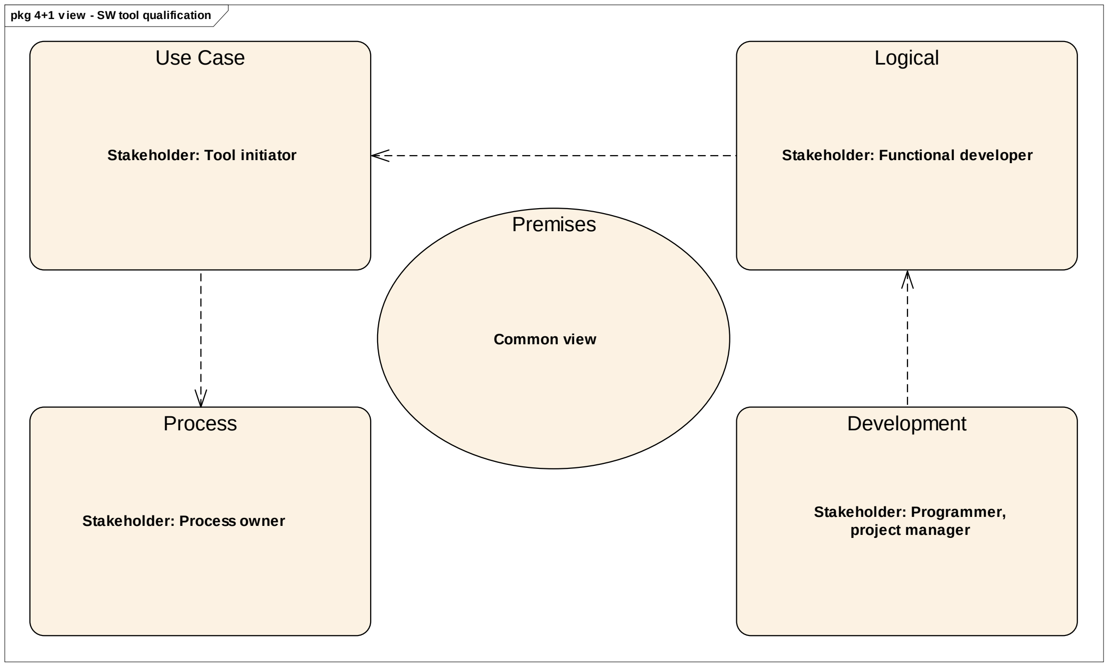

pkg 4+1 view - SW tool qualification
Use Case
Stakeholder: Tool initiator
Logical
Stakeholder: Functional developer
Premises
Common view
Process
Stakeholder: Process owner
Development
Stakeholder: Programmer, project manager

Figure 3.3: Qualification view model based on Kruchten's "4+1 View Model"

Premises The physical view in [Kru95] is replaced by the premises view. The premises view is the common view of the qualification process and relevant for all stakeholders. As the name suggests, this view defines the software package that is considered for a software tool qualification. It offers the considered version of the software tool to be qualified, the maximum ASIL and the determined TCL with a link to the software tool evaluation report. The version numbers of the operation system and of all supporting software tools shall be listed, too. If different versions of the hardware components that are needed for tool execution have no influence on the correct functioning of the considered software tool, only software aspects need to be considered in this view. The persons who are responsible for the software tool qualification are also listed.

Process The process view shows the development process within the the software tool is embedded. Diagrams and documents describing the development process relevant for the tool shall be included in the process view. The stakeholders of this view are represented by the process owners.

Use Cases This view depicts all the use cases of the software tool to be qualified. These use cases are derived from the content of the process view. It is proposed to show the interactions between the user and the software tool in an use case diagram. Furthermore, the user manual of the software tool shall be contained by this view. The software tool initiator represents the stakeholder of this view.

Logical According to [Kru95], the logical view supports

> what the system should provide in terms of services to its users.

For a software tool qualification the same meaning of this view as defined by Kruchten is used. That means, the logical view specifies the functionalities and behavior of each use case of the software tool. The stakeholder of this view is the functional developer of the software tool. It is proposed to specify the software tool requirements within the logical view.

Development The development view depicts how the functions of the software tool shall be implemented. For the proposed approach of a software tool qualification this view includes an architectural design and a unit design of each module of the software tool. To depict the software tool architecture class diagrams (static description) and sequence diagrams (dynamic description) shall be used. Each use case shall be linked with a sequence diagram, to group the software tool modules by use cases. In addition data flow diagrams and package diagrams shall be used to provide a general overview on the software tool. Each process, defined in the data flow diagram, shall be linked with the classes of the software tool that are implementing the corresponding process. Furthermore it is proposed to use activity diagrams to show the internal structure of each software tool module. [Kru95] defines, that

> The development view serves as the basis for requirement allocation, for allocation of work to teams (or even for team organization), for cost evaluation and planning, for monitoring the progress of the project, for reasoning about software reuse, portability and security. It is the basis for establishing a line-of-product.

That means, the stakeholders of the development view are programmers and project managers.

In addition to the five views, the order of specifying each view is shown in Figure 3.3, too. It must be mentioned, that the premises view as the common view is not included in this consideration, because it is related to all the other views. First, process view shall be specified because it initiates the use case view. This circumstance is shown by the dependency relationship between the process view and the use case view. This is based on the fact, that an in-house developed software tool with its use cases is required to support the internal development process. When all the use cases are defined, the concrete functions of the use cases must be specified and after that the development view shall be filled with information.

As already shown in section 3.1, the application of [ISO11a] is advantageous. Considering the qualification view model, it becomes apparent that the logical view and the development view represent the design phases of [ISO11a].

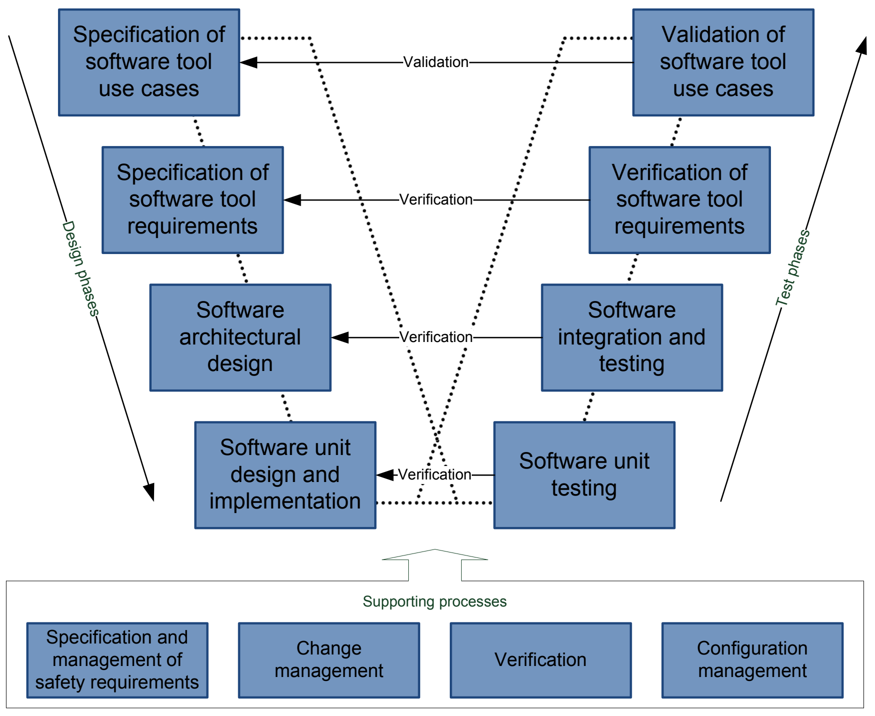

Specification of software tool use cases
Validation
Validation of software tool use cases
Specification of software tool requirements
Verification
Verification of software tool requirements
Software architectural design
Verification
Software integration and testing
Software unit design and implementation
Verification
Software unit testing
Design phases
Test phases
Supporting processes
Specification and management of safety requirements
Change management
Verification
Configuration management

Figure 3.4: Reference phase model for the SW tool development based on ISO 26262-6

The test phases of the applied safety standard [ISO11a] are located outside of the qualification view model. Although the test phases are not directly represented by the model, every phase of testing is related to a specific view on the software to be qualified. The use case view and the process view are relevant to all test cases considering the validation of the software tool. The verification of the software tool requirements is based on the diagrams and documents of the logical view. For software integration and testing the class diagrams, sequence diagrams and data flow diagrams specified in the development view are relevant. The activity diagrams specified in the development view are pertinent to the software unit testing. Last but not least, the test results are valid for the version and environment of the software tool to be qualified, provided by the premises view.

Beside the Kruchten "4+1 View Model", the applied safety standard [ISO11a] has to be tailored for a software tool qualification, too. Figure 3.4 shows the reference phase model including the design and test phases for a software tool qualification. As can be seen, the first phase "Specification of software tool use cases" and the last phase "Validation of the software tool use cases" represent the major extension of the safety standard. For all other phases the basic considerations can be taken from [ISO11a]. The specification of software tool use cases serves as an software tool specific access into [ISO11a], the validation[2] of the software tool use cases checks the tool behavior with an increased focus on anomalous operating conditions.

[2]The term "Validation" is related to the act of validation as understood by [ISO11b].

The validation of the software tool is based on [Som05]. This paper presents an approach for use case validation with scenarios. [Som05] describes a scenario as follows:

> Scenarios describe interactions between systems and actors. (...) A scenario may be "positive" or "negative". A positive scenario describes interactions that must be supported, while a negative scenario describes interactions that need to be avoided. (...) Scenarios and use cases are related. A use case consists of a primary scenario and 0 or more secondary scenarios.

After defining all use cases and scenarios, the tests can be executed. [Som05] proposes, that

> The requirements analyst needs to identify a deviation point in a scenario where the behavior resulting from simulation deviates from the expected behavior.
>
> - For a positive scenario that fails, the deviation point is the event at which the simulation stopped.
> - For a negative scenario that passes, the deviation point is the first event (trigger, assertion) in the scenario that shouldn't have been accepted, or the first system reaction that shouldn't have been produced by the system.

In addition to the approach presented by [Som05], parts of [WWI+12] are applied to the software tool validation, too. As proposed by [WWI+12], a test run without fault injection is performed at first. After executing these test cases, a test run using fault injection is performed. The fault injection considers the anomalous operating conditions of the software tool.

But not only the application of [ISO11a] is proposed. Furthermore the application of a subset of supporting processes, defined in [ISO11b], is necessary for a successful software tool qualification, too. Besides the implicit required supporting processes "Specification and management of safety requirements" ([ISO11b] clause 6), "Change management" ([ISO11b] clause 8) and "Verification" ([ISO11b] clause 9), also the supporting process "Configuration management" ([ISO11b] clause 7) shall be applied.

In case of a software tool qualification according to the proposed approach the work product of the qualification is the qualification view model with its related test results. The view model contains all the information necessary for deriving a software tool criteria evaluation report and a software tool qualification report. Figure 3.5 shows the steps of the proposed software tool qualification approach.

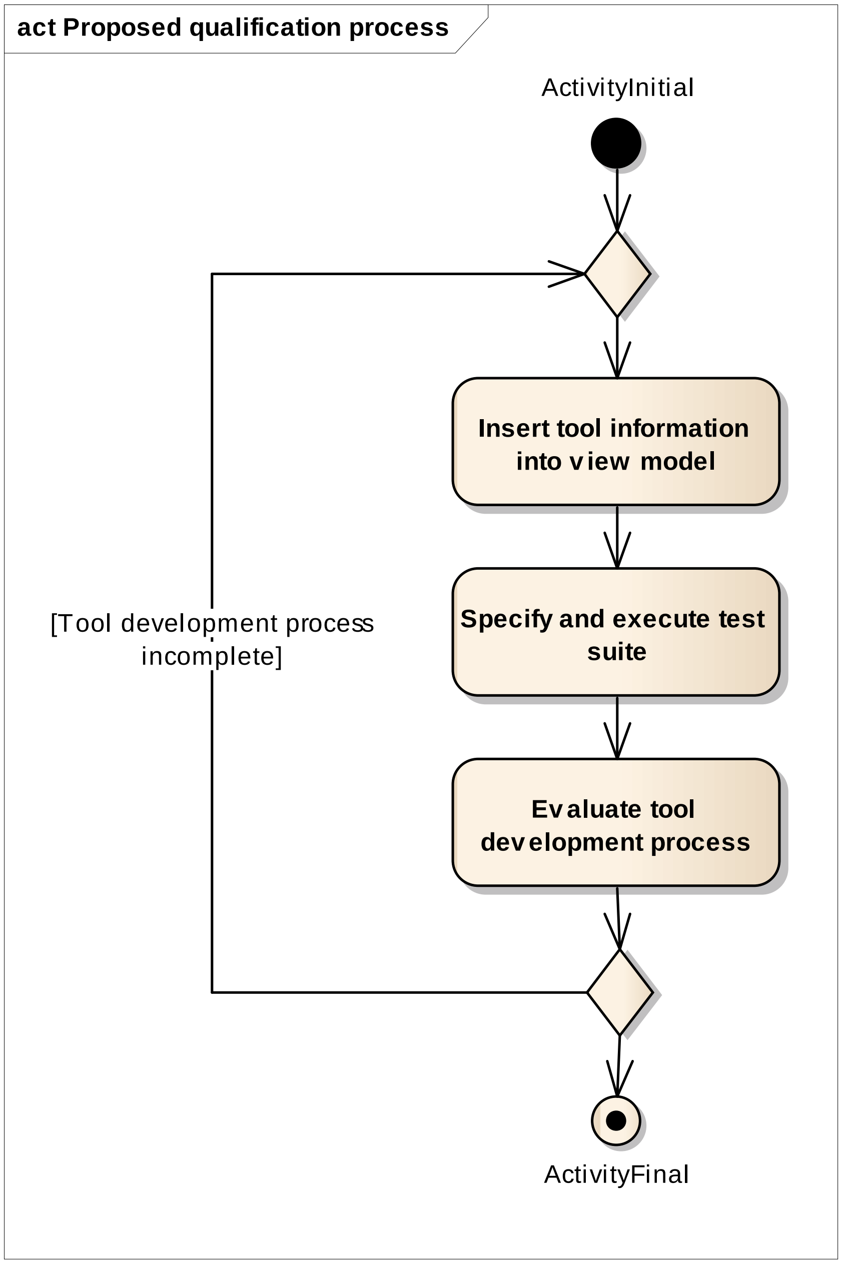

act Proposed qualification process
ActivityInitial
Insert tool information into view model
Specify and execute test suite
Evaluate tool development process
[Tool development process incomplete]
ActivityFinal

Figure 3.5: Proposed qualification process of a software tool qualification in accordance with ISO 26262 based on the proposed approach

3.3 Correlation with the Standard

Each requirement of clause 11 in [ISO11b] shall now be analyzed to find out, if it is covered by the proposed approach.

11.4.1.1 This requirement specifies the circumstances that make a software tool qualification necessary. It is the reason for a qualification and automatically fulfilled by performing the software tool qualification.

11.4.2.1 When qualifying a software tool, assumptions according to the usage of the software tool are necessary to be made. If those assumptions are translated to practice, this subclause is fulfilled. It must be mentioned, that this is a very tool specific requirement and therefore not part of the proposed generic software tool qualification approach.

11.4.3.1 The subclause requires the usage of exactly the same software tool version that is qualified. The proposed methodology focuses on the qualification of software tools, so this subclause is out of scope.

11.4.4.1 By deciding to perform the proposed software tool qualification approach, this requirement is satisfied. The identification and version number and the software tool environment as well as the maximum ASIL are considered within the premises view, the configuration is depicted within the logical view and the qualification methods are already defined by applying the approach.

11.4.4.2 The information for a proper usage or evaluation of the software tool is required by this subclause. All the information is provided by the adapted "4+1 View Model" and the application of [ISO11a]. A description of the features and functions is given in the logical view. Furthermore the logical view contains a description of the expected behavior of the software tool under anomalous operating conditions (specification of the error handling). Information concerning the correct use of the tool is provided by the process view. The applied change management, required by [ISO11a], contains the description of known (detected) tool malfunctions and their avoidance by commissioning a change in the code lines responsible for the malfunctions. The proposed approach provides a verification and a validation of the software tool. Thus, the information on the measures for the detection of malfunctions of the software tool identified during TCL determination is provided.

11.4.5 This subclause specifies the correct evaluation of the software tool to be qualified. For the application of the proposed approach for qualification, the resulting TCL from software tool evaluation is not relevant. Due to the applied qualification methods, the approach can be used for TCL 2 as well as for TCL 3.

11.4.6.1 The applied qualification methods are adopted from this subclause and represent a combination of the methods 1b, 1c, and 1d. As a result, the considered requirement is fulfilled. The proposed qualification procedure is designed to comply with the most stringent safety requirements for a software tool qualification (ASIL D) and thus can be used for all maximum ASIL classifications.

11.4.6.2 The proper documentation of a software tool qualification is described by this requirement. The unique version number of the software tool and the determined TCL with a link to the software evaluation report are given in the premises view. The maximum ASIL that can be violated by malfunctions of the software tool, the configuration environment and the persons who carried out the qualification are also listed in the premises view. A description of the proposed approach, as given in section 3, complies with the demands on the documentation of the applied qualification methods. The measures applied to qualify the software tool are the verification and the validation of the tool. By applying [ISO11a] the results of the measures applied to qualify the software tool are documented within the applied change management process. However, the detected malfunctions are documented in the corresponding test cases.

11.4.7 Method not applied.

11.4.8 The proposed methodology assumes, that the compliance of the qualification process with ISO 26262 is evaluated. The specification of the evaluation criteria is out of scope of this thesis.

11.4.9.1 The qualification method "Validation of the software tool" is applied, as recommended by [ISO11b].

11.4.9.2 By applying the safety standard [ISO11a], it is verified if the software tool complies with its specified requirements. Furthermore the detected software tool malfunctions are analyzed and proper measures to correct them are derived by the applied change management process. As mentioned above, the validation of the software tool focuses intensively on the anomalous operating conditions. That means the proposed approach complies with this subclause.

11.4.10 This subclause addresses the review of the software tool qualification performed by an external organization. Thus, it is out of scope of the proposed approach.

When analyzing the proposed approach for the qualification of in-house developed software tools, it becomes apparent that the approach covers all requirements of [ISO11b] concerning a software tool qualification.

Chapter 4

Qualification of the software tool ATool

At *Magna Powertrain* functional software is developed by a model-based approach. The modules of an automotive embedded software are implemented by means of dSpace TargetLink blocks within Mathworks Simulink. For each change and configuration management checkpoint of a software module implementation, c-code is generated based on the correspondent Simulink module. To comply with [ISO11a], each Simulink module as well as its generated c-code must be verified on the module (= unit) level, in accordance with [ISO11a], clause 9. At *Magna Powertrain* the generation of test environments used for software module testing is a generic process. Individual software module test environments differ in using different software modules under test (MUTs). As a result the handling of module tests follows a generic process, too.

A manual creation of test environments and a manual handling of the specified test cases would be extremely time-intensive. Furthermore, the risk of implementing errors into the test environment is relatively high, same for test case handling. To increase the efficiency of software module testing and to minimize the risk of human error, the software verification tool ATool was developed by *Magna Powertrain*.

The tool is implemented within MATLAB and supports the handling of module-in-the-loop (MIL) tests and software-in-the-loop (SIL) tests as well as the handling of an observer function (OBS) implemented by the module tester. Result evaluation is automatically performed for MIL against OBS as well as for MIL against SIL.

ATool features seventeen use cases to a module tester. These use cases are realized by thirty-one MATLAB modules that are implemented by approximately ten thousand lines of code. ATool communicates with MS Excel, Mathworks Simulink, MKS Source Integrity and dSpace TargetLink.

For a correct usage of ATool according to ISO 26262:2011(E), the software tool is qualified based on the qualification approach described in chapter 3.

4.1 Tailoring of ISO 26262-6:2011

As explained in chapter 3, the applied safety standard [ISO11a] must be tailored for a software tool development. In this section the tailoring of [ISO11a] for the ASIL D development of ATool shall be discussed. To keep this section simple, only the requirements that are not or not completely applied for the development of ATool are shown and explained in the following subsections.

4.1.1 Initiation of product development at the software level

5.4.4 Adapted; no hardware development phase is applied to the development of ATool. Instead of applying a system development phase, software tool use cases are defined.

5.4.6 Adapted; support for embedded real time software is not necessary.

5.4.7 Adapted;

- 1b, Use of language subsets: Not applied. There are no defined and established language subsets applicable for a software tool development within MATLAB available.
- 1c, Enforcement of strong typing: Not applied. ATool is no embedded software. According to MATLAB Help all numeric values are stored as double-precision floating point by default (default type and precision cannot be changed). Any number, or array of numbers, can be chosen to store as integers or as single-precision. Integer and single-precision arrays offer more memory-efficient storage than double-precision. In case of ATool a memory-efficient storage of data is not required.

4.1.2 Specification of software safety requirements

6.4.1 Adapted; the term "software safety requirement" is replaced by the term "software tool safety requirement". Instead of "technical safety requirements" software tool use cases shall be considered.

6.4.2 Adapted; the hardware aspects of this requirement can be rejected for ATool development, as explained in section 4.2.1. Also timing constraints don't need to be considered, because of the fact that ATool is not a real-time application.

6.4.3 Not applied; no ASIL decomposition is performed.

6.4.4 Not applied; no hardware-software interface is considered, as explained in section 4.2.1.

6.4.7 Not applied; for explanation see section 4.2.1

6.4.8 Adapted; a compliance and consistency with the ATool use cases shall be reached, instead of a compliance with a system design and with technical safety requirements. The hardware-software interface is not considered (see section 4.2.1).

4.1.3 Software architectural design

7.4.3 Adapted;

- 1f, Appropriate scheduling properties: Not applied; ATool is not a real time system.
- 1g, Restricted use of interrupts: Not applied; ATool is no embedded software. Furthermore, MATLAB offers no interrupts.

7.4.8 Not applicable; the entire software tool has to be qualified in accordance witch [ISO11b], clause 11.

7.4.13 Not applied since:

- Each software module is treated as safety-related without differentiation.
- No safety mechanisms are intended to cover failure modes of ATool itself.

7.4.14 Strategical decision: not applied. The relatively low complexity of ATool compared to automotive embedded software, and the fact that ATool is not a real-time software support this decision.

7.4.15 Strategical decision: not applied. The relatively low complexity of ATool compared to automotive embedded software, and the fact that ATool is not a real-time software support this decision.

7.4.17 Not applicable; this is an embedded software specific requirement.

7.4.18 Adapted; considerations on target hardware as explained in section 4.2.1. Inspection of the software architetural design will be applied.

4.1.4 Software unit design and implementation

8.4.4 Adapted;

- 1a, One entry and one exit point in subprograms and functions: Not applied only if necessary due to complexity and error handling reasons.
- 1e, Avoid global variables or else justify their usage: Avoided whenever possible. Necessary due to MATLAB variable handling purposes and to avoid unnecessary complexity.
- 1g, No implicit type conversions: In MATLAB not necessary.
- 1h, No hidden data flow or control flow: See remarks to 1a and 1e.

8.4.5 Adapted; due to the complexity and the programming language only a code inspection is applied.

4.1.5 Software unit testing

Unit level testing has the following two advantages:

1. Proper testing of the behavior in equivalence classes and at the boundary values.
2. High code coverage

Boundary values and equivalence classes are needed only in rare cases for ATool. The tool provides a test environment for modules and stores results. No internal calculation is performed. Branch coverage is measured on the integration testing level. The target code coverage is 100%.

4.1.6 Software integration and testing

10.4.1 Adapted; instead of an embedded software, ATool is considered. Therefore the dependencies between the software integration and the hardware-software integration are not applied (see section 4.2.1)

10.4.3 Adapted; the software-hardware interface is not considered (see section 4.2.1).

- 1b, Interface test: No interface tests necessary due to the low complexity and due to the variable type handling of MATLAB interfaces are tested in the validation phase implicitly.
- 1d, Resource usage test: Not necessary. Validation is performed on each target hardware.
- 1e, Back-to-back comparison test between model and code: Model-based software development is not applied on ATool

10.4.7 Adapted; instead of an embedded software, ATool is considered.

10.4.8 Validation is performed for each target hardware seperately

4.1.7 Verification of software safety requirements

11.4.2 Not applicable.

4.2 Specification of the qualification view model

The following sections detail the five views explained in chapter 3. An overview on the qualification approach applied on ATool is depicted in Figure 4.1[1].

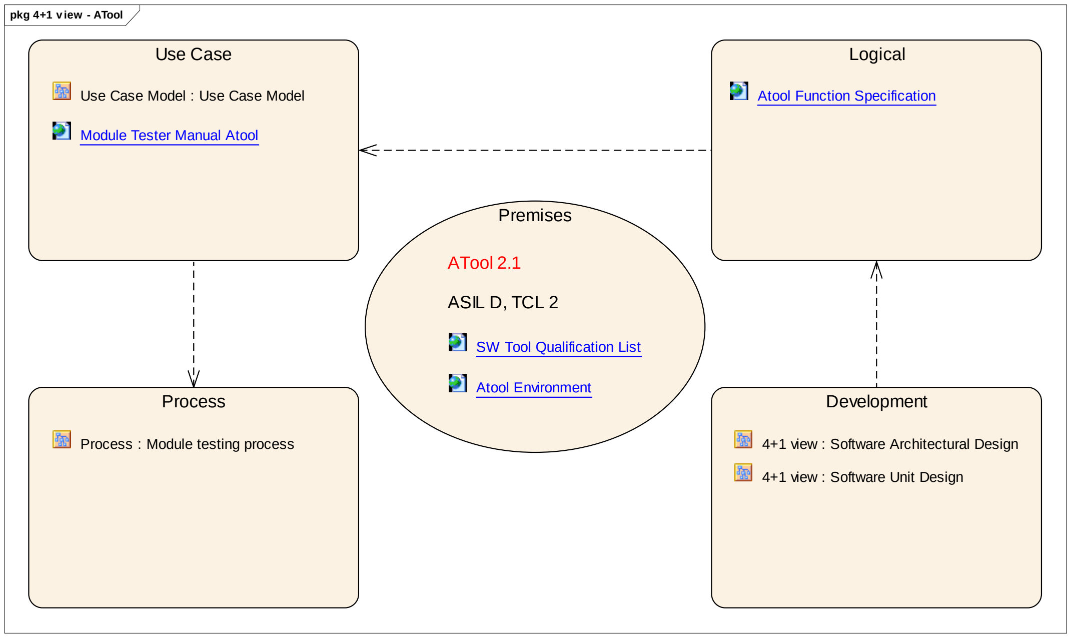

pkg 4+1 view - ATool
Use Case
Use Case Model : Use Case Model
Module Tester Manual Atool
Logical
Atool Function Specification
Premises
ATool 2.1
ASIL D, TCL 2
SW Tool Qualification List
Atool Environment
Process
Process : Module testing process
Development
4+1 view : Software Architectural Design
4+1 view : Software Unit Design

Figure 4.1: Qualification view model applied on ATool

4.2.1 Premises

First of all, the unique software version of the software tool to be qualified must be defined. Furthermore, the persons responsible for the qualification need to be stated. When speaking of ATool, the following sections consider ATool 2.1 as version to be qualified. The persons who perform the qualification are Adam Schnellbach and Benjamin Archan.

ATool evaluation

As a next step, the software tool must be evaluated. The conclusions concerning the software tool evaluation of ATool stated in the module tool list are given below.

- Inputs:
  - Developer module
  - Correspondent data dictionary
  - Observer function realized in a m-file and/or in a mdl-file
  - Test specification sheet

[1]All UML diagrams were created with Enterprise Architect 9.3.

- Outputs:
  - Test environment in Simulink
  - Output signal vectors
  - Test status
  - Code coverage report
  - Test log

The purpose of ATool is to generate test environments for automated testing of software modules developed with model-based development using a tool chain consisting of MATLAB, Simulink, Stateflow and TargetLink. ATool is executed within this tool chain environment, too. The test tool features comprise MIL/SIL comparison, MIL/OBS comparison, open/closed loop testing, C1 code coverage measurement, automatic test result evaluation and test report generation. As a result, possible failure modes are malfunctions or errors in

- test environment creation
- test case execution
- test result evaluation

That means, that the tool has to be classified as TI2, because wrong measurement results can hide malfunctions.

ATool is not used in the last verification phase. Wrong measurement results remain undetected on module level. The software of the product is tested later on the Hardware-in-the-loop (HIL) test rig and in the vehicle using different tools. TD2 is given because the test depth of that HIL and vehicle tests is worse than of the module tests.

Based on this considerations ATool is evaluated conservatively as TCL2.

Maximum ASIL

A multitude of modern vehicle components use the advantages of E/E systems with an increasing trend. Therefore, software module tests will be required for a long term period. Due to the complexity of ATool, its qualification is a protracted process. Therefore, the determination and application of another time intensive qualification approach caused by a maximum ASIL that changed to a higher level shall be avoided. As a consequence the maximum ASIL for ATool is chosen as ASIL D.

Considered environment for qualification

As explained in section 3.2, the qualification relates solely on the specified tool version and tool environment. The environment of ATool is given in Table 4.1. The entries in Table 4.1 shall be called as environment software. The validation step of the qualification also includes the testing of the software and hardware environment of ATool. This step will be repeated on each host PC after an installation of ATool. In this way the separate qualification of the software and hardware environment can be omitted.

Vendor	*Operating system*	Version
Microsoft	Windows	XP Professional 2002 SP3, 32-bit
Vendor	*Supporting tool*	Version
Mathworks	MATLAB	7.11.2.1031 (R2010b) SP2, 32-bit
Mathworks	MATLAB Compiler Settings	Lcc-win32 C 2.4.1*
Mathworks	Simulink	7.6.2 (R2010bSP2)
Mathworks	Stateflow	7.6.2 (R2010bSP2)
dSpace	MATLAB Integration Main	2.0.3
dSpace	TargetLink Data Dictionary	3.4p2
dSpace	TargetLink Production Code Generator	3.4p2
Microsoft	Office Excel	2007 (12.0.6665.5003) SP2 MSO (12.0.6662.5000)
PTC	MKS Integrity Client	2009 Build 4.10.0.9049, SP 006-01

Table 4.1: ATool environment relevant for qualification
*located in C:\PROGRA˜1\MATLAB\R2010B˜1\sys\lcc

4.2.2 Process

Figure 4.2 shows the part of the module testing process at *Magna Powertrain* relevant for ATool. This process is embedded within the engineering process landscape. A profound examination on the engineering process landscape at *Magna Powertrain* is provided by [Spo11].

The first step is to create a module test environment. This test environment is realized by means of a Simulink model.

After that, the test cases for module testing are developed. A test case can be considered as developed, when the module input signal vectors, the output signal tolerances, the used parameter set and the usage of signal feedbacks are specified within a test definition MS Excel sheet. Furthermore, the specified test cases must be synchronized with MKS RM and linked with its corresponding requirements. In addition to the test specification, an observer function must be created. The observer function implements the functional specification of the MUT as understood by the module tester and is represented by a m-file or by a mdl-file.

Before starting the automated test run, the test specification and observer function must be reviewed. Detected defects of the test specification or observer function must be eliminated.

The test case specifications and the observer function are applied to the test environment by performing an automated test run. The test results produced by the automated test run must be checked into MKS SI.

The last step of the module testing process is the analysis of the test results. Depending on the output of the analysis, the test case status in MKS RM has to be set. In case of

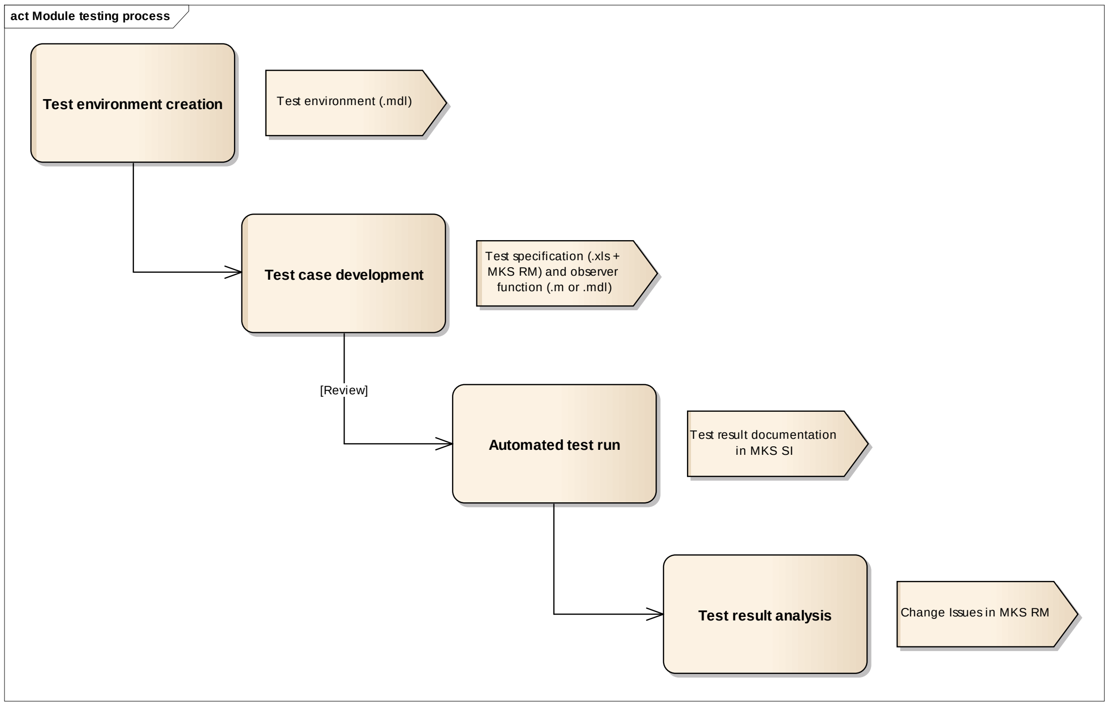

act Module testing process
Test environment creation
Test environment (.mdl)
Test case development
Test specification (.xls + MKS RM) and observer function (.m or .mdl)
[Review]
Automated test run
Test result documentation in MKS SI
Test result analysis
Change Issues in MKS RM

Figure 4.2: Overview on the complete module testing process

expected test case results, the status in MKS RM shall be set to "TC Completed". In case of unexpected test case results this must be documented by means of a Change Issue in MKS RM. The cause and a possible fix shall be handled according to the defined change process, which may lead to a new model version requiring a re-test. In addition the test case status in MKS RM must be set to "TC Failed" and the Change Issue must be linked to the TC.

4.2.3 Use cases

Based on the module testing process, the use cases of ATool are derived. Figure 4.4 shows the use case diagram of ATool. The elements of the diagram shall be understood as follows.

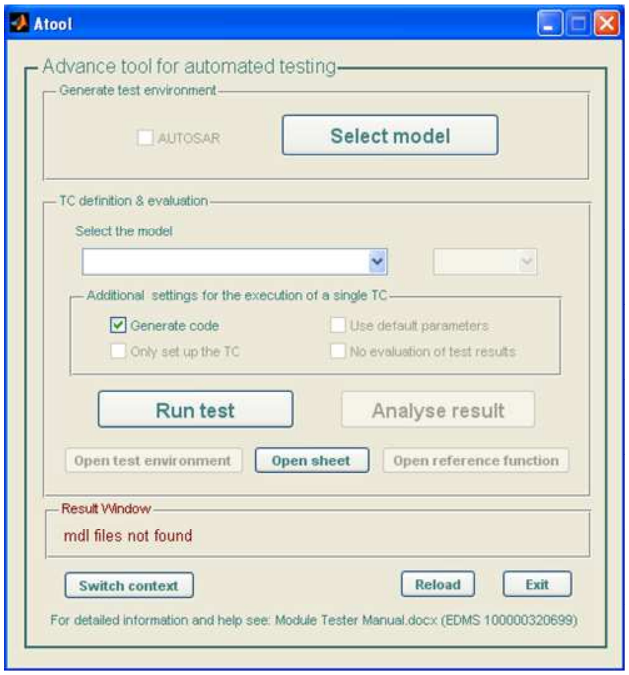
Generate new test environment As the name suggests, this use case considers the creation of a test environment for a new or modified MUT. The necessary information concerning the MUT is gathered from the project's development directory. In case of a new MUT the corresponding test definition sheet and the observer function template are generated, too. In case of a modified MUT, the existing test definition sheet and observer function may be reused and adapted. The name of the MUT is inserted into the module test overview sheet, that lists all MUTs of a considered software component. If the test environment already exists, the updated developer module is integrated and all necessary information is updated.

Edit test case The support for developing test cases is realized by this use case. When specifying a test case, the module tester performs changes within the test definition sheet. Dependent on the test case to be developed, changes within the observer function[2] could be necessary too. ATool provides the ability to open the test definition sheet in MS Excel for editing input signals/parameters, the observer function in MATLAB Editor for changing the behavior of the observer function, as well as the test environment in Simulink for checking the MUT. To provide compatibility to projects performing an older module testing process, the evaluation against observer function outputs can be disabled within the observer function.

Run single test case This use case considers the execution of a single test case including a MIL/Obs and a MIL/SIL comparison. To provide a step-by-step development of a test case, the use case can be extended in multiple ways. First, the test case can be aborted after the set up (no simulation and evaluation is performed). This makes it possible for the user to ensure that the test case inputs are specified and loaded correctly. If needed, a simulation of the test case that has been set up can be done manually. Second, the MIL/SIL comparison can be disabled by the user. This use case extension can be selected, if the observer function is in the focus of the user. Thereby, the time intensive code generation is disabled. Third, the evaluation of test results can be disabled. This feature can be used for code coverage tests, where the results can be ignored. Finally, the user-defined parameter set (if existing) can be overridden by the default parameter set. This is a feature for the tester, to check the impact of a changed parameter set on the test results while keeping the values of the user-defined parameter set.

Run all test cases The considered use case differs slightly from the use case "Run single test case". The only difference is, that here all test cases of the selected MUTs are executed. It uses the same extensions except for "Only set up TC". This is due to the fact, that the check of test case inputs can only be performed properly when a single test case is considered.

[2]NOTE: Within the ATool GUI the observer function is called "reference function".



mdl files not found

For detailed information and help see: Module Tester Manual.docx (EDMS 100000320699)

Figure 4.3: Graphical user interface of ATool

Analyze test results The analysis of test results includes the analysis of the output signal vectors within the test definition sheet. As already mentioned, ATool provides the ability to open the test definition sheet in MS Excel. The analysis of code coverage is also included within test results analysis. That is why an HTML-report containing code coverage data is generated when performing a MIL/SIL comparison. This use case is extended by the ability of analyzing input/output signal characteristics[3]. Therefore, ATool provides the plotting of signals within MATLAB.

For the sake of completeness, Figure 4.4 depicts the use case "Synchronize test results". This use case is not realized by ATool, it is realized by means of an Excel macro within the test definition sheet. The graphical user interface (GUI) of the tool, depicted in Figure 4.3, visualizes the use cases.

[3]Note: The analysis of signal characteristics is a special feature of the tool. In case of deviances between the signal plot and the values contained by the test definition sheet, the latter are valid.

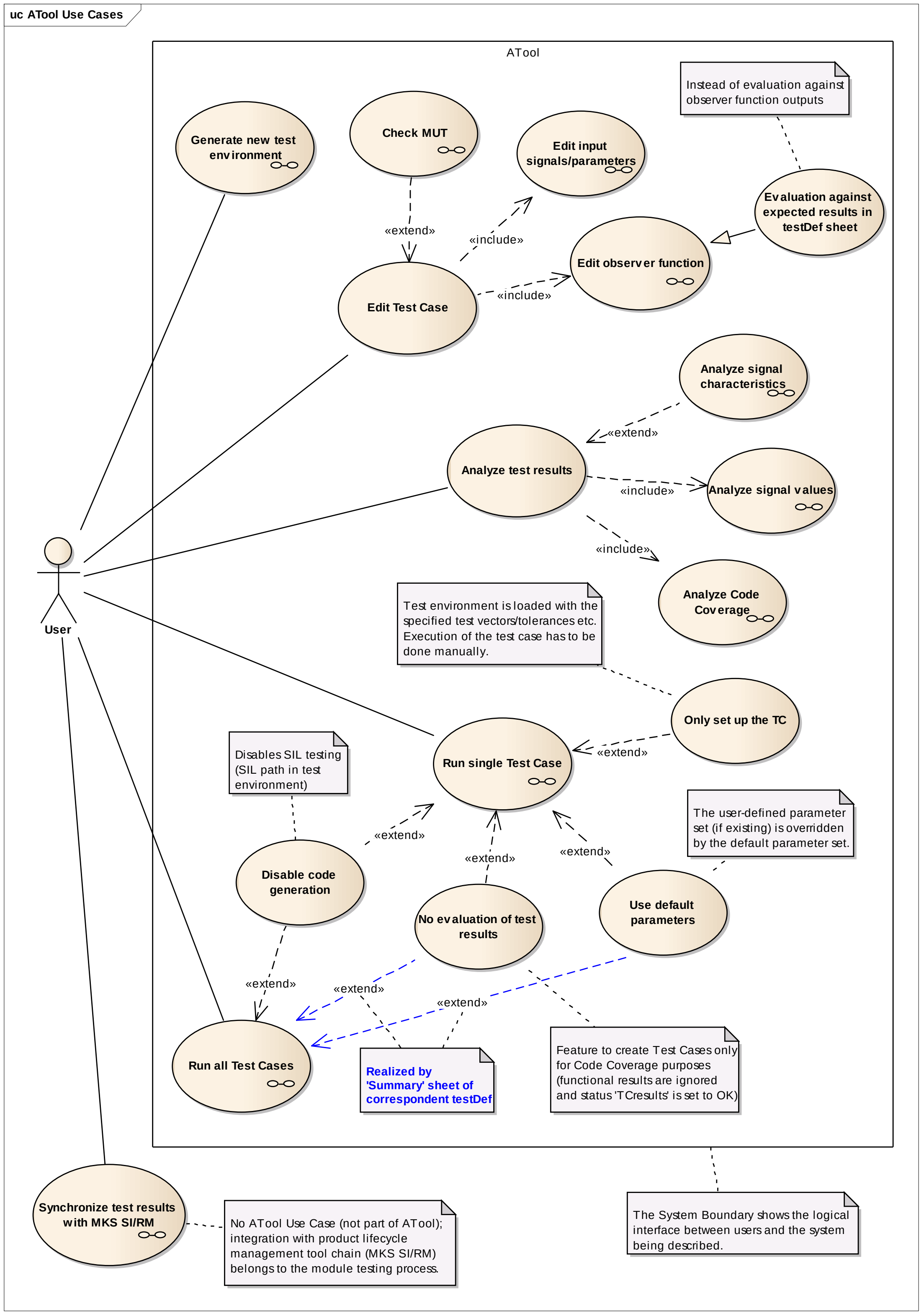

uc ATool Use Cases
ATool
Instead of evaluation against observer function outputs
Generate new test environment
Check MUT
Edit input signals/parameters
Evaluation against expected results in testDef sheet
«extend»
«include»
Edit observer function
Edit Test Case
«include»
Analyze signal characteristics
«extend»
Analyze test results
«include»
Analyze signal values
«include»
Analyze Code Coverage
User
Test environment is loaded with the specified test vectors/tolerances etc. Execution of the test case has to be done manually.
Only set up the TC
«extend»
Run single Test Case
Disables SIL testing (SIL path in test environment)
«extend»
«extend»
«extend»
The user-defined parameter set (if existing) is overridden by the default parameter set.
Disable code generation
No evaluation of test results
Use default parameters
«extend»
«extend»
«extend»
Run all Test Cases
Realized by 'Summary' sheet of correspondent testDef
Feature to create Test Cases only for Code Coverage purposes (functional results are ignored and status 'TCresults' is set to OK)
Synchronize test results with MKS SI/RM
No ATool Use Case (not part of ATool); integration with product lifecycle management tool chain (MKS SI/RM) belongs to the module testing process.
The System Boundary shows the logical interface between users and the system being described.

Figure 4.4: ATool use cases

4.2.4 Logical

This view contains the function specification of ATool in MKS RM. The linking of the requirements within MKS RM is described in section 5.2. The ATool function specification is complied with the adapted requirements of [ISO11a], clause 6 (see section 4.1). An exemplary subset of detailed ATool safety requirements as formulated in MKS RM is shown in Table 4.2. Instead of a detailed specification of ATool, a general overview on the functionality of ATool shall be given in this section.

The execution error handling shall be test case specific.
Means a list of all errors/warnings/notes shall be generated for each of the test cases. This list should be written back into the test definition sheet to the affected test case section and it should be saved as mat-file to the testData_<TCx>.mat file.
The errors should be separated into test case control and test case execution errors.

Table 4.2: Exemplary software tool safety requirements of ATool concerning the error handling

ATool is implemented in MATLAB and provides a test framework for MIL tests and SIL tests in Simulink. An exemplary test environment is depicted in Figure 4.5.

The MUT is linked from the development directory within a project into the test environment and all additional necessary data concerning the MUT is gathered from the correspondent TargetLink data dictionary. The observer function represents the MUT requirements implemented as a MATLAB script or Simulink model. This observer function is linked into the test environment, too. The created test environment provides a possibility for test case execution with previously specified test vectors. The evaluation of the test results is done by performing a MIL/OBS and a MIL/SIL comparison. Hereby, the outputs of the observer function and the outputs of the MUT after code generation (SIL) are evaluated based on the outputs of the MUT before code generation (MIL).

Furthermore ATool handles the specification and execution of MIL/SIL test cases as well as the depiction of the test case results. By means of a so called test definition sheet in MS Excel, the test cases are specified. This includes the definition of the MUT's input signal vectors, MIL/SIL deviance tolerance values, the specification of signal feedbacks, used parameter sets and the simulation time. The possibility to specify multiple test cases to be executed is provided by the test definition sheet, too. By using an interface to MS Excel, the test case specifications from the test definition sheet are loaded into MATLAB workspace and after simulation in Simulink, the test results are written back into the test definition sheet. The test results contain the output signal vectors of the MIL path. Furthermore, the output signal vectors of the SIL path and of the observer function as well as their deviations from the output signal vectors of the MIL path are contained by the test results as well as a generated code coverage report. If these deviations exceed the specified tolerances, the correspondent output signal vector is highlighted in the test definition sheet. If the deviations of the output signal vectors don't exceed the specified tolerance values, the status of the test case is set to "OK", else the status is set to "NOK". In case of detected test case execution errors, a test run is aborted and the error message is written into the test definition sheet. In addition to the export of test results to MS

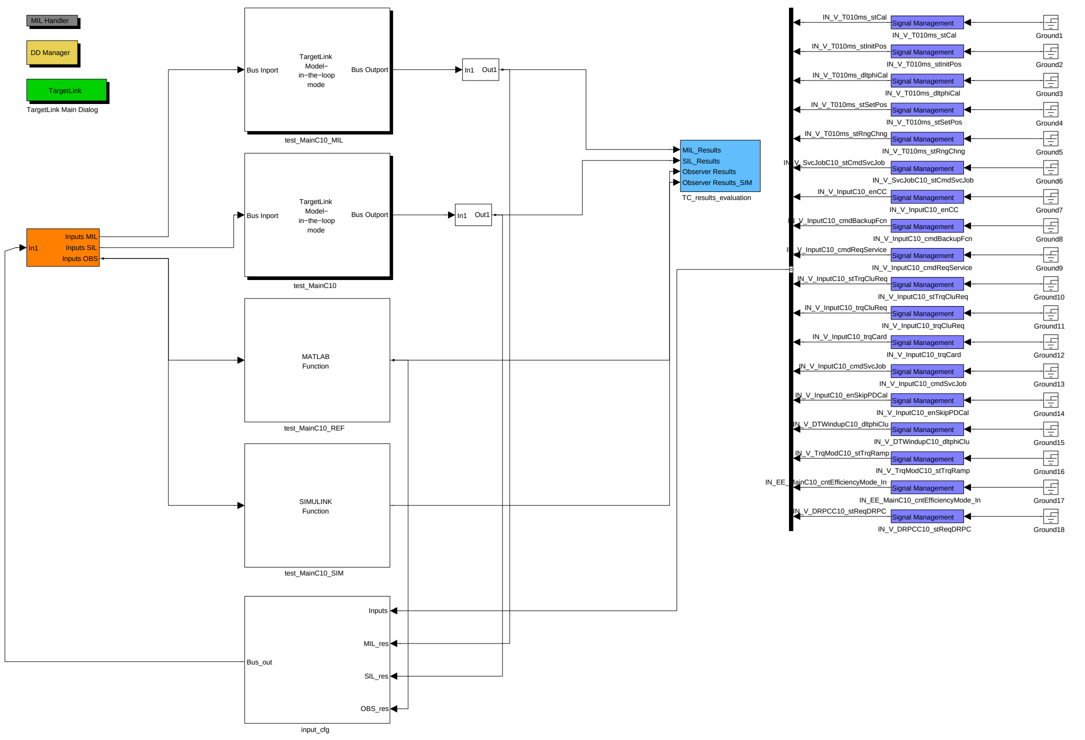


Figure 4.5: Test environment created by ATool

Excel a logging of the executed test cases is performed, by storing all input and output signal vectors as well as the used parameter set and test results in a mat-file. If the test catalog specified in test definition sheet is executed, the test logs are stored in this case in a tar-file.

Another feature of ATool is to provide support in test result analysis to the module tester. Input and output signals can be plotted in MATLAB to gain a better overview on signal characteristics. Furthermore, the tool determinates the reached code coverage and generates a code coverage report in HTML. This report can be used for analysis of MUT's test coverage.

4.2.5 Development

The development view addresses the implementation of the requirements contained by the logical view. The following deals with the architectural design and the software unit design of ATool. For the generation and review of the diagrams [Sys] was consulted.

Software architectural design

Figure 4.6 depicts the packages of ATool considered for qualification. Each package represents a folder within the ATool development project in MKS SI. These project folders contain all ATool modules. As already mentioned, ATool is implemented within MATLAB. Experience has shown that fundamental changes between different MATLAB versions are very likely. To consider that circumstance, ATool utilizes MATLAB version dependent packages. That means, all modules that had to be adapted because of a changed MATLAB version are stored in a version dependent package (see Figure 4.6, package "ML_7.11.2"). When ATool is started, the corresponding version dependent package is loaded. All modules whose content did not change over different versions are located within the "Run" and "ModelCreation" package. As the names suggest, within "ModelCreation" all necessary templates for the creation of a new test environment are stored and within "Run" all modules used for the execution of test cases are stored. The orange package shown in Figure 4.6 is not a package of ATool. However, it contains all dSpace functions called by ATool modules and is depicted to provide a better understanding of ATool. Another abstract package is "Startup", it is executed when MATLAB starts and calls ATool. The dependencies between the depicted packages are indicated by usage relationships. For instance, package "ML_7.11.2" uses package "Run", package "ModelCreation" as well as package "dSpace Functions". That means, that some modules contained by "ML_7.11.2" call some modules of the used packages ("use" their functionality).

Each module is realized by a MATLAB function stored in a m-file. These modules are represented in the UML model as a class stereotyped as module. The general description of the functionality of a module is contained by the notes field of its corresponding class. The main function of a module is specified as a public operation of the class. If a module contains subfunctions, they are specified as private class operations. The input parameters of a class operation represent the input parameters of the corresponding function within the considered module. The specification of an input parameter is composed of its name, data type and a short description of the input parameter. The same attributes are used for the specification of the function return values. However, the return values are not specified as parameters of an operation, their specification is inserted into the notes field of their correspondent operation. Additional, operations representing subfunctions contain a functional description of the subfunction within their notes field. Inputs and outputs of a module that cannot be captured by the input parameters and return values of its functions are specified within the class' notes field. For instance, loaded mat-files or generated files are listed as "Additional inputs" or "Additional outputs" by the notes field of a class representing an ATool module.

The module dependencies that are indicated by the usage relationships in Figure 4.6, are shown in detail by a set of linked composite structure diagrams. In these static diagrams a function call is represented by a usage relationship. If the used module uses another set of modules, a link to the corresponding composite structure diagram is given. The link is represented by the diagram frame of the target diagram. In that way a "module usage tree" is generated. The information concerning the module usage tree is gathered by applying the MATLAB function *fdep.m* provided by [Mat07]. Figure 4.7 depicts all modules used by the ATool GUI and represents the root of the module usage tree. One can recognize, that the module "AnalysisWindow" contains no operations or attributes. This is because of the fact, that all test results are visualized in the test definition sheet in MS Excel. The test definition sheet represents the valid output document of the software module test. However, "AnalysisWindow" provides the depiction of the signal characteristics and can be seen as an additional feature to the module tester. As a result, a detailed description of "AnalysisWindow" can be neglected. In Figure 4.8 a main branch of the module usage tree is shown. As can be seen in Figure 4.8, "testCtrl.m" uses modules provided by TargetLink. These modules are highlighted in orange and are solely depicted for a better understanding of the tool behavior.

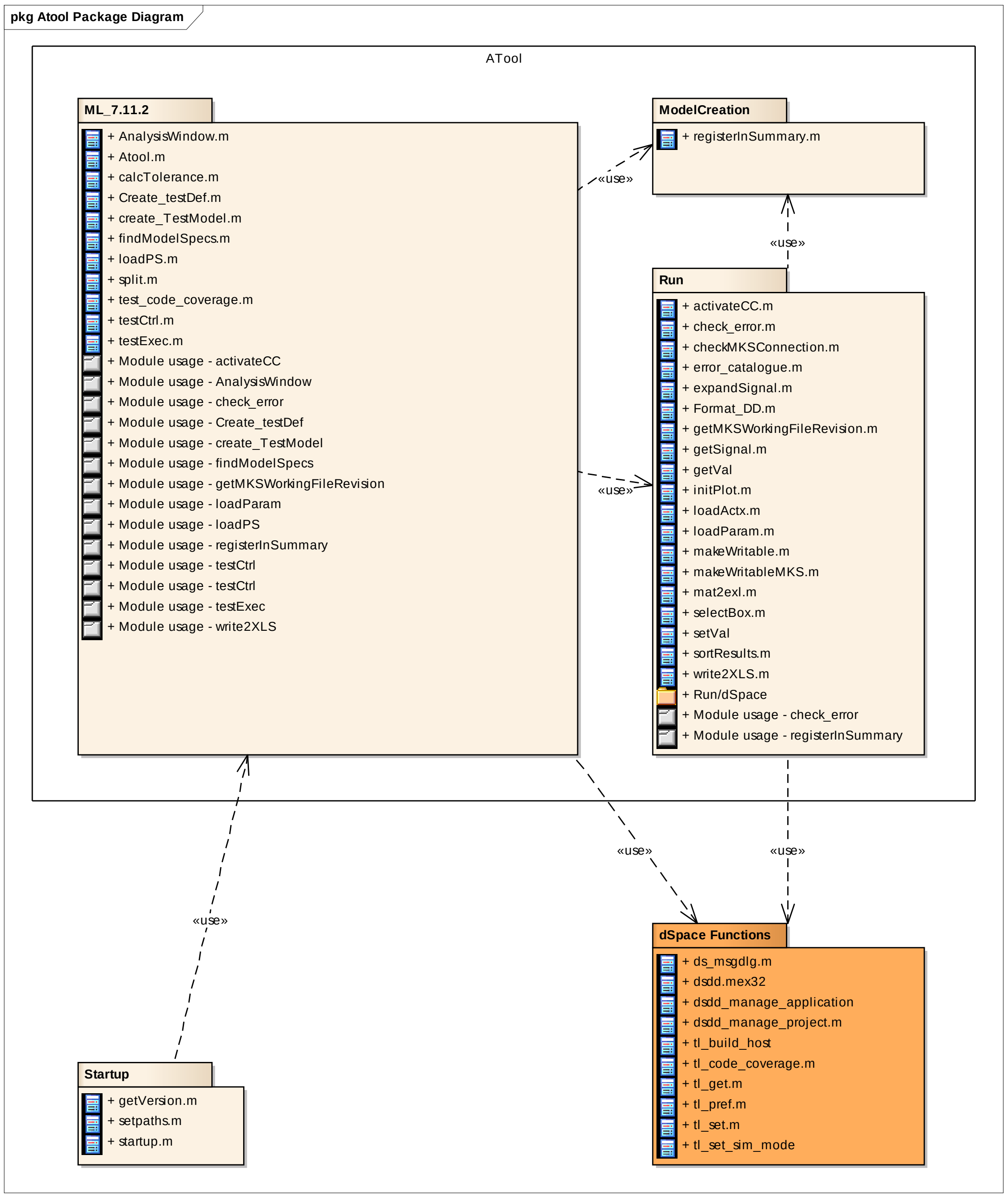


Figure 4.6: ATool package diagram

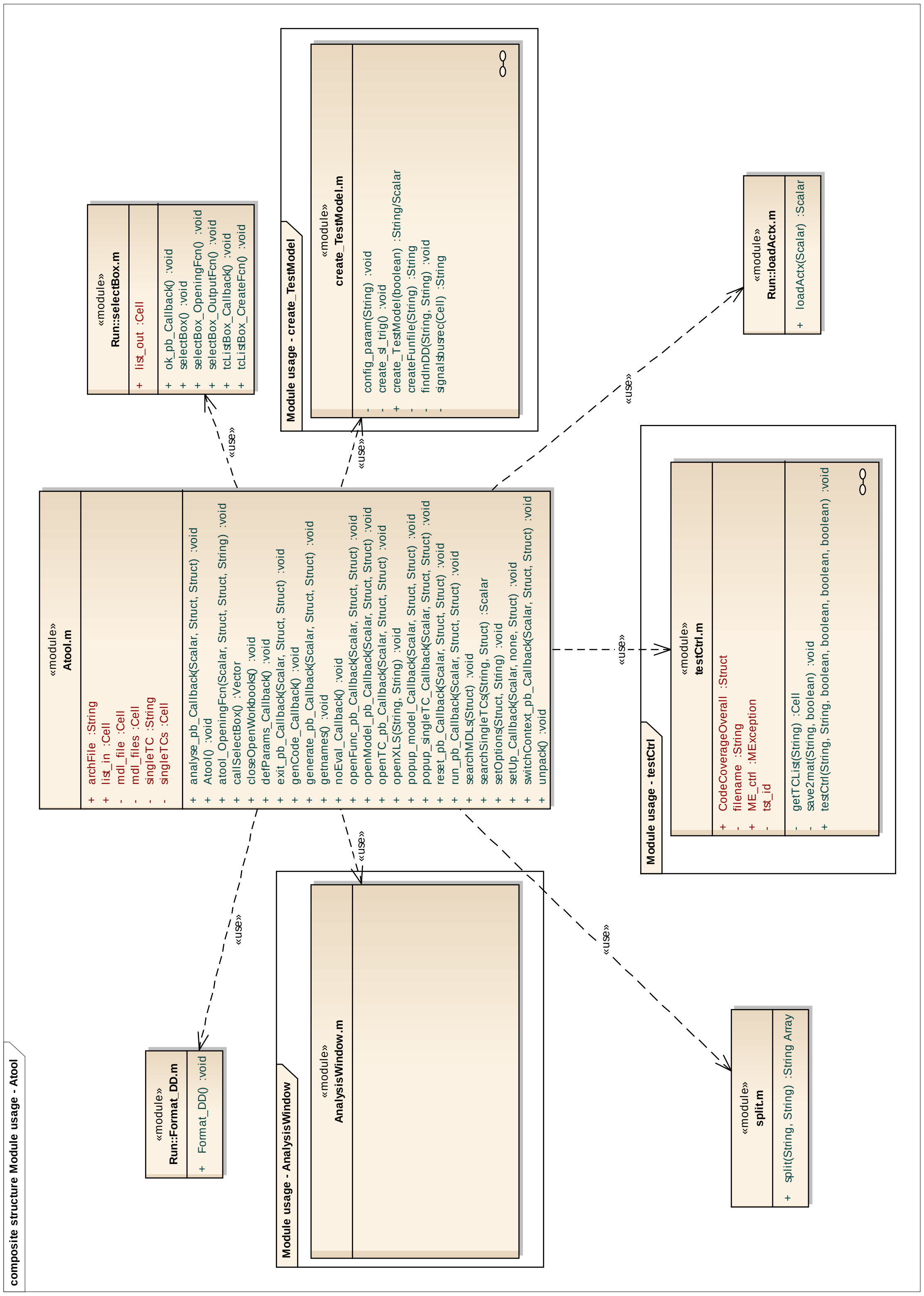

composite structure Module usage - Atool
«module»
Atool.m
+ archFile :String
+ list_in :Cell
- mdl_file :Cell
- mdl_files :Cell
- singleTC :String
- singleTCs :Cell
+ analyse_pb_Callback(Scalar, Struct, Struct) :void
+ Atool() :void
+ atool_OpeningFcn(Scalar, Struct, Struct, String) :void
+ callSelectBox() :Vector
+ closeOpenWorkbooks() :void
+ defParams_Callback() :void
+ exit_pb_Callback(Scalar, Struct, Struct) :void
+ genCode_Callback() :void
+ generate_pb_Callback(Scalar, Struct, Struct) :void
+ getnames() :void
+ noEval_Callback() :void
+ openFunc_pb_Callback(Scalar, Struct, Struct) :void
+ openModel_pb_Callback(Scalar, Struct, Struct) :void
+ openTC_pb_Callback(Scalar, Struct, Struct) :void
+ openXLS(String, String) :void
+ popup_model_Callback(Scalar, Struct, Struct) :void
+ popup_singleTC_Callback(Scalar, Struct, Struct) :void
+ reset_pb_Callback(Scalar, Struct, Struct) :void
+ run_pb_Callback(Scalar, Struct, Struct) :void
+ searchMDLs(Struct) :void
+ searchSingleTCs(String, Struct) :Scalar
+ setOptions(Struct, String) :void
+ setUp_Callback(Scalar, none, Struct) :void
+ switchContext_pb_Callback(Scalar, Struct, Struct, String) :void
+ unpack() :void
«use»
«module»
Run::selectBox.m
+ list_out :Cell
+ ok_pb_Callback() :void
+ selectBox() :void
+ selectBox_OpeningFcn() :void
+ selectBox_OutputFcn() :void
+ tcListBox_Callback() :void
+ tcListBox_CreateFcn() :void
«use»
Module usage - create_TestModel
«module»
create_TestModel.m
- config_param(String) :void
- create_sl_trig() :void
+ create_TestModel(boolean) :String/Scalar
- createFunfile(String) :String
- findInDD(String, String) :void
- signalsusec(Cell) :String
«use»
«module»
Run::loadActx.m
+ loadActx(Scalar) :Scalar
«use»
Module usage - testCtrl
«module»
testCtrl.m
+ CodeCoverageOverall :Struct
- filename :String
+ ME_ctrl :MException
- tst_id
- getTCList(String) :Cell
- save2mat(String, boolean) :void
+ testCtrl(String, String, boolean, boolean, boolean, boolean) :void
«use»
Module usage - AnalysisWindow
«module»
AnalysisWindow.m
«use»
«module»
Run::Format_DD.m
+ Format_DD() :void
«use»
«module»
split.m
+ split(String, String) :String Array

Figure 4.7: Modules used by ATool GUI

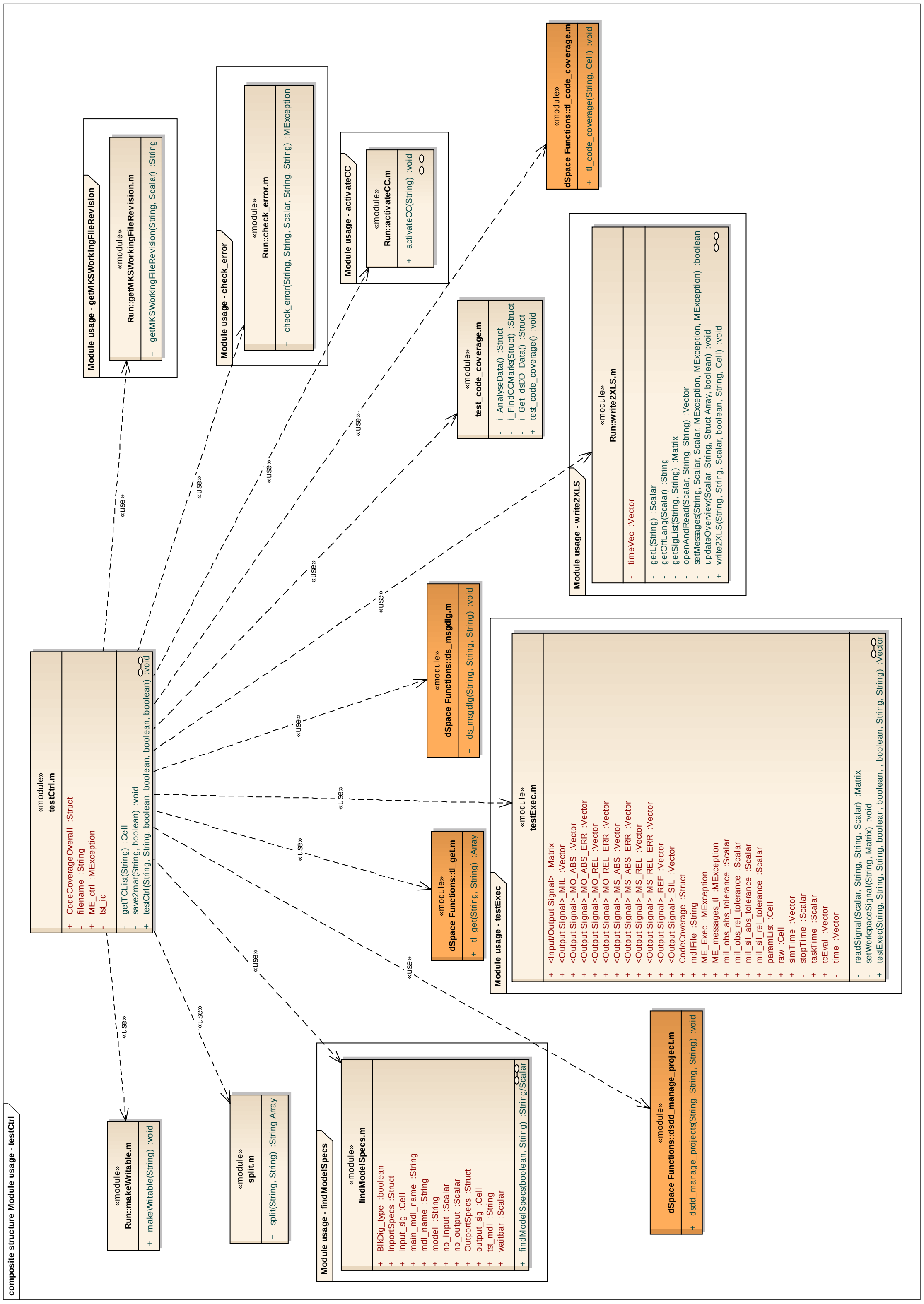

composite structure Module usage - testCtrl
«module» testCtrl.m
«module» Run::getMKSWorkingFileRevision.m
«module» Run::check_error.m
«module» Run::activateCC.m
«module» dSpace Functions::tl_code_coverage.m
«module» test_code_coverage.m
«module» Run::write2XLS.m
«module» dSpace Functions::ds_msgdlg.m
«module» testExec.m
«module» dSpace Functions::tl_get.m
«module» dSpace Functions::dsdd_manage_project.m
«module» findModelSpecs.m
«module» split.m
«module» Run::makeWritable.m
Module usage - getMKSWorkingFileRevision
Module usage - check_error
Module usage - activateCC
Module usage - write2XLS
Module usage - testExec
Module usage - findModelSpecs
«use»

Figure 4.8: Modules used by module ”testCtrl.m”

In parts data transfer within ATOOL is realized by means of global variables and base workspace variables. Within MATLAB the assignment of variables to base workspace and the evaluation of these base workspace variables is basically equal to the usage of global variables. Therefore, when speaking of global variables within the following, base workspace variables shall be considered as well. Within ATOOL each global variable is written exclusively by a single class. A global variable that is written by its class is represented by an attribute of this class. The data type and a short description of the global variable's content are contained by the notes field of the corresponding attribute. If the variable is read by more than one class, the attribute is specified as public. If the variable is just used by the subfunctions of a single class, it is specified as private. The usage of all public global variables is documented by means of multiple composite structure diagrams. It is ensured, that the depiction of a global variable is exclusive to a single diagram. Within a composite structure diagram each global variable transfer is represented by a port at the writing class, a port at the reading class and an assembly relationship. Figure 4.9 shows a data transfer realized by global variables. In case of an input port, a short description of the global variable and of its usage within the considered class is given in the notes field of the input port. The notes field of an output port contains only a remark on the class that sets the global variable.

A MS Excel sheet provides an overview on all global variables as well as all "eval" and "assign" commands[4] that are used within ATOOL. The usage of global variables is documented as follows. All functions contained by the ATOOL modules are listed in the first row of the MS Excel sheet. The second row of the sheet lists all subfunctions contained by the ATOOL functions. The first column lists all global variables. Global variables that are written by a function or subfunction are indicated by a "w" within the appropriate cell of the sheet. Global variables that are read by a function or subfunction are indicated by a "r" within the appropriate cell of the sheet. Figure 4.10 shows a screenshot of the MS Excel worksheet that provides an overview on the global variables of ATOOL. Global variables that are highlighted in yellow are only used within a single module. A global variable that is highlighted in grey is never read by any ATOOL module. Global variables that appear in green are written by a single function or subfunction and are read by more than one module. All functions or subfunction that are highlighted in blue use global variables. Compared with the documentation of global variables, the usage of "eval" and "assign" commands is documented in less detail. The usage of "eval" and "assign" commands is listed in a separate worksheet. In each line of the first two columns of that worksheet all functions and subfunctions of ATOOL are listed. For each function and subfunction the numbers of used "eval", "evalin", "evalin base", "assign", "assignin" and "assignin base" commands are listed.

[4]Datatransfer via the MATLAB base workspace is realized by means of "assignin" commands (write) and "evalin" commands (read).

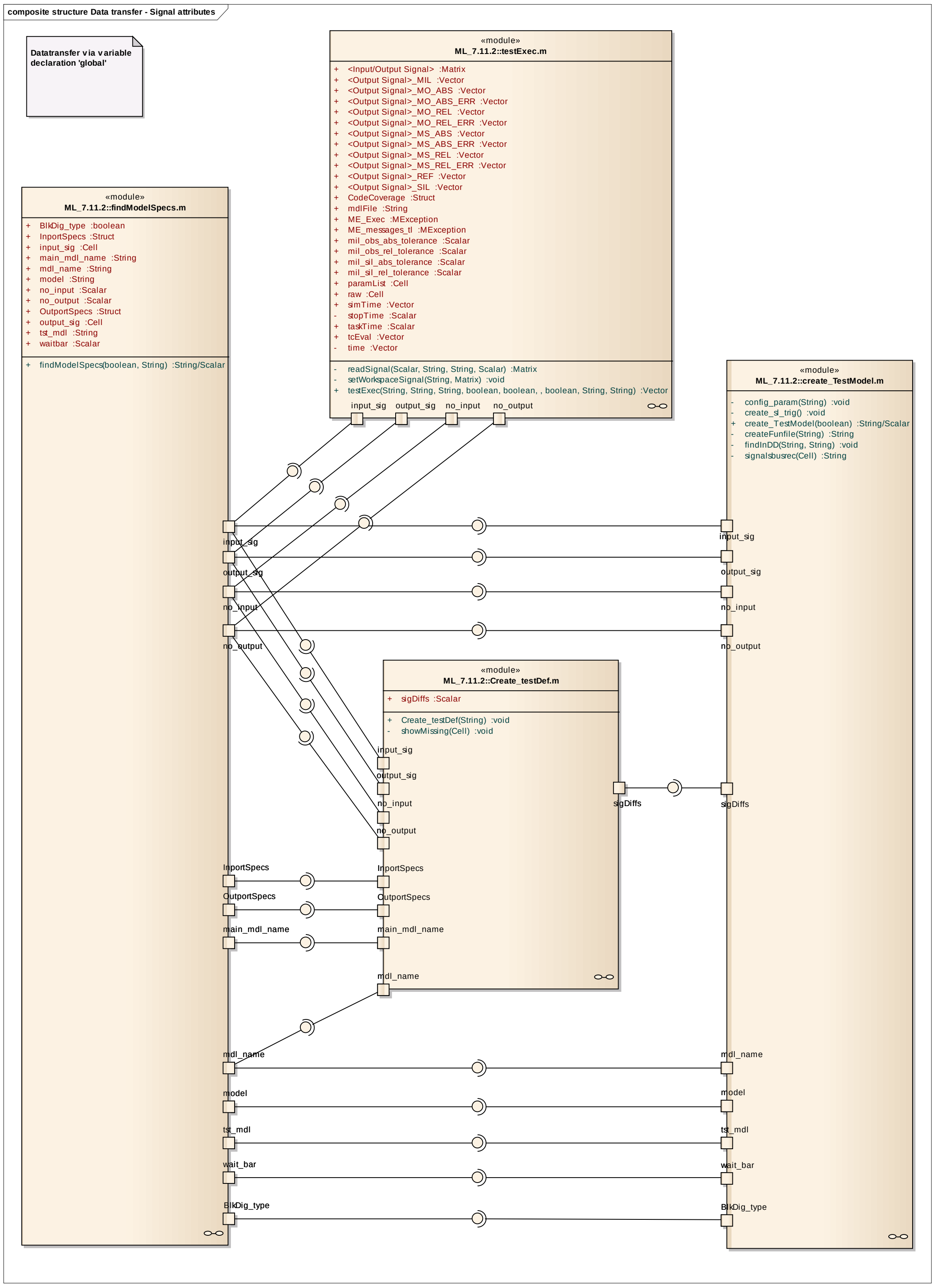
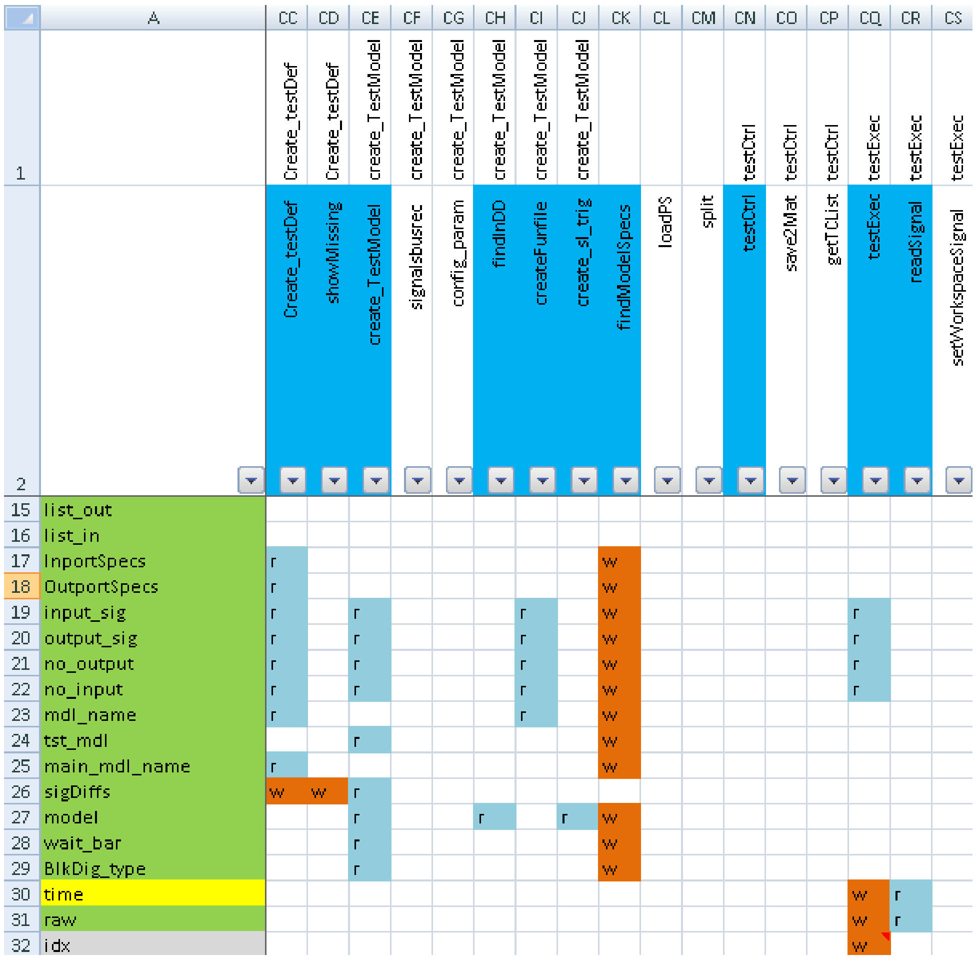

composite structure Data transfer - Signal attributes
Datatransfer via variable declaration 'global'
«module»
ML_7.11.2::testExec.m
+ <Input/Output Signal> :Matrix
+ <Output Signal>_MIL :Vector
+ <Output Signal>_MO_ABS :Vector
+ <Output Signal>_MO_ABS_ERR :Vector
+ <Output Signal>_MO_REL :Vector
+ <Output Signal>_MO_REL_ERR :Vector
+ <Output Signal>_MS_ABS :Vector
+ <Output Signal>_MS_ABS_ERR :Vector
+ <Output Signal>_MS_REL :Vector
+ <Output Signal>_MS_REL_ERR :Vector
+ <Output Signal>_REF :Vector
+ <Output Signal>_SIL :Vector
+ CodeCoverage :Struct
+ mdlFile :String
+ ME_Exec :MException
+ ME_messages_tl :MException
+ mil_obs_abs_tolerance :Scalar
+ mil_obs_rel_tolerance :Scalar
+ mil_sil_abs_tolerance :Scalar
+ mil_sil_rel_tolerance :Scalar
+ paramList :Cell
+ raw :Cell
+ simTime :Vector
- stopTime :Scalar
+ taskTime :Scalar
+ tcEval :Vector
- time :Vector
- readSignal(Scalar, String, String, Scalar) :Matrix
- setWorkspaceSignal(String, Matrix) :void
+ testExec(String, String, String, boolean, boolean, , boolean, String, String) :Vector
input_sig output_sig no_input no_output
«module»
ML_7.11.2::findModelSpecs.m
+ BlkDig_type :boolean
+ InportSpecs :Struct
+ input_sig :Cell
+ main_mdl_name :String
+ mdl_name :String
+ model :String
+ no_input :Scalar
+ no_output :Scalar
+ OutportSpecs :Struct
+ output_sig :Cell
+ tst_mdl :String
+ waitbar :Scalar
+ findModelSpecs(boolean, String) :String/Scalar
input_sig
output_sig
no_input
no_output
InportSpecs
OutportSpecs
main_mdl_name
mdl_name
model
tst_mdl
wait_bar
BlkDig_type
«module»
ML_7.11.2::create_TestModel.m
- config_param(String) :void
- create_sl_trig() :void
+ create_TestModel(boolean) :String/Scalar
- createFunfile(String) :String
- findInDD(String, String) :void
- signalsbusrec(Cell) :String
input_sig
output_sig
no_input
no_output
sigDiffs
mdl_name
model
tst_mdl
wait_bar
BlkDig_type
«module»
ML_7.11.2::Create_testDef.m
+ sigDiffs :Scalar
+ Create_testDef(String) :void
- showMissing(Cell) :void
input_sig
output_sig
no_input
no_output
InportSpecs
OutportSpecs
main_mdl_name
mdl_name
sigDiffs

Figure 4.9: Data transfer within ATool concerning signal attributes



	A	CC	CD	CE	CF	CG	CH	CI	CJ	CK	CL	CM	CN	CO	CP	CQ	CR	CS
1		Create_testDef	Create_testDef	create_TestModel	create_TestModel	create_TestModel	create_TestModel	create_TestModel	create_TestModel				testCtrl	testCtrl	testCtrl	testExec	testExec	testExec
2		Create_testDef	showMissing	create_TestModel	signalsbusrec	config_param	findInDD	createFunfile	create_sl_trig	findModelSpecs	loadPS	split	testCtrl	save2Mat	getTCList	testExec	readSignal	setWorkspaceSignal
15	list_out																	
16	list_in																	
17	InportSpecs	r								w								
18	OutportSpecs	r								w								
19	input_sig	r		r				r		w						r		
20	output_sig	r		r				r		w						r		
21	no_output	r		r				r		w						r		
22	no_input	r		r				r		w						r		
23	mdl_name	r						r		w								
24	tst_mdl			r						w								
25	main_mdl_name	r								w								
26	sigDiffs	w	w	r														
27	model			r			r		r	w								
28	wait_bar			r						w								
29	BlkDig_type			r						w								
30	time															w	r	
31	raw															w	r	
32	idx															w		

Figure 4.10: Overview on the usage of global variables within ATool

The content of the global variables overview Excel sheet is generated by means of two MATLAB scripts. The script "globalVariablesUsage.m" analyzes the code of ATool concerning the usage of global variables, "evalUsage.m" browses the code by "eval" and "assign" commands. Both scripts utilize the same principle. The principle of ATool code analysis is described as follows:

- Define the packages of ATool considered for the code analysis
- Walk through all considered packages and get all contained m-files
- Open each gathered m-file
- Load each code line of an opened m-file
- Perform actions based on the content of a code line. Therefore use regular expressions.

The activity diagram of "globalVariablesUsage.m" is depicted in Figure 4.11.

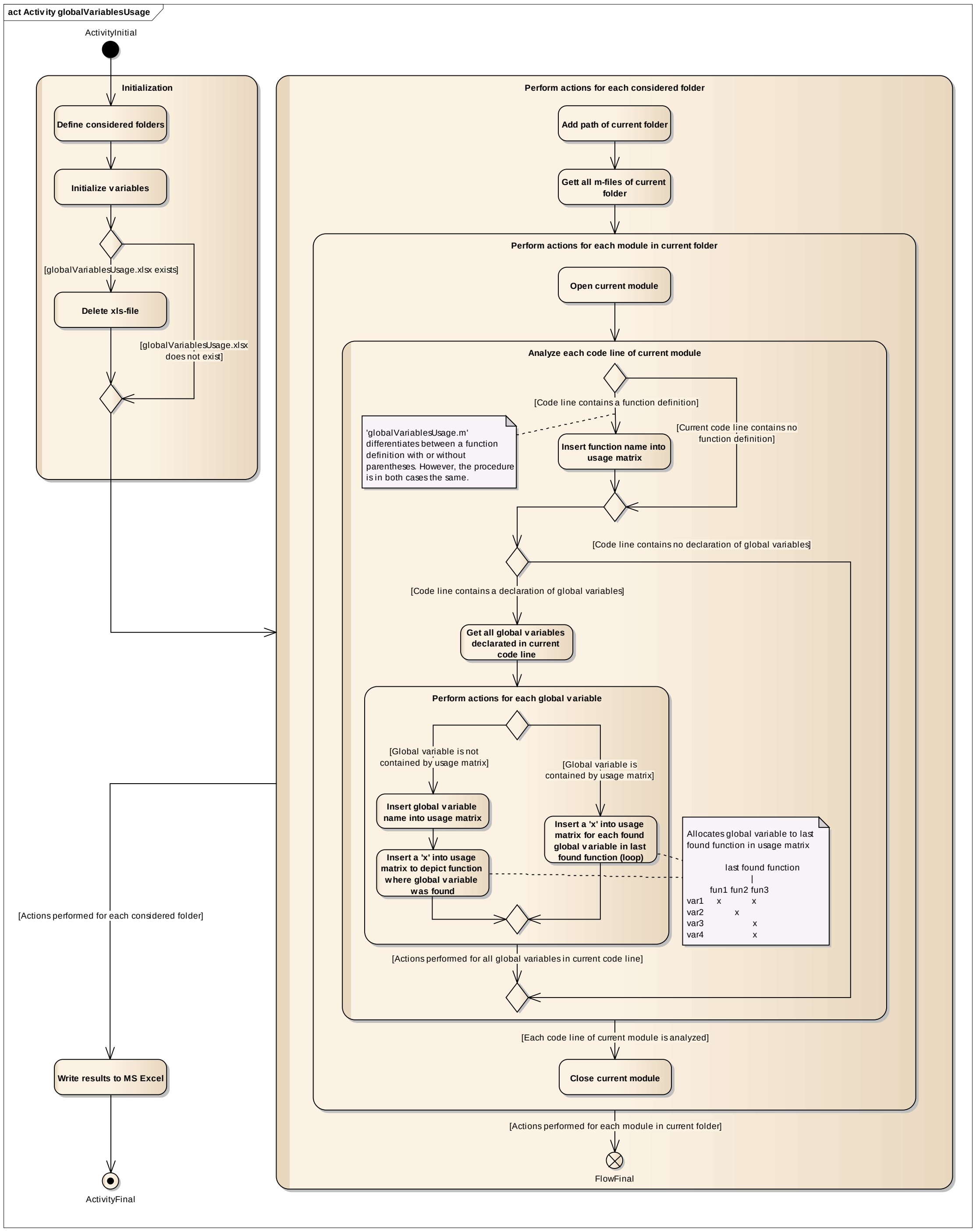

act Activity globalVariablesUsage
ActivityInitial
Initialization
Define considered folders
Initialize variables
[globalVariablesUsage.xlsx exists]
Delete xls-file
[globalVariablesUsage.xlsx does not exist]
Perform actions for each considered folder
Add path of current folder
Gett all m-files of current folder
Perform actions for each module in current folder
Open current module
Analyze each code line of current module
[Code line contains a function definition]
[Current code line contains no function definition]
'globalVariablesUsage.m' differentiates between a function definition with or without parentheses. However, the procedure is in both cases the same.
Insert function name into usage matrix
[Code line contains no declaration of global variables]
[Code line contains a declaration of global variables]
Get all global variables declarated in current code line
Perform actions for each global variable
[Global variable is not contained by usage matrix]
[Global variable is contained by usage matrix]
Insert global variable name into usage matrix
Insert a 'x' into usage matrix to depict function where global variable was found
Insert a 'x' into usage matrix for each found global variable in last found function (loop)
Allocates global variable to last found function in usage matrix
last found function
|
fun1 fun2 fun3
var1 x x
var2 x
var3 x
var4 x
[Actions performed for all global variables in current code line]
[Actions performed for each considered folder]
[Each code line of current module is analyzed]
Close current module
[Actions performed for each module in current folder]
Write results to MS Excel
FlowFinal
ActivityFinal

Figure 4.11: Activity diagram of MATLAB script to determine the usage of global variables within ATOOL

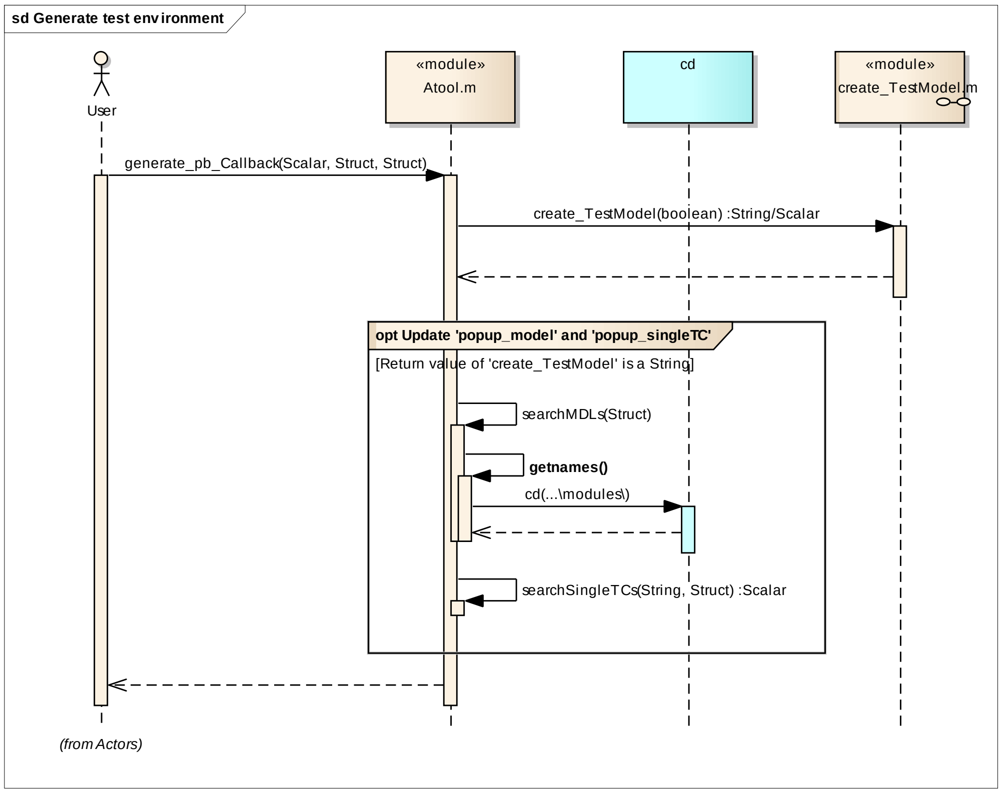

sd Generate test environment
User
«module»
Atool.m
cd
«module»
create_TestModel.m
generate_pb_Callback(Scalar, Struct, Struct)
create_TestModel(boolean) :String/Scalar
opt Update 'popup_model' and 'popup_singleTC'
[Return value of 'create_TestModel' is a String]
searchMDLs(Struct)
getnames()
cd(...\modules\)
searchSingleTCs(String, Struct) :Scalar
(from Actors)

Figure 4.12: Sequence diagram linked to use case ”Generate test environment”

The logical sequence of data processing is depicted by means of sequence diagrams. Each use case of ATool is linked with a sequence diagram that describes the series of called operations when performing a use case. Furthermore, each ATool class is linked with its own sequence diagram. As a result, the static module usage tree is replicated dynamically by the sequence diagrams. Figure 4.12 shows the sequence diagram linked to the use case ”Generate test environment”. In Figure 4.13 the linked sequence diagram of the ATool module ”create_TestModel” is depicted. To provide a better overview on the behavior of ATool, lifelines of some MATLAB functions as well as lifelines of TargetLink functions are depicted, too. MATLAB lifelines appear in turquoise and TargetLink lifelines appear in orange to provide an unambiguous distinction to all lifelines of ATool modules.

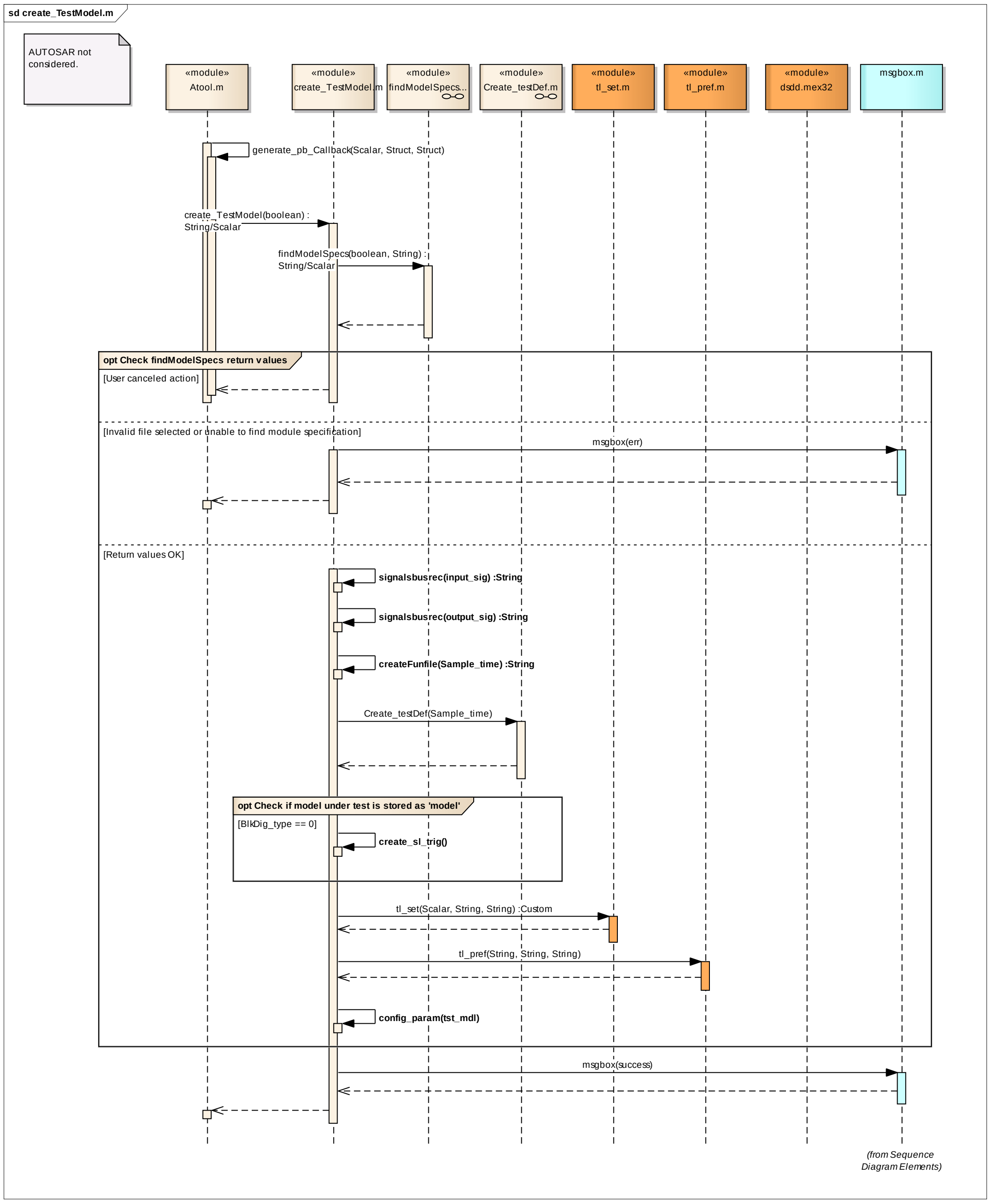

sd create_TestModel.m
AUTOSAR not considered.
«module» Atool.m
«module» create_TestModel.m
«module» findModelSpecs...
«module» Create_testDef.m
«module» tl_set.m
«module» tl_pref.m
«module» dsdd.mex32
msgbox.m
generate_pb_Callback(Scalar, Struct, Struct)
create_TestModel(boolean) : String/Scalar
findModelSpecs(boolean, String) : String/Scalar
opt Check findModelSpecs return values
[User canceled action]
[Invalid file selected or unable to find module specification]
msgbox(err)
[Return values OK]
signalsbusrec(input_sig) :String
signalsbusrec(output_sig) :String
createFunfile(Sample_time) :String
Create_testDef(Sample_time)
opt Check if model under test is stored as 'model'
[BlkDig_type == 0]
create_sl_trig()
tl_set(Scalar, String, String) :Custom
tl_pref(String, String, String)
config_param(tst_mdl)
msgbox(success)
(from Sequence Diagram Elements)

Figure 4.13: Sequence diagram of module "create_TestModel.m"

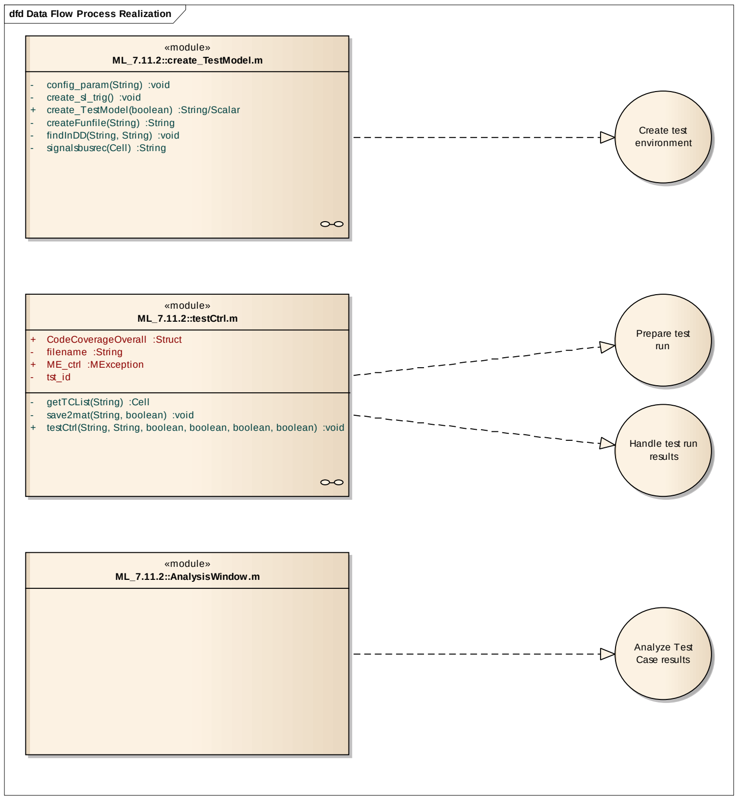

dfd Data Flow Process Realization
«module»
ML_7.11.2::create_TestModel.m
- config_param(String) :void
- create_sl_trig() :void
+ create_TestModel(boolean) :String/Scalar
- createFunfile(String) :String
- findInDD(String, String) :void
- signalsbusrec(Cell) :String
«module»
ML_7.11.2::testCtrl.m
+ CodeCoverageOverall :Struct
- filename :String
+ ME_ctrl :MException
- tst_id
- getTCList(String) :Cell
- save2mat(String, boolean) :void
+ testCtrl(String, String, boolean, boolean, boolean, boolean) :void
«module»
ML_7.11.2::AnalysisWindow.m
Create test environment
Prepare test run
Handle test run results
Analyze Test Case results

Figure 4.14: Realization of the processes depicted in the data flow diagram

The external interfaces of ATool are described in a data flow diagram. Therefore, tool processes are defined. It must be mentioned, that the defined tool processes and the tool use cases are not the same. However, the ATool processes represent overall tool functions that are implemented in MATLAB. To provide consistency over all diagram types, the classes that realize the defined ATool processes are shown in Figure 4.14. Software tools that support ATool are depicted as solid blocks, datastorages are represented by two parallel lines. The data flows are visualized by arrowed lines. The labels of those lines contain the data that is sent or received, respectively.

Figure 4.15 shows the external data flow of ATool. As can be seen, each process is linked with a note element. Within a note element the sequence of the processes is described. Furthermore it can be seen, that ATool sets Simulink parameters when creating a test environment and preparing a test run. An overview on all Simulink parameters set by ATool is contained by a MS Excel sheet called ATool parameter list. This file provides a worksheet for each Simulink layer where parameters are modified by ATool. For each parameter modified by ATool the parameter name, the parameter value, the ATool module that assigned the parameter value and additional comments concerning the considered parameter are given. The working steps that need to be performed when updating the ATool parameter list are given in worksheet "Methodology".

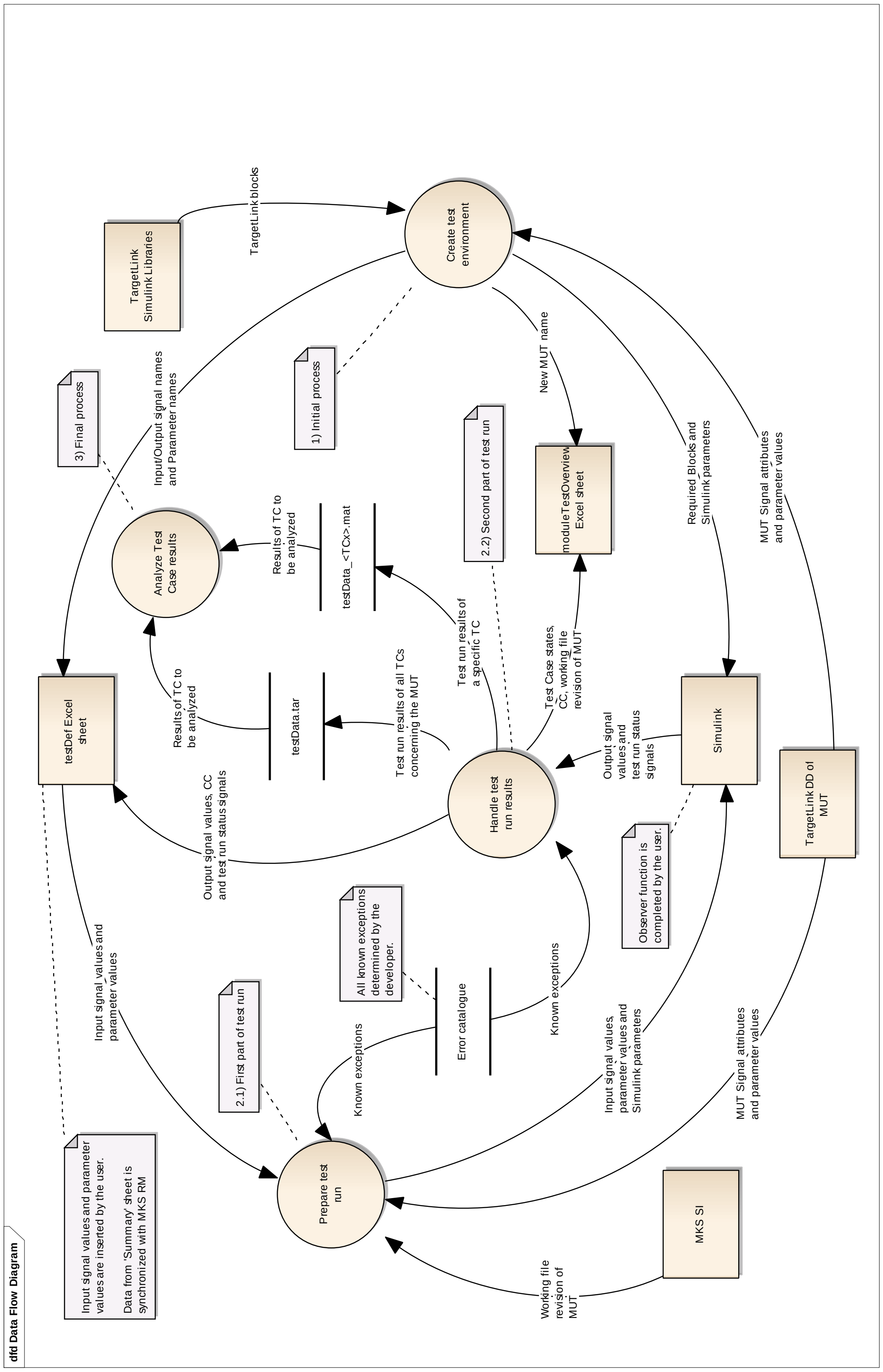

dfd Data Flow Diagram
TargetLink Simulink Libraries
TargetLink blocks
Create test environment
1) Initial process
New MUT name
moduleTestOverview Excel sheet
Required Blocks and Simulink parameters
MUT Signal attributes and parameter values
Input/Output signal names and Parameter names
3) Final process
Analyze Test Case results
Results of TC to be analyzed
testData_<TCx>.mat
2.2) Second part of test run
Test run results of a specific TC
Test Case states, CC, working file revision of MUT
testDef Excel sheet
Results of TC to be analyzed
testData.tar
Test run results of all TCs concerning the MUT
Handle test run results
Output signal values and test run status signals
Simulink
TargetLink DD of MUT
Output signal values, CC and test run status signals
Observer function is completed by the user.
All known exceptions determined by the developer.
Error catalogue
Known exceptions
Known exceptions
Input signal values and parameter values
2.1) First part of test run
Input signal values, parameter values and Simulink parameters
MUT Signal attributes and parameter values
Prepare test run
Input signal values and parameter values are inserted by the user.
Data from 'Summary' sheet is synchronized with MKS RM
MKS SI
Working file revision of MUT

Figure 4.15: ATool data flow diagram

Software unit design

The software unit design is realized by means of activity diagrams. Each ATool module is related to an activity diagram that describes the behavior within the module. However, no activity diagrams are created for modules that show a low complexity. The unit design of those modules is represented by an informal description within the class' notes field. A special case is given for the module called "Atool". This module handles the ATool GUI. No activity diagram is created for "Atool" because of the fact, that an activity diagram for a GUI handling is too much effort by providing less overview. Instead of a semi-formal description of the GUI handling, an informal explanation is given within the notes field of each operation of "Atool".

The symbols that are used for the ATool activity diagrams and their meaning within an ATool m-file is described in Table 4.3. It must be mentioned that actions elements can represent parts of the code that includes for instance if-else commands. To provide clear and expressive activity diagrams, less important parts of the documented code are summarized to action elements. However, the behavior of the summarized code is described within the notes field of its corresponding action element.

Activity diagram elements	*Meaning within a m-file*
Action	Functional unit
Activity	Set of actions that have a logical context (e.g. init actions, for-loops, while-loops, ...)
ActivityInitial	"function" command
ActivityFinal	return
Decision	if-elseif-else, swich-case-otherwise
Interruptible activity region	try-block
Exception handler	catch-block
Receive	Exception that causes the execution of the catch-block

Table 4.3: Used activity diagram elements as understood for the documentation of a m-file.

An exemplary activity diagram is given in Figure 4.16. It shows how ATool gathers signal specific information from the TargetLink data dictionary.

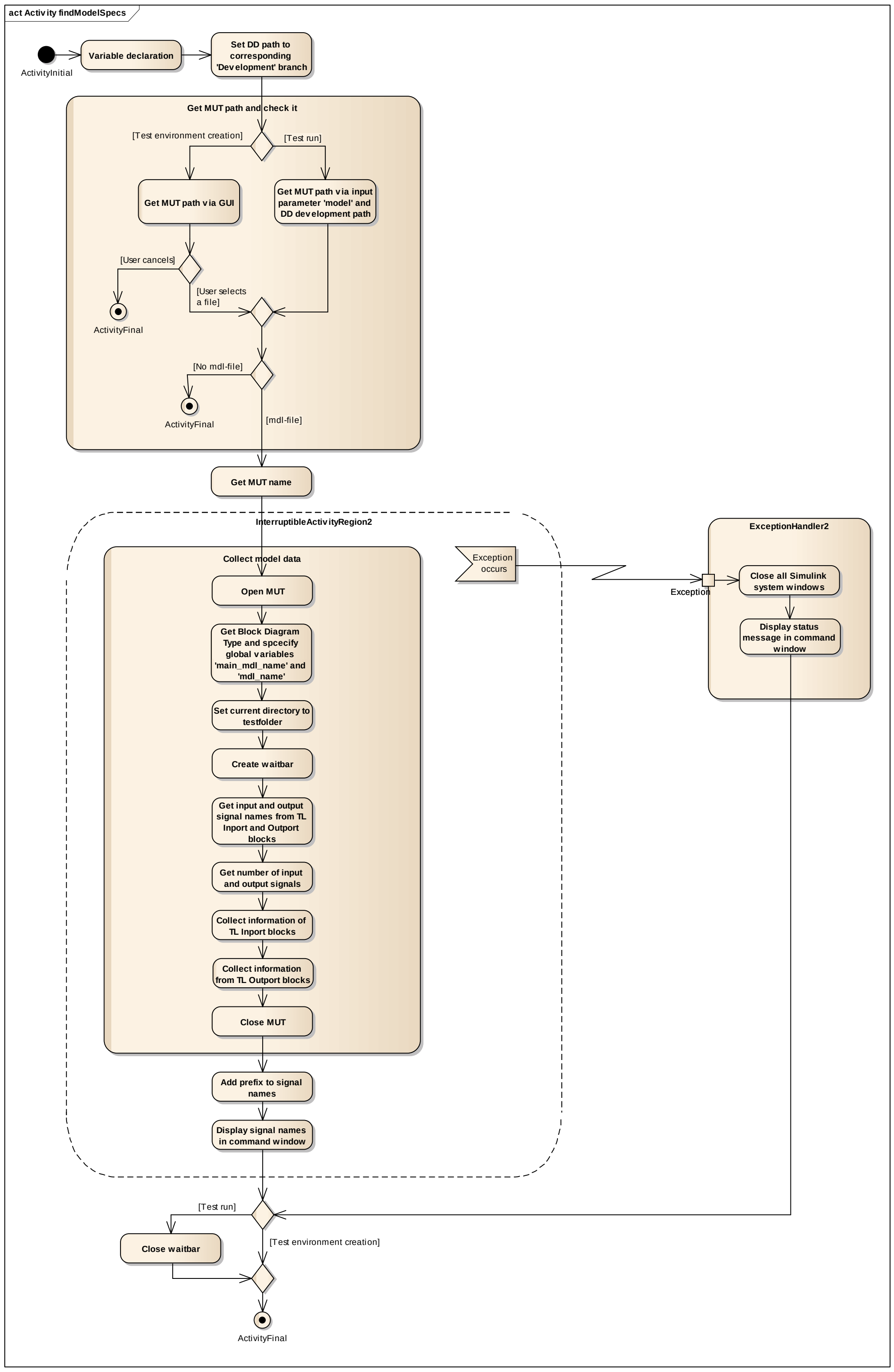

act Activity findModelSpecs
ActivityInitial
Variable declaration
Set DD path to corresponding 'Development' branch
Get MUT path and check it
[Test environment creation]
[Test run]
Get MUT path via GUI
Get MUT path via input parameter 'model' and DD development path
[User cancels]
[User selects a file]
ActivityFinal
[No mdl-file]
ActivityFinal
[mdl-file]
Get MUT name
InterruptibleActivityRegion2
Collect model data
Open MUT
Get Block Diagram Type and spcecify global variables 'main_mdl_name' and 'mdl_name'
Set current directory to testfolder
Create waitbar
Get input and output signal names from TL Inport and Outport blocks
Get number of input and output signals
Collect information of TL Inport blocks
Collect information from TL Outport blocks
Close MUT
Add prefix to signal names
Display signal names in command window
Exception occurs
ExceptionHandler2
Exception
Close all Simulink system windows
Display status message in command window
[Test run]
Close waitbar
[Test environment creation]
ActivityFinal

Figure 4.16: Activity diagram of module "findModelSpecs.m"

4.3 Validation and verification

Compared with an automotive embedded software ATool shows a low complexity. Therefore, most of the applicable verification steps required by [ISO11a] are covered by the validation process. The verification step that must be performed in addition to the ATool validation is a design and code inspection. All ATool defects and errors can be detected by stimulating the GUI and modifying the inputs of ATool's modules. The validation is realized by checking the output of ATool for each ATool scenario. The validation is composed of two phases. First, a test run without any provoked errors is performed. At this phase the inputs of the ATool modules are not modified. After that, the second phase of validation starts. By means of fault injection tests each known error is provoked. This is performed, by corrupting the inputs of the ATool modules. The error handling of ATool must detect all inserted faults and as a result ATool has to be stopped. These two phases of tool validation are contained by the ATool test suite. To avoid negative impacts on ATool caused by the configuration of the workstation, the test suite is executed after each installation.

The data flow of the ATool test suite is shown in Figure 4.17. Again, the circles represent the processes performed for an ATool validation. Data stores external to MATLAB are represented by solid blocks and internal MATLAB data stores are represented by two parallel lines. The arrowed lines show the transferred data. A description of the process and its sequence is given by a yellow comment, a gray comment signalizes the path of stored data. As can be seen in Figure 4.17, the ATool validation is performed according to the following steps:

1. Automatic insertion of code coverage flags
2. Execution of test cases specified in MKS RM (test suite)
3. Test result analysis

The subsequent sections deal with those validation steps and explain them in detail.

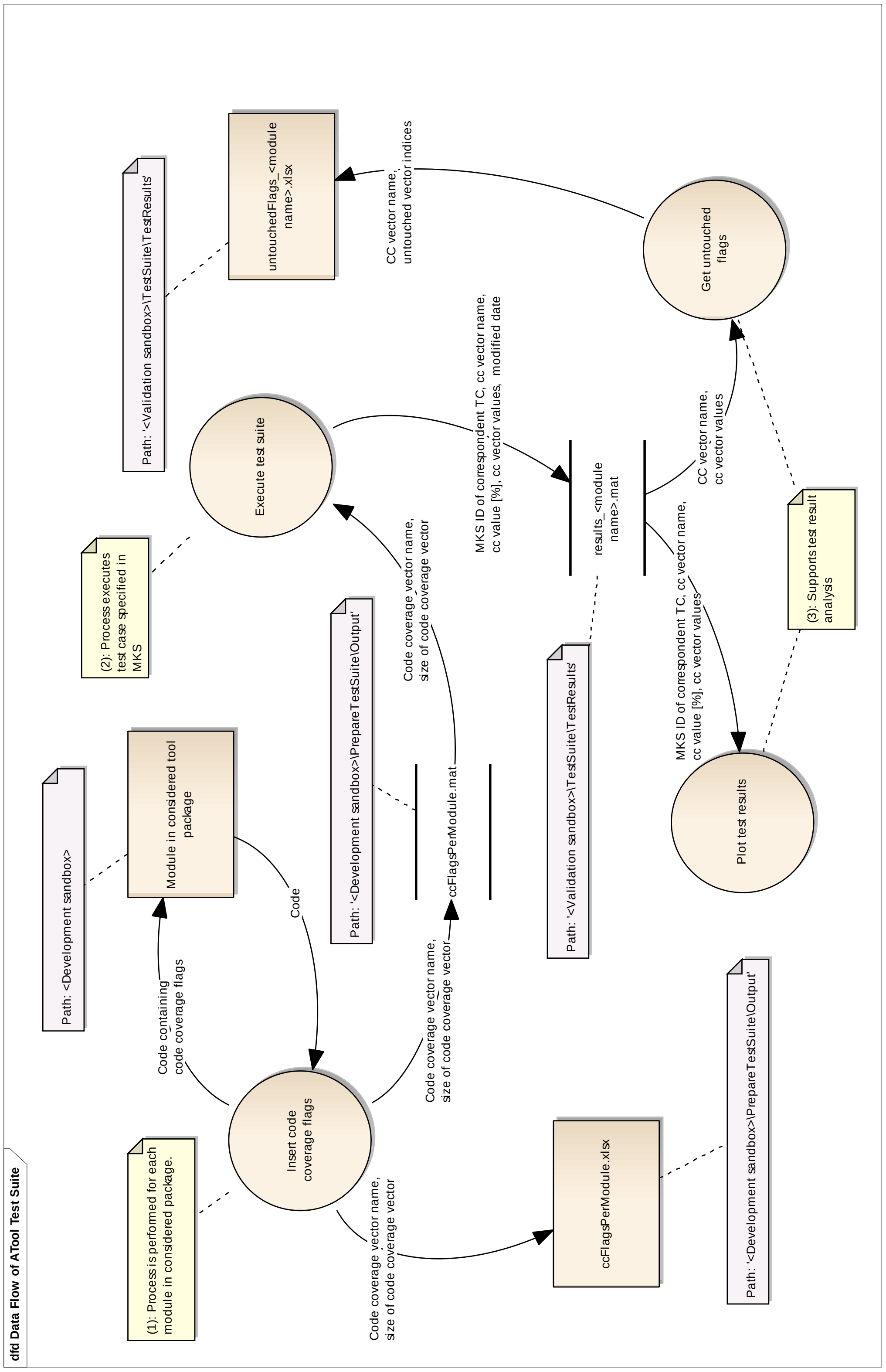

dfd Data Flow of ATool Test Suite
(1): Process is performed for each module in considered package.
Insert code coverage flags
Code coverage vector name, size of code coverage vector
ccFlagsPerModule.xlsx
Path: '<Development sandbox>\PrepareTestSuite\Output'
Path: <Development sandbox>
Module in considered tool package
Code containing code coverage flags
Code
Code coverage vector name, size of code coverage vector
ccFlagsPerModule.mat
Path: '<Development sandbox>\PrepareTestSuite\Output'
Code coverage vector name, size of code coverage vector
(2): Process executes test case specified in MKS
Execute test suite
Path: '<Validation sandbox>\TestSuite\TestResults'
untouchedFlags_<module name>.xlsx
CC vector name, untouched vector indices
MKS ID of correspondent TC, cc vector name, cc value [%], cc vector values, modified date
results_<module name>.mat
Path: '<Validation sandbox>\TestSuite\TestResults'
Get untouched flags
CC vector name, cc vector values
MKS ID of correspondent TC, cc vector name, cc value [%], cc vector values
(3): Supports test result analysis
Plot test results

Figure 4.17: Data flow of ATool test suite

4.3.1 Code instrumentation

As already mentioned, before executing the ATool test suite the code of ATool has to be prepared. That means, measures to provide the determination of test coverage must be taken. Therefore, the ATool code is instrumented to establish function coverage and branch coverage. The code of ATool is modified by inserting code coverage flags. Therefore, a global vector containing elements of type boolean is added to each ATool module. The name of such a global vector is composed of the prefix "atoolCC_" and the corresponding module name (e.g. atoolCC_testExec). These vectors are called code coverage vectors, the vector elements represent the code coverage flags. The code coverage flags are initialized with zero. After a key word within the ATool code a code line is added, that sets a code coverage flag to one.

The key words signalizing the different types of code coverage:

- Function coverage:
  - function
- Branch coverage:
  - if
  - elseif
  - else
  - case
  - otherwise
  - try
  - catch

The vectors must be declared as global variables, because then the code coverage information is available for the test suite. This is ensured, even when the execution of a module is stopped because of an exception occurred. Furthermore, the insertion of global variables into the ATool code reduces the necessary changes of the code to a minimum.

In Figure 4.18 the sequence of function calls when inserting code coverage flags is depicted. Again, lifelines representing MATLAB functions that are shown for a better interpretation of the diagram are highlighted in turquoise. As the name suggests, the module "insertCodeCoverageFlags.m" inserts the code coverage flags to each module contained by a folder of the considered ATool package. In addition an overview list in MS Excel called "ccFlagsPerModule.xlsx" is generated (see Figure 4.17). This list contains the names of all created code coverage vectors and their size.

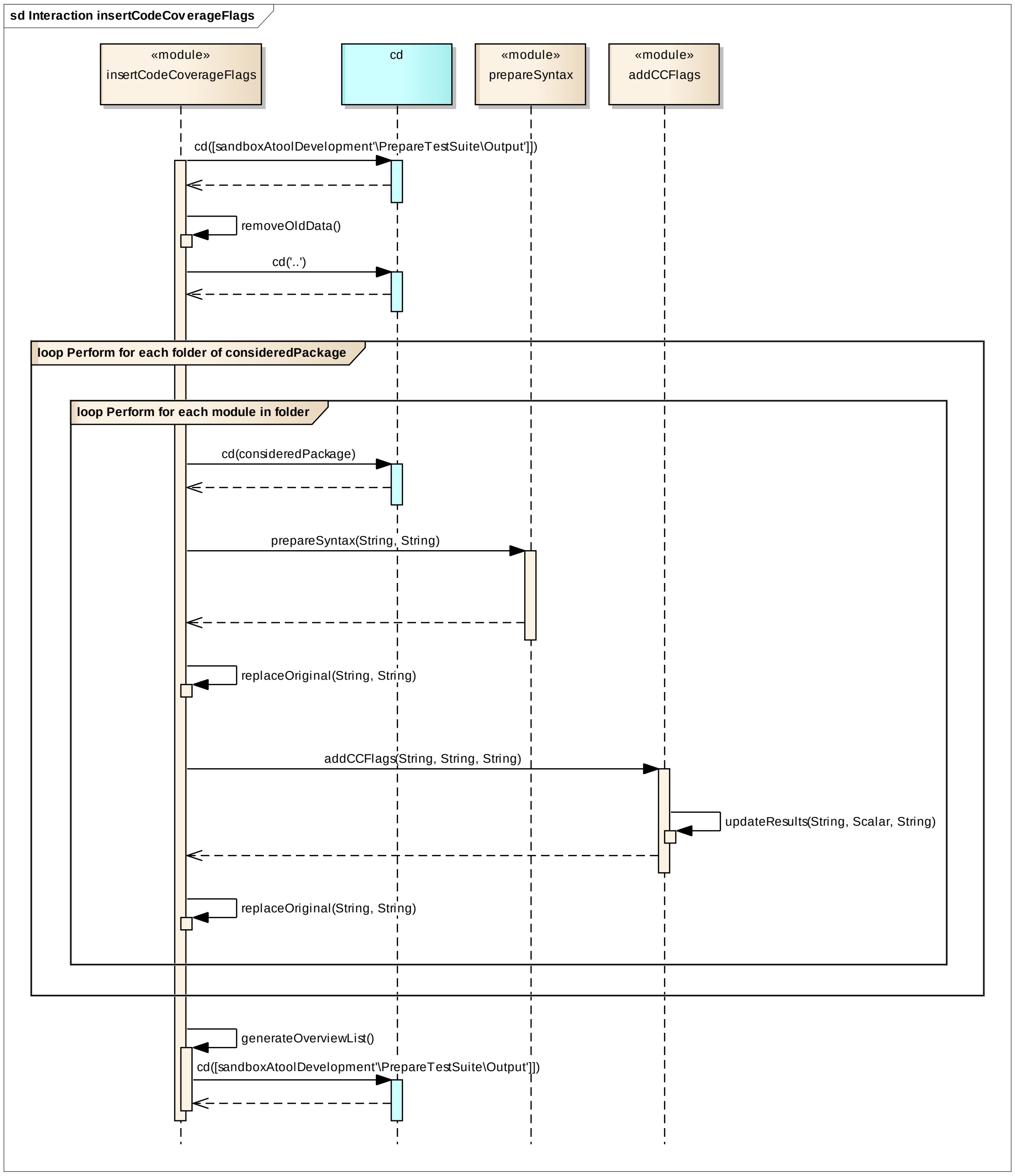

sd Interaction insertCodeCoverageFlags
«module»
insertCodeCoverageFlags
cd
«module»
prepareSyntax
«module»
addCCFlags
cd([sandboxAtoolDevelopment'\PrepareTestSuite\Output'])
removeOldData()
cd('..')
loop Perform for each folder of consideredPackage
loop Perform for each module in folder
cd(consideredPackage)
prepareSyntax(String, String)
replaceOriginal(String, String)
addCCFlags(String, String, String)
updateResults(String, Scalar, String)
replaceOriginal(String, String)
generateOverviewList()
cd([sandboxAtoolDevelopment'\PrepareTestSuite\Output'])

Figure 4.18: Sequence of function calls when inserting code coverage flags

Before executing	*After executing*
	if(condition) %MODIFIED. . .
if(condition);statement;end	statement;
	end
	if(condition) %MODIFIED. . .
	statement;
if(condition); statement; else; statement; end	else
	statement;
	end
	try %MODIFIED. . .
	statement;
try; statement; catch; statement; end	catch
	statement;
	end
	catch %MODIFIED. . .
catch; statement; end	statement;
	end

Table 4.4: Syntax changed by ”prepareSyntax”

As can be seen in Figure 4.18, the function ”prepareSyntax” is called before adding the code coverage flags to a module of ATool. This function ensures that a line break appears after a key word in a code line. Table 4.4 lists code lines before and after execution of ”prepareSyntax”. Only the key words if, else, try and catch are concerned by ”prepareSyntax”. All other key words are indicators for a higher complexity, hence an implementation within a single code line will not be performed by a programmer. The modified code of the input file of ”prepareSyntax” is stored in an output file. Deviations between the input and the output file are signalized by comments at the end of a modified line. Figure 4.19 and Figure 4.3.1 show the activity diagrams of ”prepareSyntax”.

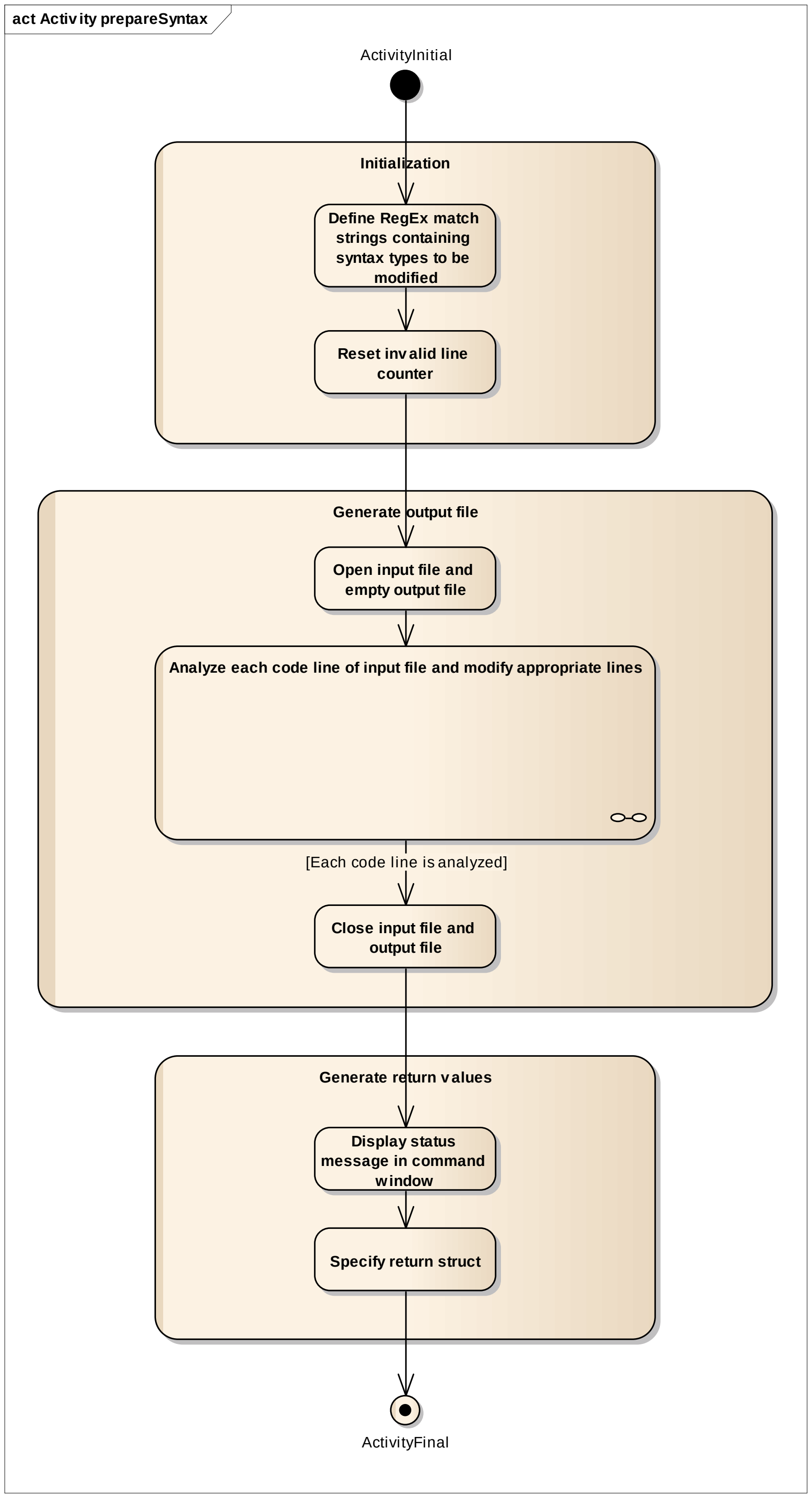
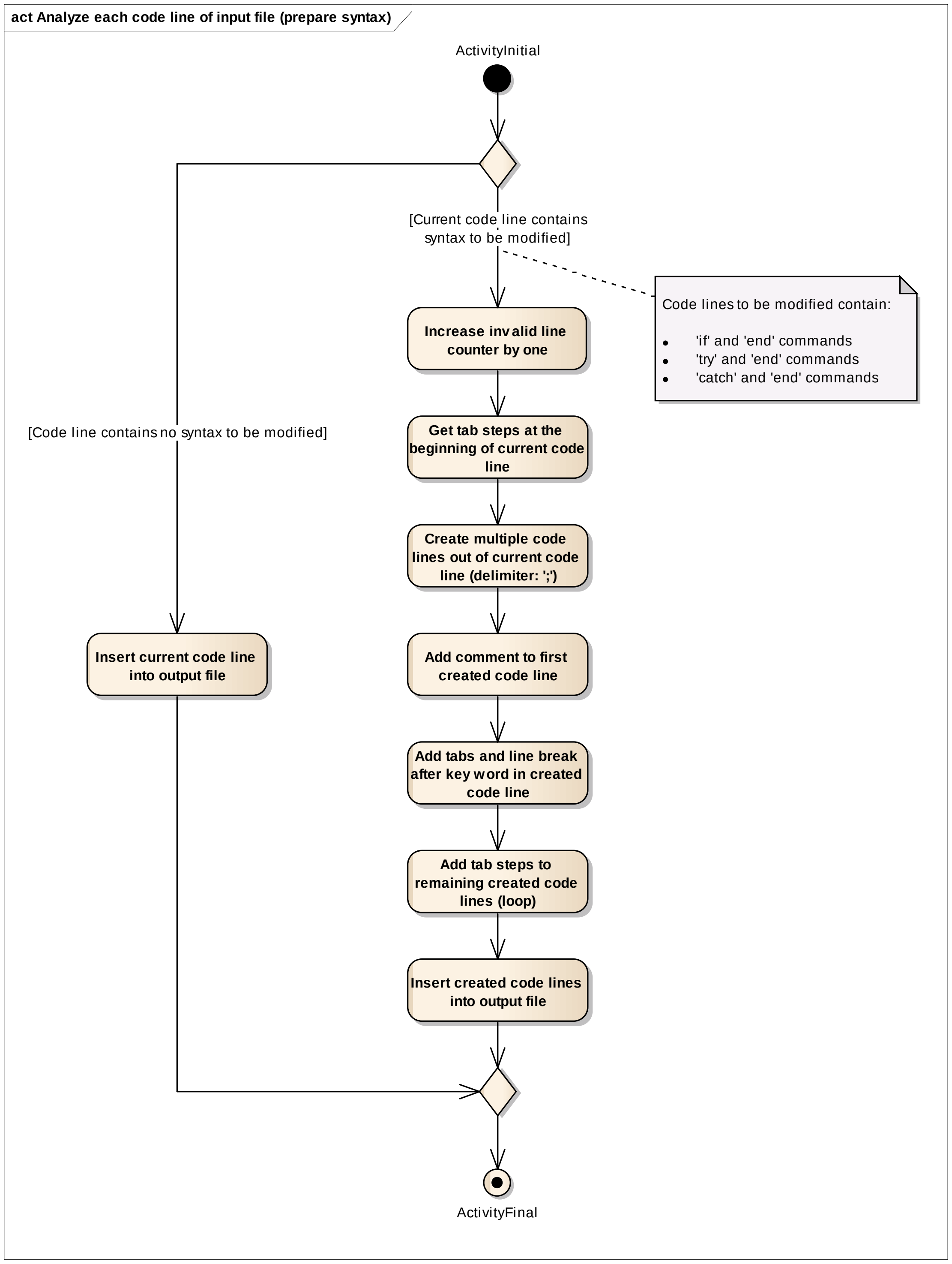

act Activity prepareSyntax
ActivityInitial
Initialization
Define RegEx match strings containing syntax types to be modified
Reset invalid line counter
Generate output file
Open input file and empty output file
Analyze each code line of input file and modify appropriate lines
[Each code line is analyzed]
Close input file and output file
Generate return values
Display status message in command window
Specify return struct
ActivityFinal

Figure 4.19: Activity diagram of "prepareSyntax.m"


act Analyze each code line of input file (prepare syntax)
ActivityInitial
[Current code line contains syntax to be modified]
Code lines to be modified contain:
• 'if' and 'end' commands
• 'try' and 'end' commands
• 'catch' and 'end' commands
Increase invalid line counter by one
Get tab steps at the beginning of current code line
Create multiple code lines out of current code line (delimiter: ';')
Add comment to first created code line
Add tabs and line break after key word in created code line
Add tab steps to remaining created code lines (loop)
Insert created code lines into output file
[Code line contains no syntax to be modified]
Insert current code line into output file
ActivityFinal

Before executing	*After executing*
function statement	function statement
	global atoolCC_<module name>
	atoolCC_<module name>(<counter value>) = 1;
if(condition)	if(condition)
statement;	atoolCC_<module name>(<counter value>) = 1;
elseif(condition1 && ...	statement;
condition2)	elseif(condition1 && ...
statement;	condition2)
else	atoolCC_<module name>(<counter value>) = 1;
statement;	statement;
end	else
	atoolCC_<module name>(<counter value>) = 1;
	statement;
	end
switch	switch
case(condition)	case(condition)
statement;	atoolCC_<module name>(<counter value>) = 1;
otherwise	statement;
statement;	otherwise
end	atoolCC_<module name>(<counter value>) = 1;
	statement;
	end
try	try
statement;	atoolCC_<module name>(<counter value>) = 1;
catch	statement;
	catch
end	atoolCC_<module name>(<counter value>) = 1;
	end

Table 4.5: Modifications of the ATool code caused by "addCCFlags"

Figure 4.18 shows, that after preparing the syntax of a module the original module of ATool is replaced by the generated output file of "prepareSyntax". The prepared module of ATool is used as the input file of "addCCFlags". The behavior of "addCCFlags" and "prepareSyntax" is similar. Instead of inserting line breaks after a key word, a code line that sets a code coverage flag is inserted to the prepared code by "addCCFlags". Table 4.5 lists exemplary code lines before and after execution of "addCCFlags". Furthermore, the structure of a code coverage flag is shown in the table. As a result of the correspondence between these two modules Figure 4.19 showing the activity diagram of "prepareSyntax" and Figure 4.20 showing the activity diagram of "addCCFlags" are similar. The two main differences are the execution of different actions when analyzing a code line (compare Figure 4.3.1 with Figure 4.21) and the actions performed at the generation of return values. The return values differ, because "addCCFlags" updates the size of an added code coverage vector within "ccFlagsPerModule.mat" (see Figure 4.17).

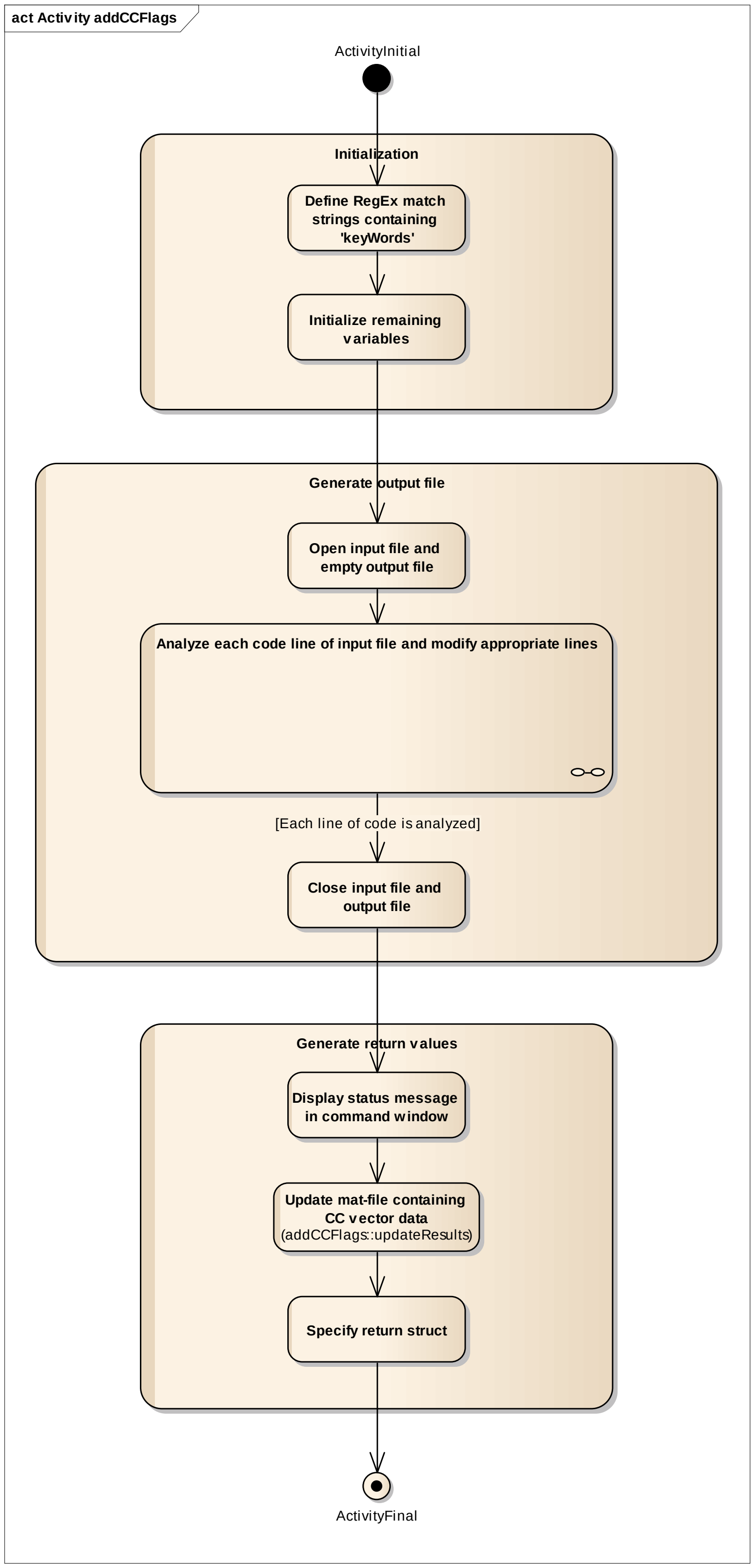

act Activity addCCFlags
ActivityInitial
Initialization
Define RegEx match strings containing 'keyWords'
Initialize remaining variables
Generate output file
Open input file and empty output file
Analyze each code line of input file and modify appropriate lines
[Each line of code is analyzed]
Close input file and output file
Generate return values
Display status message in command window
Update mat-file containing CC vector data (addCCFlags::updateResults)
Specify return struct
ActivityFinal

Figure 4.20: Activity diagram of "addCCFlags.m"

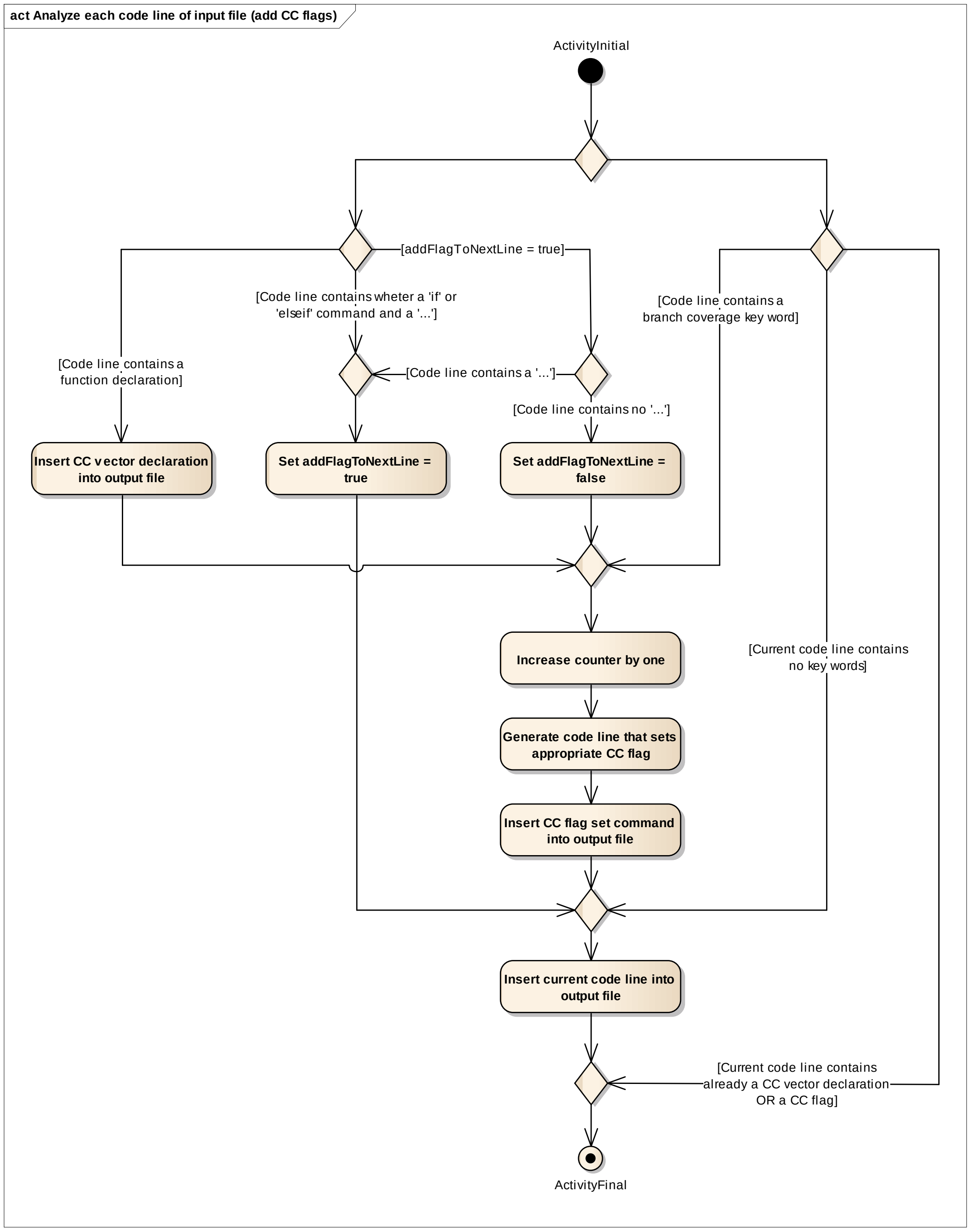

act Analyze each code line of input file (add CC flags)
ActivityInitial
[addFlagToNextLine = true]
[Code line contains wheter a 'if' or 'elseif' command and a '...']
[Code line contains a branch coverage key word]
[Code line contains a function declaration]
[Code line contains a '...']
[Code line contains no '...']
Insert CC vector declaration into output file
Set addFlagToNextLine = true
Set addFlagToNextLine = false
Increase counter by one
[Current code line contains no key words]
Generate code line that sets appropriate CC flag
Insert CC flag set command into output file
Insert current code line into output file
[Current code line contains already a CC vector declaration OR a CC flag]
ActivityFinal

Figure 4.21: Activity that analyzes each code line at ”addCCFlags.m”

4.3.2 Test suite

The ATool test suite is controlled under configuration management. That means, within MKS SI a project is created for the purpose of ATool validation. This validation project in MKS SI contains the ATool test suite and the ATool test environment. The test environment containing the MUTs is based on a real development project at *Magna Powertrain*. As a result, the input test vectors for ATool validation are represented by real MUTs.

Figure 4.22 shows the steps to be performed to attain the test specification of the ATool test suite. The document is based on the content of the UML model. Therefore, the first step is to generate a class report that addresses all ATool modules. Within the report a short description of the module functionality as well as all inputs and all outputs are listed for each module. This means the report shall contain the class notes as well as the embedded elements, operations and attributes of a class. For the next step the use case diagram has to be considered. A set of scenarios has to be defined for each use case of ATool. The defined scenarios can be positive as well as negative (error handling) and represent an ATool test case. The appropriate scenario path has to be followed within the corresponding sequence diagrams for each scenario. By doing this, the sequence of called ATool modules of a scenario is determined. By consulting the class report, the inputs and outputs of a called module are used to specify a test case. The expected behavior of an ATool module is represented by its output. To specify the first test run only the module outputs and their sequence have to be considered. However, the module inputs that shall be corrupted have to be considered for the fault injection tests as well.

The module "atoolTestSuite" handles the execution of a test case for the purpose of ATool validation. The sequence of function calls of "atoolTestSuite" is shown in Figure 4.23. As can be seen, first the generic commands concerning an initialization of the code coverage vectors are performed. The commands address the global variable declaration of each code coverage vector as well as the initialization of each code coverage vector with zero. The required information for the generation of those generic commands is gathered from "ccFlagsPerModule.mat" (see Figure 4.17). After that, the ATool startup modified for ATool test environment is called. The difference between the original ATool startup and the test environment startup is, that the original ATool startup is automatically called after a MATLAB start. However, the test environment startup must be called after code coverage vector initialization. Furthermore, all variables of test environment startup used for ATool initialization must be assigned to base workspace explicitly to provide a correct functioning of ATool at the test environment. It is mentioned, that the test environment startup contains code coverage flags. The test environment startup opens the ATool GUI within the test environment. Subsequently, a test case of the ATool test suite is performed by the tool tester. When a test case is finished the tool tester presses any key and the results of the executed test case are saved. Each code coverage vector that contains code coverage flags set to one is stored in its corresponding mat-file. The name of the mat-file is composed of the prefix "results_" and the corresponding module name (see Figure 4.17).

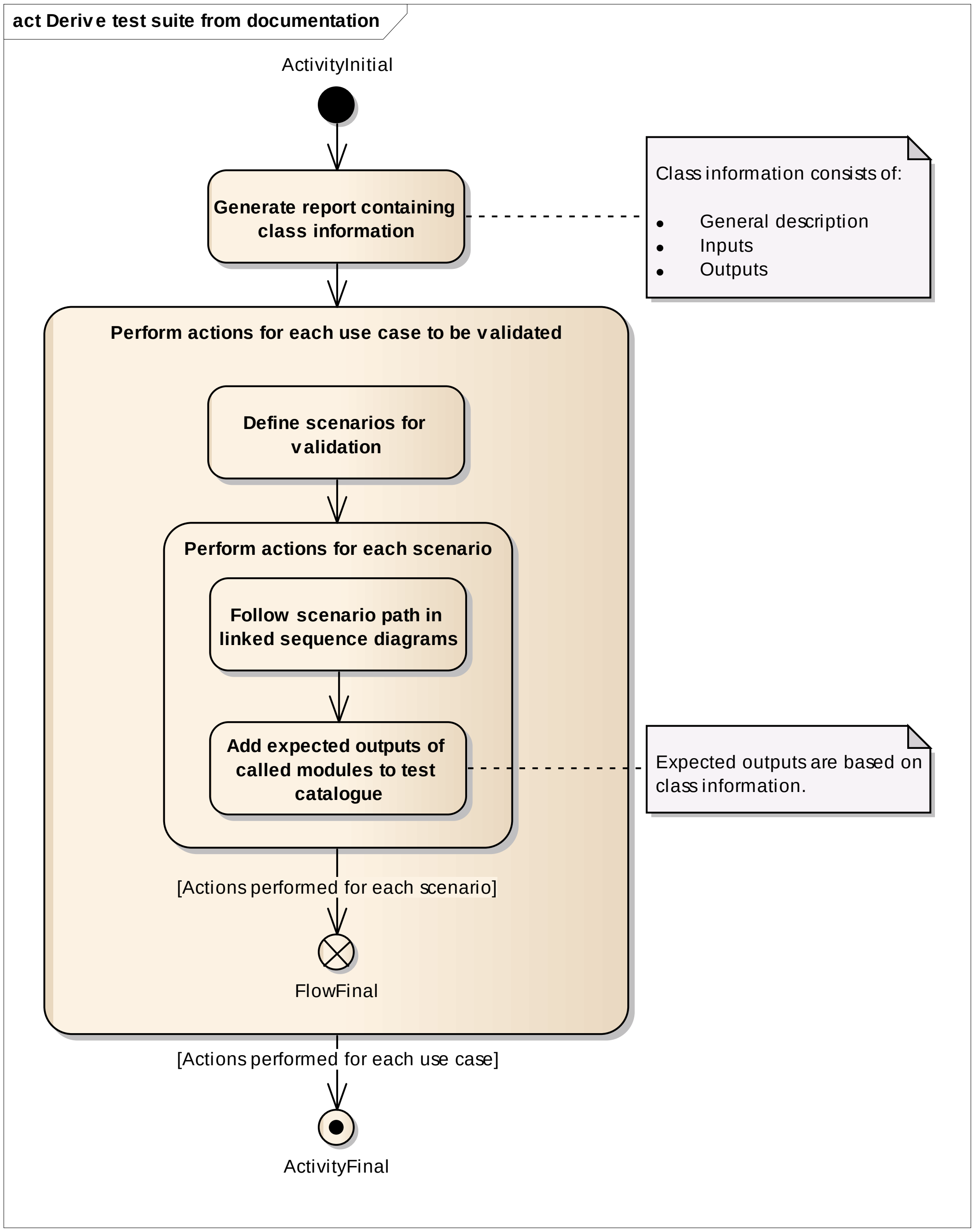

act Derive test suite from documentation
ActivityInitial
Generate report containing class information
Class information consists of:
• General description
• Inputs
• Outputs
Perform actions for each use case to be validated
Define scenarios for validation
Perform actions for each scenario
Follow scenario path in linked sequence diagrams
Add expected outputs of called modules to test catalogue
Expected outputs are based on class information.
[Actions performed for each scenario]
FlowFinal
[Actions performed for each use case]
ActivityFinal

Figure 4.22: Actions to be performed when deriving the test specification from the existing ATOOL documentation

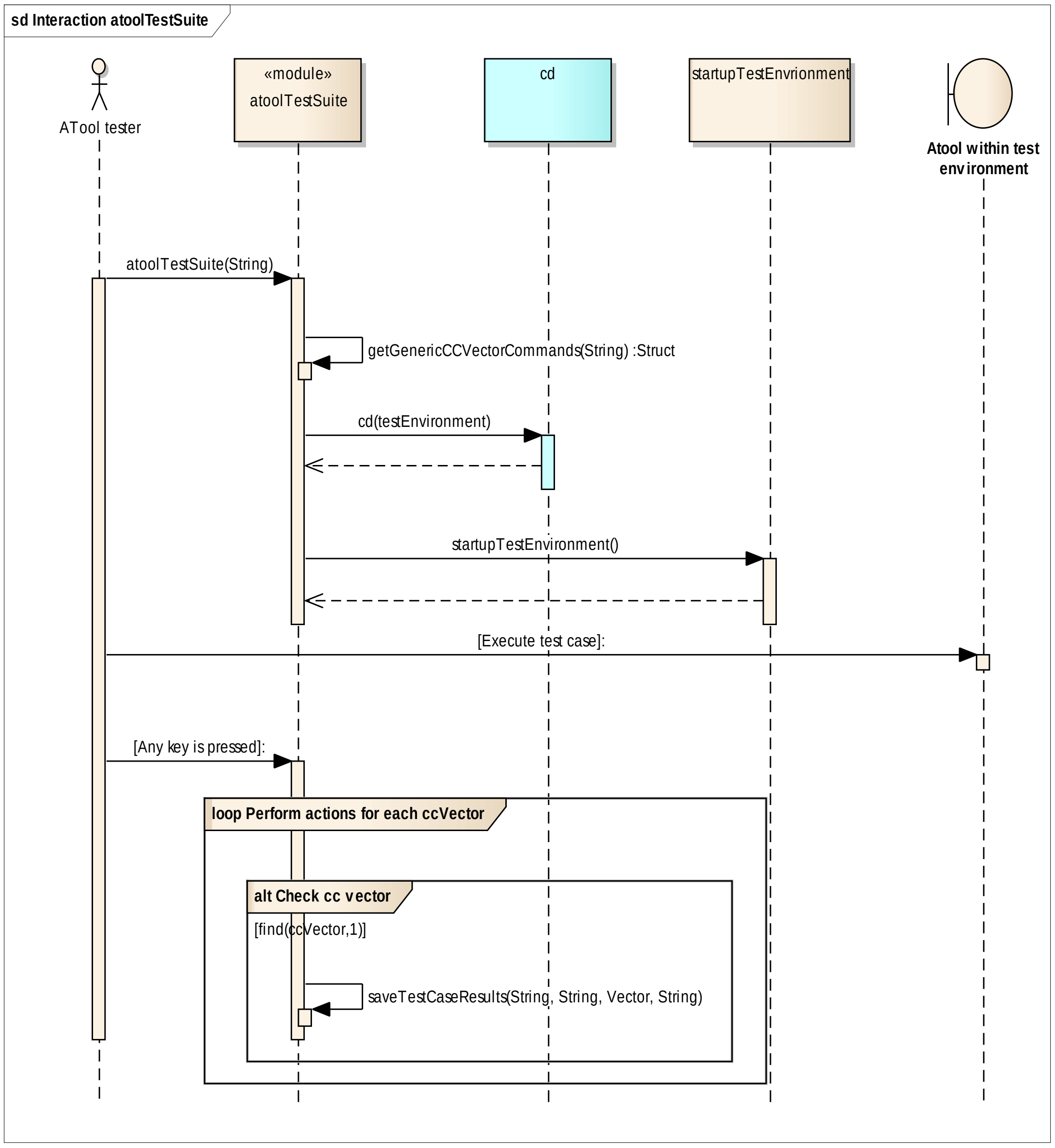


Figure 4.23: Sequence of function calls at ”atoolTestSuite.m”

Each mat-file representing the test results of an ATool module contains a test case specific struct including the following fields:

mksTcId: MKS ID of executed Test Case

ccVectorName: Name of code coverage vector

ccPercent: Code coverage in percent

ccVector: Values of code coverage vector (value type: boolean)

modifiedOn: Output of MATLAB command *datestr(now)*

In case of a repeated test case execution, the existing test results are overridden by the results obtained from the last executed test run.

4.3.3 Test result analysis

The MATLAB functions "viewTestResults" and "getUntouchedFlags" support the result analysis of an executed test case.

The code coverage vectors of all executed test cases of an ATool module are or linked and plotted by "viewTestResults". An exemplary result plot is shown in Figure 4.24. Code coverage flags that are set appear as a blue bars, code coverage flags that are not set appear as red bars. The target of the ATool validation is to reach a code coverage of 100%. An indicator for an incomplete scenario definition is given, when an 100% code coverage is not reached by all executed test cases. It is noted that the code coverage provides no testimony concerning the test result. The test result depends exclusively on the deviation of the expected outputs from the real outputs of a module.

Code coverage flags that remain set to zero are called untouched flags. All code coverage flags of a single ATool module that are not touched by any of the executed test cases are gathered by "getUntouchedFlags". This MATLAB function generates a MS Excel sheet providing a framework to analyze the reason of an untouched flag. The MS Excel sheet name is composed of the prefix "untouchedFlags_" and the name of the considered module. When calling "getUntouchedFlags" multiple times, the existing MS Excel sheet is replaced by the new generated one. To avoid losses of the analysis of untouched flags, flag analysis data is copied to a separate xlsx-file. Figure 4.25 shows an excerpt of the untouched flags list corresponding to the test results plot shown in Figure 4.24.

4.3.4 Configuration and version management

Figure 4.26 shows the project structure within MKS SI. The validation is performed for each major version of the tool. Changes of ATool caused by the insertion of the code coverage flags are checked in as a new branch of the project ("Tool preparation branch"). This ensures, that no code coverage flags are contained by the code of the ATool development main branch. As a result, the readability of the code located at the development main branch is provided. The ATool version containing code coverage flags is shared into the test environment located at the validation project. After each performed ATool validation a checkpoint on the validation project is set.

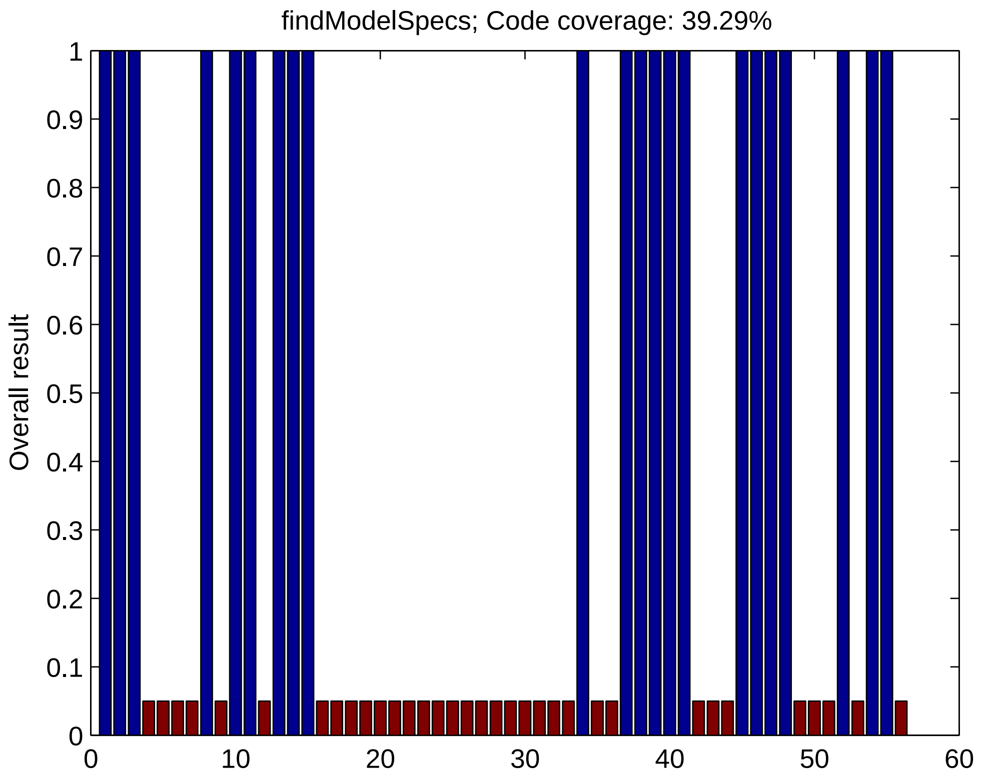
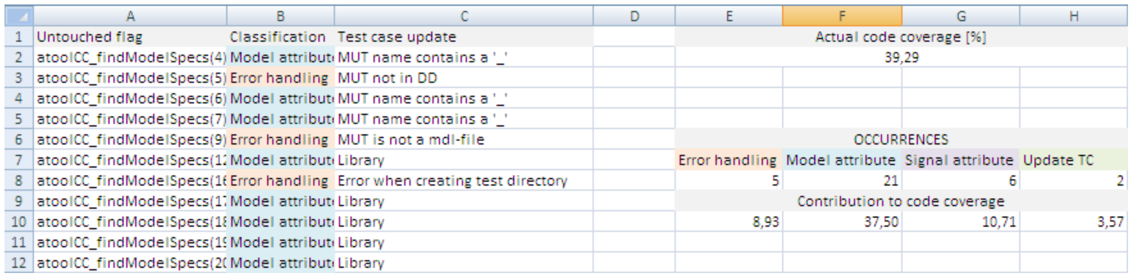


Figure 4.24: Exemplary overall test results



	A	B	C	D	E	F	G	H
1	Untouched flag	Classification	Test case update		Actual code coverage [%]			
2	atoolCC_findModelSpecs(4)	Model attribut	MUT name contains a '_'		39,29			
3	atoolCC_findModelSpecs(5)	Error handling	MUT not in DD					
4	atoolCC_findModelSpecs(6)	Model attribut	MUT name contains a '_'					
5	atoolCC_findModelSpecs(7)	Model attribut	MUT name contains a '_'					
6	atoolCC_findModelSpecs(9)	Error handling	MUT is not a mdl-file		OCCURRENCES			
7	atoolCC_findModelSpecs(1	Model attribut	Library		Error handling	Model attribute	Signal attribute	Update TC
8	atoolCC_findModelSpecs(1	Error handling	Error when creating test directory		5	21	6	2
9	atoolCC_findModelSpecs(1	Model attribut	Library		Contribution to code coverage			
10	atoolCC_findModelSpecs(1	Model attribut	Library		8,93	37,50	10,71	3,57
11	atoolCC_findModelSpecs(1	Model attribut	Library					
12	atoolCC_findModelSpecs(2	Model attribut	Library					

Figure 4.25: Exemplary excerpt of a MS Excel sheet containing analyzed untouched flags

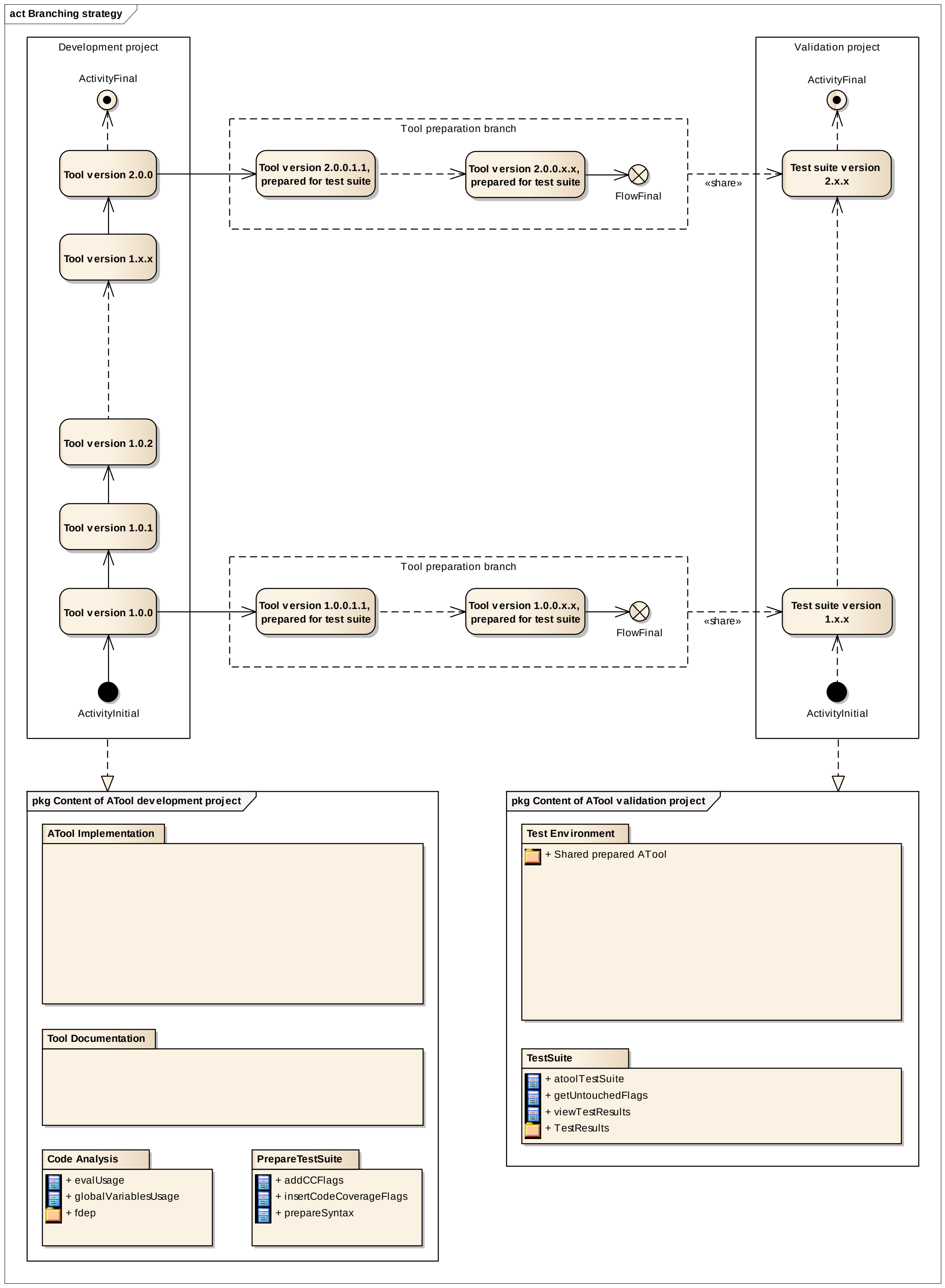


Figure 4.26: Branching strategy for ATOOL validation

Chapter 5

Conclusion and future work

The objective of this master thesis is to acquire knowledge concerning the qualification of software tools in accordance with *ISO 26262-8:2011(E): Road vehicles – Functional safety – Part 8: Supporting processes* ([ISO11b]). The approach is to perform a rigorous qualification of ATool to check which qualification actions are necessary and which are too stringent. Due to this approach, not all specified activities for a software tool qualification in accordance with [ISO11b] have been performed. The following sections deal with the findings from this thesis and the open issues concerning a successful qualification of ATool.

5.1 Conclusion

The applied combination of the qualification methods "Development in accordance with a safety standard" and "Validation of the software tool" is inevitable for an ASIL D qualification. When applying a safety standard on the development of the software tool, an application of the qualification method "Evaluation of the tool development process" is obvious. It is in the interest of the tool developer to ensure that the safety standard has been applied in a proper way for the software tool development.

As mentioned by [ISO11b], no safety standard is fully applicable to a software tool development. Due to the broad range of software tools no software tool safety standard is available at the moment. That means, a profound analysis of the applied safety standard cannot be avoided. However, each requirement of the safety standard has to be analyzed concerning its applicability on the software tool development.

It is proposed to apply [ISO11a] on the development of a software tool used in the automotive industry. This provides manifold advantages. First, existing tools and processes used for automotive software development can be used/adapted for software tool development. Second, developers and testers are familiar with those tools and processes. Third, the qualification method "Evaluation of the tool development process" can be performed as an internal assessment. Last but not least, it is possible to cover several verification requirements of [ISO11a] by applying the qualification method "Validation of the software tool".

The verification of a software tool is represented by the testing phases required by [ISO11a]. A software tool validation stimulates the tool only at the GUI. That means, the validation is performed as required in the [ISO11a] verification step "Software integration and testing". Furthermore, the behavior of the software tool is examined at the validation phase. As a result, the software tool validation covers the [ISO11a] verification step "Verification of software safety requirements", too.

Nevertheless, a successful application of [ISO11a] for a software tool development requires a software tool specific access into the V-model of the standard. Such an access is provided by means of a specification of software tool use cases. The testing of those use cases represent the validation of the software tool. It must be ensured, that the requirements of [ISO11b] concerning the "Validation of the software tool" are satisfied for the validation of the software tool use cases. In-house developed software tools as well as off-the-shelf software tools can be validated. However, only in-house developed software tools can be verified.

ATool shows a low complexity compared to an automotive embedded software. As a result of the low complexity, the validation of ATool in addition to the target of 100% function coverage and branch coverage represents a software integration and testing as well as a verification of software tool safety requirements described in [ISO11a]. As a result of the functionality of ATool, no calculations are performed within the tool. Thus, explicit module testing is not required for ATool.

To ensure that the hardware and software configuration of the used workstation has no impact on the correct functioning of the software tool, the test suite representing the software tool validation must be performed for each installation or update of the software tool. However, it is questionable if the tool execution environment including the hardware and the software (e.g. operating system) is relevant for the software tool qualification. The additional risk implied by the different execution environment seems to be significantly lower than the risk due to the development failures of the tool.

The structuring of the software tool qualification by means of the proposed qualification view model represents a clear way to gather all the required tool information (Figure 5.1 shows the qualification view model). This procedure represents the most time consuming activity of the qualification and therefore a structured way of working is useful. In case of a reverse engineered specification of the qualification view model, the stakeholders of each view represent the interview partners to allocate view specific information.

In chapter 2.3 a methodology to establish confidence in the usage of software tools ([HRM$^+$11]) is discussed. Due to the scope of this master thesis an application of the methodology proposed by [HRM$^+$11] on this thesis makes no sense. However, an application of the methodology proposed by [HRM$^+$11] and the usage of the qualification view model is advantageous. The benefit of applying [HRM$^+$11] for a software tool evaluation is, that in case of a subsequent qualification the processes view, the use case view and parts of the premises view are already specified.

The documentation of ATool by means of an UML-model contributes to an increased comprehensibility of the tool behavior. Furthermore the generation of the test specification is facilitated by the existing classes and sequence diagrams. For the testing of the error handling, the activity diagrams are helpful. However, by implementing code coverage flags and providing a framework to analyze those code coverage flags after a test run, the creation of activity diagrams provides no significant advantages for tools like ATool. Tools that are designed to perform calculations or similar activities require explicitly performed module tests and as a result activity diagrams for such tools are necessary.

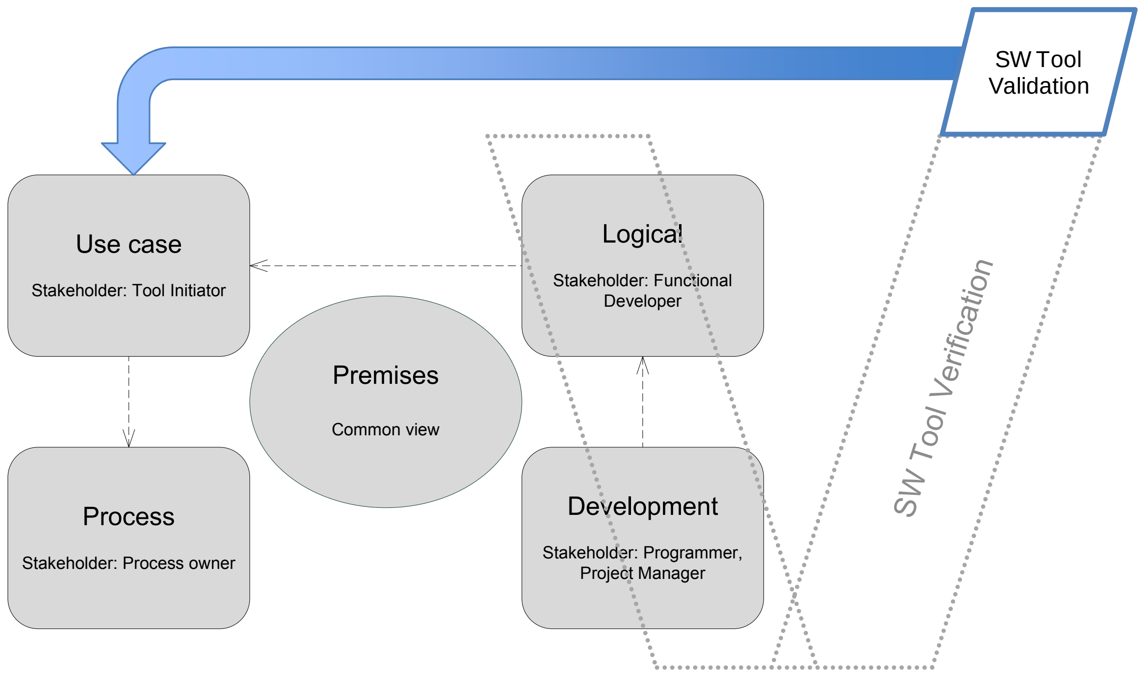

SW Tool Validation
Use case
Stakeholder: Tool Initiator
Logical
Stakeholder: Functional Developer
Premises
Common view
Process
Stakeholder: Process owner
Development
Stakeholder: Programmer, Project Manager
SW Tool Verification

Figure 5.1: Structuring of the software tool qualification by means of the proposed qualification view model

The application of a SW FMEA on a software tool development is possible in principle. In case of ATool no ASIL decomposition and no software partitioning is performed and no further safety measures are intended to be derived. Therefore the SW FMEA was not finished. However, the methodology used for the SW FMEA shall be applied at the concept phase of the software tool if possible. Hence, the failure risks of the software tool can be determined at this early development phase and as a result the monitor concept proposed by [IIW12] can be applied.

As already explained, in-house developed software tools can be validated and verified. However, in case of off-the-shelf tools the user has only one option: validation. As a result, the qualification of an off-the-shelf tool cannot be performed adequately by the tool user because only the tool vendor knows the source code of the software tool. Hence, only the tool vendor is able to determine the test coverage when qualifying the tool. Some use the argument, that only the tool customer is able to perform a software tool qualification, because only the tool customer knows about the applied use cases of the tool. This argument might be true for complex and generic software tools like MATLAB or MS Excel. In fact, most of the software tools used for the development of automotive embedded software are very specific (e.g. vector CANalyzer). Due to the standardized development approach within the automotive industry, the use cases of off-the-shelf tools applied at different tool customers are nearly identical. This circumstance is also known by the software tool vendors. It is proposed that the software tool vendor determines a typical set of use cases for which the standard qualification applies. The tool user has to ensure that its development process complies with this typical set of use cases.

5.2 Future work

ATool has already been developed when the decision for a qualification was made. As a result, the specification of the qualification view model is reverse engineered. ATool is composed of about ten-thousand lines of code, multiple interfaces to external software tools exist and global variables are used. Hence, the specification of the qualification view is extremely time-consuming. The following describes the open issues that must be implemented for a successful ATool qualification.

Initialization of all variables: The initialization of variables must be guaranteed for all modules of ATool. Therefore, the modules must be reviewed. All variables that are used in a MATLAB function must be initialized at the beginning of the corresponding MATLAB function.

Inspection: The design description and the code of ATool must be reviewed by means of an inspection as required by [ISO11a].

Software tool development: The target ASIL for the qualification of ATool is ASIL D. Therefore the methods of [ISO11a] required for an ASIL D development are considered for ATool development. This procedure must be called into question. Although [ISO11b] requires a software tool development in accordance with a safety standard for ASIL D qualifications, [ISO11b] makes no statement on the classification of the development. That means, theoretically an application of [ISO11a] in compliance with an ASIL A classification would be possible. However, the appropriate classification of the software tool development shall be considered with respect to the TCL of ATool.

Documentation within PTC Integrity: A "SOW Requirement Document" documenting the ATool use cases has to be created in MKS RM. The existing MKS Document representing the ATool function specification has to be reviewed to ensure that the content of the function specification deals with the ATool version to be qualified. Furthermore, the function specification has to be adapted to comply with [ISO11b], clause 6. Each requirement defined in the function specification has to be linked with its corresponding use case within MKS RM. Function specification requirements that are linked to a use case have to be classified as functional. Requirements that cannot be allocated to a single use case (e.g. requirements concerning the error handling) have to be classified as non-functional.

Furthermore, the existing test specification in MKS RM has to be adapted in two ways. First, the test specification has to be reviewed to ensure that the content addresses the ATool version to be qualified. Second, the reviewed test cases have to be linked with the appropriate requirements in MKS RM.

Linking between enterprise architect and MKS RM: A linkage between the elements in Enterprise Architect and the requirements in MKS RM has to be established. A possibility to establish such a linkage is to add the MKS ID as a tagged value of the corresponding element in Enterprise Architect. Remark: Tagged values can be made visible in a diagram by selecting them at "Feature Visibility".

Evaluation of the tool development process: An evaluation of the ATool development process has to be performed. The evaluation shall be complied with [ISO11b]. Possible adaptions of the development process resulting from the evaluation must be considered.

Generation of the qualification report: Based on the content of the qualification view model the qualification report has to be generated.

In order to optimize the process of a re-qualification of ATool the following points shall be considered:

Automation of the test suite: At the moment all test cases contained by the ATool test suite have to be executed manually. Therefore, the MATLAB function "atoolTestSuite" shall be extended with code representing the execution of the specified test cases. User inputs can be reproduced by calling appropriate ATool GUI callback functions. The error handling shall be tested by using intentionally corrupted inputs. The environment that provides a corrupted ATool input shall be stored within the validation project. The extensions of "atoolTestSuite" shall be documented in Enterprise Architect.

The need for a verification of the automated test suite shall be considered.

Simplify unit design: To increase the maintainability and readability of the ATool code, the unit design shall be adapted (e.g. remove global variables of "loadParam.m").

Appendix A

Abbreviations

ASIL	Automotive safety integrity level
E/E system	Electrical and/or electronic system
E/E/PE system	Electrical/electronic/programmable electronic system
FMEA	Failure mode and effects analysis
GUI	Graphical user interface
HIL	Hardware-in-the-loop
MIL	Model-in-the-loop
MKS RM	MKS Requirements Management
MKS SI	MKS Source Integrity
MPT	Magna Powertrain
MS	Microsoft
MUT	Module under test
OBS	Observer function
SIL	Software-in-the-loop
SOW	Statement of work
SW	Software
TC	Test case
TCL	Tool confidence level
TD	Tool error detection
TI	Tool impact
UML	Unified modeling language

Bibliography

[BM08] Graham Bath and Judy McKay. *The Software Test Engineer's Handbook.* Rocky Nook, June 2008.

[BW11] Andreas Bärwald and Doris Wild. *Softwarewerkzeuge für sicherheitsgerichtete Entwicklungen - Anforderungen und Lösungsansätze.* TÜV Süd Automotive GmbH, Garching, Germany, November 2011.

[CMR10] Mirko Conrad, Patrick Munier, and Frank Rauch. Qualifying Software Tools According to ISO 26262. In *MBEES*, pages 117–128, 2010.

[EN 11] EN 50128:2011: Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems. Standard, European Committee for Electrotechnical Standardization, Brussels, B, 2011.

[Glo08] Tilman Gloetzner. IEC 61508 Certification of a Code Generator. In *3rd International Conference on System Safety 2008*, pages 134–137, 2008.

[HRM$^+$11] Joachim Hillebrand, Peter Reichenpfader, Irenka Mandic, Hannes Siegl, and Christian Peer. Establishing Confidence in the Usage of Software Tools in Context of ISO 26262. In *SAFECOMP 2011*, pages 257–269, 2011.

[IEC06] IEC 60880:2006: Nuclear power plants – Instrumentation and control systems important to safety – Software aspects for computer-based systems performing category A functions. Standard, International Electrotechnical Commission, Geneva, CH, April 2006.

[IEC10a] IEC 61508-3:2010: Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements. Standard, International Electrotechnical Commission, Geneva, CH, April 2010.

[IEC10b] IEC 61508-4:2010: Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 4: Definitions and abbreviations. Standard, International Electrotechnical Commission, Geneva, CH, April 2010.

[IEE10] IEEE Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations. Standard, IEEE Power & Energy Society, New York, USA, August 2010.

[IIW12] Viacheslav Izosimov, Urban Ingelsson, and Andreas Wallin. Requirement Decomposition and Testability in Development of Safety-Critical Automotive Components. In *Computer Safety, Reliability, and Security, 31st International Conference, SAFECOMP2012, Magdeburg, Germany*, pages 74–86, 2012.

[ISO08] ISO 12207:2008(E): Systems and software engineering – Software lifecycle processes. Standard, International Organization for Standardization, Geneva, CH, February 2008.

[ISO11a] ISO 26262-6:2011(E): Road vehicles – Functional safety – Part 6: Product development at the software level. Standard, International Organization for Standardization, Geneva, CH, November 2011.

[ISO11b] ISO 26262-8:2011(E): Road vehicles – Functional safety – Part 8: Supporting processes. Standard, International Organization for Standardization, Geneva, CH, November 2011.

[Izo12] Viacheslav Izosimov. *Qualification of Test Tools for Safety Critical Systems with Fault Injection and a Monitor*. EIS By Semcon AB, Stockholm, Sweden, March 2012. URL: http://www.kth.se/polopoly_fs/1.300633!/Menu/general/column-content/attachment/EIS_SlavaIzosimov.pdf, visited on July 16th 2013.

[Kru95] Philippe Kruchten. Architectural Blueprints – The "4+1" View Model of Software Architecture. In *IEEE Software 12 (6)*, pages 42–50, 1995.

[Mat07] Mathworks. www.mathworks.com/matlabcentral/fileexchange/17291-fdep-a-pedestrian-function-dependencies-finder, November 2007. Visited on September 11th 2013.

[RTC11] RTCA DO-178C: Software Considerations in Airborne Systems and Equipment Certification. Standard, Radio Technical Commission for Aeronautics, Washington DC, USA, December 2011.

[Sch11] Adam Schnellbach. *Qualification of SW tools - Guideline.* Magna Powertrain, Lannach, Austria, December 2011.

[SIG10] Automotive SIG. Automotive SPICE® Process Assessment Model. Guideline, The SPICE User Group, May 2010.

[Som05] Stephane S. Some. Use Cases based Requirements Validation with Scenarios. In *Proceedings 13th IEEE International Conference on Requirements Engineering (RE 2005)*, pages 465–466, 2005.

[Spo11] Gunther Spork. Efficient Requirements Management Considering Automotive Standards: Best Practice Sharing of Mechatronic Engineering within MAGNA Groups. Master's thesis, Institute of Production Science and Management, Graz University of Technology, 2011.

[Sys] Sparx Systems. http://www.sparxsystems.eu/resources/project-development-with-uml-and-ea/. Visited on March 17th 2014.

[WWI$^+$12] Q. Wang, A. Wallin, V. Izosimov, U. Ingelsson, and Z. Peng. Test tool qualification through fault injection. In *17th IEEE European Test Symposium (ETS)*, 2012.