Masterarbeit

Handling Variability in Unit Testing of Automotive Control Software

Wolfgang Raschke

Institut für Technische Informatik
Technische Universität Graz

Begutachter: UA DI Dr. Christian Kreiner.
Betreuer: DI Andrea Leitner.

Graz, im Februar 2012

Kurzfassung

Derzeit werden Hybridfahrzeuge in Hinblick auf Massenfertigung entwickelt. Tatsächlich gibt es sehr viele mögliche Konfigurationen eines Hybridfahrzeuges, etwa Vollhybrid oder Mildhybrid. Da es so viele mögliche Hardwarekonfigurationen gibt, ist es sehr schwer, mit konventionellen Praktiken der Softwareentwicklung eine generische und wiederverwendbare Steuerungssoftware zu entwickeln.
Automotive Steuerungssoftware muss getestet werden. Die getestete Software besteht aus mehreren Units. Auf Grund der hohen Komplexität automotiver Steuerungssoftware gibt es sehr viele mögliche Kombinationen, einzelne Units in das Gesamtsystem zu integrieren. Die Auswahl der Tests und ihre Automatisierung muss diese Komplexität ebenfalls berücksichtigen. Zusätzlich hängt die Steuerungssoftware auch von einer Parametrisierung ab. Daher ist es schwer, wiederverwendbare Tests zu implementieren.
Der Zugang zu diesem Thema in dieser Masterarbeit orientiert sich an Testartefakten. Diese beinhalten: Testpläne, Testfälle, Testdatensätze und Testsoftware sowie Testskripte. Es wurde ein Testframework implementiert, das es ermöglicht, die Tests in der Sprache zu formulieren, in der auch die getestete Software implementiert ist: Simulink. Zusätzlich können diese Tests in Simulink mit Variabilitätsinformation angereichert werden, sodass eine nahtlose Integration in die Software Product Line möglich ist.
In dieser Arbeit definieren Testpläne Variabilität auf Komponentenebene. Nur Units, die in das Gesamtsystem integriert sind, werden von den Testplänen abgedeckt. Testpläne repräsentieren einen Blickwinkel der Software Product Line - den *problem space*. Der *solution space* hingegen ist der technische Blickwinkel und wird durch Testfälle, Testsoftware und Testskripte abgedeckt.
Weil die getestete Software parametrisiert ist, wird Testfallwiederverwendung erst möglich, wenn Testfälle auch Variabilität beinhalten. Testdatensätze werden von einer globalen Repräsentation der Parameter sowie von Requirement-Modellen abgeleitet. Testdatensätze gehören zum *problem space* der vorgeschlagenen Vorgehensweise. Der *solution space* wird durch variable Teile innerhalb der Testfälle repräsentiert.
Es hat sich herausgestellt, dass das Testen von mehreren Units vollständig automatisiert werden kann. Testfallwiederverwendung kann durch variable Teile innerhalb der Testfälle stark erhöht werden. Die Haltung von Testplänen und Testdaten in einem Konfigurationsmanagementsystem hat die Wartbarkeit des entwickelten Testframeworks erhöht.

Schlüsselwörter: Software Product Lines, Testartefakte, Test-Driven Development

Abstract

At the moment, hybrid vehicles are developed with respect to mass-customization. In fact, there are lots of possible configurations of a hybrid vehicle, for instance full hybrid or mild hybrid. Because there are so many possible hardware configurations it becomes hard to develop a generic and reusable control software with conventional software engineering practices.
Automotive control software has to be tested. The software under test consists of units. Due to the complexity of automotive control software, the possible combinations of integrated units are very high. The selection of unit tests and the automation of their execution has to regard this complexity, as well. Moreover, automotive control software depends on variable parametrization and, thus, makes it difficult to build up a base of reusable tests.
The approach followed in this thesis is centered around test artifacts. They include: test plans, test cases, test reports, test data sets and test software and scripts. A test framework has been developed that enables writing tests in the same language that is used to develop the software under test: Simulink. Additionally, the tests written in Simulink can be enhanced with variability information which allows seamless integration into a Software Product Line.
Test plans in the scope of this thesis define variability on component level. Only units that are integrated in a product will be covered by the test plans. Test plans define the *problem space* of the developed Software Product Line. The *solution space* covers all technical realization aspects and is implemented by means of test cases and test software and scripts.
Because the software under test is parametrized, test case reuse can only be achieved, if test cases contain variability, as well. Test data sets are derived from a global representation of parameters and from requirements models. Test data sets define the *problem space* of the proposed methodology. The *solution space* is represented by the variable parts within test cases.
It turned out, that testing of several units can be automated fully. Test case reuse can be increased to a high degree by introducing variable parts within test cases. Moreover, keeping test data and test plans in a configuration management system has improved the maintainability of the developed test framework.

Keywords: Software Product Lines, Test Artifacts, Test-Driven Development

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

This master thesis has been carried out at the Institute for Technical Informatics, Graz University of Technology in cooperation with the virtual vehicle Forschungsgesellschaft mbH. I wand to give thanks to the team of the virtual vehicle who kindly supported my work with lots of help. DI (FH) Wolfgang Ebner helped a lot by answering questions.
I would like to thank Dr. Christian Kreiner and DI Andrea Leitner from the Institute for Technical Informatics for their enthusiastic help and support.
Especially, I want to thank my family for great support and encouragement.

Graz, im Februar 2012 Wolfgang Raschke

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
CCFM	Consul Component Family Model
CTM	Classification Tree Method
CTM/ES	Classification Tree Method/Embedded Systems
DSL	Domain Specific Language
EC	Equivalence Class
FODA	Feature Oriented Domain Analysis
FURPS	Functionality Usability Reliability Performance Supportability
GQM	Goal Question Metric
GUI	Graphical User Interface
HF	High Frequency
HTML	Hypertext Markup Language
PC	Personal Computer
RPM	Revolutions Per Minute
SPL	Software Product Line
SW	Software
TC	Test Case
TD	Test Data
TDD	Test-Driven Development
TP	Test Plan
TPT	Time Partition Testing
TS	Test Software and Scripts
UML	Unified Modeling Language
V	Variant
VP	Variation Point

1. Introduction

1.1. Motivation

At the moment hybrid vehicles are on the brink of being developed and manufactured with respect to mass-customization. This encompasses a handful of compelling problems: hybrid vehicles are in the prototype stage and control software has to cover many possible configurations for fine-tuning. Myriads of possible vehicle configurations lead to more diverse control software configurations. Due to its multitude of hardware assembly variants, hybrid vehicles confront developers with even more variant configuration possibilities as composed to conventional vehicles.

1.2. Background

In the *HybConS* project currently a Software Product Line is under development. Software Product Lines differentiate between commonalities and variabilities. This is especially important in the context of control units for hybrid vehicles because they share some common features and differentiate enormously due to their huge amount of possible hardware configurations. In order to cope with these problems, the *HybConS* project is defining a generic and highly reusable software architecture.

Automotive software is highly safety-relevant. For this reason testing efforts add a huge amount to development costs. Testing in Software Product Lines is a present challenge. Test reuse is an important cost-cutting issue. Test management as well as test data storage and the possibility of their integration in Software Product Lines will have a major impact onto competitiveness.

1.3. Purpose and Scope

A control unit for hybrid electrical vehicles is under development with respect to software reuse and adaptibility. Therefore, a Software Product Line is implemented by to cover all possible functional variants.

For instance, a functional variant may be the implementation of recuperation during service braking. This function would be desired by some vehicles but not by all. So, it is, simply spoken a goal of the SPL to present the developer a simple switch where he could determine the presence of the functional variant.

Dependent on the software configuration it should be possible to produce a selection of corresponding test cases. Moreover, it will be able to create some generic test cases and

fill them automatically with test data.

Currently, unit testing is performed by means of a unit testing software, called *Poet*. Despite the fact that this is a proprietary company-internal product, it is insufficient for the reasons:

- It is not possible to select test cases on the basis of a formal representation. Selections have to be done by the tester on unit test level.
- Neither a connection between different units nor between test cases and requirements are provided. Test cases can not be organized by any means of hierarchy. Testing issues are just local (unit level) concerns.
- No global test data management is provided.
- Specified signals as well as signal evaluation criteria can only be defined as low level data. No abstraction of signals and evaluation criteria is supported. No chance of test reuse is provided.

This leads to the negation of reuse. Signals have to be drawn in a signal editor for each test case. A lot of redundant hand-crafted work has to be done. The manual test data generation is very error-prone and changes in test data cause a lot of work.

The scope of this thesis encompasses first, a systematic test case selection mechanism based on a formal representation, and second, the connection between test data management, test signals and test evaluation criteria. This includes a more abstract view on signals which leads to an abstract reception of test cases. Those generic test cases provide a missing link between test data and test cases.

1.4. Disposition

In **Chapter 2** a first introduction to test related terminology is given. **Section 2.2** gives an overview on Test-driven development and the xUnit test paradigm. The mlUnit and slUnit implementations of xUnit are presented. **Section 2.3** deals with Software Product Line principles in general. Software Product Line testing concerns are discussed in **Section 2.4**. Finally, **Section 2.4** treats problems and possible solutions that are apparent in testing automotive embedded software.

Chapter 3 depicts general ideas that lead to the interlacing of pure::variants and Simulink, first. Second, the implementation of test plans in pure::variants, Simulink and other test frameworks is discussed. In problem space, modeling of requirements models, as well as vehicle and software configuration models is described. Finally, the link between solution space and test frameworks is explained.

In **Chapter 4** the implementation of the concepts is described. Since the creation of feature models can be done in pure::variants easily, it is not described here. The main part of the implementation is the link between the test framework with family models and

the generation of family models out of Simulink with the help of user interaction.

In **Chapter 5**, a case study has been carried out. First, the system under test and the test conditions are described. The system under test is tested with two different software configurations in order to prove the concept of Software Product Line testing techniques. The results will be discussed.

Chapter 6 concludes this theses. Lessons learned are listed and future work is proposed.

2. Related Work

2.1. Aspects of Software Quality

Software quality aspects go beyond the correct fulfillment of the functional specifications. Most of the time the functional specification is tested. Besides, some aspects are tightened together so that a level of quality concerning aspect A may influence aspect B's quality [Vig10].

Software quality aspects can be described by using the FURPS-model. It is described in [Vig10] and [CdPL09] and consists of the following parts:

Functionality covers the whole spectrum of demanded specifications.

- *Adequacy* means that the realization should stick to the requirements.
- *Correctness* means that the realizations sticks to the specification.

Usability covers all aspects of human computer interaction, f.i. comprehensibility, learnability, ease of use and documentation.

Reliability is the ability to work in a predictable way.

- *Maturity* is a measure of how often software fails.
- *Fault Tolerance* is the ability to deal with faults that have been caused by problems.
- *Recoverability* is the ability to establish desired performance after a faulty state in a certain time.

Performance is the ability to satisfy defined boundary conditions, such as:

- *Timing Resources* needed by an application.
- *Memory Resources* needed by an application.

Supportability is the ability to run on different platforms.

- *Adaptibility* How easy can software be adapted to different needs?
- *Installability* How easy can software be installed?
- *Portability* How easy can software be ported to different platforms?

2.2. Test-Driven Development

TDD is the acronym for *Test-driven development* which is a more sophisticated approach than simple and intuitive testing during development. TDD favors documentation and check per requirement.

2.2.1. The Utmost Maxime: Test First

This approach should be viewed under a pragmatic perspective: each requirement is linked with one or more tests [GSM04]. Before implementing a new functionality, the corresponding tests have to be created.

One may compare writing software to writing texts. Human beings are very likely to commit a lot of typing errors and even if they would re-read the written words they often do not get aware of the committed errors. It seems somehow that human beings do not read in a letter-after-letter manner but more likely in some more complex contexts of meaning. They build up an inner *world model* that prevents them from finding errors. A possible solution to detect mistakes is to let another person read the text. As well another way to detect many mistakes is to read the text beginning at the end reading word after word in the reverse order. This inversion of order somehow relates to TDD. A case study has been conducted in [WMV03] that indicates that TDD helps reducing defect density.

2.2.2. TDD Process

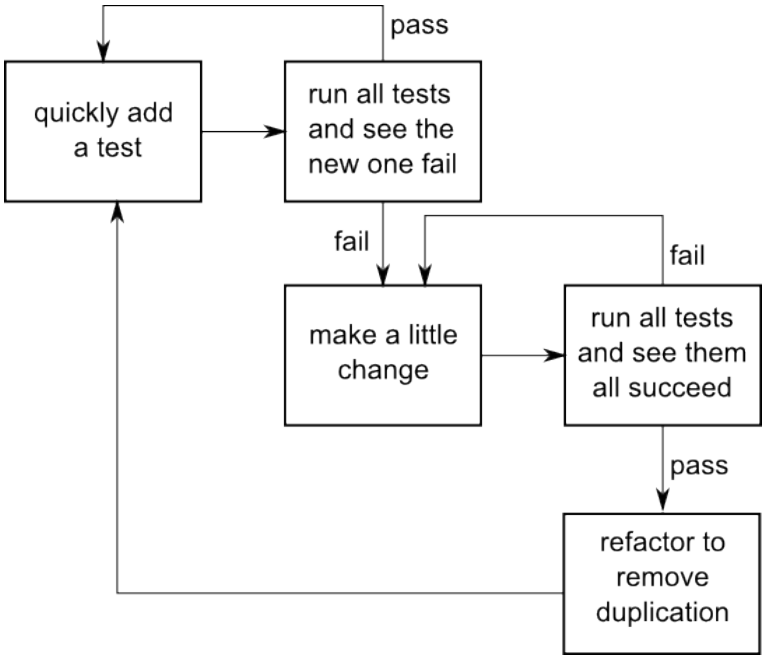

Figure 2.1.: Development Cycle in TDD

The process of TDD (see Figure 2.1) has been described in [Bec09] and consists of the

following five steps:

Quickly Add a Test

A test is added in order to cover a requirement. A requirement may need more than one test. In this step only one test is added at a time. In TDD it is not clearly defined who writes the tests: *"The unit tests might be written by the same developer or by a designated tester"* [GP07].

Run All Tests And See The New One Fail

The new test should be executed to test if it requires new code. If the new test case does not fail, it does not require new code to satisfy its test specification and thus the test is of no use. In this case the test has to be rewritten.

Make a Little Change

In this phase new code is implemented in order to let pass the latest test case. Only as much new code should be written as is needed to achieve a pass when applying the new test. This should guarantee, that all code branches are traversed.

Run All Tests And See Them Succeed

After each code change, all tests are executed automatically. If they pass, the process continues at the next step.

Refactor to Remove Duplication

Refactoring relates to improving the written code by whatever aspect necessary. Regression tests can now be easily run after each code change. After the refactoring has been accomplished, the process continues at the step, where a new test is added.

Link any Test Case with Meaning

Any test case must be linked with meaning [Eng03]. This is one of the advantages of TDD since without these actions many test cases are thrown away or simply carried out by typing a number (e.g. 9) somewhere (see Section 3.5.1). Apparently, this behavior does not contribute to test case reuse. Each requirement is linked with a test case and provides a form of documentation.

2.2.3. Additional Tools: Fakes and Mock Objects

Units are tested isolated and independently from each other. In unit testing, sometimes test data generated from other units are needed. Because of the absence of other units, test data have to be generated by means of *fakes* or *mock objects*.

Fakes

Fakes are simple objects, providing a simple workaround to prevent compiler errors [Vig10].

Mock Objects

Mock objects support test data generation up to a very complex extent. They are simpler than the real implementation counterparts but are able to imitate the behavior in an appropriate way [Vig10].

2.2.4. xUnit

xUnit is a language independent unit testing framework and was developed and explained first in [Bec11]. xUnit is free of the typical restrictions of a programming language and can serve as a reference model to the *diverging and different* implementations of mlUnit[1] and slUnit[DG07][Doh08]. mlUnit follows the object oriented paradigm, whereas slUnit follows the paradigm of graphical programming in Simulink.

xUnit consists of five patterns:

- Test Case [Bec11]
- Test Suite [Bec11]
- Fixture [Bec11]
- Check [Bec11]
- Composite Pattern[DG07][Doh08]

The Test Case Pattern

A *test case* is an autonomous unit of testing. The preconditions of a test case have to be created by means of a *setup* method. Because a test case is autonomous and preconditions are defined, the order of execution of multiple test cases is independent.

The executable code of a test case is called *test method*. The name of a test method starts with *test*.

Test Suite Pattern

A test suite consists of a set of test cases and test suites (see Figure 2.2). All parts of the test suite are executed.

Fixture Pattern

A fixture is part of a test case, it provides the common configuration or the desired precondition. A fixture can be implemented by means of *setup* and *teardown* methods in object oriented test frameworks (see Section 2.2.5) or otherwise (see Section 2.2.6).

For instance, such a common configuration needed by a test case may be a connection

[1] http://sourceforge.net/projects/mlunit/

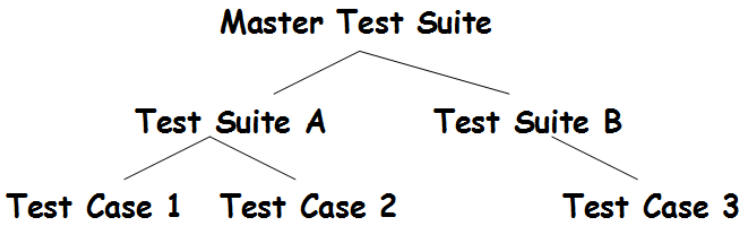

Figure 2.2.: In xUnit it is Possible to Build a Tree Structure of Tests [Vig10]

to a database. The *setup* method is then responsible for creating this connection. After the execution of the test case, the *teardown* method is responsible for closing the connection.

Check Pattern

Feedback is an essential part of testing. The check pattern returns two possible results: *failure* or *error*. A failure is reported, if the test result does not correspond to the specification. An error is reported in case of incorrect behavior, f.i. caused by divisions by zero, exceptions not caught and the like. As a result of any execution of a test case or a test suite, a test report is generated.

Composite Pattern

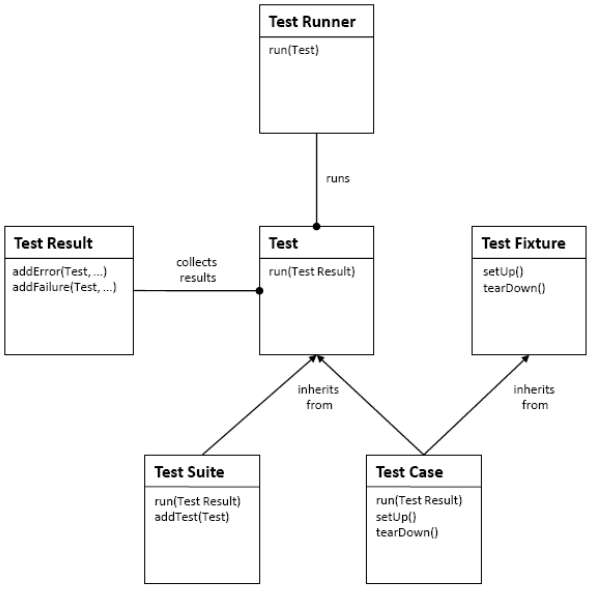

Figure 2.3.: xUnit Class Diagram [Doh08]

In Figure 2.3 *Test* is the abstract basic class. One can derive a *Test Case* or a *Test Suite* from it. As we can see, it is possible to build up a *test tree*. Inner nodes are always test suites and leaves are test cases (Figure 2.2).

2.2.5. mlUnit

mlUnit[2] is an implementation of xUnit in Matlab and is built up similar to other object oriented frameworks such as jUnit[3]. mlUnit is built upon xUnit patterns, as well:

Test Suite Pattern A *test suite* consists of several test cases. All test cases in a test suite are executed, if the test suite itself is executed.

Test Case Pattern A *test case* is executable and may contain several test methods. All test methods within a test case are executed. By convention, all test methods begin with the prefix *test*.

Fixture Pattern Each test case owns the two methods *setup* and *teardown* that constitute the pre- and postcondition and are responsible for creating and restoring the test context.

Check Pattern Assertions correspond to the xUnit check pattern. Assertions enable automatic evaluation of test routines. They compare expected results with the test results. Assertions may detect failures that are forwarded to the test report. Errors are reported as well.

2.2.6. slUnit

In slUnit[DG07][Doh08] the five patterns are mapped to Simulink (see Figure 2.4). The so-called test objective is the subsystem under test.

Test Method Pattern Test methods are the slUnit counterparts of test cases. They provide test signals and have access to the signals generated by the test objective. Besides, they do not have *setup* or *teardown* methods.

All Tests Pattern The all tests pattern corresponds to the test suite pattern. It means, that a set of test methods is executed one after another. By connecting one after another test method to the test objective, the multiplexer arranges a suite of tests.

Assertions Pattern Assertions are defined within a test method. Assertions are represented by special assertion blocks, that are part of slUnit. If an assertion is violated, the corresponding block changes its color from green to red. The test method block, that contains the assertion turns to red as well. In this manner, a failure is indicated in slUnit.

[2] http://sourceforge.net/projects/mlunit/
[3] http://www.junit.org/

Fixture and Composite Pattern The example of a test composite is gray shadowed in Figure 2.4. The *common code* block is the realization of the fixture pattern (as opposed to the setup-method in mlUnit). Additional test methods and test composites can be attached to the test composite. We can see, the composite pattern can form a recursive tree structure.

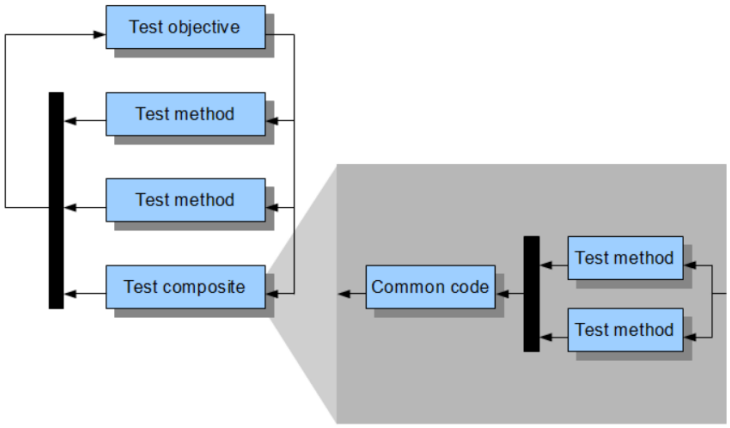

Figure 2.4.: Overview of all the Patterns in slUnit [DG07]

2.3. Software Product Lines

2.3.1. Motivation

Henry Ford is said to be the inventor of the assembly line and industrial mass production. His philosophy was to build a cheap car so that every american would be able to buy one. Due to his policy of cheap production there was only *one* model - the Model T - available in *one* configuration [C+09].

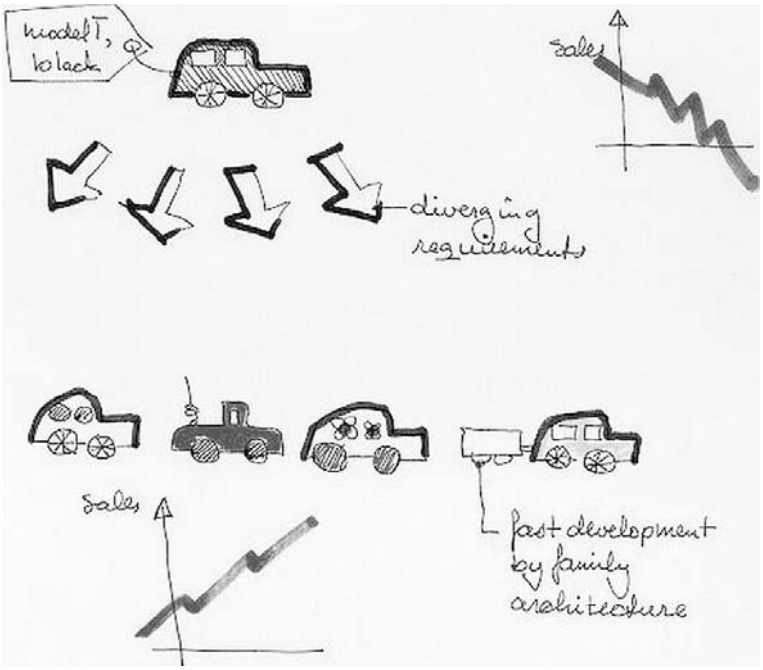

Figure 2.5.: Single System Engineering vs. Product Line Engineering in Principle [PBvdL05]

It was not until the year 1924 that Alfred Sloan from the General Motors Corporation discovered diversity as a key factor to success. His approach did address not only the practical needs of customers. Moreover he discovered that a car is more than a means of transportation. People associate their success and social status with their cars. Additionally, he detected a need for comfort and luxury. He introduced a business strategy he called *a car for every purse and purpose* [Pel06].
In the end, the Ford Motor Company was out-competed by the General Motors Corporations as Americas' most successful car manufacturer.

As can be seen, diversity of products is a key factor to success in whatever business that may outweigh conservative key factors such as time-to-market or cost. Lessons learned for the software business: the time of mass customized standard products is over.

Systematic reuse has been successful in several areas, such as automobile development, architecture and last but not least - software engineering [CN07].

2.3.2. Different Types of Reuse

Buy vs. Build

Following [Bro11] *"The most radical possible solution for constructing software is not to construct it at all"*. Buying standard software components seems to be the best reuse strategy if possible. Consider operating systems: almost every company uses standard PC's with an operating system running on it. On the other hand, only very few companies can afford to build an operating system on their own because it is tremendously expensive. A license for an operating system becomes very cheap in contrast because of massive reuse.

Higher Level Languages

Higher level languages have a high impact on productivity: compare programming in assembly language to programming in C. Basically, higher level languages make use of reuse [Bro11].

Domain Specific Languages

DSL are tailored to specific *domains* or fields of application [vDKV00]. Examples for such DSL are:

- Database Query Languages
- HTML
- Matlab/Simulink

DSL have the impact that experts in specific domains (such as control theory, physics, mechanics) can easily use them without a deep knowledge of software engineering and programming. For an example, consider Simulink, a graphical programming language that is mainly used in physics, mechanics, signal processing and so on has improved productivity of code generation enormously. On the other hand, DSL address only a narrow scope of possible applications and thus are very unflexible compared to general purpose programming languages [vDKV00]. In Simulink, simulating non-linear differential equations becomes very easy; programming a sorting algorithm is nearly impossible.

Libraries

Reuse with libraries is often done halfhearted: it happens that a small team of software engineers is assigned the task of writing libraries. Those libraries are seldom used for several reasons [Str00]:

- The team, that develops libraries is not integrated in the development process of application. Thus, the libraries would not match the need of the application developers.
- There is no emphasis on reuse. No regular training on how to use the libraries takes place. Application engineers often do not know how to use the libraries in the right way.

- Reinventing the wheel is rewarded. If productivity is measured in lines of code, why should anyone use libraries. Reusing libraries does produce few code, compared to reinventing the wheel.

Clone and Own

In [CN+11a] clone and own is described as follows: *"Suppose you are developing a new system that seems very similar to one you have built before. You borrow what you can from your previous effort, modify it as necessary, add whatever it takes, and field the product, which then assumes its own maintenance trajectory separate from that of the first product. What you have done is what is called 'clone and own.' ".*

In [CN+11a] it is argumented that the clone and own principle differs from Software Product Lines in at least two ways: First, Software Product Lines are explicitly designed for reuse. Second, in the clone and own principle, each product is viewed as a single entity, thus a single entity will have no influence on the architecture of the common code base.

2.3.3. Systematic Reuse

Systematic reuse is *"domain focused, based on a repeatable process and concerned primarily with the reuse of higher level lifecycle artifacts, such as requirements, designs and subsystems"* [FS94].

In the *domain "families of related systems"* [FS94] are created in order to derive coherent and similar applications. The reusable parts that are created in the *domain* are called *core assets* (see [CN+11b]) or *domain artifacts* (see [PBvdL05]).

Repeatable processes help formalizing reuse. In [PD93] it is stated *"that substantial quality and productivity payoffs will be achieved only if if reuse is conducted systematically and formally"*. Such *repeatable processes* are described in Section 2.3.4. If there are no dedicated processes, reuse is exploited informally (*ad-hoc*) [PD93].

2.3.4. Two Development Cycles

In Software Product Line engineering, emphasis is put explicitly on reuse. Because, besides technical issues, organizational issues are important, the development of products is split into two development cycles: *domain engineering* and *application engineering*. Both of them state explicitly several process steps, none of them may be leaved out for successful product development [CN07].

Domain Engineering

Domain engineering is design for reuse [FS94]. In domain engineering (see Figure 2.6) a common base of reusable core assets is developed [vdLSR07].

Product Management In product management, a set of similar products is defined. Not only actual products are listed but possible future products as well. Product management includes scoping: in scoping the bandwidth of products is defined. If the

scope of products is narrow, benefits of reuse may be too small; if it is too large, commonalities would shrink to a minimum, reverting the potential of a common architecture. Scoping should be performed on evidence, such as return on investment, marketing aspects and reuse economics [Sch02].

Domain Requirements Engineering In domain requirements engineering, all requirements concerning the core assets are gathered, explicitly mentioning all commonalities and variabilities among themselves.

Domain Design In domain design, an architecture is constructed, including all variable parts.

Domain Realization In domain realization, a detailed software realization is implemented with the help of variability mechanisms (see Section 2.3.5).

Domain Testing Domain testing covers all testing artifacts (see Section 2.4.1).

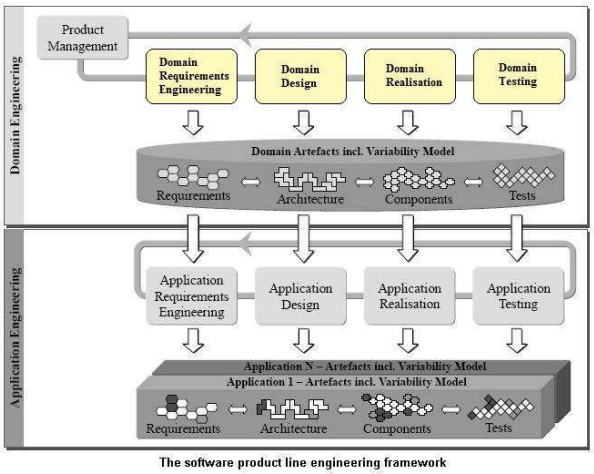

Figure 2.6.: Domain Engineering Activities on the Upper Layer [PBvdL05]

Application Engineering

In application engineering (see Figure 2.7), domain artifacts have to be instantiated, first. Based on that, domain artifacts are extended to a complete product [vdLSR07].

Application Requirements Engineering In application requirements engineering, requirements are gathered that haven't yet been covered by domain requirements.

Application Design All variabilities in the domain design are bound: it is instantiated. Starting from this initial system, additional application aspects are covered, by extending it by the application design.

Application Realization In application realization phase, the product instance is built from reusable domain realization artifacts. Additionally, product-specific functionality has to be implemented.

Application Testing In application testing, the finished product is tested; test artifacts previously defined in the domain cycle have to be completed.

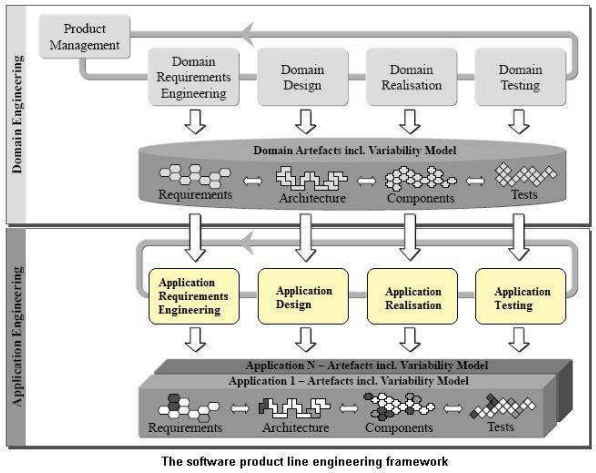

Figure 2.7.: Application Engineering Activities on the Lower Layer [PBvdL05]

2.3.5. Commonality vs. Variability

Variability Modelling

When defining a portfolio of products, at least two presumptions will hold true: first, products will have common parts in requirements, architecture, implementation and documentation. Second, products will have different parts in requirements, architecture, implementation and documentation.

As a first step, it is reasonable to separate the common parts from the variable parts and to find a notion of variability.

In [PBvdL05], three questions are proposed to define variability:

What does vary? *"Answering this question means identifying precisely the variable item or property of the real world. The question leads us to the definition of the term variability subject"* [PBvdL05]. A *variability subject* is defined as follows: *"A variability subject is a variable item of the real world or a variable property of such an item"* [PBvdL05].

Why does it vary? *"There are different reasons for an item or property to vary: different stakeholder needs, different country laws, technical reasons, etc."* [PBvdL05].

How does it vary? *"This question deals with the different shapes a variability subject can take. To identify the different shapes of a variability subject we define the term variability object"* [PBvdL05]. A *variability object* is defined as follows: *"A variability object is a particular instance of a variability subject"* [PBvdL05].

Variability Representation

Variation Point A *variation point* is the SPL abstract view onto the real world *variability subject*. A variation point may find its concretization within all kinds of possible domain artifacts, such as architecture, requirements, test plans, etc. [PBvdL05].

Variant A *variant* is the SPL counterpart to the real world *variability object*. Variants are owned by variation points and may include variation points, as well [PBvdL05].

Variability can be expressed explicitly in a graphical variability model [vdLSR07]. A set of useful building blocks is depicted in Figure 2.8. Such a model consists of the aforementioned *variation points* and *variants*. Since both of them have to be linked together, the concept of *variability dependencies* is introduced. Regarding the example in Figure 2.9, an *intrusion detection* system may contain either a *camera surveillance* or *motion sensors* as a variant but not both at the same time. This is called an *alternative choice*. The variant *cullet detection* is optional and is not influenced by the other two variants [PBvdL05].

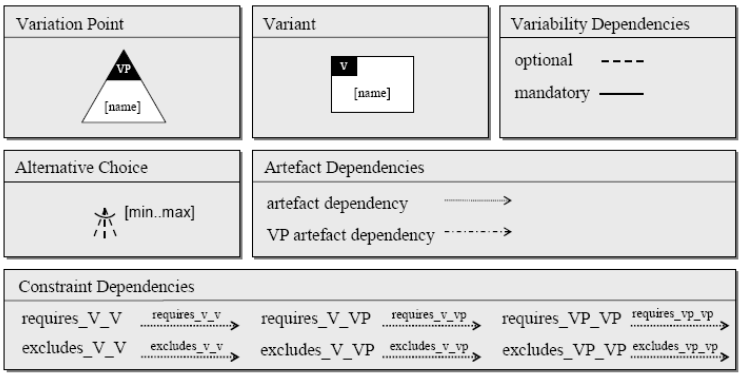

Figure 2.8.: Building Blocks for a Graphical Representation of Variability [PBvdL05]

So-called *constraint dependencies* are used to model constraints between variations or variations and variation points. They require each other or they exclude one from another. Regarding the example in Figure 2.9, the selection of variant *basic* requires the selection of the variants *motion sensors* and *keypad* [PBvdL05].

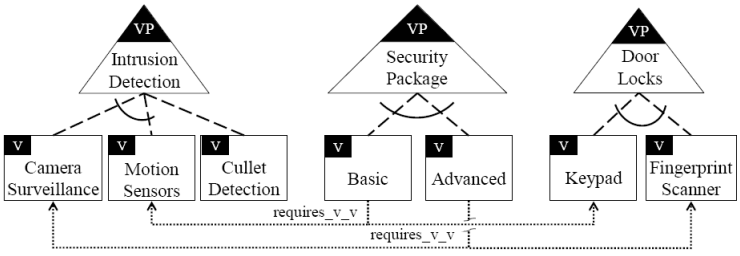

Figure 2.9.: Sample Graphical Representation of Variability [PBvdL05]

The graphical representation is explicit and not embedded in the realization artifacts, such as requirements, documentation, design. The concurrency of both, explicit graphical notation as well as realization artifacts is favorable and called *orthogonal variability model* [PBvdL05]. Figure 2.10 presents an example, where the graphical variability model is related with a domain artifact (use case diagram). The interrelation between variants and artifacts is modeled with *artifact dependencies*, that are graphical building blocks (regard Figure 2.8) [PBvdL05].

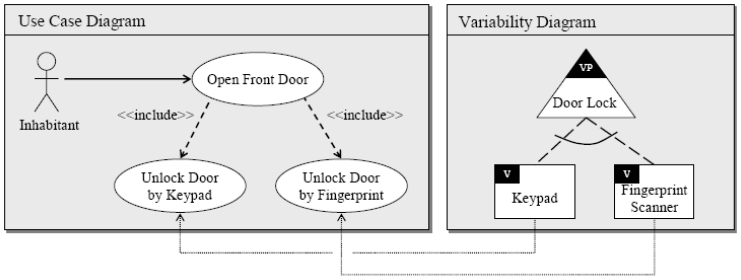

Figure 2.10.: Sample Graphical Representation of the Orthogonal Variability Model [PBvdL05]

Variability Realization

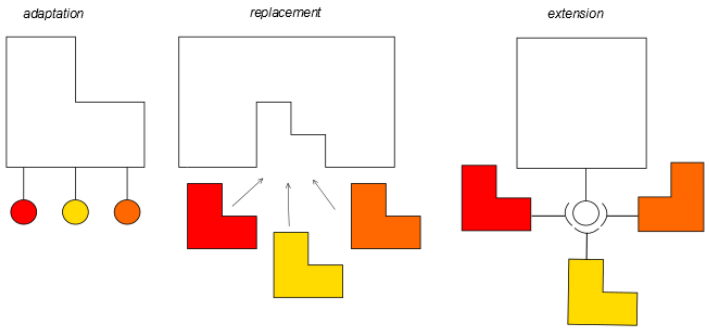

Figure 2.11.: Basic Variability Mechanisms [vdLSR07]

After [vdLSR07] there are three basic of variability mechanisms: *adaptation*, *replacement* and *extension*.

Adaptation In adaptation technique, only one implementation is available; changes to the software can be done via interfaces that might be: configuration files, run-time parametrization, etc.

Replacement In replacement technique, there exist several variants of the implementation which can be replaced; one of them can be chosen, or a product-specific version can be implemented.

Extension In extension technique, the architecture has to provide an interface to allow addition of components. Those can be product-specific or not. For instance, plugins relate to the extension mechanism.

2.4. Testing Software Product Lines

2.4.1. Test Artifacts

Test artifacts are defined in [PBvdL05] as follows: *"Test artifacts are products of the test process containing plans, specifications, and test results"*.

After [McG11] test artifacts include:

- Test plans
- Test cases
- Test reports
- Test data sets
- Test software and scripts

In [McG11] it is stated, that *"All test artifacts are under the control of the configuration management system so that when a specific build of a system is recreated, the appropriate test artifacts are also available"*.

2.4.2. Domain Testing Activities

Domain testing cannot be viewed completely without any interrelation to application testing. Regarding Software Product Line testing strategies, both, domain activities and application activities have to be taken into account in parallel (see also Section 2.4.5).

Since Software Product Lines incorporate the virtue of early testing [PBvdL05], testing in domain engineering appeals to be desirable. As testing is difficult in the construction phase (regard requirements engineering, domain design), other validation techniques should be considered. In [McG11] static-testing techniques are described for validating Software Product Lines, namely *inspections* and *architecture evaluation*.

Inspections

After [McG11], inspections in the SPL context *"are intended to determine whether an artifact is correct and complete"*. A product is derived by taking choices at all variation points, following some criteria [McG11]:

"An artifact is correct if it matches some standard deemed accurate by experts" [McG11].

"An artifact is complete if it addresses the full range of possible values as defined in its

specification or the product line scope document" [McG11].

"An artifact is consistent if it does not contain any contradictions among its internal components and does not contradict any other product line artifact" [McG11].

Architecture Evaluation

Architecture evaluation has been stated as a static testing technique in [McG11] and is described in [PBvdL05] as *"a means to assess the architecture according to certain selected quality requirements. The architecture is tested against a set of development scenarios"*. After [PBvdL05] those architecture evaluations take place under certain *quality issues* against whom they are tested. Quality issues may be security or safety concerns, for example.

With advancing maturity of the domain scope implementation, dynamic validation techniques, namely tests become applicable. A compelling problem in domain scope testing are absent variants [PBvdL05]. Absent variants are not developed in domain scope, if they are needed for one or only few product instances. Generally spoken, the existence of extension points leads to problems in domain testing since a fully composed product is not available (see also Section 2.4.5). Dynamic domain validation can be subdivided in the following processes: unit testing and integration testing.

Unit Testing

Domain unit testing resembles application unit testing, since both share the same problems or strategies. It is in the nature of things that unit testing takes place without the context of other units. If unit testing is too hard to accomplish, domain design should be reconsidered [PBvdL05]. The problem of not instantiated extension points or absent variants may be solved by test-driven techniques that help to create a unit test context (see Section 2.2.3).

Integration Testing

Integration testing in domain engineering is hard to accomplish fully since components occupy variants. The number of possible combinations often becomes very high, even if few variation points are defined [PBvdL05]. Therefore, exhaustive integration testing in domain engineering is not possible. Some SPL test strategies deal with domain integration testing (see Section 2.4.5). Nevertheless they cannot be viewed completely isolated from application integration testing.

2.4.3. Application Testing Activities

Even, if testing has been performed in the domain engineering phase it has to be repeated in the application phase to reassure no side effects take place [PBvdL05]. Reuse of domain test artifacts is desirable.

Unit Testing

Test cases for common units are available, at least in form of a skeleton. Partly, it will be necessary to change test data according to changes in application space. Variable test cases and so-called absent variants will have to be implemented, as well [PBvdL05].

Integration Testing

In application engineering, all variants have to be selected [PBvdL05]. Integration testing becomes applicable, since no exhaustive number of combinations of modules or components has to be tested (see Section 2.4.5).

2.4.4. Interrelation

In Figure 2.12, *Variant A* is extended either by *Module B* or by *Module D*. The structure of the product is reflected by the test structure. The *Test Variant A* is extended either by *Test Module B* or by *Test Module D*. Following this argumentation, test cases can be composed from partial test cases dependent on the product composition [McG11].

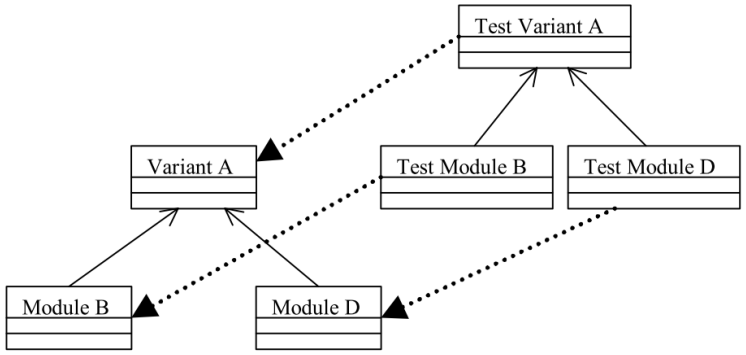

Figure 2.12.: Composing Partial Test Cases [McG11]

Tests Related to Variability

Since variants have to be selected during application engineering phase many configuration problems may occur:

Absence of Variants It can happen that a variant has not been selected. [PBvdL05] proposes some means to test this absence of variants. As well, some FODA-tools provide means to check those dependencies and report configuration errors.

Application Dependencies [PBvdL05] states the problems of dependencies, which means that selections are interrelated and dependent.

One has to check whether:

- *"the presence of variants violates any restrictions"* [PBvdL05].
- *"the absence of variants violates any restrictions"* [PBvdL05].

FODA-tools, such as pure::variants allow to define restrictions as well as constraints and, therefore, provide powerful means to prevent those configuration errors.

2.4.5. SPL Test Strategies

Brute Force Strategy

According to [PBvdL05] the brute force test strategy includes tests at all levels (unit test, integration test and system test) for all possible configurations in domain space only.

Benefits are early uncovered defects, compliant with the idea of early testing. In integration testing the number of possible configurations rises enormously with the number of variation points, so that even with some few variation points, the computing resources would not suffice. Another drawback are so-called absent variants and extension points. Using stubs incorporates other problems: after [PBvdL05] stubs do not reflect the reality and stubs may contain defects as well. Therefore, stubs would have to be tested, too.

Pure Application Strategy

According to [PBvdL05] application testing is performed in application space only; domain testing is neglected. This violates the domain space process, where testing is pointed out explicitly (see Section 2.3.4). Going further, this neglecting means that benefits of Software Product Lines can not be mapped to testing. In other words: testing is not stated in the domain process and systematic test reuse does not take place. Test reuse is only performed through means of test artifacts, if they are defined in domain space. The benefits of early testing are lost: domain space components are tested in application space at the moment they are released.

On the other hand, application testing is easy to accomplish, since it resembles single-system testing. All variants are bound and available. All extension points are implemented.

Sample Application Strategy

With this test strategy (see [PBvdL05]), one or some few sample applications are built up already in domain engineering and testing is carried out, using this sample application. The benefits are clear: no absent variants cause problems in testing. Early testing is possible. Nevertheless, sample application testing is not able to address all possible configurations, it can cover at least all commonalities with testing.

Commonality and Reuse Strategy

After [PBvdL05] domain testing aims at testing common parts and preparing test artifacts for variable parts. So, in domain testing test artifacts have to cover commonalities and regard variable parts as well. Some of the variable test artifacts are then implemented in application space.

This approach is the most sophisticated and perhaps most appealing. Despite this, one has to take into account that implementing a test framework for this test strategy is a real challenge and may pay-off only in variant-intensive Software Product Lines.

2.4.6. Testing Roles

The issue of testing roles is much discussed in literature [McG11][RGW03] regarding responsibilities and desired skills. According to [McG11] the following testing roles are defined:

Test Manager

The test manager is in charge of assigning test resources to a specified product according to quality requirements. This includes risk analysis, priorisation of tests and test cases and selecting a subset of tests that should be conducted.

Test Architect

The test architect is responsible for the provision of the test and test automation framework. In the scope of Software Product Lines, those frameworks are of high complexity. According to [McG11] the architect specifies the test artifacts of the SPL. These test artifacts are produced by different testing roles.

Tester

The tester implements the test cases in order to unveil actual and possible defects. The testers role in TDD differs a little from the traditional one. In TDD (see Section 2.2), testers put the cart before the horse: tests are created before implementing of the corresponding functionality. In this context, tests assure that implemented functionality sticks to requirements.

2.5. Testing in the Automotive Domain

2.5.1. Classification Tree Method/Embedded Systems

CTM/ES [Con04][KM10][Wol11][Vig10] is the extension of CTM that enables discrete-time testing. First we will have a shorthand look upon CTM and investigate CTM/ES later on.

Equivalence Classes

Equivalence classes are described in [Vig10]. They are will be explained by a short example, below.

Consider an automotve software unit that alerts the driver if the velocity of the vehicle is higher than 130 km/h. It is useful to distinguish the possible input values in three equivalence classes:

```
EC1: velocity  <  0 (no alert)
EC2: velocity > 130 (alert)
EC3: 0 <= 130 <= velocity (no alert)
```

It is apparently not very efficient and of no use to test all values of every EC. The better way is to find values at the borders of the ECs f.i.

```
EC1, test1: velocity = -10
EC1, test2: velocity = -1

EC2, test2: velocity = 0
EC2, test3: velocity = 10
EC2, test4: velocity = 120
EC2, test5: velocity = 130

EC3, test6: velocity = 131
EC3, test7: velocity = 140
```

Classification Tree Method

The Classification Tree Method is described in [Vig10] and is a more sophisticated way to break down input values into different ECs. It is based on a graphical notation and thus is more intuitive for test engineers. As the name states, CTM forms a tree. The root represents the name of the unit under test. The nodes model so-called aspects that have influence on the information processing inside the black box. By assuming aspects, testers use some knowledge of the test objective's inner implementation. This is called it a grey box approach. The nodes constitute the equivalence classes.

The process can be viewed as a four step cycle [Vig10]:

- Identify aspects that have influence on the systems behavior
- For every input parameter find reasonable equivalence classes
- Specify test cases: find a combination of equivalence classes for each of it
- Run the test cases

The Classification Tree Method has a severe drawback: it does not fit to discrete-time systems. It is hardly possible to create time-varying test-input signals. Therefore, CTM has been extended to the so-called CTM/ES.

Classification Tree Method / Embedded Systems

With CTM/ES [Con04][KM10][Wol11][Vig10] it is possible to create time-varying signals in an easy and graphical way.

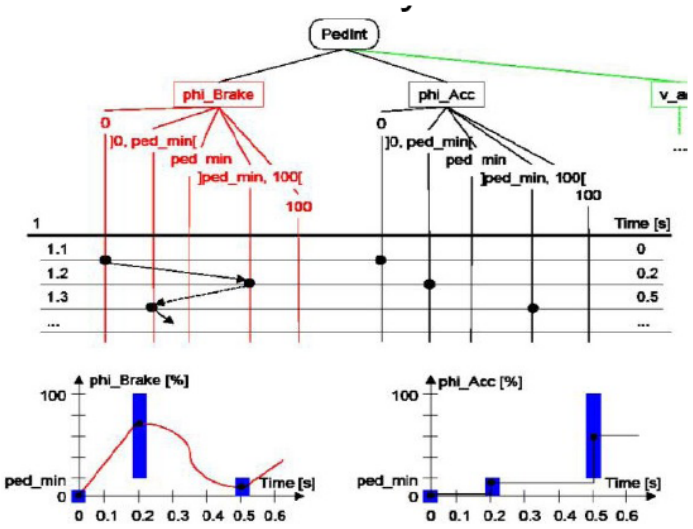

Figure 2.13.: CTM/ES Creating Time-Discrete Signals by Example [Wol11]

In Figure 2.13 some points in time are added that mark changes or transitions from one equivalence class to another. These points in time are displayed as horizontal lines. By marking this points the test engineer determines values of input signals at that specific moment. Between this time points other values have to be created by means of interpolation. These interpolations or transitions are displayed as a line between the marked points. F.i. if a linear interpolation between two points is desired, the line would be dotted, if a cubic spline interpolation is desired, it would be continuous.

The process is a five-step cycle (see also [Vig10] and [Wol11]):

- Identify aspects that have influence on the systems behavior
- For every input parameter find reasonable equivalence classes
- Find adequate points in time
- Specify test cases:
  - Find a combination of equivalence classes for each point in time (mark a point)
  - Find the corresponding transition (by determining the line style)
- Run tests

2.5.2. Time Partition Testing

TPT is a common and widespread test description language in the embedded systems domain and has been developed in [Leh04].

TPT-Diagrams

The TPT language is based on graphical notation for the purpose of intuitive understanding. A test scenario is defined as a sequence of states. Each state can be linked with a natural-language description (see Figure 2.15).

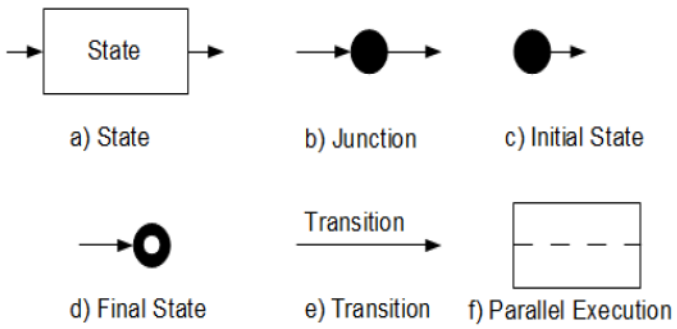

Figure 2.14.: Components of Time Partitioning Testing [Con04]

TPT-Diagrams consist of the following components (see Figure 2.14):

States A state represents a current phase in the test. It is entered and leaved by transitions. A state may consist of a sub-scenario. In TPT, a state is displayed as a rectangle with a short natural language description.

Junctions Junctions can embrace two or more transitions or branch them out. They are no states, just a semantic graphical component.

Initial States An initial state is an entry point to a scenario.

Final States A final state terminates the scenario.

Transitions Transitions are connections between states, junctions initial and final states. They represent the order, direction and orientation by which the diagram may be traversed. Transitions are linked with a formal transition condition. They are depicted as arrows.

Parallelization TPT-diagrams allow parallelization of automata. This parallelization is represented by a horizontal dotted line.

Variability Mechanisms

Introducing variability can enhance test case reuse as can be seen in the example from [Leh04]. A test scenario may be starting an engine and simulating an engine speed error. The scenario remains the same for many different engines and pre-conditions on a high level. The generic sequence is:

- Start engine
- Wait for a specified engine speed
- Simulate the error.

An example of possible variations is given in Figure 2.15. Generating a specific test case means to choose the appropriate variation. In terms of TPT, the structure with all the possible variations is called testlet. By choosing a combination, a specific test scenario is created.

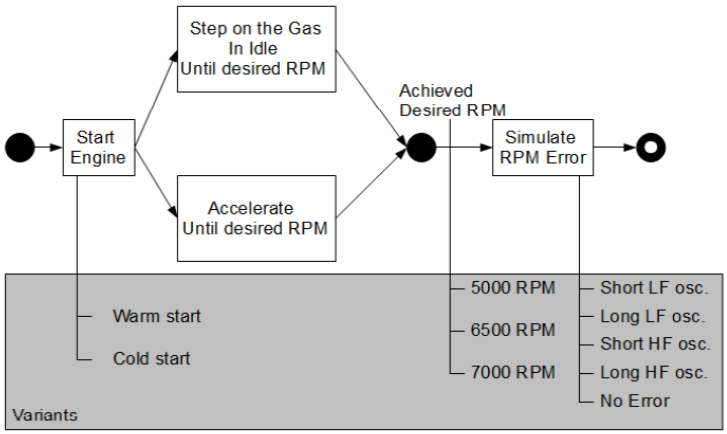

Figure 2.15.: Example of Variablity Mechanisms in Time Partitioning Testing

2.5.3. A Signal Feature Concept

Signal features serve as a means to analyze signals. Zander-Nowicka [ZN09] has considered signal features as a way to generate signals, as well.

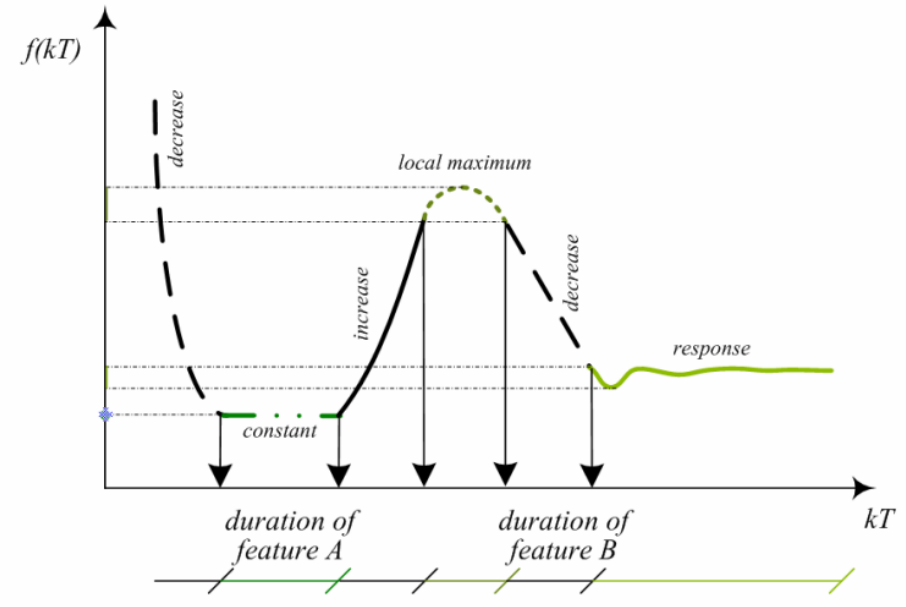

Figure 2.16.: Several Signal Features by Example [ZN09]

The drawback of many regression test evaluation methods is that a reference signal is needed. This constitutes a problem, since:

- In TDD tests are created before the functionality is implemented. A reference signal is usually obtained by recording the outputs of a module. Since there is no module for creating a reference signal at the time tests are written, this is apparently a contradiction.
- Of course, a reference signal can be created handcrafted, which causes a lot of work.
- In discrete-time signal processing, signals are treated as a series of values in time. There is no abstract perception on signals that can serve for variability issues

In Figure 2.16 the two perceptions on signals are shown. The line illustrates the traditional discrete-time signal processing viewpoint. The signal feature concept *extracts* so-called *signal features*. For example, in the diagram, the signal feature A (*constant*) is displayed as a green line. Additionally, the signal feature has a duration time.

In the opinion of the author, signal features are important because they provide an easy means to formulate expected results.

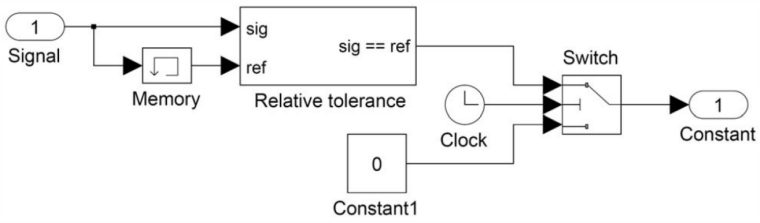

Figure 2.17.: Extraction of a *Constant* Signal Feature in Simulink [ZN09]

A simple example has been described in [ZN09] and will be explained shortly. In Figure 2.17 a feature extraction of a constant in Simulink is depicted. The *Signal* is compared with a time-shifted version of itself and the *Signal* is the *sig* input to the *Relative tolerance Block*. The shifted *Signal* is the *ref* input to the *Relative tolerance Block*. The *Relative tolerance Block* compares the two signals and allows a *specified* relative tolerance. After [ZN09] *"the clock and the switch were introduced to prevent bad outputs derived from an unfortunate choice of the initial output of the memory block. They provide no effect after the first time step."*

2.6. Hypothesis

The goal of this thesis is to introduce a Software Product Line testing approach to automotive control software, that is developed within the Matlab/Simulink environment.

As a first outcome of the analysis and synthesis of related work, a test-driven development approach seems to be compelling. Test-driven development solutions are possible to implement with the help of two test frameworks: mlUnit and slUnit. Another basic idea of test-driven development is that a test case represents a requirement. Connecting tests with requirements is a desirable feature since they provide a requirements-based test selection mechanism and favor test case reuse as well.

When regarding Software Product Lines, testing issues emerge to be more complex than in single-system development. Dealing with the higher complexity demands is challenging but comes with the price of systematic test reuse. Test case selection as mentioned above does not reflect the whole spectrum of Software Product Line testing. The construction of test artifacts seems to be sufficient to represent and classify all facets: test plans, test cases, test data, test reports, test software ans scripts. All those parts will have to be tightened together by some means. FODA-tools incorporate an appropriate way to connect them all.

Finally, the problem of evaluating tests is very compelling. The signal feature concept presented in Section 2.5.3 seems to be an appropriate test evaluation mechanism that is applicable to test-driven development. Signals features are not limited to test evaluation. They enable test signal generation as well. The idea of enhancing signal features with variability in a Software Product Line context was rather inspiring and helped to bridge the gap between Software Product Line Engineering and testing variant-intensive embedded software.

3. Concepts

3.1. Selection of Test Frameworks

3.1.1. slUnit

slUnit was previously described in Section 2.2.6. Several reasons have led to the decision to use slUnit:

slUnit is embedded in Simulink. This means that the tests can be written in the same language as the implementation.

The source code of slUnit is open source: thus, it is easy to change slUnit in order to integrate it with the other frameworks used in this thesis. slUnit has been changed in two ways: first, the slUnit test automation mechanism has been removed because it provides no possibility to create test reports in a textual form. The test execution is controlled by mlUnit test methods in the framework developed in this thesis, instead. Second, slUnit assertions have been replaced by Simulink assertions because slUnit assertions have a very poor timing performance.

slUnit is not very complex. In order to build a prototypical implementation, it is desirable that the underlying frameworks are not too complicated because it would take a lot of time to get familiar with them, otherwise.

slUnit is explicitly designed for unit testing. It follows the patterns of xUnit.

3.1.2. mlUnit

mlUnit was previously described in Section 2.2.5. Several reasons have led to the decision to use mlUnit:

mlUnit is a Matlab based test framework. Thus, it is relatively easy to use the Matlab/Simulink API within the mlUnit test methods f.i. to start simulations and log signals.

The source code of mlUnit is open source: thus, it is easy to use and understand the API of mlUnit.

mlUnit provides a very good means to generate test reports. It is possible to create a test report for each unit under test. Additionally, it is possible to add together the test reports of all units in one global test report.

mlUnit is explicitly designed for unit testing. It follows the patterns of xUnit.

3.2. Separation of Spaces

FODA-tools are based on a separation of spaces: besides the two development cycles (domain engineering, application engineering), implementations are split up into problem space and solution space. Separation of spaces is important because Software Product Lines are usually built up by experts out of different domains.

In the current thesis, the pure::variants software enables such a separation of spaces. In Figure 3.1 the four different spaces of a pure::variants SPL are illustrated.

In domain space, feature models represent the *what* of the Software Product Line. The *what* is modeled by domain experts that do not have to be familiar with the technical solution that is implemented by software experts. Feature models are a generic definition of problem, a specific definition is done in application space.

Family models are also located in domain space. The represent the *how* of the SPL: the technical solutions that are usually created by software engineers. Family models are connected with implementations of Simulink libraries. In order to connect them to a family model, these Simulink libraries have to be enhanced with variability information.

The separation between feature models and family models is important because domain experts have different views onto the SPL:

The requirements engineer does not know, how the requirements are implemented with test cases. Thus, the requirements engineer has a non-technical view on the Software Product Line. He creates the *requirements model* in pure::variants as a feature model.

The general features of a car, such as maximum velocity and acceleration are usually not determined by automotive engineers. They are determined by management based on a carefully scoped product portfolio. Thus, the general features provide in principle the business side of the SPL. Therefore it is represented by a feature model: the *vehicle config model*.

Which units are integrated into the software system represents the *what* of the SPL. It is represented by the so-called *software system config model*, a feature model.

The unit test model is the technical counterpart of the *requirements model*. Thus, the unit test model represents the *how* of the SPL and is represented by a family model.

The *test registry* provides test scripts in order to automate testing of several units. Thus, it is a technical solution and is represented by a family model.

In application space, the variant description model is derived from the feature model, which means that the generic problem definition is turned into a specific problem definition by selecting variants.

When the problem is defined, the variant result model represents a specific technical solution. The last step in pure::variants is the generation of desired implementation. In Figure 3.1 the outcome of the generation are: Simulink unit tests, mlUnit tests and test scripts. These implementations can be executed automatically and provide test reports and test signals as a result.

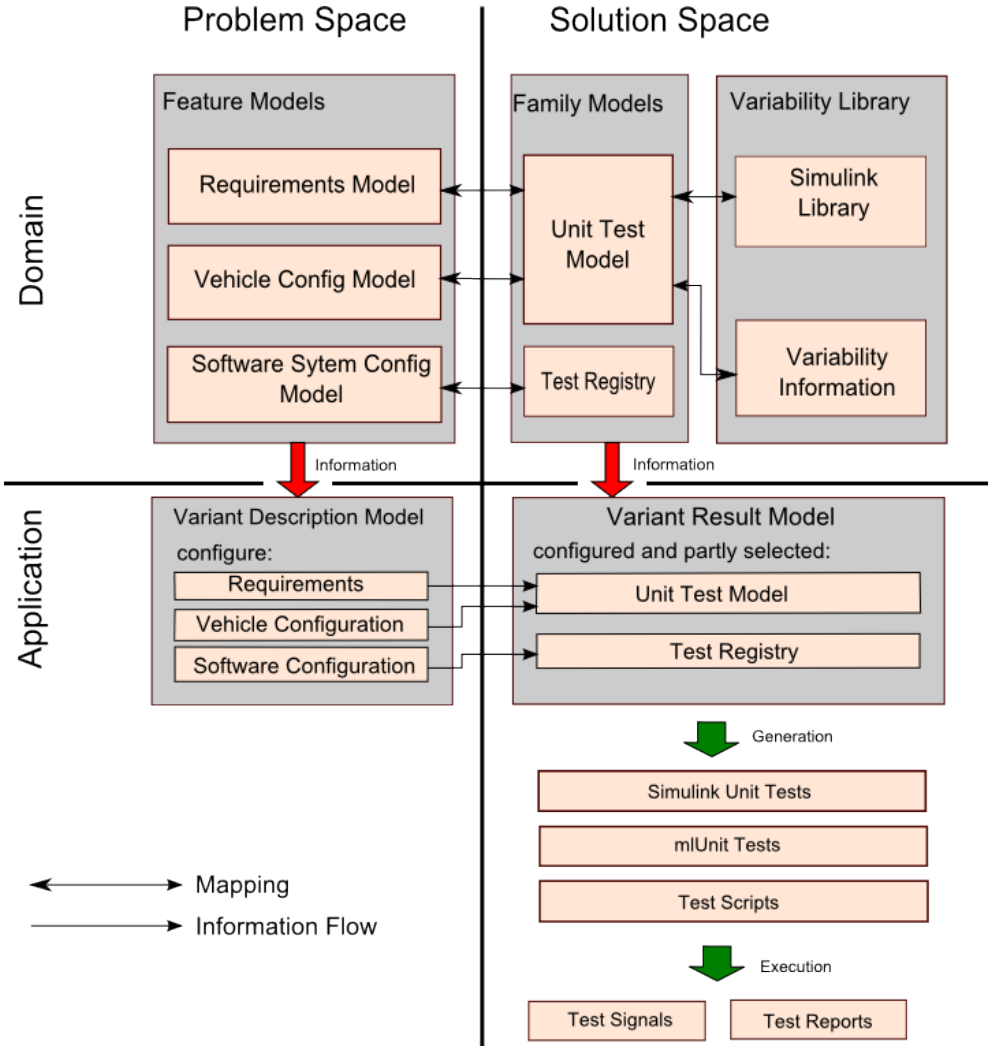

Figure 3.1.: Transformation Process of the Proposed Methodology Including Artifacts

3.3. Development Process

As can be seen in Figure 3.2, the development process is split up into two parts: domain engineering process and application engineering process. The process steps are connected with arrows which indicates the sequence of their execution. The involvement of the different roles is also illustrated in Figure 3.2. Roles in the context of this development process are a model to describe activities that are related and coherent. It is not mandatory that each of the roles is assigned a different person.

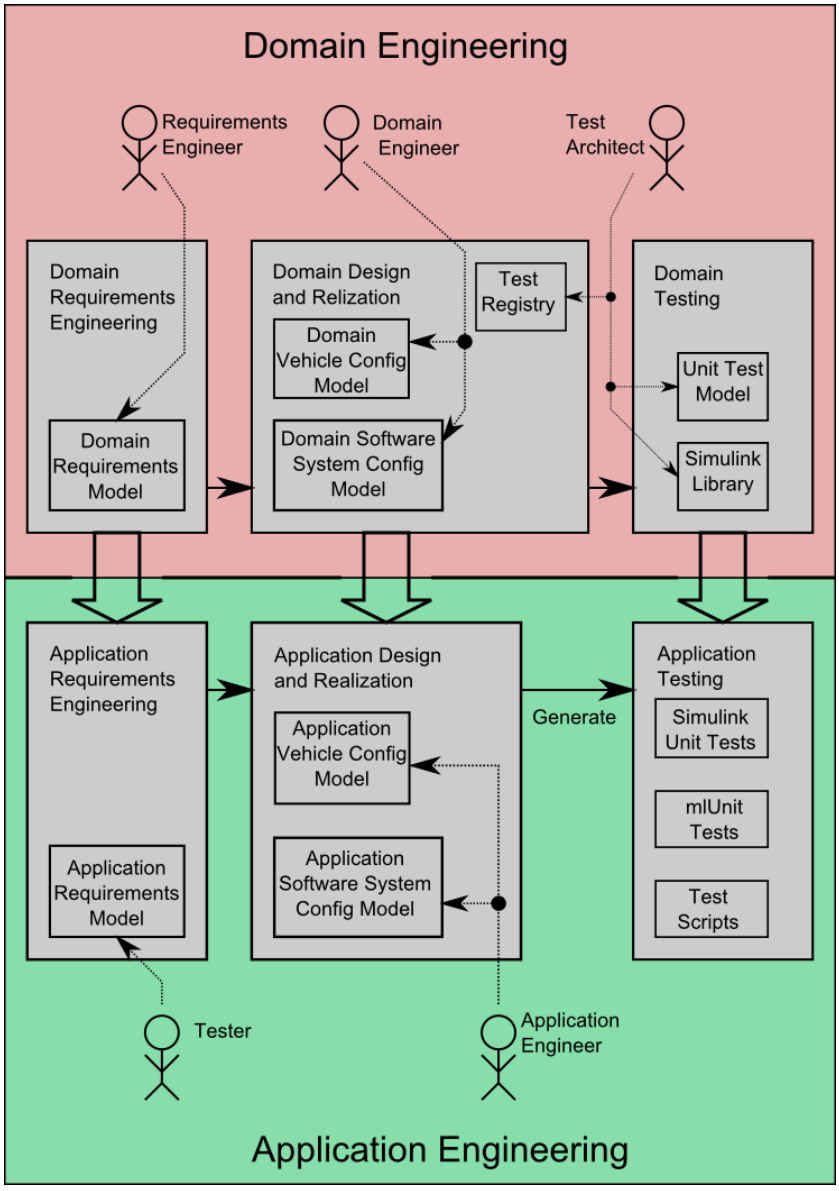

Figure 3.2.: Development Process Including Roles, Responsibilities and Artifacts

3.3.1. Domain Development Process With Focus On Unit Testing

Domain Requirements Engineering

In domain requirements engineering, the requirements engineer is responsible for gathering reusable requirements, documenting them and deriving a *domain requirements model* (see Section 5.3). The *domain requirements model* is realized as a pure::variants feature model, which has to be created manually (see Section 5.3).

Domain Design and Realization

In domain design and realization, domain engineers are in charge of building the *domain software system config model* and the *domain vehicle config model*.

The *domain software system config model* lists all units that are part of the software system under test. It is a pure::variants feature model and has to be created manually.

The *domain vehicle config model* contains features that may change from one vehicle to another. Such a feature may be for instance the maximum velocity of a vehicle. It is a pure::variants feature model and has to be created manually.

Domain engineers in the context of this process are responsible for all non-testing activities, such as software engineering.

The *test registry* connects all software units under test and allows the full automation of the test process via test scripts. After the creating the test registry manually, the test architect has to map the test registry with the *domain software system config model*.

Domain Testing

The test architect is responsible for creating reusable Simulink test libraries and enhancing them with variability information; the connection of both constitutes the *variability library* (see Section 4.3.4). Second, the variability library can be transformed to a family model automatically: the *unit test model*. The last step to be done is to connect the unit test model with the corresponding *domain requirements model* and the *domain vehicle config model*. This step has to be carried out manually.

3.3.2. Application Development Process With Focus On Unit Testing

Application Requirements Engineering

Application requirements engineering means to configure the *application requirements model* which is derived from its domain engineering counterpart. The configuration is done by the tester who has to type in the appropriate test data, here.

Application Design and Realization

In application design and realization, the application engineers have to create a working instance of the SPL. They accomplish this task by configuring the *application vehicle config*

model and the *application software system config model*. These models are derived from their domain counterparts and are used to configure the actual software.

Application Testing

In application testing, executable test artifacts are generated by the FODA-tool. The Matlab script *test_registry.m* (see also Section 3.6.4) has to be started manually. This script navigates to the folder where the unit under test is contained and calls the *run_all.m* script there (see also Section 3.6.1). The *run_all.m* script launches the *mlUnit test suite* that controls all *slUnit test cases* in order to test the unit under test. After the execution of all executable test artifacts, a global test report is generated automatically.

3.4. Test Artifacts

As described in Section 2.4.1, test artifacts include: test plans, test cases, test reports, test data sets, test software and scripts. Figure 3.3 gives an overview of the representation of test artifacts within different tools.

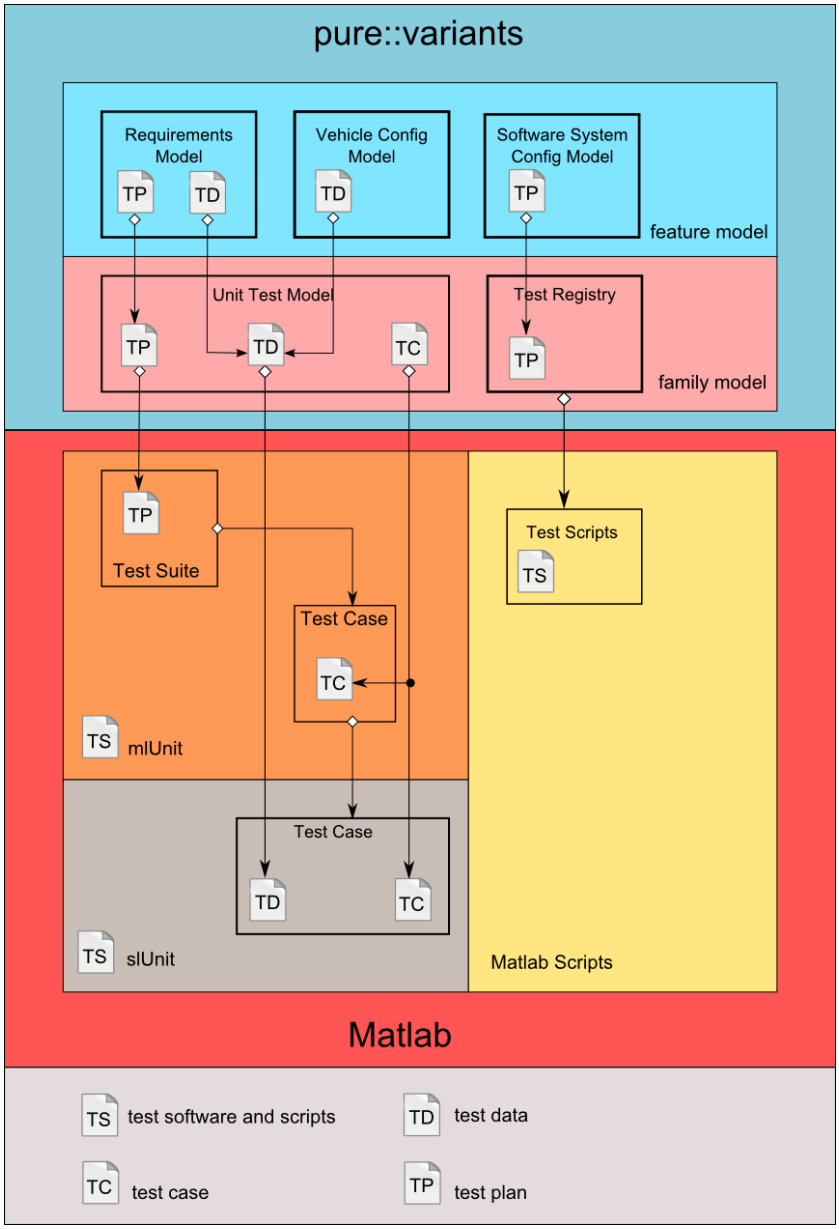

Figure 3.3.: Test Artifacts are Represented and Implemented Within Different Tools

3.4.1. Test Plans

Test plans define which testing activities should be performed. In Figure 3.3 there are two feature models depicted, that influence the selection of test cases:

Relation to Requirements Model The *requirements model* is described in Section 3.5.1. As stated in Section 2.2.2, each test case should be linked with a requirement. Requirements can be selected in the feature models; by implication only those test cases will be conducted, that are linked with a selected requirement.

Relation to Software System Config Model The *software system config model* (see Section 3.5.3) describes, which units are contained in the software. Test scripts are then generated by the *test registry* that is linked with the *software system config model*. The generated test scripts will only launch tests for the units selected in the *software system config model*.

3.4.2. Test Data Sets

Test data are values needed by the test cases (see Figure 3.3). The test data sets are provided by two feature models:

Relation to Requirements Model The *requirements model* is a feature model derived from natural language requirements (see Section 5.3). It presents an interface to the tester (see Section 3.5.1). He has to type in values here.

Relation to Vehicle Config Model The *vehicle config model* (see section 3.5.2) represents values that can be derived from the components of a vehicle, f.i. the maximum torque of the engine. The *vehicle config model* is desirable, since the corresponding test data change with the variable properties of a car's components.

3.4.3. Test Software and Scripts

Relation to Test Software The test software consists of slUnit blocks as well as of mlUnit test cases and test suites. Such a composition of test software is able to run the tests for a whole unit under test.

Relation to Test Scripts The test scripts are Matlab scripts to enable testing of more than one unit under test in a proper way. Additionally, the test scripts save all signals and put together all test reports.

3.4.4. Test Reports

Test reports are generated by the *mlUnit* framework. Those test reports are generated for every unit under test. Since more then one unit can be tested, they have to be put together. This task is accomplished via the test scripts generated by the *test registry*.

3.4.5. Test Cases

Test cases are executeable test artifacts. Test cases are represented in Simulink as slUnit *TestCase* blocks and in Matlab as mlUnit *test_case* classes. All these parts have to be connected to a pure::variants family model (see Figure 3.6, Figure 3.7): it is called *unit test model* and is described in more detail in Section 3.6.1.

3.5. Problem Space

3.5.1. Requirements Model

To benefit from Software Product Line principles it is useful to derive a semi-formal requirements model. The requirements model is implemented in pure::variants and helps to provide test case reuse. It delivers a possibility for the user (the tester) to type in test data that have to be manually calculated. In fact it is a better practice to provide values in a requirements model than just typing it directly into the test case. This may seem trivial but in fact, linking a test case with a requirement is the major driving force for this thesis. Because it is so important, the author will cite the basic source of his inspiration, that was stated anecdotal in [Eng03]:

"When making test cases the test engineer tends to write the test case immediately, and instruct the tester f.I. the value '9'. It is not explained however why this is. The test engineer may initially have a good reason for choosing this value (say it is a boundary value), but if it is not connected to a requirement or implementation decision, and if the 'why' is not known the test case may become virtually worthless, after a change in requirements or in a new environment, f.I. A new product line, in which the component is used."

Summarized, arguments for a requirements model are:

- The tester is presented explicitly an envelope (the requirements model). Values must not be typed in the test case directly, because some test data may be derived from other models (f.i. *vehicle config model*).
- Since the requirement is linked with a purpose and a description the tester knows the *why* of the test case. Otherwise the test case would become worthless, if the requirements associated with it changes.
- Because every value in the requirements model is treated as an envelope, it enables test case reuse, since only variable values have to be entered. This can be done in a FODA-tool that enables automatic test case generation.
- A requirement can be linked with more than one test cases. Therefore, values that remain the same for some test cases just have to be typed in once. Error-prone manual retyping of values is minimized, thus.

3.5.2. Vehicle Config Model

The vehicle config model contains all vehicle-specific values that change from one vehicle to another. Some of the values are boundary values, they can be considered as vehicle specific equivalence classes. Many test data can be directly mapped to those vehicle-dependent values or derived from them in a mathematical way. The benefits are obvious: these values must be typed in *once* for a specific vehicle and can be linked to test cases. Much nasty, work-intensive and error-prone retyping can be avoided. For instance, if a value changes, it has to be retyped once in the vehicle config model and not in all linked test cases. If the number of test cases is high, testers would run into severe trouble finding all test cases where values must be changed, otherwise.

3.5.3. Software System Config Model

In the *HybConS* project it is planned to configure the software components with the help of a tool chain including pure::variants. This tool chain is currently in work and not available yet. Therefore the *software system config model* is implemented as a simple feature model that provides switches so that the tester can choose the software units under test. In Figure 3.4 the user can choose whether the unit *ReqRecupMotrn* is tested or not. Based on the selection of units in the variant description model the corresponding test registry is configured that is responsible for creating the appropriate test scripts for test automation.

Addionally, it is possible to select two alternative implementations of the unit *ReqRecupMotrn*: *ReqHybMotrn_comp1* or *ReqHybMod_comp2*. Based on the selection the implementation, the *test case implementation* is determined (see Section 3.7.4).

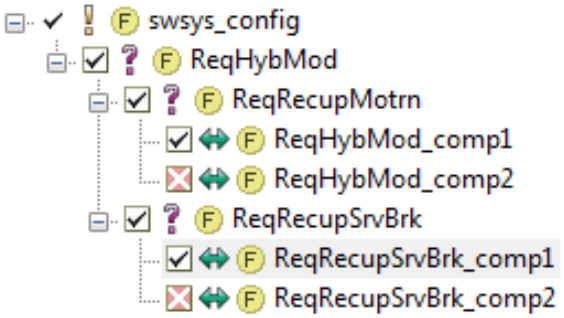

Figure 3.4.: Sample *software system config model* Variant Description Model

3.6. Solution Space - pure::variants

3.6.1. Unit Test Model

The *unit test model* is the main part of the framework. In the scope of one unit under test, it is responsible for the construction of:

- mlUnit test suite and test cases. In Figure 3.5 this is represented by the ps:component *test*.
- The testing composition and parametrization in Simulink. This is represented by the ps:component *system* in Figure 3.5.
- A Matlab script called *run_all* that covers aspects of automation in the scope of the unit under test. In Figure 3.5 it is represented by the ps:component *automate*.

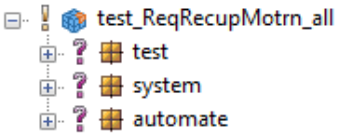

Figure 3.5.: Sample Unit Test Model, Main Parts

3.6.2. Unit Test Model: test

The ps:component *test* consists of two other ps:components:

- *test_cases*
- *test_suites*

In Figure 3.6 the ps:component *test_cases* owns for each test case again a ps:component, in this example they are named *ref_TestA_1* and *ref_TestB_1*. For each of the ps:components representing a test case, a subdirectory is created. Within this subdirectory four files are generated:

- *setUp.m* is created by the ps:part *test_case:tc_setUp*
- *tearDown.m* is created by the ps:part *test_case:tc_tearDown*
- *createTest.m* is created by the ps:part *test_case:tc_createTest*
- The constructor of the mlUnit test case is created by the ps:part *test_case:tc_createConstructor*

The ps:component *test_suites* is responsible for the creation of a mlUnit test suite. The ps:part *test_suite:ts_ref_TestComposite* creates one file that is responsible for the execution of the test suite. In this example it consists of three ps:fragments:

- *test_ref_TestComposite*: This fragment creates a Matlab function and code for a test suite, but no test cases are added in this fragment.
- *test_ref_TestA_1*: In this fragment a line of code is generated, that adds the test case *ref_TestA_1* to the test suite.
- *test_ref_TestB_1*: In this fragment a line of code is generated, that adds the test case *ref_TestB_1* to the test suite.

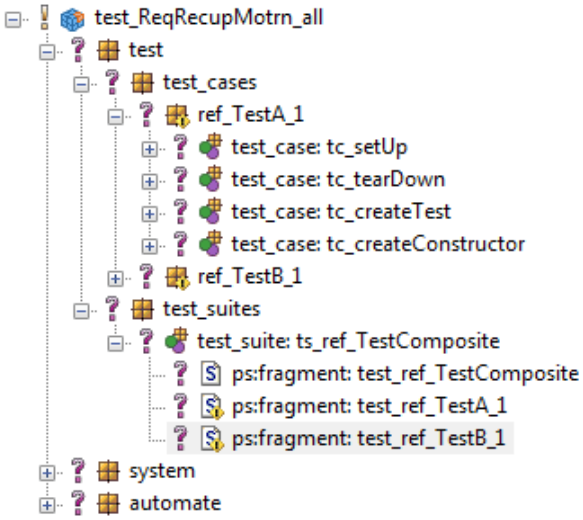

Figure 3.6.: Sample Unit Test Model, Test Part Expanded

3.6.3. Unit Test Model: system

In Figure 3.7 the ps:component named *system* and its subtree is responsible for building up and parametrizing the test system in Simulink. The ps:component *system* has no functionality, it is inserted for structural reasons. All other ps:components represent a counterpart in Simulink. They are either inner nodes or leaves:

- *Inner nodes*: Inner nodes always represent *referenced subsystems* (see Section 4.3.4). Referenced subsystems can be test composites, test cases or normal subsystems.
- *Leaves*: Leaves may be all other types of Simulink blocks, f.i. *Simulink Constant Blocks*.

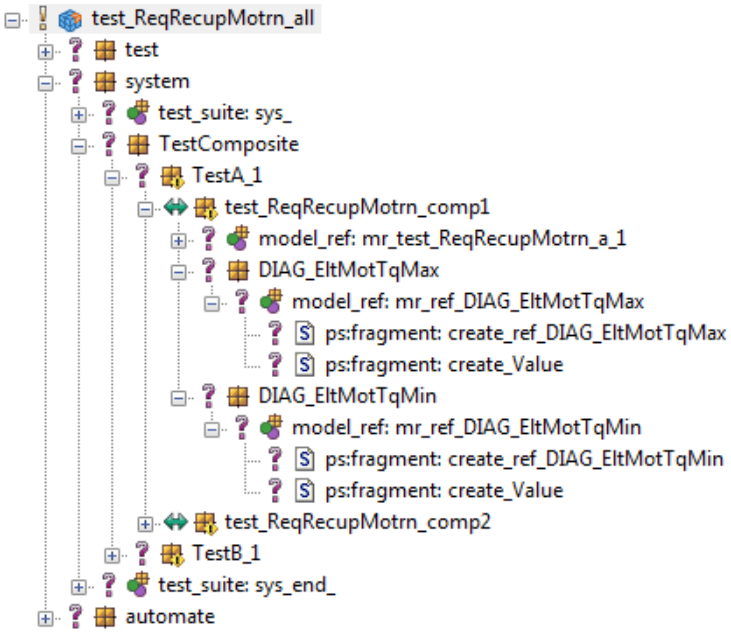

Figure 3.7.: Sample Unit Test Model, System Part Expanded

Usually, the test code will be implemented in Simulink. For variable test data it is desirable to influence them by means of a variant management tool.

3.6.4. Test Registry

The *test registry* (see Figure 3.8) is the part of software that helps to automate testing in Matlab. During model transformation the Matlab script *test_registry.m* is generated. It is the only Matlab script that has to be started manually in order to run all tests (see also Section 3.3.2).

In the *config_all* part, code for the configuration of the implemented framework is generated. This is all in all, adding framework functions to the search path as well as initialising the library.

The ps:component *ConfigPoet* is responsible for setting up the test environment.

The other parts of the test registry change the path to the unit under test and call the automation script *run_all* (see Section 3.6.1). Each of these parts can be linked with a restriction and thus, the appropriate selection of tested units can be done by means of a feature model.

The family model of the test registry has to be created manually which is not too complicated and can easily be done by non-experts, too.

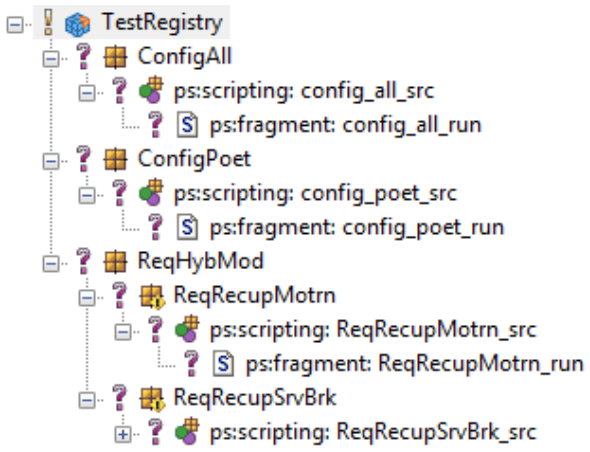

Figure 3.8.: Sample Representation of a Test Registry in pure::variants

3.7. Solution Space - Simulink

3.7.1. Test Bed

The unit under test is *ReqRecupMotrn* (see Figure 3.9). The *test block* (see also Section 4.2.1) is called *test_ReqRecupMotrn_all*. It provides the test code in Simulink. It generates test data and evaluates the feedback from the unit under test.

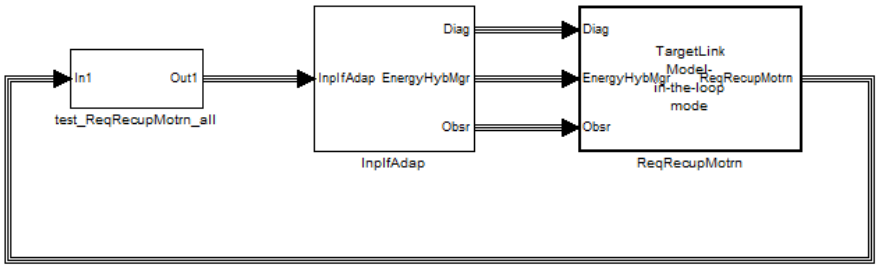

Figure 3.9.: Sample *Test Bed* Including *Test Block* and Unit Under Test

3.7.2. Test Block

Figure 3.10 shows the contents of the *test block*. The *test composite* contains all *test cases*. The *Subsystem* connects the output signals (test data) of the *test composite* with a bus.

The *test block* is represented as a ps:family (see Figure 3.7). The corresponding family model in pure::variants is the so-called *unit test model* (see Section 3.6.1).

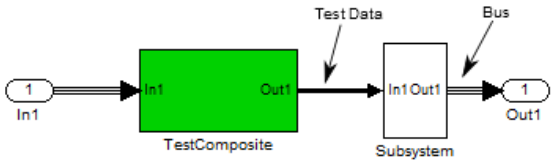

Figure 3.10.: Contents of the *Test Block*

3.7.3. Test Composite

The *test composite* (see also Section 4.2.2) in Figure 3.11 contains two test cases. The *Multiplexer1* is a *slUnit multiplexer* (see also Section 4.2.2). It connects the output of exactly one *test case* with the unit under test.

The *test composite* is represented in the unit test model twice. First, the ps:component *test_suites* in Figure 3.6 is responsible for controlling the execution of *test cases* within the *test composite*. For instance, the *test case Test_A1* is only executed, if the ps:fragment *test_ref_Test_A1* is added to the ps:part *test_suite:ts_ref_TestComposite* (see Figure 3.6).

The ps:fragment *test_ref_Test_A1* itself is linked to a feature model, the *requirements model* by means of a restriction. It is only added if the corresponding requirement is selected. A sample mapping between requirements and *test cases* is listed in Section 5.2.

Second, it is represented by the ps:component *TestComposite* in Figure 3.7 for structural reasons, only.

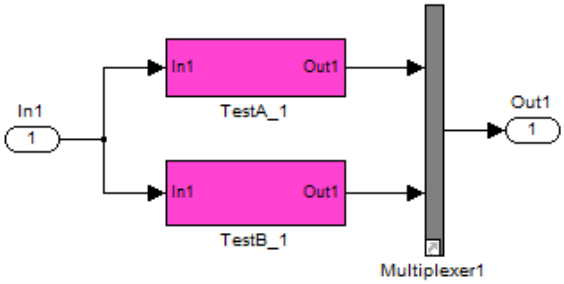

Figure 3.11.: Contents of the *Test Composite* Including *Test Cases* and *slUnit Multiplexer*

3.7.4. Test Case

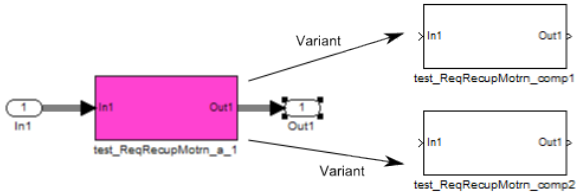

Figure 3.12.: Contents of a *Test Case* Including the *Test Case Implementation*

Figure 3.12 shows the contents of the *test case TestA_1*. The *test case* (see also Section 4.2.3) contains simply a subsystem called *test_ReqRecupMotrn_a_1*. It is a placeholder that enables variable subsystems (see also Section 4.3). The placeholder can be filled during model transformation with the *test case implementation* which is either *test_ReqRecupMotrn_comp1* or *test_ReqRecupMotrn_comp2*. Variable *test case implementations* make it possible to test different implementations of the unit under test.

Each *test case implementation* is represented in the unit test model (see Figure 3.7): the *test case implementation test_ReqRecupMotrn_comp1* is represented by the ps:component *test_ReqRecupMotrn_comp1* in the unit test model. The *test case implementation test_ReqRecupMotrn_comp2* is represented by the ps:component *test_ReqRecupMotrn_comp2* in the unit test model. Because only one of them can be selected, both of them have the variation type *ps:alternative*.

Each representation of the *test case implementation* (see Figure 3.7) is linked with the *software system config model* (see Section 3.5.3) with a restriction.

3.7.5. Test Case Implementation

In Figure 3.13 the Simulink Constant Blocks colored in magenta create the output signals (test data):

- *DIAG_EltMotTqMin*
- *OBEM_EltMotTqMinContns*

Both Simulink Constant Blocks are represented in the unit test model (see Figure 3.7). Thus, the values of the Constant Blocks can be influenced by pure::variants. The representation of the Simulink Constant Blocks in the unit test model can be linked with the *requirements model* (see Section 3.5.1) or with the *vehicle config model* (see Section 3.5.2).

The feedback from the unit under test is evaluated in the *validate* subsystem. The feedback data are compared with the signal *DIAG_EltMotTqMin*.

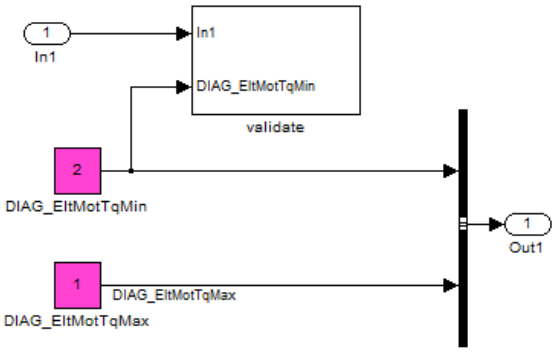

Figure 3.13.: Contents of the *Test Case Implementation*

3.8. Signal Evaluation

Within a *test case* it has to be checked if the feedback from the unit under test is correct. The signal evaluation follows the principles explained in Section 2.5.3 and will be explained by an example, below.

Consider that you have to check if a signal remains between an upper level and a lower level. In this example, the signal that has to be checked is called *sig*. It must be higher than -10 and lower than 10. In Figure 3.14 the subsystem *is_lower* checks whether *sig* is lower than 10. If yes, its output will always be a logic high. The subsystem *is_higher* checks whether *sig* is higher than -10. If yes, its output will always be a logic high. The

outputs of *is_higher* and *is_lower* are conjuncted with an *and*. If both inputs of the *and* block are a logical high, the input of the *assertion* is high, and the *assertion* is not violated. If one of the inputs of the *and* block changes to a logical low, the *assertion* is violated and listed in the test report.

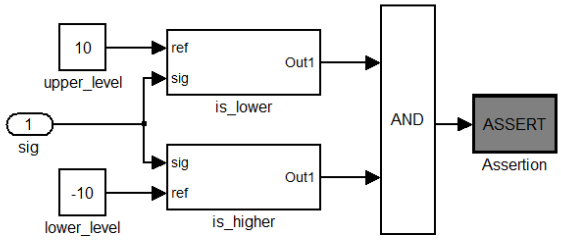

Figure 3.14.: Sample Test Evaluation Code in Simulink

3.9. A Concept of a Pivot

The concept of a pivot was rather inspiring throughout this thesis. All in all, the pivot concept consists of the pivot itself (*intermediate object hierarchy*) and two sides (see Figure 3.15).

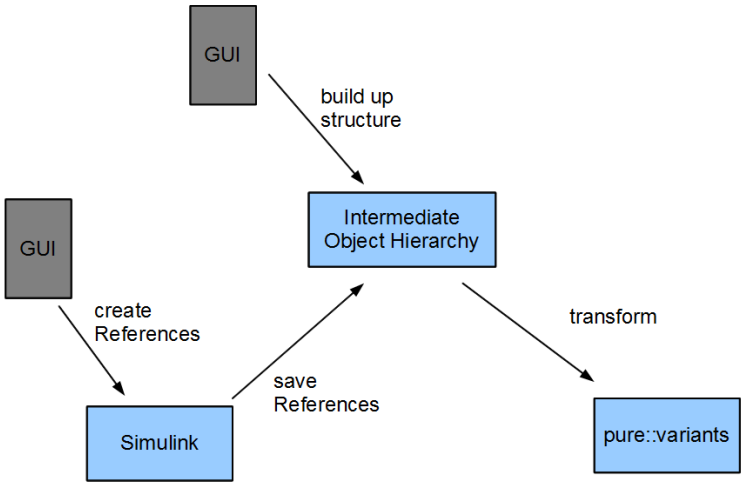

Figure 3.15.: A Concept of a Pivot

The left hand side is Simulink space. With the help of several GUIs it is possible to create and save references to Simulink blocks. After the references have been created, it is possible to build up a hierarchy of objects, forming a tree, which includes the references. The intermediate object hierarchy can be composed and saved with the help of a tree editor.

Finally, it is possible to transform the object hierarchy to a pure::variants a family model. The transformation to pure::variants is carried out by calling the *toCCFM* transformation function on each of the objects in the hierarchy.

3.10. Hooking into Simulink

In fact, hooking into Simulink is not too difficult. It is possible to register a callback with the Simulink context menu. By right-clicking on a Simulink block and selecting the appropriate menu item, the callback function *createModelRef* is called. The callback receives a so-called *callbackInfo* object. This object contains all necessary information about the location and parameters of the selected Simulink block. The idea behind providing a context menu function is to enhance usability.

4. Implementation

4.1. Overview of the Implementation

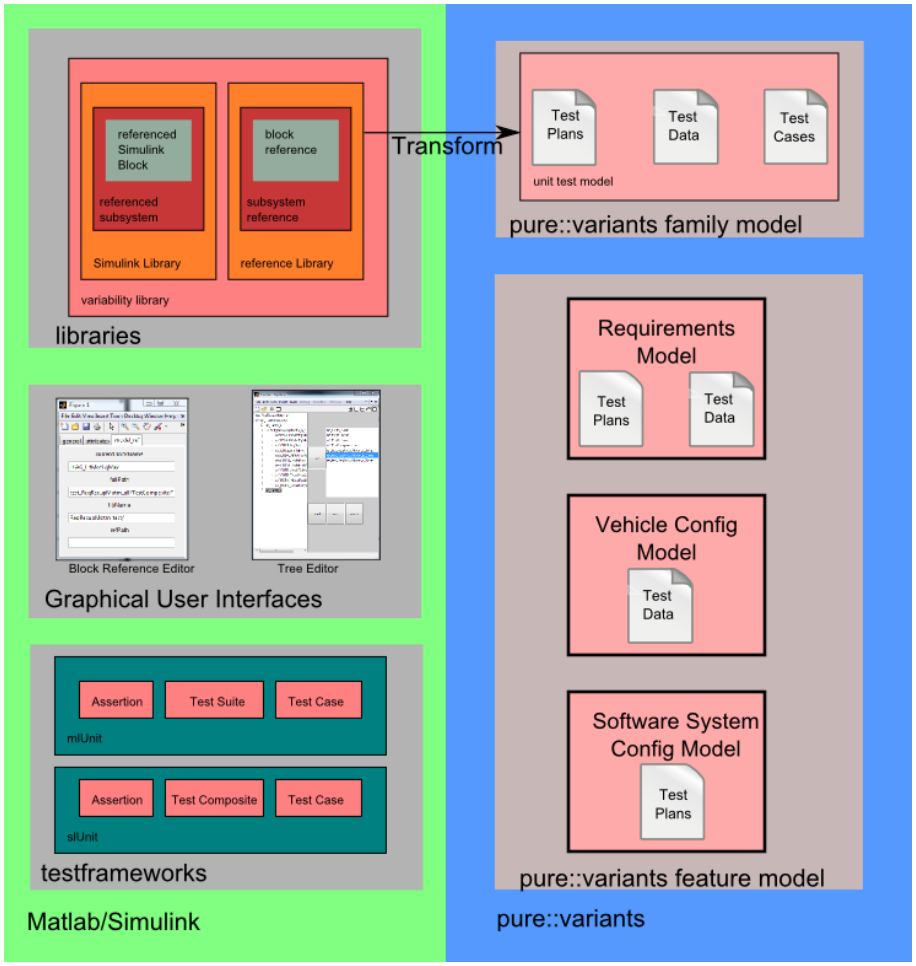

Figure 4.1.: Overview of the Implementation

The implementation (see Figure 4.1) includes two test frameworks:

- mlUnit (see Section 2.2.5)
- slUnit (see Section 2.2.6)

slUnit has three main building blocks in Simulink: *test composite* (see Section 4.2.2), *test case* (see Section 4.2.3) and assertions (see Section 4.2.4).

Test composites are an envelope, that contain several *test cases*. In the original *slUnit* implementation, a *test composites* has a test automation mechanism. This test automation mechanism has been removed, because no test reports in textual form can be retrieved. Instead, the automation of *test cases* inside a *test composite* is controlled by *mlUnit test cases*. So it is possible to create textual test reports. The second adaption was to mark a *test composite* with the type *TestComposite*. Therefore the software can detect that the *test composite* is of the type *TestComposite* when the *test composite* is referenced.

Test cases are part of the *slUnit* implementation, as well. They contain test code. Again, they have been adapted by marking them with the type *TestCase*. Thus, the software can determine that the *test case* is of the type *TestCase*, when it is referenced.

slUnit assertions check whether a condition is fulfilled. Since the *slUnit* assertion has a very poor timing performance, it has been replaced by Simulink assertions.

mlUnit is a Matlab script based framework. It contains three main building blocks: *test suite*, *test case* and *assertion*. *mlUnit* is used for controlling the execution of *slUnit test cases* because *slUnit* is not capable of creating test reports in a textual representation.

A *test suite* is a collection of *test cases* that have to be executed. The *mlUnit test suite* replaces the test automation that was used in *slUnit* before.

A test case is responsible for selecting a *slUnit test case* and starting the simulation. During the simulation, test signals are recorded. After the simulation, a test report is generated.

The libraries are responsible for keeping all information about the test code. Test code is implemented in Simulink libraries and may contain Assertions, TestComposites and TestCases. Moreover, it is possible to create references to Simulink blocks (see Section 4.4) in order to influence them by means of a FODA-Tool. Variable subsystems are partly implemented. In order to keep track of these parts, it is necessary to careful define the structure of libraries. A part of the library, the *subsystem reference* (see Section 4.3.2) can be transformed to a pure::variants family model, the so-called *unit test model* (see Section 3.6.1). The unit test model contains all information necessary to configure tests for one unit under test.

Graphical User Interfaces are a means to enhance Simulink models with variability information. The gathered information is stored in the *reference library*.

The *block reference editor* enables enhancing a Simulink block with variability information and storing this information as a *block reference* in the *reference library*.

The *tree editor* is able to connect all *block references* in the *reference library*. The connection information is then stored as a *subsystem reference* that can be transformed to a pure::variants family model.

4.2. The Test Framework

4.2.1. Embedding Tests in Simulink

In Figure 4.2 the unit testing context is illustrated. The *unit under test* contains the code that should be tested. The *test block* includes all test cases for this unit.

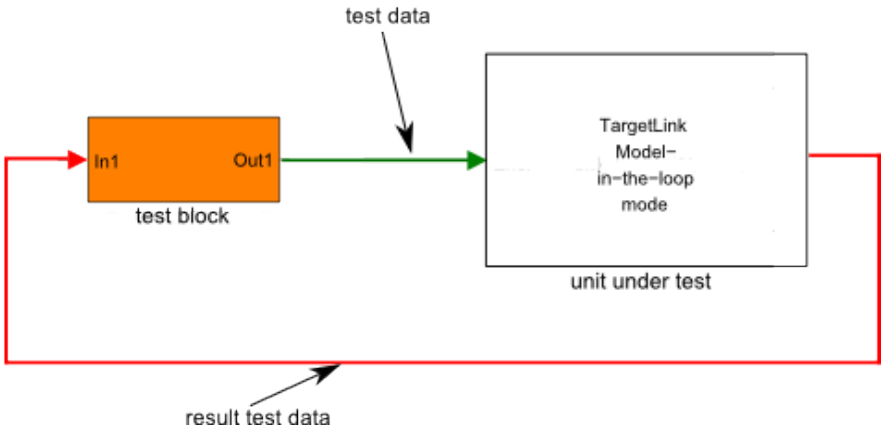

Figure 4.2.: Embedding Tests in Simulink

The signal flow in Simulink follows the alignment of the arrows. Therefore, the test data are generated in the test block and are directed towards the *unit under test*. Test data are represented by green arrows.

The *unit under test* generates output data (as a response to the input data). They are displayed in red color and are feedback to the *test block*, where the resulting test data are evaluated.

The test block is a placeholder; it is empty and filled with a test composite including test cases by executing a Matlab script that is generated by model transformation.

4.2.2. Test Composite

Test Composite in Simulink

In slUnit, a *test composite* is represented by a subsystem that contains *test cases* (see also Section 2.2.6). The original *slUnit* implementation of a *test composite* has been modified in several ways:

First, it does not use the test automation mechanisms that are inherent in *slUnit test composites*. *slUnit test composites* can not create textual test reports an thus the execution of *test cases* is controlled by *mlUnit test cases* in contrast.

Second, a *test composite* is marked as a subsystem of the type *TestComposite*. This is important, when the *test composite* is referenced, because the software has to check somehow that it is a *test composite*. The information is then stored within the *model_ref* class (see Section 4.4.3).

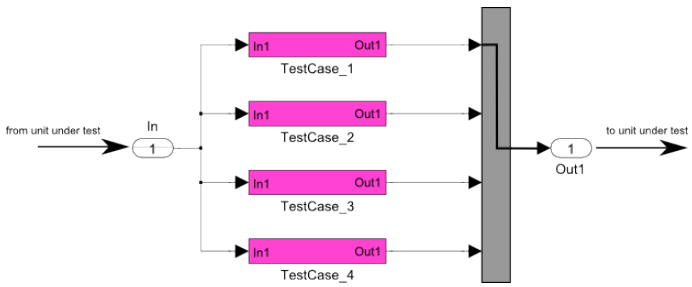

Figure 4.3.: Sample Components of a Test Composite

The *test composite* in Figure 4.3 contains four *test cases*. The output of the *test cases* are forwarded to a *slUnit multiplexer*. The *slUnit multiplexer* connects the output of exactly one *test case* (*TestCase_1*) with the unit under test. The selection of the forwarded signal can either be done manually via a mask or automatically. It is done automatically by executing the *createTest* test method within a *mlUnit test case*.

Test Composite in mlUnit

The *mlUnit* counterpart of a *slUnit test composite* is the *test suite*. Each of the *test cases* that should be executed must be listed in the *mlUnit test suite*. *mlUnit test suites* are automatically generated from a pure::variants feature model configuration.

Test Composite in pure::variants

For the test composite, a Matlab script file named *test_ref_TestComposite.m* is created. Contents of that script are determined by several ps:fragments that are part of a family model, the so-called *unit test model*. The ps:fragments are pieces of text inserted one after another.

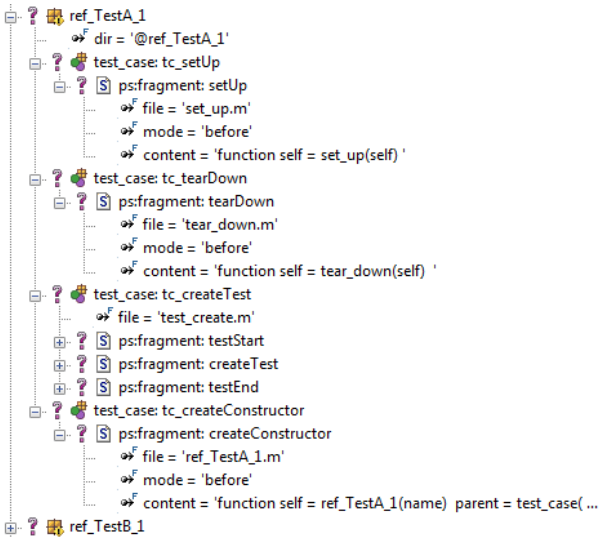

Figure 4.4.: Sample Representation of a Test Composite in pure::variants

The first ps:fragment named *test_ref_TestComposite* contains code for an empty mlUnit test suite. For each test case in this test suite, one line of code has to be generated, in order to add a test case. This is done by means of all the other ps:fragments. Each and every test case fragment has a restriction that is linked to a feature model.

4.2.3. Test Case

Test Case in Simulink

Test cases used in this framework are modified *slUnit test cases* (see Section 2.2.6). They have been modified in two ways.

First, *slUnit test cases* have been modified, so that they do not use the *slUnit* test automation mechanism. The *slUnit* test automation mechanism has been removed, because it can not create test reports in a textual form. Instead, the *slUnit test cases* are controlled by *mlUnit test cases* which provides test reports in a textual form.

Second, the *slUnit test case* has been extended by the type *TestCase*. Therefore, the software can determine its type, when the *test case* is referenced. This information is then stored within the *model_ref class* (see Section 4.4.3).

In Figure 4.3 the *test case TestCase_1* is connected via the *slUnit multiplexer* with the unit under test. Thus, the test data are generated within the *test case* and are directed towards the unit under test. The unit under test generates a response, the test result. The test result is feedback to the *test case*, where it is evaluated.

Test Case in mlUnit

In *mlUnit* a *test case* is a folder with the name of the *test case* itself. The folder contains a constructor with the same name as the *test case*. Additionally, there is a *set_up* and a *tear_down* method. Moreover, one *test method* is created automatically, the method *createTest*. Within this method, the multiplexer (see Figure 4.3) is controlled, in order to connect the corresponding *slUnit test case* with the unit under test. When the simulation is started, all the input and output signals are recorded. A test report is generated. All *slUnit assertions* are checked within the *test method createTest*.

Test Case in pure::variants

The *test case* is reflected in a pure::variants family model (see Figure 4.5). In this example, the ps:component *ref_TestA_1* represents a *test case*. For this *test case*, a directory *@ref_TestA_1* is created. This directory is intended to contain all necessary files for a *mlUnit test case*:

- *set_up.m*
- *tear_down.m*
- *createTest.m*
- *createConstructor.m*

The ps:part *test_case:tc_setUp* is responsible for creating the *set_up.m* method. At the moment, *set_up.m* is an empty skeleton and might be filled manually, if needed. The *set_up* method is called by the *mlUnit* framework before *test methods* are executed. It is part of the fixture pattern (see Section 2.2.5).

The ps:part *test_case:tc_set_tearDown* is responsible for creating the *tear_down.m* method. At the moment, *tear_down.m* is an empty skeleton and might be filled manually, if needed. The *tear_down* method is called by the mlUnit framework after test methods are executed. It is part of the fixture pattern (see Section 2.2.5).

The ps:part *test_case:tc_createTest* is responsible for generating the *test_create.m* method. The file *test_create.m* carries out all steps for creating the test and creating a test report. Because this is not trivial, the file *test_create.m* is composed of three text fragments.

The ps:part *test_case:tc_createConstructor* is responsible for creating the constructor of the *test case*. By convention, the constructor must be named equally than the *test case*.

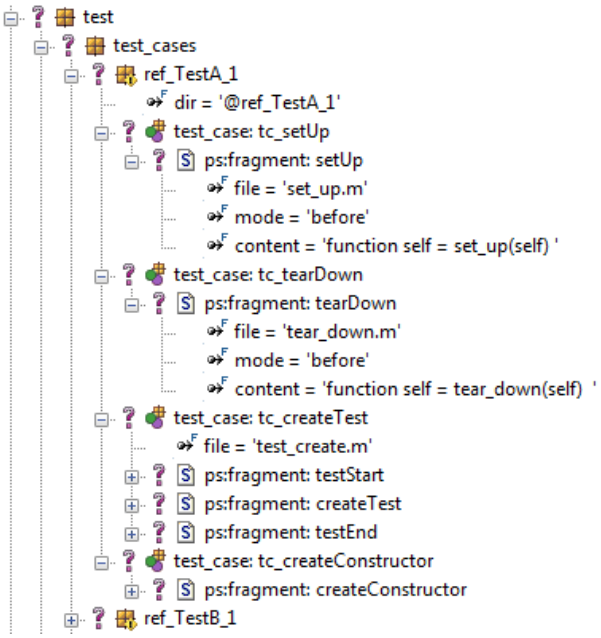

Figure 4.5.: Sample Representation of a Test Case in pure::variants

4.2.4. Assertions

Assertions in Simulink

Assertions are provided by the *slUnit* framework but for the reason of a very poor timing performance, they have been replaced by adapted Simulink assertions. Within such assertion blocks, it is possible to enter a callback that is executed in case of a violated assertion. The callback simply turns the background color of the assertion block from green to red. A *slUnit test case* is started by the *test_create.m* method within the corresponding *mlUnit test case*. Before the execution of the *test case*, the *test_create.m* method assures that all assertion blocks within this *test case* are colored green. During the execution of the *test case*, an assertion block will turn its background color to red, if it is violated. After the execution of the *test case*, the *test_create.m* method checks, if any assertion blocks have changed their color to red. If yes, the *test_create.m* method lists all assertions in a test report (see Figure 4.6).

Assertions in mlUnit

Code in the *create_test* method assures, that all Simulink assertions turn green before the execution of the *test case*. After the *test case* has been executed, all violated assertions will be colored red. Therefore, the *create_test* method checks, if any assertion blocks have changed their color to red. If yes, *mlUnit* assertions are called. *mlUnit* assertions are automatically listed in the test report, if they are violated (see Figure 4.6). It is assured, that the path to the violated Simulink assertion is contained in the test report.

4.2.5. Test Report

A test report is the result of all tests. All violated assertions are listed here. In fact, the test report in Figure 4.6 is the sum of all test reports of all tested units.

```
**********************************************************************
**********************************************************************
Running Tests For SUT: ReqRecupMotrn

----------------------------------------------------------------------
 Ran 20 tests in 51.189s

 FAILED (errors=0, failures=1)

 ======================================================================
 FAIL: ref_TestB_2('test_create')
 ----------------------------------------------------------------------
 Traceback (most recent call first):
   In C:
   In D:
   In C:
   In C:
   In C:
   In C:
   In D:
   In D:
 AssertionError: wlUnitAssertion: sys=ReqRecupMotrn_testsys_assert=ReqRecu



**********************************************************************
**********************************************************************
Running Tests For SUT: ReqRecupSrvBrk

----------------------------------------------------------------------
 Ran 20 tests in 53.245s

 OK
```

Figure 4.6.: Sample Test Report Showing the Test Results of Two Units

4.3. Variability Issues in Simulink

4.3.1. The Variable Subsystem

Variable subsystems enable the possibility for modeling differences in structure. They are needed to implement variable *test case implementations* (see Section 3.7.5). In the current thesis, variable subsystems are implemented as placeholders. They are filled with concrete subsystems and data during model transformation.

Regarding a simple example: a sensory data preprocessing unit. Usually sensor data come in from an analogue-to-digital converter as a number out of a range of values. This number does not correspond to a physical unit and has to be converted mathematically by some kind of calculation. The formula in charge will differ from one kind of a/d converter to another. So, a variable subsystem is desirable to model those differences.

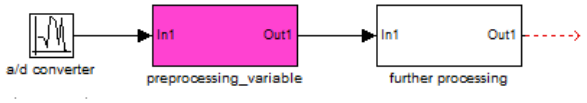

Figure 4.7.: Example of a Subsystem Placeholder in Simulink, colored in magenta

In Figure 4.7 the placeholder is the subsystem block called *preprocessing_variable*. At the beginning it has no contents. Because this block is linked from a library (see Figure 4.8), the link information is accessible and further variability information can be gathered. All variant subsystems of the preprocessing unit must be contained within this library. The term library includes Simulink libraries and additionally saved variability information.

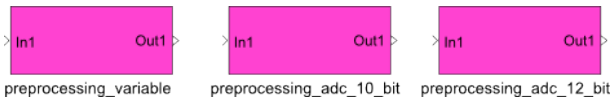

Figure 4.8.: Example Simulink Library Including Alternatives of a Variable Subsystem

4.3.2. Variability within a Subsystem

A subsystem is an encapsulation but despite this, a developer may want to parametrize some items inside it. References provide the possibility for the user to export block specific attributes to pure::variants family models. They are assembled interactively in Simulink with the help of editors. pure::variants family models are capable of representing all kinds of Simulink blocks as well as variable subsystems.

4.3.3. Simulink Organization

Before the library organization is explained in Section 4.3.4, the basic organization of Simulink is outlined now. Simulink is a programming language with graphical notation and consists of the following four main components:

Simulink Models are executable programs within the Simulink interpreter. They contain Simulink Blocks and Simulink Subsystems that are interconnected with signals.

Simulink Libraries are a non-executable collection of Simulink code.

Simulink Blocks represent states (such as Constant Blocks) or operations (such as mathematical operations).

Simulink Subsystems are containers around defined functionality. They may be compared to functions in the functional programming paradigm.

4.3.4. Library Organization

The library organization is responsible for providing *variable subsystems* (see Section 4.3.1) and *variability within a subsystem* (see Section 4.3.2). Both issues are necessary for implementing variability within tests. The library has been developed only for the test system. Because the implementation of the framework is a prototype, the library system is kept as simple as possible. Searching for variants, subsystems and references is error-prone if two items share the same name. Thus, the author of this thesis has avoided setting search paths in the library. Due to this constraints a hierarchical library was considered as too labor-intensive. The library is organized linear which works well for small libraries. In Figure 4.9 the library organization is illustrated in UML. For a better understanding, the library organization is shown additionally in an exploded view in Figure 4.10. In detail the library is organized as follows:

root directory: lib All implementations of tests in Simulink, f.i. TestComposite, TestCases have to be kept in a dedicated directory, because the software has to search and find items within this directory. The location of this directory can be chosen freely, the path to it has to be entered in the configuration script *config_all.m* (see also Section 3.6.4).

variability library The variability library supports variable subsystems that are needed for testing. For each variable subsystem, one variability library has to be created. The information concerning all alternatives of a variable subsystem is kept in this library. A variability library is represented as a folder in the root directory. It contains exactly one Simulink library and for each variant of a subsystem a so-called *reference library*.

Simulink library This is an ordinary Simulink library, it contains all variants of a variable subsystem: so-called *referenced subsystems*. The Simulink library is contained in the variability library folder. It does not contain any information about variability.

referenced subsystem A referenced subsystem is a top level subsystem in the Simulink library. Referenced subsystems may contain referenced Simulink blocks.

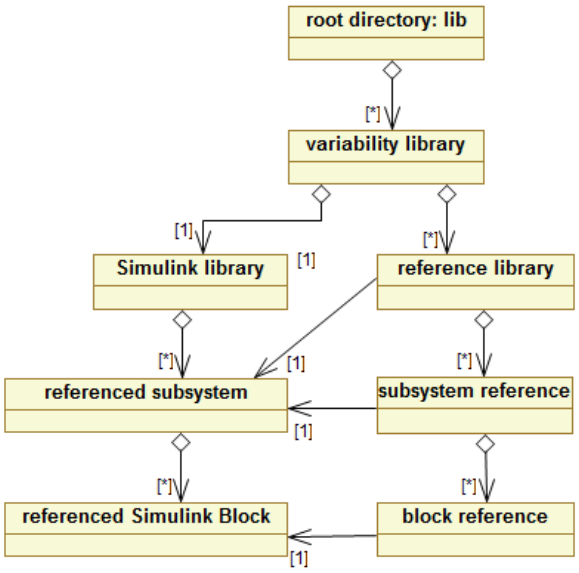

Figure 4.9.: Library Representation in UML

reference library For each referenced subsystem, a reference library has to be created that contains all information about the dedicated referenced subsystem. The reference library is represented as a folder within the variant library folder.

subsystem reference For each referenced subsystem, a subsystem reference has to be created; it connects all the block references created in this subsystem. The subsystem reference is built up with the tree editor depicted in Figure 4.16.

referenced Simulink Block This is an ordinary Simulink block after it has been referenced. They can be referenced with a right mouse-click, and choosing the appropriate item in the context menu. During the process of referencing, several settings can be chosen with the help of an editor (see Figure 4.13).

block reference After the process of referencing a Simulink block, a block reference is saved within the reference library folder.

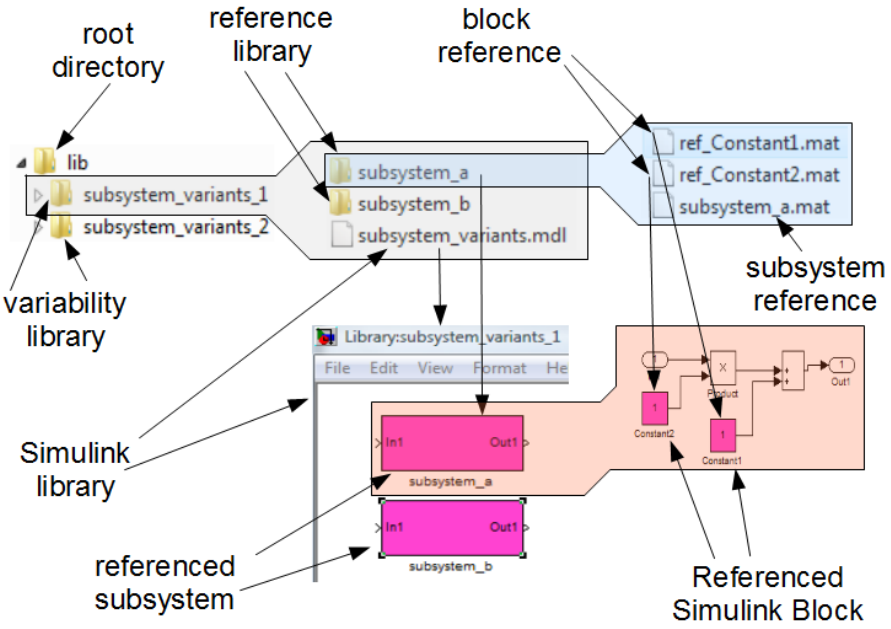

Figure 4.10.: Exploded View of a Sample Variability Library

4.4. Referenced Simulink Blocks

4.4.1. Overview of References

Referenced Simulink blocks provide to a means to export variability information to a pure::variants family model. The corresponding parts and relations are depicted in Figure 4.11. In principle, the UML diagram reflects the concept provided in Section 3.9. The classes in the middle of Figure 4.11 represent the *pivot* and are implemented in Matlab. The pivot itself has two sides: the right hand side consists of Simulink parts. The left hand side forms a pure::variants family model, namely the unit test model (see Section 3.6.1).

Developers will usually start with the definition of tests in Simulink space. Simulink blocks can be enhanced with variability information, if needed: developers can specify which *Simulink Block Properties* can be referenced by a pure::variants family model. It is also possible to create variable subsystems. The process of exporting variable subsystems to a pure::variants family model is not fully automated, at the moment.

The *pv_model* class is just an encapsulation that contains all information that is necessary to enhance a *Simulink Block* with variability information. The variability information is referenced by the classes: *general*, *model_ref* and *attributes* and will be described further in the following.

Simulink blocks are located in a subsystem that is contained in a Simulink library. In principle, it is possible to reference a Simulink block only by its name. But this has

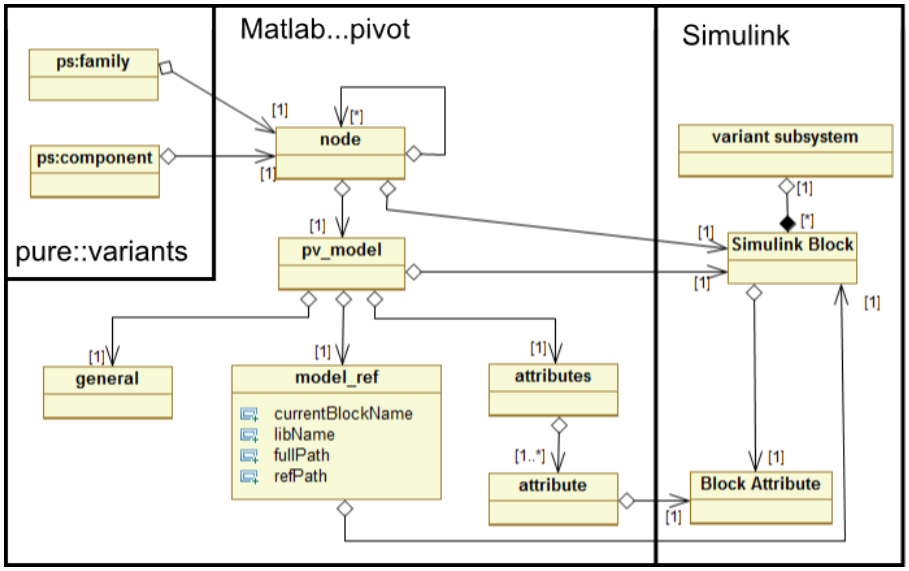

Figure 4.11.: Relation of the *node* class to all the Other Parts in pure::variants and Simulink

a severe drawback: it is not possible that two referenced Simulink blocks can have the same name, even if they are located in different subsystems. So, the following location information has to be extracted:

currentBlockName Name of the referenced Simulink block.

libName Name of the *Simulink library* where the reference is created.

fullPath Path to the referenced Simulink block beginning from the first level subsystem in the Simulink library.

refPath If the referenced Simulink Block is of the type *Subsystem* and is linked from another Simulink library, the path to the original Subsystem is displayed.

This location is needed in the pure::variants family model, because it generates a Matlab script that is capable of modifying the referenced Simulink block. The model_ref class is responsible for referencing this location information.

Each Simulink block has so-called *Simulink Block Properties*. *Simulink Block Properties* may be f.i. the *Value* of a *Constant Block*. The developer has to decide, which of these *Simulink Block Properties* should be exported to the family model. Each *Simulink Block Property* is referenced by the *attribute* class. The *attributes* class is a list of *attributes* in order to enable referencing more than one *Simulink Block Property*. The process of referencing is supported by a GUI (see Section 4.4.4).

The *general* class is responsible for pure::variants parametrization that has no relation to Simulink. A GUI enables the input of the parametrization. These pure::variants parameters are f.i.: *variation_type* or *default_selected*. They are explained in more detail in

Section 4.4.2.

All hierarchical information is represented with the help of the *node* class. The *node* class has a member of a *pv_model* class, that represents all non-hierarchical information. Since the *node* class is capable of representing a hierarchy, it can build up a tree. Moreover, this class implements the ability of composing trees within a GUI. The GUI itself loads node information from references that have been saved in files. Thus, the *node* class must be able to save all its containing information to *.mat* files and to load all it from a *.mat* file. In Matlab, objects can be represented as *.mat* files.

Finally, a tree of *node* objects can be transformed to a pure::variants family model. The root *node* is then of the type *ps:family* (see Section 3.6.1), all the other *nodes* are of the type *ps:component*.

4.4.2. The general Class

The pure::variants representation of the *general* class is highlighted in fuchsia in Figure 4.12. The ps:component named *DIAG_EltMotTq* has a name and a variation type. They correspond to the *unique_name* and *variation_type* member variables in the *general* class. Additionally, but not visible in Figure 4.12, the pure::variants *default_selected* property is aligned to the *default_selected* field of the *general* class.

In Figure 4.12 the creation of a block reference is illustrated: with the *general* tab selected, it is possible to select the *variation_type* of the reference which could be: *and*, *or*, *andor*. The mapping between *variation_type* and *pure::variants variation_type* is listed in Table 4.1.

variation_type	*pure::variants variation_type*
and	*ps:mandatory*
or	*ps:optional*
andor	*ps:alternative*

Table 4.1.: Mapping Between *variation_type* and *pure::variants variation_type*

Also, the *default_selected* property can be chosen by the user. The *unique_id* of the pure::variants counterpart, namely the ps:component *DIAG_EltMotTq*, is displayed. Even if it is not displayed in the pure::variants part of Figure 4.12, a *unique_id* is needed by each and every ps:component. It is calculated automatically but may be changed within the editor.

The *unique_name* property is determined automatically by the Simulink block name and is equal to the *currentBlockName* in the *model_ref* class. It may be changed within this editor as well. Thus, more than one reference to a single block can be created. For example, if a variable subsystem has two variants, for each of the variants a reference with a *different* name can be created. This is needed to enable variable *test case implementations* (see Section 3.7.4). At the moment the whole process of variable subsystems is not fully implemented.

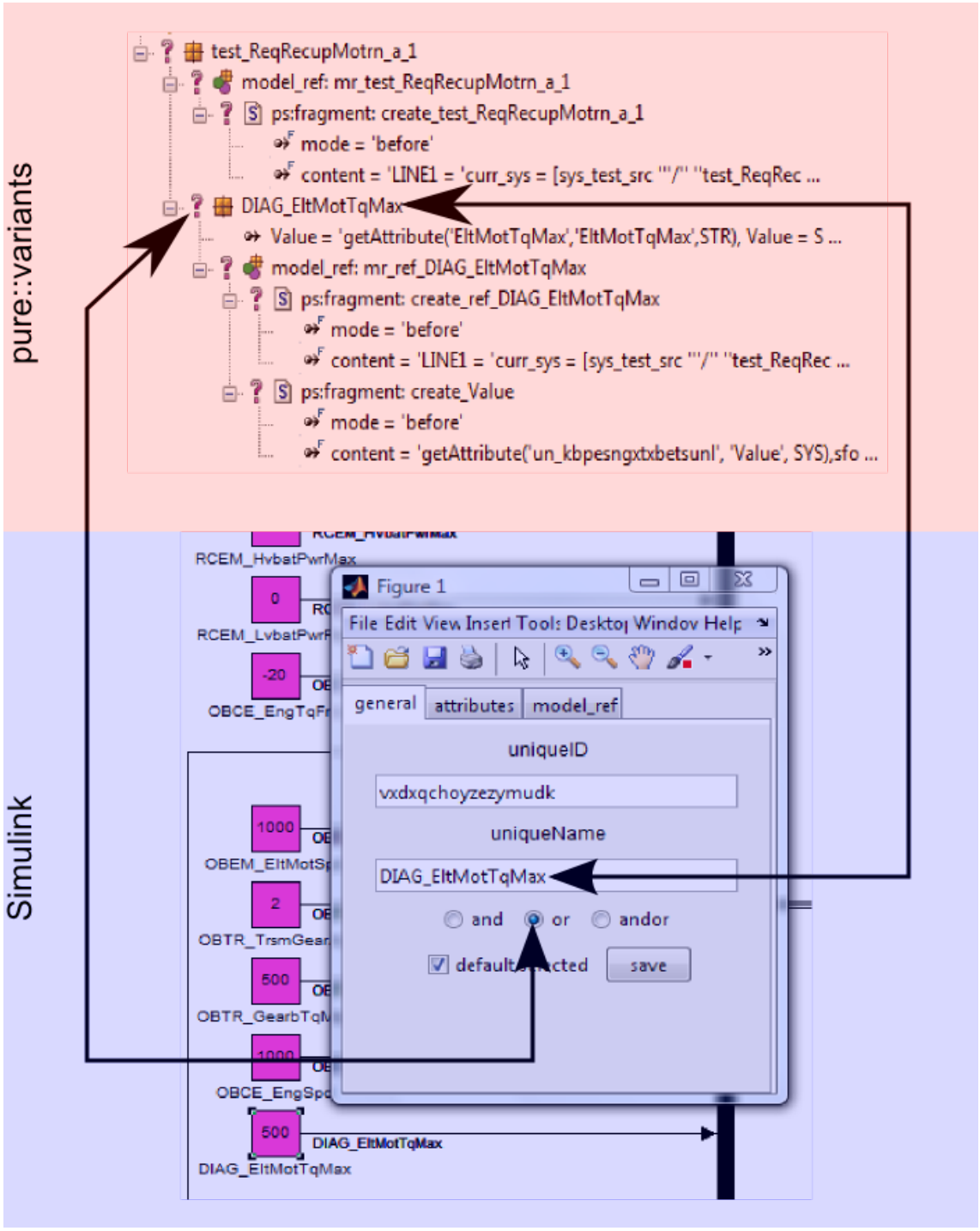

Figure 4.12.: Creating a Reference in Simulink With the *general* Tab Selected

4.4.3. The model_ref Class

In Figure 4.13 the rectangle shaded in fuchsia embraces the pure::variants representation of the *model_ref* class. In the example it consists of the ps:fragment (outlined by the rectangle) named *create_ref_DIAG_Elt_Mot_Tq_Max*. Within this fragment a piece of code is implemented that enables the composition of Simulink models. The path to the referenced block is assigned to the Matlab variable *curr_sys*. If the referenced block is a linked subsystem, this piece of code copies the subsystem from the library to the placeholder and breaks the link to the library. Otherwise, it is a referenced Simulink block and must not be linked from a Library. In this case the code neither copies anything nor breaks links to a library.

In Figure 4.13 the cyan shaded part displays the GUI used to create a block reference to the Simulink Constant Block *DIAG_Elt_Mot_Tq_Max* with the *model_ref* tab selected. The *model_ref* tab consists of 4 text fields that contain the four different path information necessary:

currentBlockName Name of the referenced Simulink block.

libName Name of the *Simulink library* where the reference is created.

fullPath Path to the referenced Simulink block beginning from the first level subsystem in the Simulink library.

refPath If the referenced Simulink Block is of the type *Subsystem* and is linked from another Simulink library, the path to the original Subsystem is displayed. This is a kind of a pointer.

Those four parameters are calculated automatically from the Simulink model and do not have to be entered in the GUI.

Additionally, three parameters not depicted in the GUI are calculated in Simulink:

is_subsystem This value can be determined by the Simulink *BlockType* parameter. If this parameter has the value *Subsystem*, *is_subsystem* is set to 1, otherwise 0.

is_test_case TestCase blocks are subsystems, too. The only difference between subsystems and test case blocks is the Simulink Block Parameter *UserData*. If the text within this parameter is *TestCase*, *is_test_case* is set to 1, otherwise 0. To ensure, that the *UserData* field contains the appropriate content, the block must be inserted from the slUnit library.

is_test_suite *is_test_suite* is set to 1 if the *UserData* field contains *TestComposite*. The test composite block has to be inserted from the slUnit library.

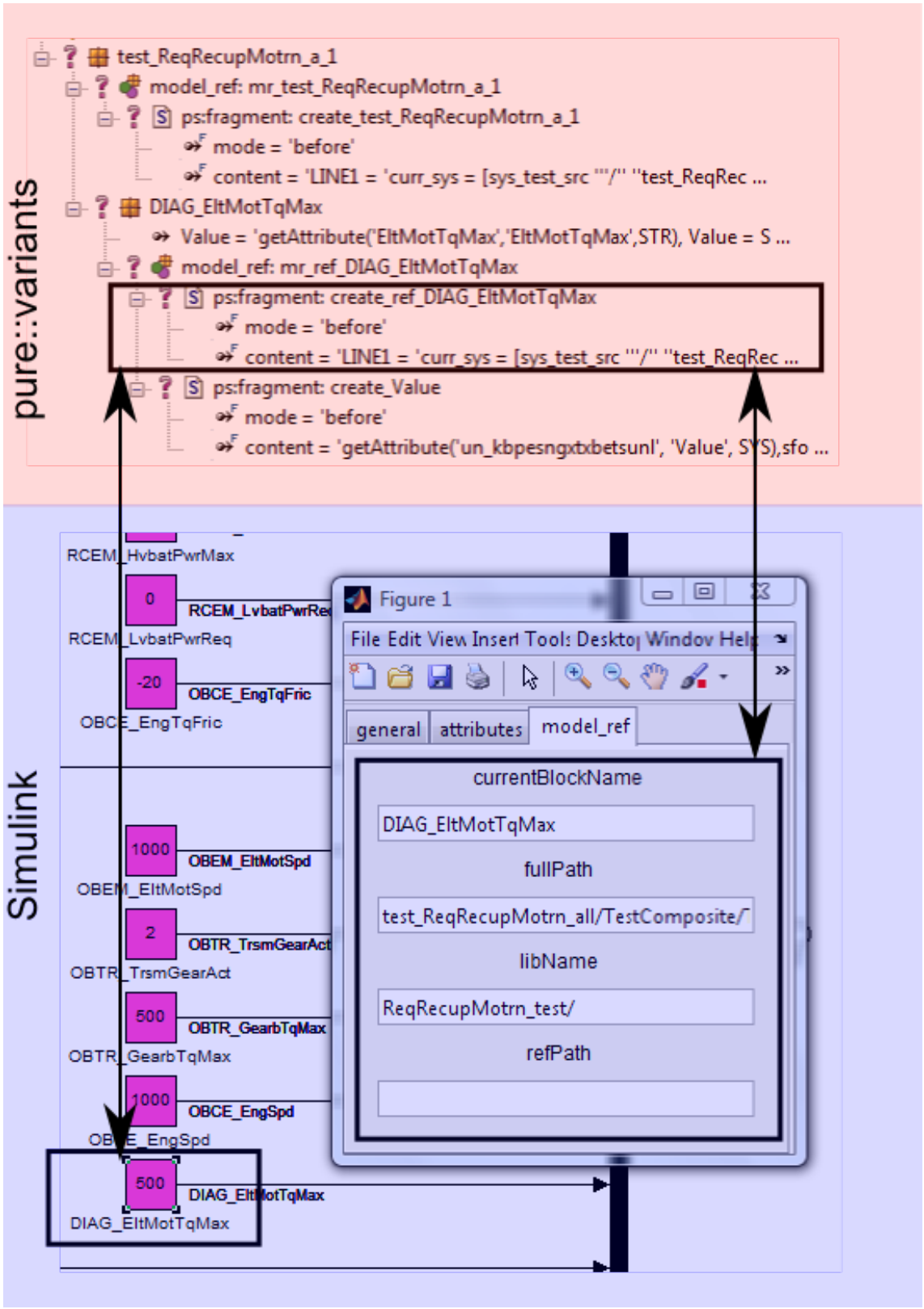

Figure 4.13.: Creating a Reference in Simulink With the *model_ref* Tab Selected

4.4.4. The attributes and attribute Classes

The pure::variants part of the *attribute* and *attributes* classes is colored fuchsia in Figure 4.14. For each *attribute* one ps:fragment is created, which contains Matlab code that sets the *attribute* of the corresponding Simulink block to the desired value. In this example, the attribute is named *Value*. Therefore, the name of the ps:fragment is *create_Value*

In Simulink all blocks have so-called *Block Parameters*. In the Simulink part of Figure 4.14 within the Simulink Constant Block *DIAG_EltMotTqMax* the number *500* is shown. This *Simulink Block Parameter* is called *Value*. Thus, the *attribute name* is called *Value*. The number *500* itself is the corresponding *Simulink Value* of the *Simulink Block Parameter Value*. Thus, the *attribute value* is *500*. For a better understanding, the relation between *Simulink Block Parameters* and the *attribute* class is depicted in Table 4.2 for this example.

Simulink Block Parameter	*attribute name*
Value	Value
Simulink Value	*attribute value*
500	500

Table 4.2.: Mapping Between *Simulink Block Parameter* and *attribute name*

It can be chosen, whether the attribute is *fixed* or not. Additionally the type of the attribute can be selected either as *integer* or as *string*.

Simulink blocks may have more than one attribute. For example, the *Simulink Pulse Generator Block* has the following parameters: *amplitude*, *period*, *pulse with* and *phase delay*. Thus, there is a need for editing more than one attribute. The solution is a list of attributes, called the *attributes* class. Within the editor it is possible to create, delete and edit attributes.

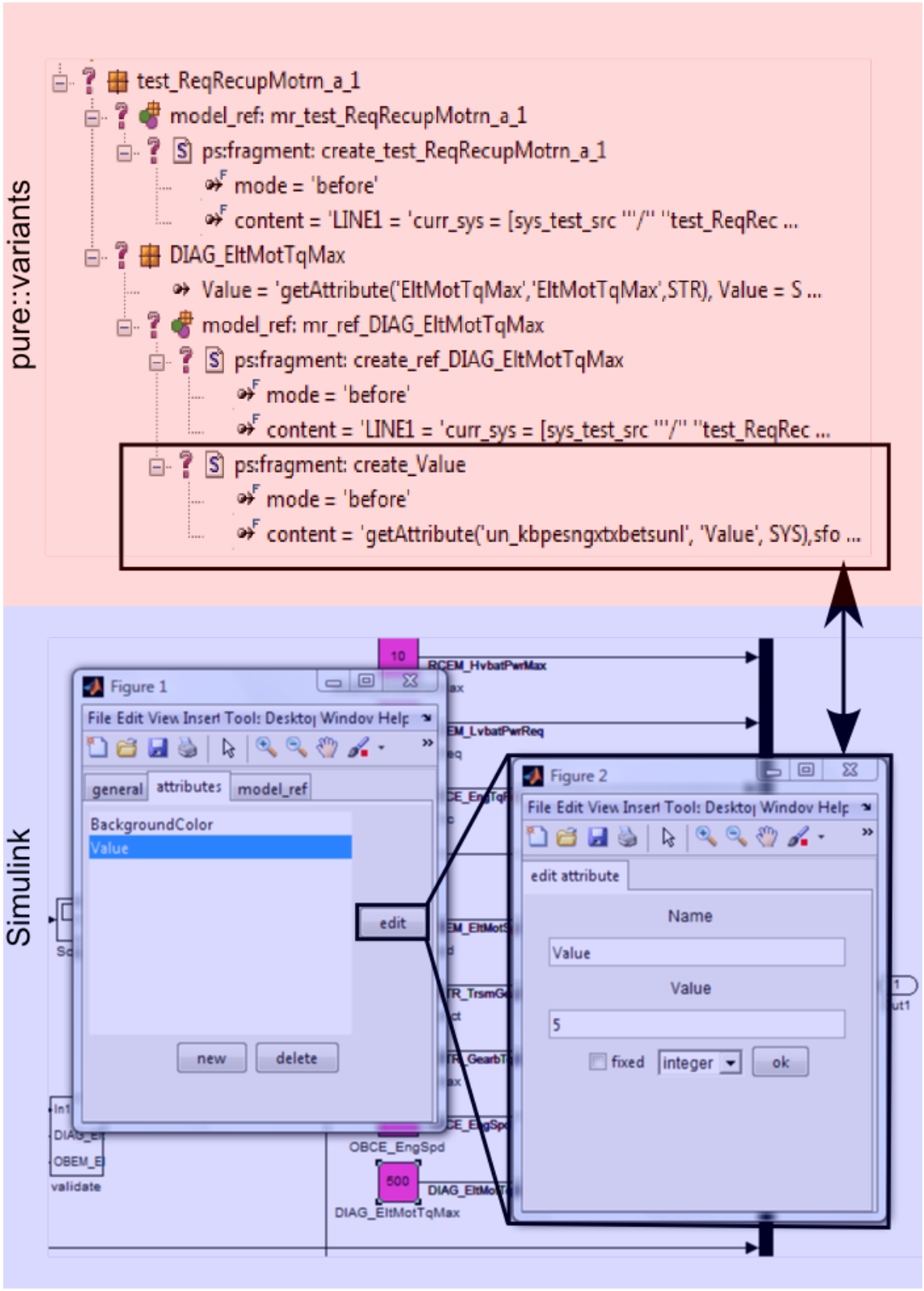

Figure 4.14.: Creating Attributes in Simulink

4.5. Building up the Skeleton With Containers: The node class

In Simulink, subsystems can contain subsystems itself. So, subsystems can build up a tree of hierarchy (see Figure 4.15). Each of the subsystems in Figure 4.15 is a *referenced subsystem*. It is desirable to keep the structure of hierarchy in the pure::variants family model.

All ps:components in Figure 4.15 represent their Simulink *referenced subsystem* counterparts. The arrows indicate the connection between parts in Simulink and pure::variants.

Because *referenced subsystems* can build up hierarchies, one Matlab class must be capable of containing itself: the *node* class (see Figure 4.11). In fact, the *pv_model* class contains all information of the location (*currentBlockName*, *fullPath*, *libName*, *refPath*) and parameters of referenced Simulink blocks or referenced subsystems. The *node* class is just a wrapper around the *pv_model* class to enable a hierarchical tree structure of all referenced parts. All *subsystem references* and *block references* are in fact node objects. In Matlab, it is possible to save objects to *.mat* files and load objects from *.mat* files.

In Figure 4.16 a tree editor for manually composing the hierarchy of *nodes* is depicted. On the left hand side of the editor, the current tree of *nodes* is displayed. On the right hand side, all *block references* and *subsystem references* are listed that are part of the *test_ReqRecupMotrn_all* reference library:

- *ref_TestA_1*
- *ref_TestB_1*
- *ref_TestC_1*
- *ref_TestComposite*
- *ref_test_ReqRecupMotrn_a1*
- *ref_test_ReqRecupMotrn_b1*
- *ref_test_ReqRecupMotrn_c1*

Inserting a new *node* in the tree structure is very simple - in Figure 4.16 in each of the windows, a *node* is selected: *ref_TestB_1* in the left hand side window,
ref_test_ReqRecupMotrn_b_1 on the right hand side. By clicking the ⟨⟨ button the selected *node* on the right hand side is inserted as a child of the selected *node* on the left hand side. If the inserted *node* represents a subtree, the whole subtree is inserted and displayed unfolded in the left window. When composed, the tree can be saved as a *subsystem reference*. In the case of the aforementioned example, the name of the *subsystem reference* is *test_ReqRecupMotrn_all.mat*, which equals the name of the root *node* of the tree. Since this *subsystem reference* is on top of the hierarchy, it can be transformed into a pure::variants family model. In fact, all *subsystem references* can be transformed to pure::variants, but not all of the resulting family models make sense; thus the responsible test architect has to take care.

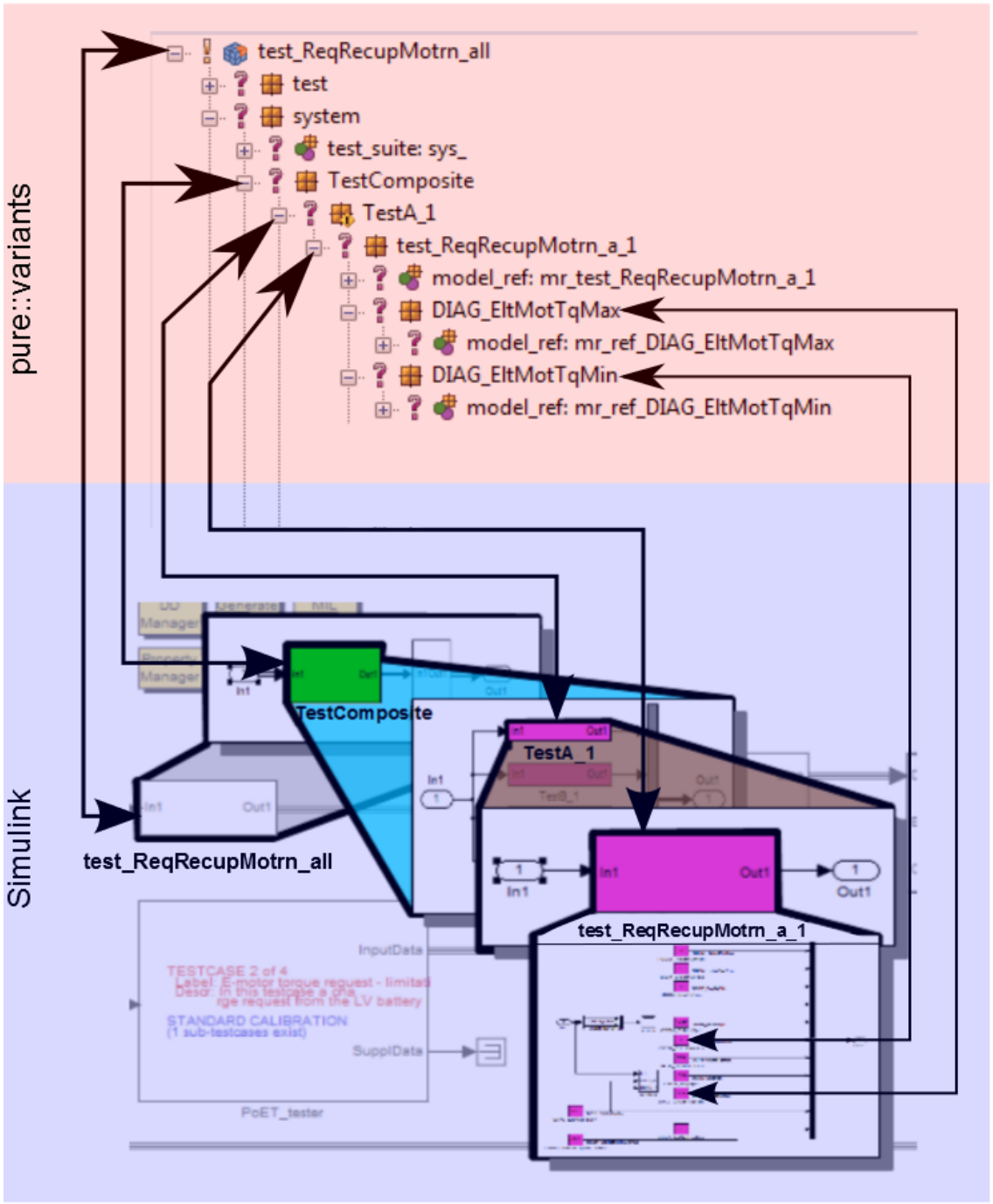

Figure 4.15.: *Node* Hierarchy in Simulink and Their Counterpart in pure::variants

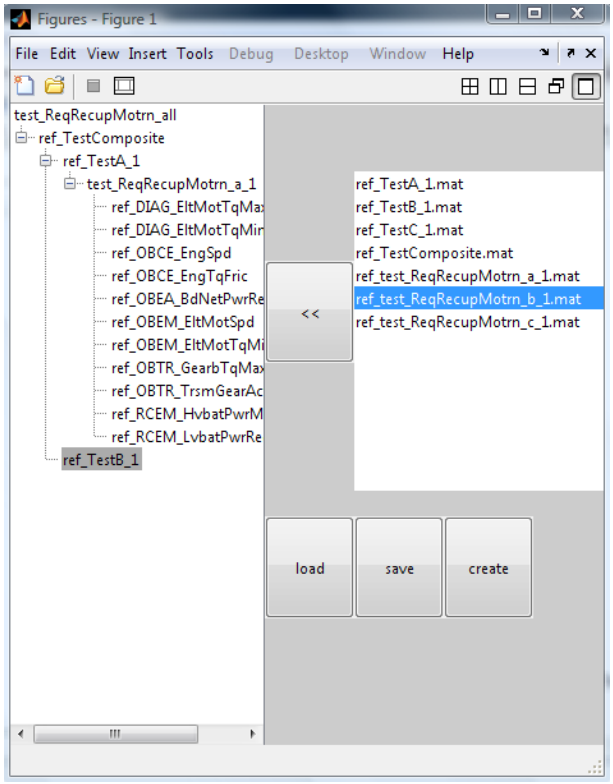

Figure 4.16.: Creating a Tree With *node* Objects in the Tree Editor

4.6. Create a Variability Library

The process of creating a variability library is rather complex. The flowchart in Figure 4.17 shows all necessary steps. For a better understanding all steps numbered in Figure 4.17 are explained in more detail in the following.

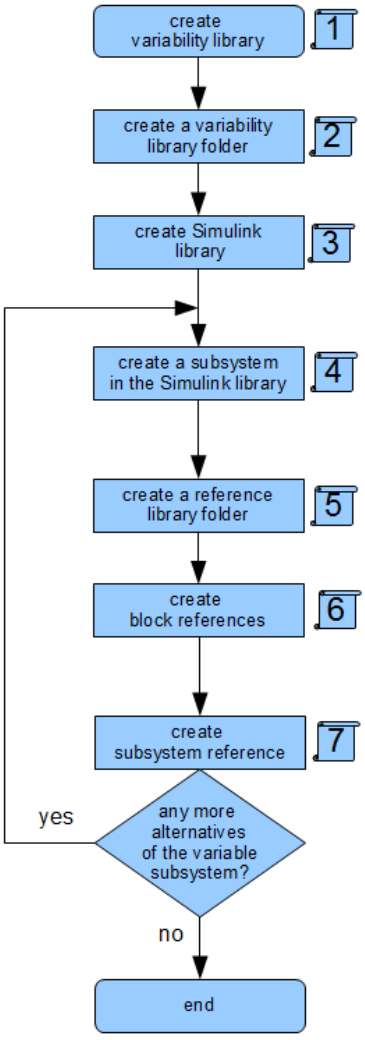

Figure 4.17.: Basic Steps When Creating a Variability Library

4.6.1. Step 1: Create a Variability Library

In the first step, developers have to decide which of the subsystems should be variable. In fact, this step is the starting point of the process.

4.6.2. Step 2: Create a Variability Library Folder

In step 2 a variability library folder is created within the root directory. In this example, the root directory is called *lib* and the variability library folder *subsystem_variants_1* (see Figure 4.18). All information concerning the variability library is stored within this folder.

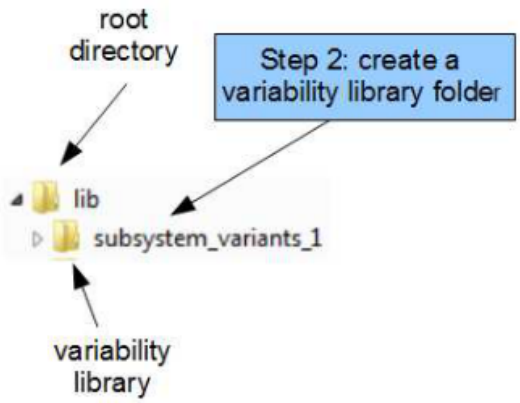

Figure 4.18.: Step 2: Create a Variability Library Folder

4.6.3. Step 3: Create a Simulink Library

In step 3, an empty Simulink library is created that will contain all subsystem variants. In Figure 4.19 it is named *subsystem_variants.mdl*. It must be stored in the variability library folder.

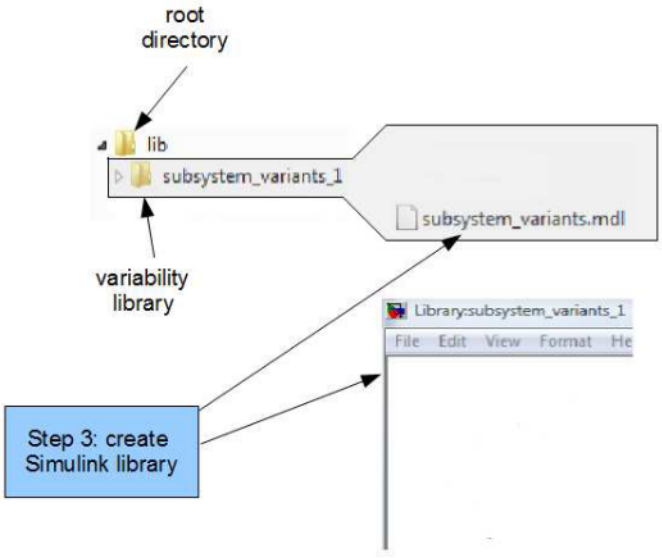

Figure 4.19.: Step 3: Create a Simulink Library

4.6.4. Step 4: Create a Subsystem in the Simulink Library

In this step, a subsystem has to be created with the desired functionality (see Figure 4.20). This subsystem is intended to have subsystem alternatives. Thus, the process repeats at step 4 for each alternative subsystem, which is indicated by the loop in the flowchart in Figure 4.17.

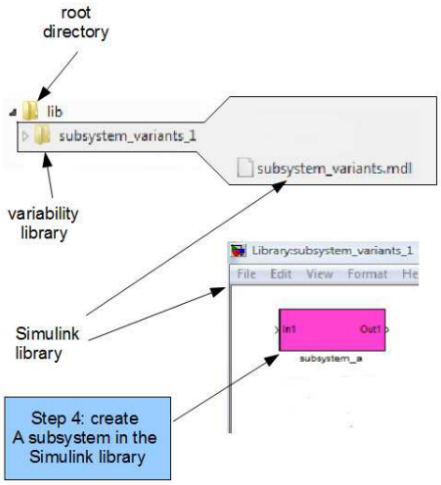

Figure 4.20.: Step 4: Create a subsystem in the Simulink Library

4.6.5. Step 5: Create a Reference Library Folder

To each variant subsystem defined in step 4, a reference library folder is assigned. This folder has the same name as the assigned subsystem and must be contained within the variability library folder (see Figure 4.21). All references to Simulink blocks that belong to the corresponding subsystem will be stored in this folder.

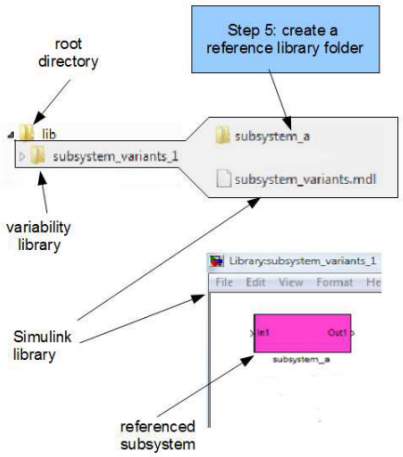

Figure 4.21.: Step 5: Create a Reference Library Folder

4.6.6. Step 6: Create Block References

In step 6, all block references are created (see Figure 4.22). The developer has to right-click the desired Simulink block, first(1). By selecting the appropriate context-menu item the GUI pops up(2). Within the GUI some parametrization is necessary. After that, the block reference is stored automatically(3).

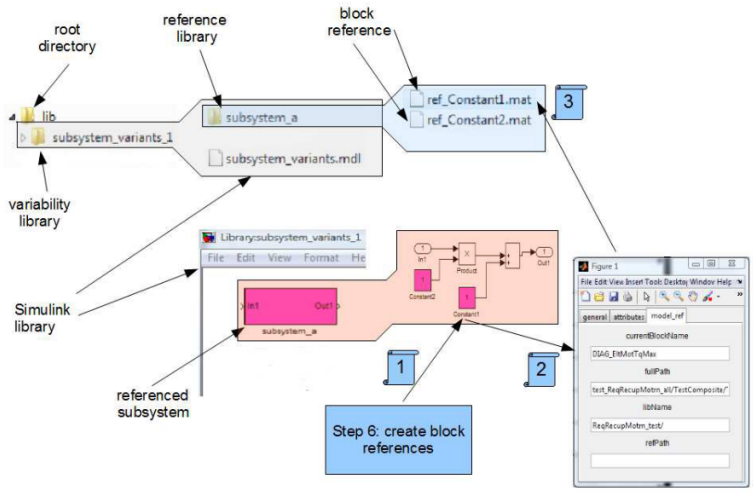

Figure 4.22.: Step 6: Create Block References

4.6.7. Step 7: Create Subsystem References

Subsystem references define the relationships between all block references within a subsystem (see Figure 4.23).

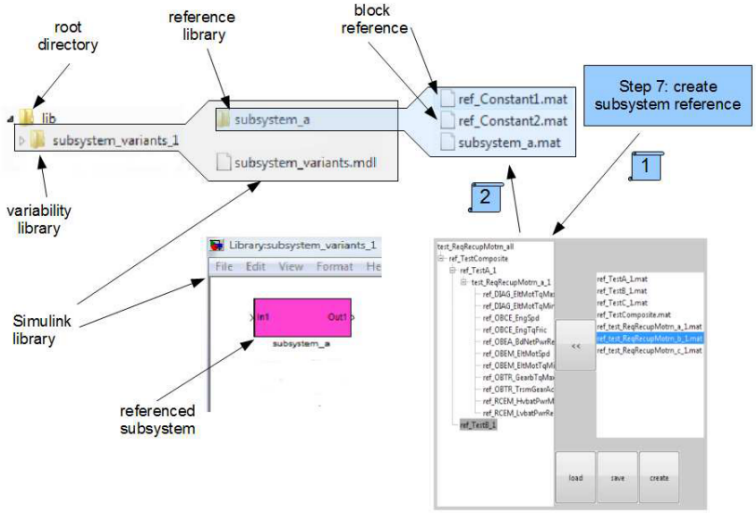

Figure 4.23.: Step 7: Create Subsystem References

First, the developer has to start the tree editor(1). With the help of the tree editor, the developer interactively defines the relations between the block references. This process is described in more detail in Section 4.5. Second, the gathered informations have to be saved as a subsystem reference within the reference library folder(2).

If there are any alternative subsystems left, the process repeats at step 4. If not, the variability library is fully defined. At this point, the variability library can be transformed to a pure::variants family model by invoking the *toCCFM* method on the subsystem reference.

5. Case Study

5.1. System Description

In the *HybConsProject*, a generic automotive control software is under development. To test and demonstrate the developed test methodology and the corresponding tool support, a small sample project (see Figure 5.1) has been provided. To keep the case study simple and small enough, the author has focused on a few components out of this sample project for a better understanding. These components will be described shortly.

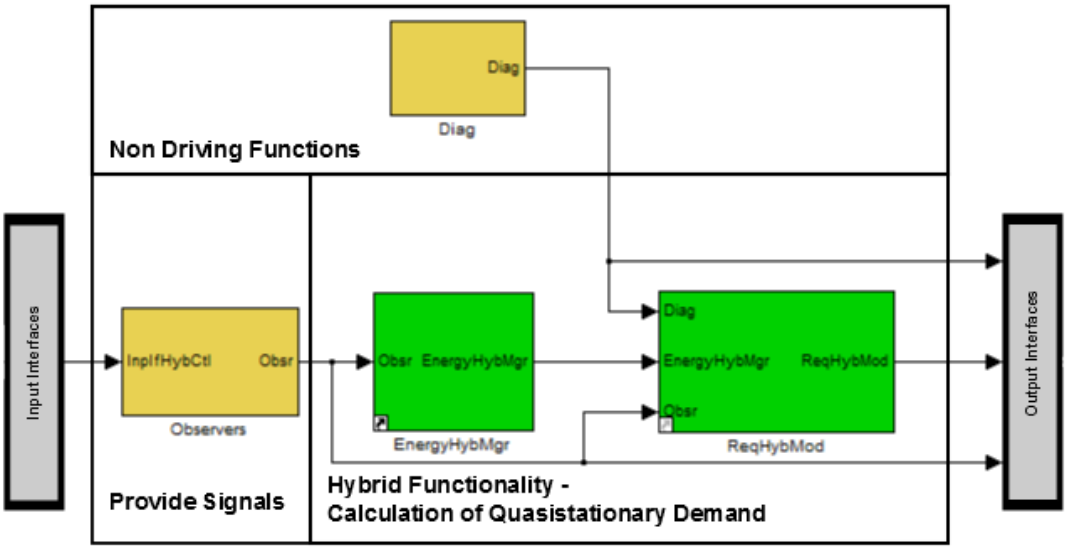

Figure 5.1.: Overview of the Sample Project Software Architecture

5.1.1. Overview of the Sample Project Software Architecture

The system consists of two main modules, that will be discusses below.

Provide Signals

This module is providing standardized signals for the next module (calculation of quasistationary demands). The input data are sensory signals. Therefore, this module encompasses signal preprocessing and normalization of signals [Kor10].

Calculation of Quasistationary Demand

In this module, there are two main parts. The *ReqHybMod* (see [Kor10]) module calculates demands for parameters (e.g. electric motor torque) for each of the hybrid modes. This module will be described in detail, later on. The *EnergyHybMgr* (see [Kor10]) calculates the available energy for each hybrid operation mode.

5.1.2. Request Hybrid Modes

Since the module *ReqHybMod* will be used as the system under test, it will be described in more detail here. Within this module several functional variants calculate the quasistationary demands (requests) for the dedicated generic hybrid modes. For the two functional variants chosen for case study, the corresponding generic hybrid modes will be outlined shortly.

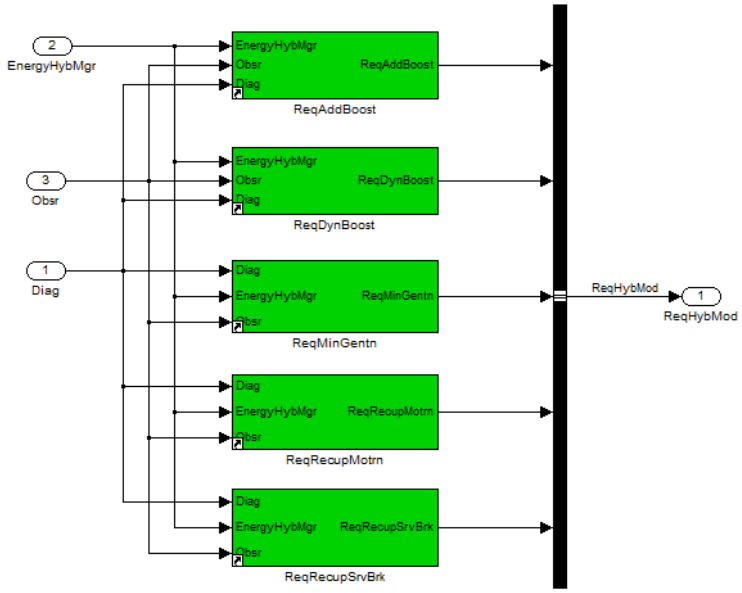

Figure 5.2.: Functional Variants of Request Generation for Hybrid Modes

Hybrid Mode: Recuperation at Engine Motoring

After [Kor10], the generic hybrid mode *Recuperation at Engine Motoring* is defined as follows:

"Recuperation means to convert the kinetic energy of the vehicle into electric power by operating the E-Motor in generator mode at vehicle braking events. The generated energy is used to supply the electrical devices on board and to charge the energy storage system.

The mode Recuperation at motoring is activated when driver demands vehicle deceleration by engine braking (usually when both acceleration and brake pedals are not activated)

desired clutch torque is negative. In conventional vehicles, the engine is set to motoring fuel cut-off operation mode. In this situation, some amount of negative E-Motor torque can be added to the drive line thus increasing the braking effect. The kinetic energy can more efficiently be recuperated and drivability improved, if combustion engine can be decoupled from the drive line, and the E-Motor torque simulates the total engine drag torque."

Hybrid Mode: Recuperation at Service Brake

After [Kor10], the generic hybrid mode *Recuperation at Service Brake* is defined as follows:

"Recuperation means to convert the kinetic energy of the vehicle into electric power by operating the E-Motor in generator mode at vehicle braking events. The generated energy is used to supply the electrical devices on board and to charge the energy storage system.

When the service brake is activated, some amount of negative E-Motor torque can be added to the drive line, thus increasing the braking effect. In vehicles with brake-by-wire systems, the negative E-Motor torque can to some extent substitute the effect of service brakes."

5.2. Problem Space of the Case Study

The units under test are:

- ReqRecupMotrn
- ReqRecupSrvBrk

Which of the units are tested is defined by the *software system config model*. The short name of it is *swsys_config* (see Figure 5.3). The user can select here, which units should be tested.

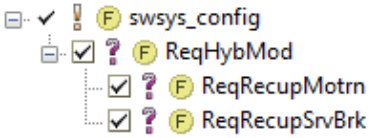

Figure 5.3.: The User Can Select the Units Under Test in the *software system config variant description model*

The *vehicle config model* defines vehicle-specific data, that change from one vehicle to another. These data are listed in Table 5.1.

signal	*vehicle 1*	*vehicle 2*	*unit*
OBVM_VehV	0-300	0-120	km/h
OBEA_BdNetPwrReq	0-2	0-5	kW
OBLG_HvbatPwrDchaMaxPeak	20	50	kW
OBLG_HvbatPwrChaMaxPeak	20	50	kW
OBSN_HvbatSocNorm	0-100	0-100	percent
OBDA_DrvrTqDmd	0-350	0-2500	Nm
OBDA_CluTqReq	0-350	0-2500	Nm
OBEM_EltMotSpd	0-7000	0-2500	rpm
OBCE_EngSpd	0-7000	0-2500	rpm
OBVM_VehA	-10-10	-5-5	m/s2
OBTR_TrsmGearAct	-1-8	-5-21	-
OBCE_EngTqMax	350	2500	Nm
OBCE_EngTqFric	- 30-0	- 60-0	Nm
OBTR_GearbTqMax	500	3000	Nm
DIAG_EltMotTqMin	200	350	Nm
DIAG_EltMotTqMax	200	350	Nm

Table 5.1.: Varying Values for two Different Vehicles

The user can specify the data in the so-called *vehicle config model*. A snapshot of this model is illustrated in Figure 5.4. The signal *OBVM_Veh_V* has a lower and an upper value (see Table 5.1). In Figure 5.4 the lower value is represented by *OBVM_Veh_V_L*. The upper value is represented by *OBVM_Veh_V_H*.

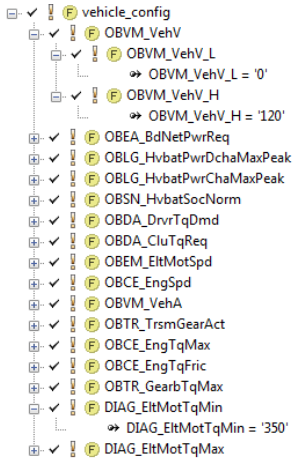

Figure 5.4.: The *vehicle config model* is represented in a pure::variants Feature Model

Each unit under test is tested with 20 different test cases. The same test cases are used for both units. Figure 5.5 illustrates a variant description model that contains all four *requirements models*. Each *requirements model* is linked with four *test cases*. The four test cases linked with one *requirements model* differ only by the signal *OBTR_TrsmGearAct*. The mapping between *requirements models*, *test cases* and *OBTR_TrsmGearAct* is listed in Table 5.2. *N* is the number of positive gears, and corresponds to the upper value of *OBTR_TrsmGearAct* in Table 5.1.

Test Case	*Requirement_1*	*Requirement_2*	*Requirement_3*	*Requirement_4*	*OBTR_TrsmGearAct*
TestA_1	x				1
TestB_1		x			1
TestC_1			x		1
TestD_1				x	1
TestA_2	x				2
TestB_2		x			2
TestC_2			x		2
TestD_2				x	2
TestA_3	x				3
TestB_3		x			3
TestC_3			x		3
TestD_3				x	3
TestA_4	x				N - 1
TestB_4		x			N - 1
TestC_4			x		N - 1
TestD_4				x	N - 1
TestA_5	x				N
TestB_5		x			N
TestC_5			x		N
TestD_5				x	N

Table 5.2.: Mapping Between Test Cases, Requirements and *OBTR_TrsmGearAct*, N is the Number of Positive Gears

The tester is supposed to type in test data in the requirements model. How a requirements model can be derived is explained in Section 5.3.

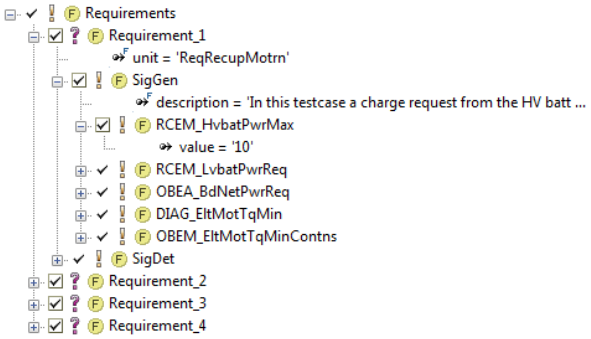

Figure 5.5.: The Requirements Model is Configured in a pure::variants Variant Description Model

5.3. Deriving a Requirements Model by Example

The derivation of a requirements model is shown on the example of one test case: *E-motor torque request - limitation by motoring torque.*

- *Name:* E-motor torque request - limitation by motoring torque
- *Description:* In this testcase a charge request from the LV battery is simulated. The charge power request (*RCEM_LvbatPwrReq*) is set to a value. The other two power requests (*RCEM_HvbatPwrMax* and *OBEA_BdNetPwrReq*) are set to zero. All inputs which could limitate the requested E-motor torque at the output are configured in a way that no limitation takes place (*DIAG_EltMotTqMin*, *OBEM_EltMotTqMinContns*).
- *Verdict Description:* This testcase passes if the requested E-motor torque at the output (*RMRE_EltMotTqReq*) is at the given -50Nm what is given by the maximum applyable torque in between the other limitations.
- *Evaluation Criterion:*

```
if(resultvector('RMRE EltMotTqReq') == -50)
    setPassed(true);
end
```

To come to a semi-formal requirements model one has to remove redundancy from the description of the test case. It is done andectodal here, as an example how it can be done.

```
'In this testcase a...'
```

This holds for each and every test case description and can be left away.

```
'. charge request from the LV battery is simulated.'
```

This is a useful information but a test engineer won't be interested in. It should be left in the description. Leaving away those parts, leads to version 1.1 of the new requirements model:

```
The charge power request (RM_LvbatPCEwrReq) is set to a value.
The other 2 power requests (RCEM_HvbatPwrMax and OBEA_BdNetPwrReq)
are set to zero. All inputs which could limitate the requested
E-motor torque at the output are configured in a way that no
limitation takes place (DIAG_EltMotTqMin, OBEM_EltMotTqMinContns).
```

Signals don't have to be explained explicitely:

```
'The charge power request (RCEM_LvbatPwrReq) is set to a value.'
```

Only the signal name should be used here. This leads to no loss of information. Version 1.2 states:

```
RCEM_LvbatPwrReq is set to a value
RCEM_HvbatPwrMax and OBEA_BdNetPwrReq are set to zero.
DIAG_EltMotTqMin, OBEM_EltMotTqMinContns are configured
 in a way that RMRE_EltMotTqReqs is not limited.
```

Signals should not be conjuncted with an *and*:

```
RCEM_HvbatPwrMax and OBEA_BdNetPwrReq are set to zero.
```

It it better practice to mention each of them explicitely. The result is version 1.3:

```
RCEM_LvbatPwrReq is set to a value
RCEM_HvbatPwrMax is set to zero.
OBEA_BdNetPwrReq ist set to zero.
DIAG_EltMotTqMin is configured in a way that RMRE_EltMotTqReqs
is not limited
OBEM_EltMotTqMinContns isc configured in a way that RMRE_EltMotTqReqs
 is not limited
```

This version is quite readable at the moment but language redundancy can be removed further and will lead to an even more precise formulation. The author decided that language constructs, such as active or passive, do not provide any information. Example: 'RCEM LvbatPwrReq is set to a value' has the same meaning with respect to testing as 'Set *RCEM_LvbatPwrReq* to a value'. To get away from those nasty natural language constructs it is a good idea to introduce some math: for example the sign =.

According to this, version 1.4 states:

```
RCEM_LvbatPwrReq = value_1
RCEM_HvbatPwrMax = 0
OBEA_BdNetPwrReq =  0
DIAG_EltMotTqMin =  value_2;  RMRE_EltMotTqReqs is not limited
OBEM_EltMotTqMinContns = value_3; RMRE_EltMotTqReqs is not limited
```

At the moment, there is no meaning for *value_1*, *value_2* and *value_3*. One can give them meaning by introducing datatypes:

```
int value_1;
int value_2;
int value_3;
```

By introducing variables, it is communicated to the tester to assign a value. By introducing datatypes, the range of the value is determined. Therefore the final version 1.5 states:

```
int value_1;
int value_2;
int value_3;

RCEM_LvbatPwrReq = value_1
RCEM_HvbatPwrMax = 0
OBEA_BdNetPwrReq =  0
DIAG_EltMotTqMin =  value_2;  RMRE_EltMotTqReqs is not limited
OBEM_EltMotTqMinContns = value_3; RMRE_EltMotTqReqs is not limited
```

The model is semi-formal, because *DIAG_EltMotTqMin* and *OBEM_EltMotTqMinContns* incorporate signal descriptions. They will be called *attributes* from now on. At this point the requirement is ready to be transformed to a requirements model in pure::variants (see Figure 5.6).

The current requirement is modeled as *Requirement_1* in the feature model.

Values (*RCEM_HvbatPwrMax*, *OBEA_BdNetPwrReq*) that must not be changed by the tester are set to a fixed *value* and can not be changed in the variant description model. Fixed attributes are indicated by an *F* on the left hand side of the attribute name (*value*).

Those values that should be typed in by the tester (*RCEM_LvbatPwrReq*, *DIAG_EltMotTqMin*, *OBEM_EltMotTqMinContns*) do not contain a value. The value itself has a datatype in the FODA-tool: ps:integer.

A brief signal description is provided as an attribute for the signals *DIAG_EltMotTqMin* and *OBEM_EltMotTqMinContns*.

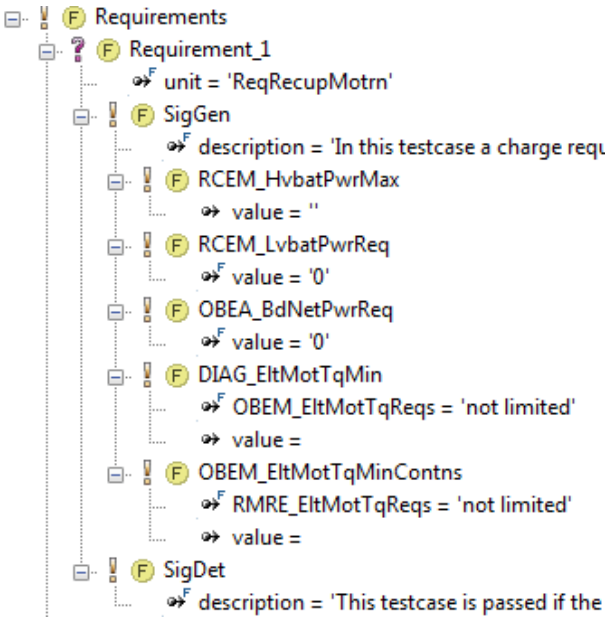

Figure 5.6.: Sample Requirements Model

5.4. Solution Space of the Case Study

The units under test are:

- ReqRecupMotrn
- ReqRecupSrvBrk

For each of them tests have been implemented. Because both units are tested with the same code, the test implementation is explained for only one unit under test.

5.4.1. Test Bed

The unit under test is *ReqRecupMotrn* (see Figure 5.7). The *test block* (see also Section 4.2.1) is called *test_ReqRecupMotrn_all*. It provides the test code in Simulink. It generates test data and evaluates the feedback from the unit under test.

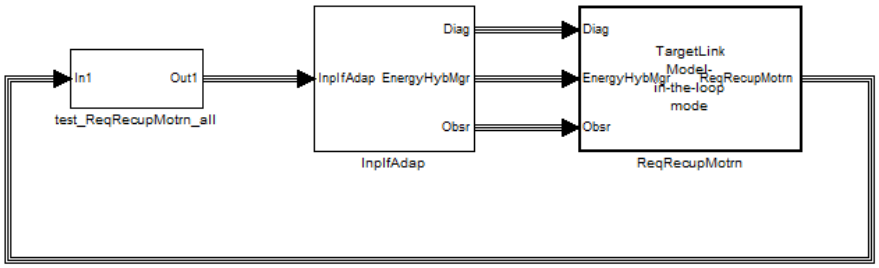

Figure 5.7.: Sample *Test Bed* Including *Test Block* and Unit Under Test

5.4.2. Test Block

Figure 5.8 shows the contents of the *test block*. The *test composite* contains all *test cases*. The *Subsystem* connects the output signals (test data) of the *test composite* with a bus.

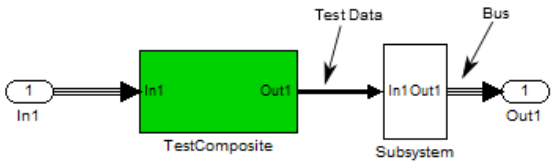

Figure 5.8.: Contents of the *Test Block*

5.4.3. Test Composite

The *test composite* (see also Section 4.2.2) contains all 20 test cases (see Figure 5.9). The *Multiplexer1* is a *slUnit multiplexer* (see also Section 4.2.2). It connects the output of exactly one *test case* with the unit under test.

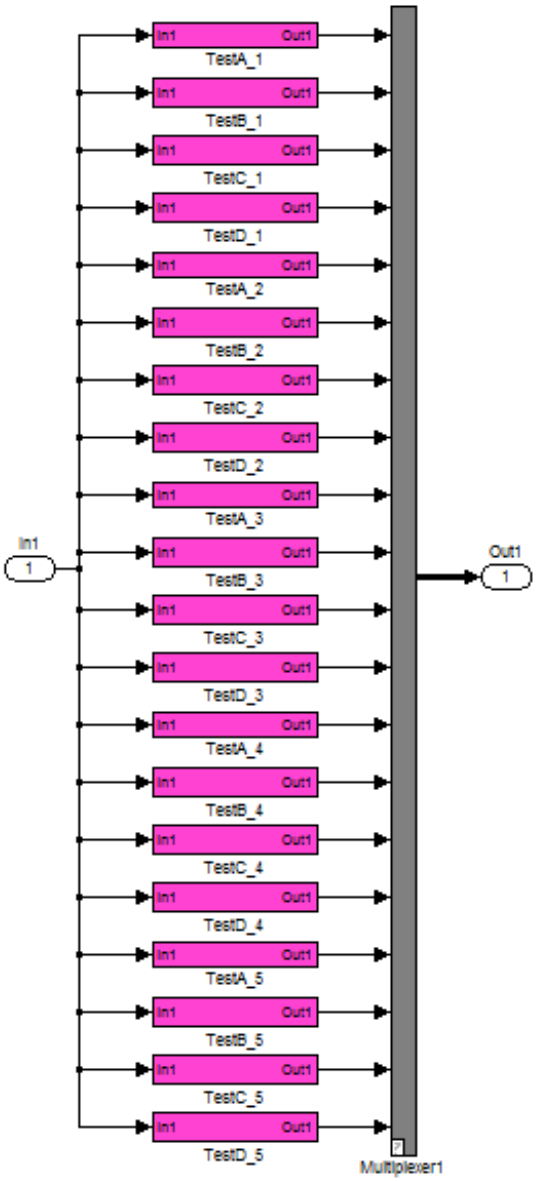

Figure 5.9.: Contents of the *Test Composite* Including *Test Cases* and *slUnit Multiplexer*

5.4.4. Test Case

Figure 5.10 shows the contents of the *test case TestA_1*. The *test case* (see also Section 4.2.3) contains a subsystem called *test_ReqRecupMotrn_a_1*. It is the *test case implementation*.

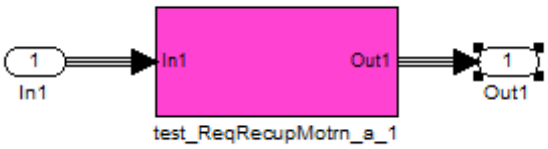

Figure 5.10.: Contents of a *Test Case* Including the *Test Case Implementation*

5.4.5. Test Case Implementation

In Figure 5.11 the Simulink Constant Blocks colored in magenta create the output signals (test data). The feedback from the unit under test are evaluated in the *validate* subsystem. The feedback data is comparerd with the following signals: *DIAG_EltMotTqMin* and *OBEM_EltMotTqMinContns*.

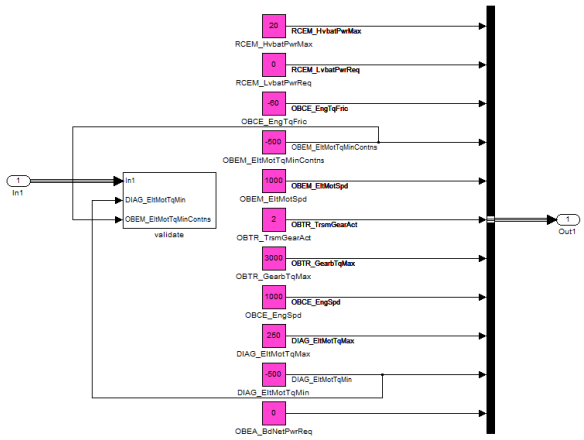

Figure 5.11.: Contents of the *Test Case Implementation*

5.4.6. Unit Test Model: test

Figure 5.12 lists a ps:component for each of the 20 *test cases*, f.i. *ref_TestA_1*. These ps:components are responsible for creating *mlUnit test cases*. Each *test case* is linked with a *requirements model* by means of a restriction: a *mlUnit test case* is only created, when the corresponding *requirement* is selected in the *requirements model variant description model* (see Figure 5.5).

A *mlUnit test suite* connects all *mlUnit test cases*. This *test suite* is represented by the ps:component *test suites*.

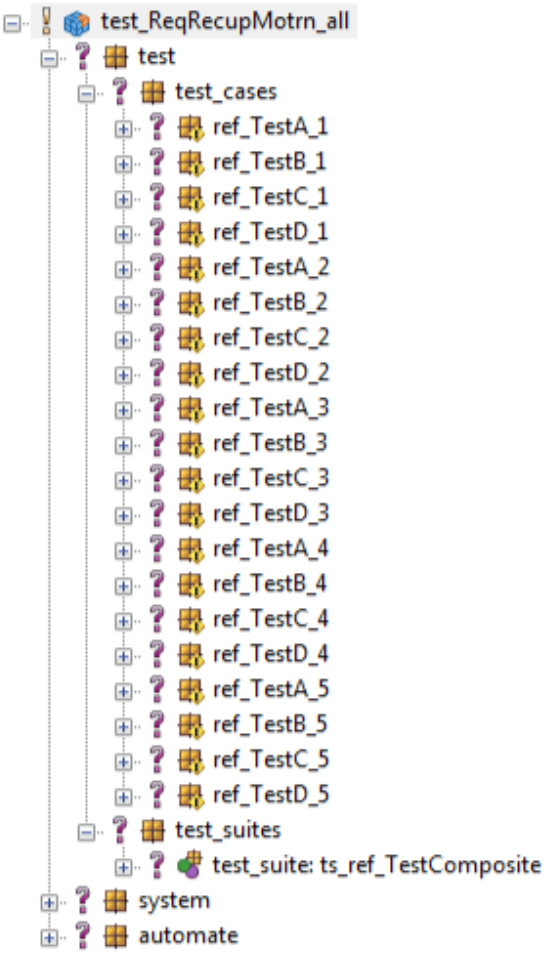

Figure 5.12.: Unit Test Model Showing the Representation of *Test Cases* and the *Test Suite*

5.4.7. Unit Test Model: system

Figure 5.13 represents the structure of the Simulink test code. The *test bed* described in Section 5.4.1 is represented by the ps:family *test_ReqRecupMotrn_all*.

The *test composite* described in Section 5.4.3 is represented by the ps:component *TestComposite*.

slUnit test cases are represented by all children of the ps:component *test composite*. The *test case* described in Section 5.4.4 is represented by the ps:component *TestA_1*.

The *test case implementation* described in Section 5.4.5 is represented by the ps:component *test_ReqRecupMotrn_a_1*. Within the *test case implementation* (see Figure 5.11) the Simulink Constant Blocks are responsible for creating the test data. The following Simulink Constant Blocks are represented by a ps:component:

- *DIAG_EltMotTqMax*
- *DIAG_EltMotTqMin*
- *OBCE_EngSpd*
- *OBCE_EngTqFric*
- *OBEA_BdNetPwrReq*
- *OBEM_EltMotSpd*
- *OBCEN_EltMotTqMinContns*
- *OBTR_GearbTqMax*
- *OBTR_TrsmGearAct*
- *RCEM_HvbatPwrMax*
- *RCEM_LvbatPwrReq*

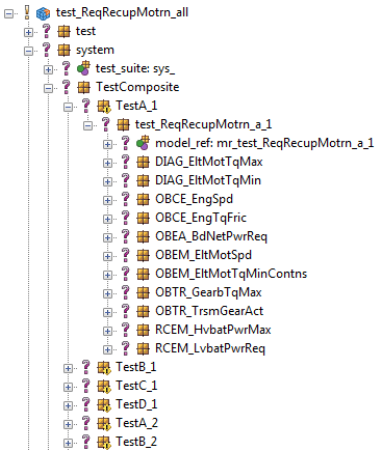

Figure 5.13.: Unit Test Model Showing the Representation of the Simulink Test Code

5.5. Test Data

The *test data* generated by the developed framework for the two vehicles are listed in Table 5.3 and Table 5.4.

Test Case	*RCEM_HvbatPwrMax [kW]*	*RCEM_LvbatPwrReq [kW]*	*OBCE_EngTqFric [Nm]*	*OBEM_EltMotSpd [rpm]*	*OBTR_TrsmGearAct [-]*	*OBTR_GearbTqMax [Nm]*	*OBCE_EngSpd [rpm]*	*DIAG_EltMotTqMax [Nm]*	*DIAG_EltMotTqMin [Nm]*	*OBEM_EltMotTqMinContns [Nm]*	*OBEA_BdNetPwrReq [kW]*
TestA_1	20	0	-30	1000	1	500	1000	200	-200	-200	0
TestB_1	0	10	-30	1000	1	500	1000	200	-200	-200	0
TestC_1	0	0	-30	1000	1	500	1000	200	-200	-2	0.25
TestD_1	20	0	-30	1000	1	500	1000	200	-200	-200	0
TestA_2	20	0	-30	1000	2	500	1000	200	-200	-200	0
TestB_2	0	10	-30	1000	2	500	1000	200	-200	-200	0
TestC_2	0	0	-30	1000	2	500	1000	200	-200	-2	0.25
TestD_2	20	0	-30	1000	2	500	1000	200	-200	-200	0
TestA_3	20	0	-30	1000	3	500	1000	200	-200	-200	0
TestB_3	0	10	-30	1000	3	500	1000	200	-200	-200	0
TestC_3	0	0	-30	1000	3	500	1000	200	-200	-2	0.25
TestD_3	20	0	-30	1000	3	500	1000	200	-200	-200	0
TestA_4	20	0	-30	1000	7	500	1000	200	-200	-200	0
TestB_4	0	10	-30	1000	7	500	1000	200	-200	-200	0
TestC_4	0	0	-30	1000	7	500	1000	200	-200	-2	0.25
TestD_4	20	0	-30	1000	7	500	1000	200	-200	-200	0
TestA_5	20	0	-30	1000	8	500	1000	200	-200	-200	0
TestB_5	0	10	-30	1000	8	500	1000	200	-200	-200	0
TestC_5	0	0	-30	1000	8	500	1000	200	-200	-2	0.25
TestD_5	20	0	-30	1000	8	500	1000	200	-200	-200	0

Table 5.3.: Test Data for Vehicle 1

Test Case	*RCEM_HvbatPwrMax [kW]*	*RCEM_LvbatPwrReq [kW]*	*OBCE_EngTqFric [Nm]*	*OBEM_EltMotSpd [rpm]*	*OBTR_TrsmGearAct [-]*	*OBTR_GearbTqMax [Nm]*	*OBCE_EngSpd [rpm]*	*DIAG_EltMotTqMax [Nm]*	*DIAG_EltMotTqMin [Nm]*	*OBEM_EltMotTqMinContns [Nm]*	*OBEA_BdNetPwrReq [kW]*
TestA_1	50	0	-60	1000	1	3000	1000	350	-350	-350	0
TestB_1	0	10	-60	1000	1	3000	1000	350	-350	-350	0
TestC_1	0	0	-60	1000	1	3000	1000	350	-350	-2	0.25
TestD_1	50	0	-60	1000	1	3000	1000	350	-350	-200	0
TestA_2	50	0	-60	1000	2	3000	1000	350	-350	-350	0
TestB_2	0	10	-60	1000	2	3000	1000	350	-350	-350	0
TestC_2	0	0	-60	1000	2	3000	1000	350	-350	-2	0.25
TestD_2	50	0	-60	1000	2	3000	1000	350	-350	-200	0
TestA_3	50	0	-60	1000	3	3000	1000	350	-350	-350	0
TestB_3	0	10	-60	1000	3	3000	1000	350	-350	-350	0
TestC_3	0	0	-60	1000	3	3000	1000	350	-350	-2	0.25
TestD_3	50	0	-60	1000	3	3000	1000	350	-350	-200	0
TestA_4	50	0	-60	1000	20	3000	1000	350	-350	-350	0
TestB_4	0	10	-60	1000	20	3000	1000	350	-350	-350	0
TestC_4	0	0	-60	1000	20	3000	1000	350	-350	-2	0.25
TestD_4	50	0	-60	1000	20	3000	1000	350	-350	-200	0
TestA_5	50	0	-60	1000	20	3000	1000	350	-350	-350	0
TestB_5	0	10	-60	1000	20	3000	1000	350	-350	-350	0
TestC_5	0	0	-60	1000	20	3000	1000	350	-350	-2	0.25
TestD_5	50	0	-60	1000	20	3000	1000	350	-350	-200	0

Table 5.4.: Test Data for Vehicle 2

5.6. Goal Question Metric - Defined Goals of the Case Study

The GQM (Goal Question Metric) (see [vSB99]) is a means to assess if the *goals* of a project have been met. Several *questions* help to identify aspects of each *goal*. *Metrics* give answers to the *questions*.

The goals and questions defined to assess the results of the proposed framework are illustrated in Figure 5.14.

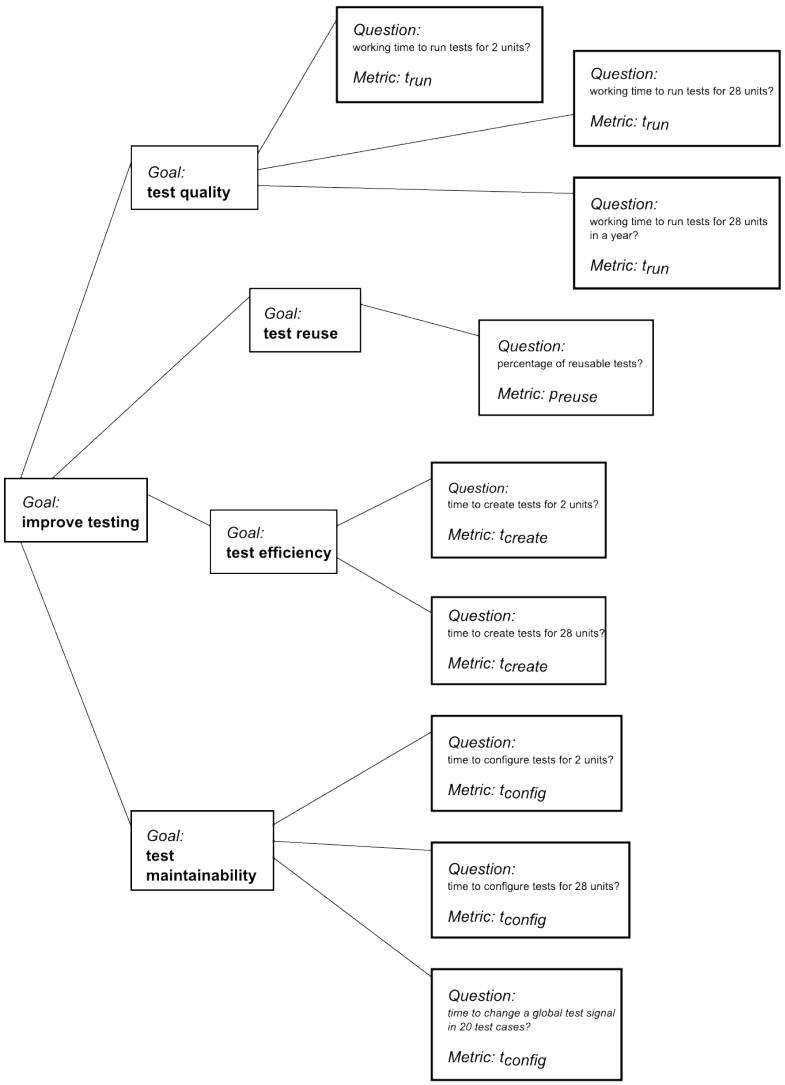

Figure 5.14.: Defined Goals, Corresponding Questions and Metrics

5.7. Goal Question Metric - Metrics of the Case Study

The metrics used in this case study are described in Table 5.6.

Metric: t_{create}
Description: The metric t_{create} denotes the time necessary to create tests for one unit.
Metric: t_{run}
Description: The metric t_{run} describes the working time necessary to run all tests for the selected units.
Metric: t_{change}
Description: The metric t_{change} describes the working time necessary to change a signal.
Metric: t_{config}
Description: The metric t_{config} is the time needed to configure a *variant description model* in pure::variants.
Metric: p_{reuse}
Description: The metric p_{reuse} denotes the ratio between reusable test cases and not reusable test cases in percent. A test case is reusable, if it does not change its test data from one vehicle to another or if it can be configured in a *variant description model*.

Table 5.6.: Metrics in This Case Study and Their Description

5.8. Goal Question Metric - Results of the Case Study

The GQM has been conducted for the former test framework that is called *Poet* and for the *SPL test framework* which is the proposed methodology. The same tests have been implemented with both approaches.

5.8.1. Results for Poet

Goal: Test Efficiency
Question: How much effort does it take to create tests for 2 units?
Metric: $t_{create} = 40\ m$
Description: The time t_{create} has been measured when creating tests for 2 units under test. Each unit under test is tested by 20 test cases. For each test case 11 constant signals have to be drawn.
Question: How much effort does it take to create tests for the sample project (28 units) ?
Metric: $t_{create} = 9.3\ h$
Description: The time t_{create} for 28 units has been extrapolated by the assumption that the effort for creating the above mentioned units constitutes the average effort throughout the sample project.

Table 5.7.: Test Efficiency in *Poet*

Goal: Test Quality
Question: How much effort does it take to run tests for 2 units?
Metric: $t_{run} = 2\ m$
Description: The time t_{run} to run tests for 2 units has been measured. The time t_{run} is the time to open the first unit, run the tests for it plus the time to open the second unit and run the tests for it.

Question: How much effort does it take to run tests for the sample project (28 units) ?
Metric: $t_{run} = 56\ m$
Description: The time t_{run} to run tests for 28 units has been extrapolated from the effort to run tests for 2 units.
Question: How much effort does it take to run tests for the sample project (28 units) in a year (250 days) ?
Metric: $t_{run} = 233\ h$
Description: The time t_{run} to run tests for 2 units has been extrapolated from the effort to run tests for 2 units with the assumption that tests are carried out one time each working day.

Table 5.8.: Test Quality in *Poet*

Goal: Test Maintainability
Question: How much effort does it take to configure tests for 2 units?
Metric: $t_{config} = 40\ m$
Description: In *Poet* it is not supported to configure test data from an external source. The tests have to be created newly for each variant. The time t_{config} corresponds to the time t_{create} in Table 5.7.
Question: How much working time does it take to configure tests for the sample project (28 units) ?
Metric: $t_{config} = 9.3\ h$
Description: In *Poet* it is not supported to configure test data from an external source. The tests have to be created newly for each variant. The time t_{config} corresponds to the time t_{create} in Table 5.7.
Question: How much working time does it take to change a global signal in 20 test cases ?
Metric: $t_{change} = 2\ m$
Description: The time needed to shift a certain constant signal in 20 test cases has been measured.

Table 5.9.: Test Maintainability in *Poet*

Goal: Test Reuse
Question: What is the percentage of reusable test cases for the sample project?
Metric: $p_{reuse} = 19\ \%$
Description: Reusable test cases in the context of *Poet* are test cases where test data do not change from one vehicle to another for sure. Those test cases are so-called driving cycles that remain the same for each vehicle.

Table 5.10.: Test Reuse in *Poet*

5.8.2. Results for SPL Test Framework

Goal: Test Efficiency
Question: How much effort does it take to create tests for 2 units?
Metric: $t_{create} = 16\ h$
Description: The time t_{create} has been measured when creating tests for 2 units under test. Each unit under test is tested by 20 test cases. For each test case 11 constant signals have to be drawn and referenced by pure::variants.
Question: How much effort does it take to create tests for the sample project (28 units) ?
Metric: $t_{create} = 224\ h$
Description: The time t_{create} for 28 units has been extrapolated by the assumption that the effort for creating the above mentioned units constitues the average effort throughout the sample project.

Table 5.11.: Test Efficiency in the *SPL Test Framework*

Goal: Test Quality
Question: How much effort does it take to run tests for 2 units?
Metric: $t_{run} = 1\ m$
Description: The time t_{run} to run tests for 2 units has been measured.

Question: How much effort does it take to run tests for the sample project (28 units) ?
Metric: $t_{run} = 1\ m$
Description: It is assumed that the automated tests can be executed on a dedicated data processor. Thus, the time t_{run} is the time needed to start the automated test scripts. This time remains always the same, independent of the number of units under test.
Question: How much effort does it take to run tests for the sample project (28 units) in a year (250 days)?
Metric: $t_{run} = 4\ h$
Description: It is assumed that the automated tests can be executed on a dedicated data processor. Thus, the time t_{run} is the time needed to start the automated test scripts once a day.

Table 5.12.: Test Quality in the *SPL Test Framework*

Goal: Test Maintainability
Question: How much working time does it take to configure tests for 2 units?
Metric: Time to configure vehicle config model: $t_{config} = 10\ m$ Time to configure requirements models: $t_{config} = 20\ m$
Question: How much working time does it take to configure the sample project (28 units) ?
Metric: Time to configure vehicle config model: $t_{config} = 10\ m$ Time to configure requirements models: $t_{config} = 5\ h$
Description: The time t_{config} corresponds to the time needed to configure the *vehicle config model* and the *requirements models* for the sample project.
Question: How much working time does it take to change a global signal in 20 test cases ?
Metric: Time to configure vehicle config model: $t_{change} = 1\ m$
Description: The time needed to shift a certain constant signal in 20 test cases has been measured. This is done by configuring the *variant description model*.

Table 5.13.: Test Maintainability in the *SPL Test Framework*

Goal: Test Reuse
Question: What is the percentage of reusable test cases?
Metric: $p_{reuse} = 95\ \%$
Description: Reusable test cases in the context of the SPL framework are test cases that can be configured by a *variant description model*. Some test cases differ enormously from a variant to another. In these cases, modelling differences by a *variant description model* becomes unprofitable compared to manually creating the test cases.

Table 5.14.: Test Reuse in the *SPL Test Framework*

5.9. Summary of all Metrics

The metrics determined in Section 5.8.1 and Section 5.8.2 are summarized in Table 5.15.

Metric	*Poet*	*SPL Test Framework*	*Reference (Poet / SPL)*
t_{create} (2 units)	40 m	16 h	Table 5.7 / Table 5.11
t_{create} (sample project)	9.3 h	224 h	Table 5.7 / Table 5.11
t_{run} (2 units)	4 m	1 m	Table 5.8 / Table 5.12
t_{run} (sample project)	56 m	1 m	Table 5.8 / Table 5.12
t_{run} (sample project per year)	233 h	4 h	Table 5.8 / Table 5.12
t_{config} (2 units)	40 m	30 m	Table 5.9 / Table 5.13
t_{config} (sample project)	9.3 h	5.2 h	Table 5.9 / Table 5.13
t_{change}	2 m	1 m	Table 5.9 / Table 5.13
p_{reuse}	19 %	95 %	Table 5.10 / Table 5.14

Table 5.15.: Summary of all Metrics

5.10. Break Even of the SPL Test Framework

To calculate the break even of the proposed methodology, a simple model is used. The terminology is given in Table 5.16.

Term: $t_{develop}$
Description: The time to develop the test framework $t_{develop}$ is the upfront investment.
Term: t_{create}
Description: The time t_{create} is the initial effort to create tests for all units of the sample project.
Term: t_{config}
Description: The time t_{config} is the effort to configure tests for all units of the sample project.
Term: $t_{maintain}$
Description: The time $t_{maintain}$ is the effort to create/configure tests for all units of the sample project for all variants.
Term: $N_{variants}$
Description: $N_{variants}$ is the number of variants that have to be tested.
Term: *full test run*
Description: In a *full test run* all variants are tested.
Term: *test run*
Description: In a *test run* one variant is tested.
Term: n_{run}
Description: The number of full test runs is n_{run}.
Term: t_{run}
Description: The time t_{run} is the effort that has to be spent on each test run.
Term: t_{total}
Description: The total time t_{total} that has been spent on testing is a function of. the number of test runs n_{run} and tested variants $N_{variants}$.

Table 5.16.: Terminology for the Calculation of the Break Even

The total time t_{total} that has been spent on testing is a function of the number of test runs n_{run} and tested variants $N_{variants}$ and is calculated as follows:

$$t_{total}(n_{run}, N_{variants}) = t_{develop} + t_{maintain}(N_{variants}) + t_{run} \cdot n_{run} \cdot N_{variants} \tag{5.1}$$

The time needed to maintain tests can be estimated as follows:

$$t_{maintain}(N_{variants}) \approx t_{create} + t_{config} \cdot (N_{variants} - 1) \tag{5.2}$$

The data necessary for calculation are listed in Table 5.17.

Test Framework	$t_{develop}$	t_{create}	t_{config}	t_{run}
Poet[4]	400 h	9.3 h	9.3 h	56 m
SPL Test Framework[5]	600 h	224 h	5.2 h	1 m

Table 5.17.: Effort for Developing, Creating and Running Tests

In Figure 5.15 the total effort as a function of the number of test runs is shown for both test methodologies. The break even is the point where the number of full test runs and the total effort equals for both methodologies. The break evens for $N_{variants} = 1..5$ are listed in Table 5.18.

$N_{variants}$	*Full Test Runs*	*Total Effort*
1	638	835 h
2	315	840 h
3	208	845 h
4	154	850 h
5	122	855 h

Table 5.18.: Break Evens for $N_{variants} = 1..5$

[4]The time $t_{develop}$ has been estimated for the implementation of the xUnit functionality of Poet.
[5]The time $t_{develop}$ has been measured during development.

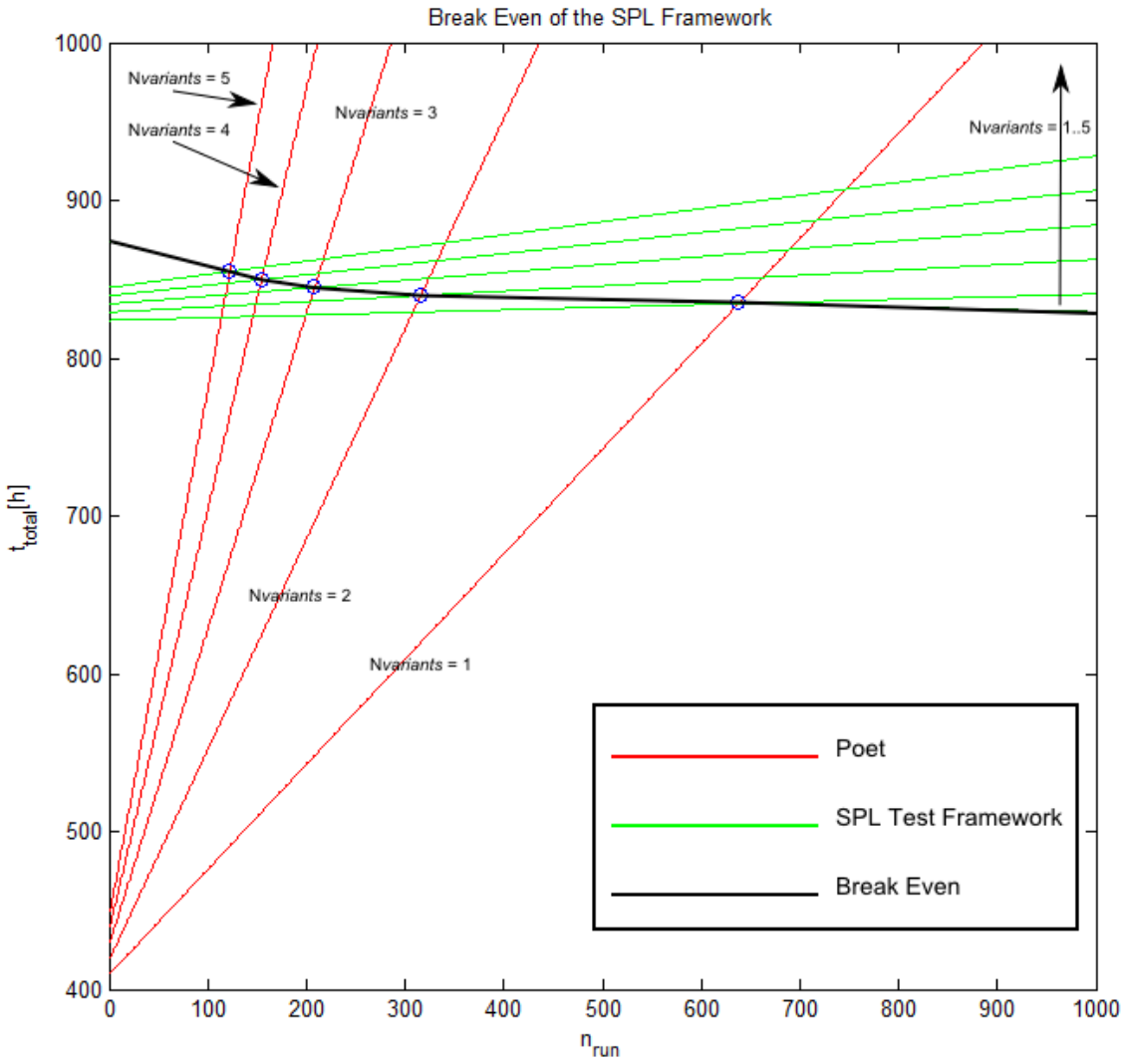

Figure 5.15.: Total Effort for Both Methodologies for $N_{variants} = 1..5$

6. Conclusion

6.1. Lessons Learned

6.1.1. Test Efficiency

Creating tests in the proposed methodology is a very time consuming task, because the implementation is a prototype. The time for creating tests could be decreased, if the implementation is optimized. Nevertheless, it turned out, that it pays off in the long term (see Section 5.10).

6.1.2. Test Quality

The effort for running tests is minimized by the proposed methodology. It is not dependent on the number of units under test. So, it can be assumed, that testing will be performed more often. Thus, defects will be discovered sooner which will cut development costs. Unfortunately, it is very hard to measure the cost savings achieved by a higher test quality.

6.1.3. Test Maintainability

It is possible to create generic test cases and configure them by means of a FODA-tool. It is also very important that a test signal that is part of several test cases can be changed in a single place: the *vehicle config model* or the *requirements model*. Typing errors can be minimized, thus.

6.1.4. Availability of Information

At the moment a lot of information is available in the field of testing embedded systems with regard to Software Product Lines. Nevertheless, this thesis was the result of the synthesis of some loose coherent cornerstones. Picking out the right parts from each of them has lead to a suitable solution.

Information concerning Matlab/Simulink was very well available. The pure::variants information was sufficient. mlUnit and slUnit are rarely documented. Since both have been developed with the TDD approach, a lot of test cases have substituted a conventional documentation in prose. All in all, TDD has proven to serve as an alternative documentation in these cases.

6.1.5. Usability

At the moment, it seems that the framework is too complicated to use. This is a problem, since no one will be able to *play around* with the framework without some training lessons. This will cost a lot of time. Since time is rare and expensive, people won't get the full

potential of the developed framework. In the end, it would have been better to build a smaller prototype with fewer features, so that people don't have problems using it.

6.1.6. Implementation of the SPL

In this thesis the full range of test artifacts has been covered. The selection of test cases and units under test works fine. Test reports are generated automatically by the framework. The most compelling problem are test data sets because it is difficult to represent time-discrete test data in a FODA-tool. A possible solution to that problem may be the signal feature concept described in Section 2.5.3 in combination with representing those signal features in a pure::variants feature model, namely the requirements model (see Section 3.5.1). It is also possible to link requirements with test cases which serves as an alternative documentation.

6.2. Future Work

6.2.1. Parameter Override

Parameter overrides are used to vary some states of the unit under test. In terms of xUnit this means to *create context* as described in Section 2.2.4. At the moment only empty *setup* and *teardown* methods are created by the framework. Parameter overrides can be done by manually writing code into the two methods. It is desirable to implement an automation of that process. It will be a real challenge to come to a solution with high usability.

6.2.2. Improve Usability

Usability is one of the major problems of this prototype. There are several ways to improve usability: First, by creating a family model, a lot of work has to be done in self-made tools. It will be better to relocate work to standard software, such as Simulink and pure::variants, as far as possible. That means: references to Simulink blocks should not be saved as node-objects but only as family models. So the user can build up a collection of references in pure::variants. The composition of those references can be accomplished in pure::variants then, which will be much simpler and more stable.

6.2.3. Integration Testing

This thesis focuses on unit testing, only. Integration testing should be considered as well. Integration testing will be much more compelling as unit testing, regarding the existent Software Product Line. In another part of the *HybConS* project, variable state machines have been implemented. Variable state machine may perhaps pave the way to integration testing. Another possibility is to derive test cases from use cases that incorporate variability, too.

6.2.4. Testing for Verification vs. Testing for Debugging

Testing for debugging (see [Zel09]) is in principle the counterpart of Test-driven development and both may complement each other well. Usually, when a defect is detected one has to reconstruct the input data that has led to the defect. So, one has to write a test case that reconstructs the defect. Further, refactoring of the test case has to be done in order to find the point, where the defect is barely uncovered. Following this principle, it is possible to link a test case with a defect.

A. Testing Software - Terminology

Quality The totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs. Customer's needs are usually translated into features and characteristics witch specific criteria. Needs may include aspects of usability, safety, availability, reliability, maintainability, economics and environment. Needs must be specified in such a manner that we know when we have satisfied them (they must be testable): The term *quality* is not an expression of a degree of excellence [IEE90].

Testing Validation [IEE90].

Test Specification Describes the test criteria and the methods to be used in a specific test to assure the performance and design specifications have been satisfied. The test specification identifies the capabilities or program functions to be tested and identifies the test environment [IEE90].

Validation The process of evaluating a product or service to ensure compliance with the specified requirements [IEE90].

Verification The process of evaluating a product or service at a point in the process (or at the end of the process) to ensure correctness and consistency with respect to the products and standards provided as input to that process [IEE90].

Requirement 1) a condition or capability needed by a user to solve a problem or achieve an objective
2) a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed document
3) a documented representation of a condition or capability as in definition 1 or 2 [IEE90].

Failure An externally visible *error* in the program behavior. Also known as *malfunction* [Zel09].

Fault Synonym for *defect* [Zel09].

Error 1) An unwanted and unintended deviation from what is correct, right or true. 2) Synonym for *infection*. 3) Synonym for *mistake* [Zel09].

Defect An error in the program - especially one that can cause an *infection* and thus a *failure*. Also known as *bug* or *fault* [Zel09].

Bibliography

[Bec09] Kent Beck. *Test-Driven Development By Example*. Signature Series. Addison-Wesley, 2009.

[Bec11] Kent Beck. Simple Smalltalk Testing: With Patterns. `http://www.xprogramming.com/testfram.htm`, visited 2011.

[Bro11] Frederick Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. `http://www.cs.nott.ac.uk/~cah/G51ISS/Documents/NoSilverBullet.html`, visited 2011.

[C+09] Barrow Colin et al. *Business Plans For Dummies*. John Wiley & Sons, 2009.

[CdPL09] Lawrence Chung and Julio Cesar Sampaio do Prado Leite. On Non-Functional Requirements in Software Engineering. In *Conceptual Modeling: Foundations and Applications*, pages 363–379, 2009.

[CN07] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns* . Addison Wesley, 2007.

[CN+11a] Paul Clements, Linda Northrop, et al. A Framework for Software Product Line Practice, Version 5.0. `http://www.sei.cmu.edu/productlines/frame_report/pl_is_not.htm`, visited 2011.

[CN+11b] Paul Clements, Linda Northrop, et al. A Framework for Software Product Line Practice, Version 5.0. `http://www.sei.cmu.edu/productlines/frame_report/coreADA.htm`, visited 2011.

[Con04] Mirko Conrad. *Modell-basierter Test eingebetteter Software im Automobil*. PhD thesis, Technische Universität Berlin, 2004.

[DG07] Thomas Dohmke and Henrik Gollee. Test-Driven Development of a PID Controller. *Software, IEEE*, 24(3):44–50, 2007.

[Doh08] Thomas Dohmke. *Test-Driven Development of Embedded Control Systems: Application in an Automotive Collision Prevention System*. PhD thesis, University of Glasgow, 2008.

[Eng03] Erwin Engelsma. Test Evolution. `http://www.esi.es/Cafe/pdf/Test_strategy_methodology_and_process.zip`, 2003.

[FS94] Wiliam B. Frakes and Isoda Sadahiro. Success Factors of Systematic Reuse. *Software, IEEE*, 11(5):12–19, 1994.

[GP07] Atul Gupta and Jalote Pankaj. An experimental Evaluation of the Effectiveness of the Test Driven Development. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 405–416, Washington, DC, USA, 2007. IEEE Computer Society.

[GSM04] A. Geras, E. M. Smith, and J. Miller. A Prototype Empirical Evaluation of Test Driven Development. In *Proceedings of the 10th International Symposium on Software Metrics*, pages 405–416, Washington, DC, USA, 2004. IEEE Computer Society.

[IEE90] IEEE Computer Society. *IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990)*, March 1990.

[KM10] Alexander Krupp and Wolfgang Müller. A systematic approach to the test of combined HW/SW systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 323–326. IEEE, 2010.

[Kor10] Evgeny Korsunsky. Generic hybrid control SW architecture. Technical Report 1.0, AVL List GmbH (internal), July 2010.

[Leh04] Eckhardt Lehmann. *Time Partition Testing - Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen.* PhD thesis, Technische Universität Berlin, 2004.

[McG11] John McGregor. Testing a Software Product Line. Technical report, CMU/SEI, visited 2011.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering.* Springer, 2005.

[PD93] Rubén Prieto-Díaz. Status Report: Software Reusability. *Software, IEEE*, 10(3):61–66, 1993.

[Pel06] William Pelfrey. *Billy, Alfred, and General Motors: The Story of Two Unique Men, a Legendary Company, and a Remarkable Time in American History* . Mcgraw-Hill Professiona, 2006.

[RGW03] Doris Rauh, Helmut Goetz, and Josef Weingaertner. Test Process and Implementation. `http://www.esi.es/Cafe/pdf/Test_modeling_and_tooling.zip`, 2003.

[Sch02] Klaus Schmid. A Comprehensive Product Line Scoping Approach and Its Validation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 593 – 603, Washington, DC, USA, 2002. IEEE Computer Society.

[Str00] Bjarne Stroustroup. *The C++ Programming Language: Special Edition.* Addison Wesley, 2000.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[vdLSR07] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007.

[Vig10] Uwe Vigenschow. *Testen von Software und Embedded Systems: Professionelles Vorgehen mit modellbasierten und objektorientierten Ansätzen*. dpunkt.Verlag, 2010.

[vSB99] Rini van Solingen and Egon Berghout. *The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development*. Mcgraw-Hill Professional, 1999.

[WMV03] Laurie Williams, E. Michael Maximilien, and Mladen Vouk. Test-Driven Development as a Defect-Reduction Practice. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 34 – 45, Washington, DC, USA, 2003. IEEE Computer Society.

[Wol11] Robert Wolff. Klassifikationsbaummethode für eingebettete Systeme (CTM/ES). `http://www2.informatik.hu-berlin.de/~hs/Lehre/2006-SS_SpezTest/10_Wolff_CTM-ES_Ausarbeitung.pdf`, visited 2011.

[Zel09] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009.

[ZN09] Justyna Zander-Nowicka. *Model-based Testing of Real-Time Embedded Systems in the Automotive Domain*. PhD thesis, Technische Universität Berlin, 2009.