eDIANA
Embedded Systems for Energy Efficient Buildings

Grant agreement no.: *100012*

D6.2-B Guidelines for automated generation of scenario to validate UML models against requirements

Author(s):

Goiuria Sagardui	**MU**
Joseba Andoni Agirre	**MU**
Lorea Belategui	**MU**
Leire Etxeberria	**MU**
Angel Diaz	**Labein**
Luis Martinez	**Labein**
Amaia Uriarte	**Labein**
Azucena Cortes	**Labein**
Jesús Benedicto	**ATOS**

Issue Date	April 2010
Deliverable Number	D6.2-B
WP Number	WP6
Status	Released

Dissemination level	
X	**PU** = Public
	PP = Restricted to other programme participants (including the JU)
	RE = Restricted to a group specified by the consortium (including the JU)
	CO = Confidential, only for members of the consortium (including the JU)

Document history			
V	***Date***	***Author***	***Description***
0.1	*15/01/2010*	*MU*	*ToC*
0.2	*26/02/2010*	*Labein*	*Contribution to Introduction and UnitTesting*
0.3	*26/03/2010*	*MU*	*Contribution to Introduction, Model based testing, System Testing and Variability management sections*
0.4	*22/04/2010*	*ATOS, Labein, MU*	*Contribution to Traceability (ATOS) and Contribution to Test Modelling (Labein), first version of summary and conclusions (MU)*
0.5	*26/04/2010*	*MU*	*Addition and correction of some sections*
0.6	*28/04/2010*	*Labein*	*Contribution to eDiana Platforms TTCN-3 Compliance*

Disclaimer

The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

The document reflects only the author's views and the Community is not liable for any use that may be made of the information contained therein.

Summary

The D6.2B Guidelines for automated generation of scenario to validate UML models against requirements is a document delivered in the context of WP6, Task 6.2: Early V&V with regard to model based testing in eDIANA.

This document is about Model Based Testing in eDIANA and different dimensions are considered: The generation of test scenarios using models of the SUT (System Under Test); test modelling which is concerned with modelling testing structure, test behaviour and testing artefacts; the traceability between test scenarios and requirements and variability management in testing.

Contents

Abbreviations

ASN.1	Abstract Syntax Notation One
CORBA	Common Object Request Broker Architecture
eDIANA	Embedded Systems for Energy Efficient Buildings
EFSM	Extended Finite State Machine
ETSI	European Telecommunications Standards Institute
FSM	Finite State Machine
HP QC	HP Quality Center
IDL	Interface Definition Language
MDD	Model Driven Development
MBT	Model-based Testing
OOSD	Object Oriented Software Development
PA	Platform Adapter
SA	SUT Adapter
SDLC	Software Development Life Cycle
SUT	System Under Test
TE	TTCN-3 Executable
TDD	Test Driven Development
TCI	TTCN-3 Control Interface
TRI	TTCN-3 Runtime Interface
TTCN-3	Testing and Test Control Notation version 3
UML	Unified Model Language
U2TP	UML 2.0 Testing Profile

XML	Extensible Markup Language

1. Introduction

The rapid growth in processor speed and memory enabled the development of modeling, simulation, and code-generation tools on the desktop. It also enables embedded-software developers to increase the functions and complexity of embedded controllers. This has highlighted the need to move beyond traditional code-development techniques using text editors and debuggers to center design on models. This development approach is known as model-based design, and it streamlines embedded control design with modeling, simulation and automatic-code generation (see Figure 1-1).

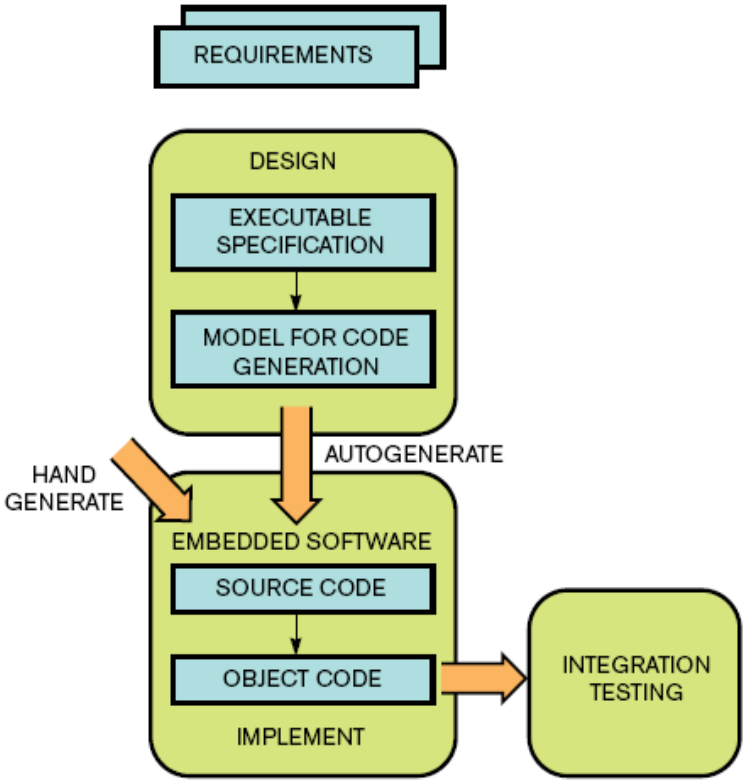

Figure 1-1: Model Based design approach

Working with model-based design means that developers use models to carry out their designs from the written requirements. These models become an "executable specification." The ability to execute the design is a huge benefit for developers or designers trying to develop and review a specification. Once the high-level model has been reviewed, it can be embellished with design details in preparation for translating it into code. Automatic code generation from the detailed design models greatly streamlines the implementation process and removes the chance of introducing translation errors going from the design to the code.

Embedded control systems have traditionally followed the V diagram as a development process. This process leaves all verification and test on the right side of the V, after design and implementation are complete (see Figure 1-2).

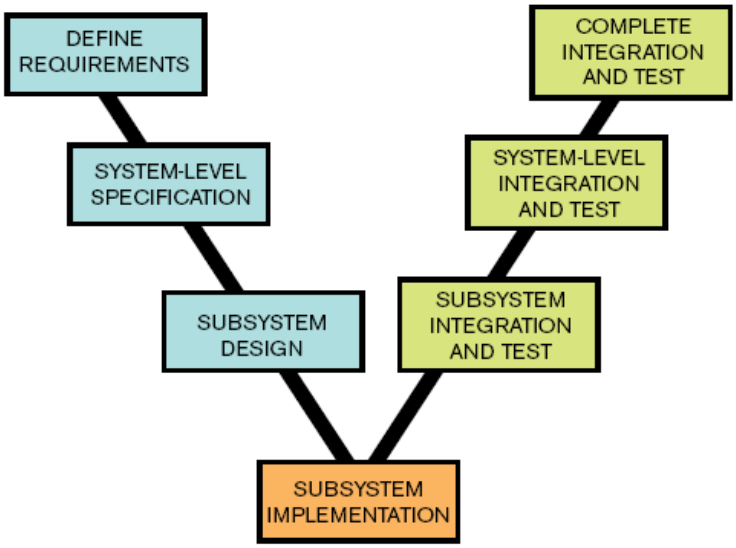

Figure 1-2: V diagram

For a traditional, C-code-based embedded-control-development process, integration testing often precedes other forms of increasingly high-level testing, such as hardware-in-the-loop testing and final system test with the actual system under control.

Although this development sequence has helped organize complex system design, it does have some drawbacks:

- The sequence does not consider verification and test until the end, when it is more expensive and time-consuming to fix any errors
- All components must be implemented to test a system
- It fails to account for iteration in a development process.

Model-based design enables new techniques for verification and validation throughout the development process. Doing test and verification as a parallel activity along every step of the development process means finding errors at their point of introduction. It is possible to reiterate, fix, and verify the design faster than in the traditional V-diagram process (see Figure 1-3). The following sections outline some best practices for achieving early verification.

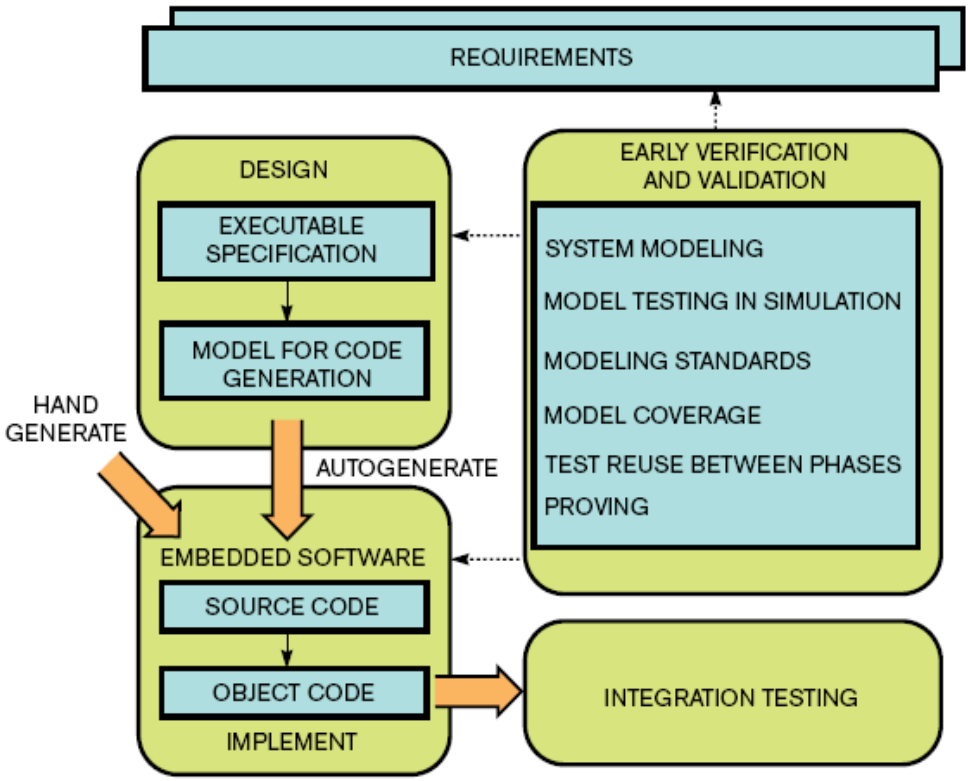

Figure 1-3: Early verification and validation through model based design

Model-based design can be considered a part of Model-based Development (MBD) methodology. MBD is a software development methodology which aims to raise the abstraction level of system specifications and increase automation in system development. It uses models at different levels of abstraction for raising the abstraction level. Automation is achieved by using model transformations: higher-level models are transformed into lower level models. One kind of model transformation is code generation. In the context of MBD, Model Based Testing (MBT) is used to describe all testing activities in the context of MBD. It relates to a process of test generation based on the model of a System Under Test (SUT) [30].

An introduction of MBT is provided in section 2. MBT is the development of testing artefacts on the basis of models. In other words, the models provide the primary information for developing the test cases and test suites, and for checking the final implementation of a system [3]. It is mainly concerned with deriving testing artefacts from models.

Related to Model-based and testing, four dimensions are addressed:

- Test modelling in section 3, which is the specification of the structural and behavioural aspects of the testing software. It is concerned with modelling testing structure, test behaviour and testing artefacts [3]. Here, it must be

mentioned UML 2.0 Testing Profile (U2TP), the UML profile that provides a means to use UML for test specification and modelling.

- The generation of test scenarios using models of the SUT (System Under Test) in section 4. The generation of test scenarios in different scopes is addressed: Unit testing, Integration testing and System testing.
- Traceability between test scenarios and requirements in section 5.
- Variability management in testing in section 6

2. Model-based Testing (MBT)

2.1 Software Testing

Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems [11]. It is one of the most important phases during the software development process with regard to quality assurance [30]. It "can never show the absence of failures" [31], but it aims at increasing the confidence that a system meets its specified behaviour. Testing is an activity performed for improving the product quality by identifying defects and problems [30].

Two of the most important dimensions during testing are test goal and test scope [30].

Test Goal: the software development systems are tested with different purposes. They can be categorized into:

- Static Test, also called review where specifications, models, source code… are reviewed or examined without execution to detect errors.
- Dynamic Test, based on execution
    - Structural Tests: They cover the structure of the SUT during test execution. The internal structure of the system must be known (white-box tests).
    - Functional Tests: Functional testing is concerned with assessing the functional behavior of an SUT against the functional requirements. They do not require any knowledge about system internals (black-box tests
    - Non-functional Tests: Similar to functional tests, they are performed against requirements specification of the system for assessing non-functional requirements such as reliability, load, or performance requirements.

Test Scope: Test scopes describe the granularity of the SUT. Due to the composition of the system, tests at different scopes may reveal different failures. Therefore, they are usually performed in the following order:

- Unit/Component Testing: Unit testing verifies the functioning in isolation of software pieces which are separately testable. Depending on the context, these could be the individual subprograms or a larger component made of tightly related units.

- Integration Testing: Integration testing is the process of verifying the interaction between software components.

- System Testing: System testing is concerned with the behaviour of a whole system.

2.2 Introduction to Model-Based Testing (MBT)

Model-based testing (MBT) is a variant of testing that relies on explicit behavior models that encode the intended behavior of a system and possibly the behavior of its environment. There are several definitions of model-based testing:

- Model-based testing is the generation of executable black-box tests from a behavioral model of the SUT (System Under Test) [4].
- Model-based Testing provides a technique for automatic generation of test cases using models extracted from software artifacts [5].
- Model-based testing is a testing technique where the runtime behavior of an implementation under test is checked against predictions made by a formal specification, or model (Colin Campbell, Microsoft Research).

There are also different approaches known as model-based testing [4]:

- Generation of test input data from a domain model
- Generation of test cases from an environment model
- Generation of test cases with oracles from a behavior model
- Generation of test scripts from abstract tests

In terms of model-based testing, the necessity to validate the model implies that the model must be simpler than the SUT, or at least easier to check, modify and maintain. Otherwise, the efforts of validating the model would equal the efforts of validating the SUT (see Figure 2-1).

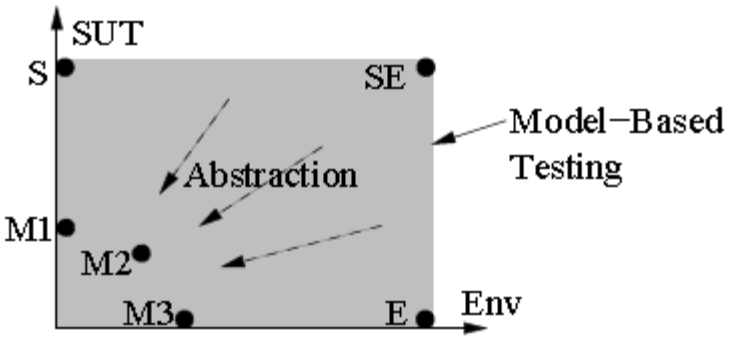

Figure 2-1: Model-based testing uses models of the SUT and its environment [9]

The model describing the SUT is usually an abstract, partial presentation of the system under test's desired behavior. The test cases derived from this model are functional tests on the same level of abstraction as the model. These test cases are collectively known as the abstract test suite. The abstract test suite cannot be directly executed against the system under test because it is on the wrong level of abstraction. Therefore an executable test suite must be derived from the abstract test suite that can communicate with the system under test. This is done by mapping the abstract test cases to concrete test cases suitable for execution [6].

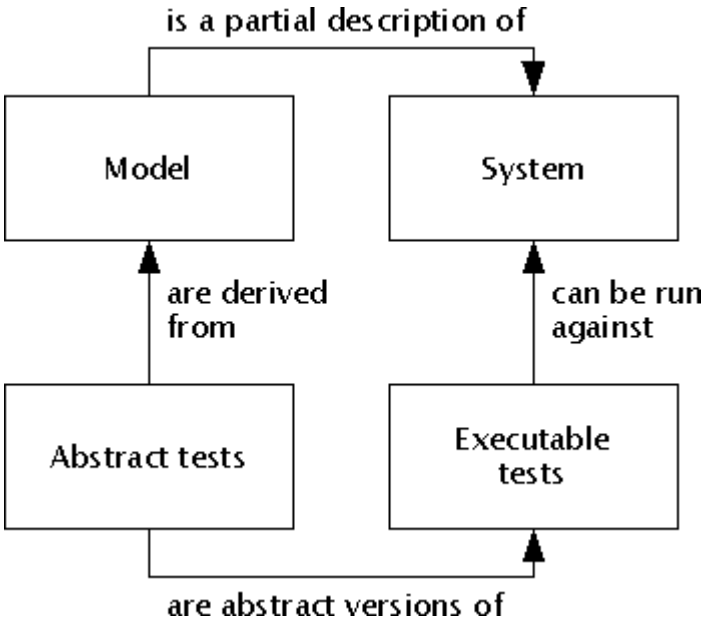

Figure 2-2: Model based testing [6]

2.3 Benefits of Model Based Testing

Some of the benefits of using Model based testing approaches are the following: easy test case maintenance: eases the updating of test suites for changed requirements, reduced costs, shorter schedules, better quality, more test cases: capability to automatically generate many non-repetitive and useful tests, early bug detection, time to address bigger test issues, improved tester job satisfaction, enhanced communication between developers and testers… [7][8]

One of the main reasons for adopting model based testing approaches is the economical one. Cost-effectiveness of using model based testing compared to traditional testing is significant (see Figure 2-3).

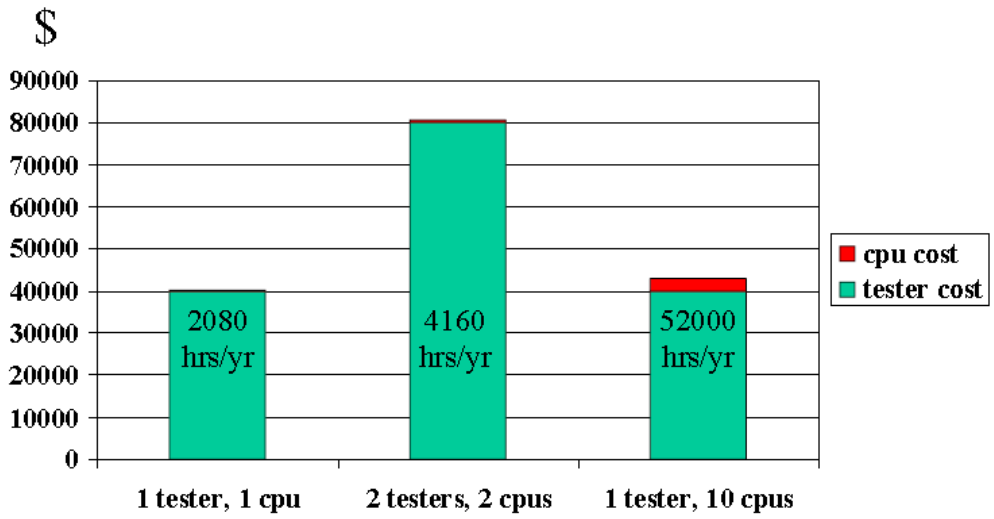

Figure 2-3: Economics of Model-Based Testing (from [7])

Model-based testing can provide a tremendous increase in testing capability but adopting model-based testing also requires an inversion and a cultural change.

2.4 The Process of Model Based Testing

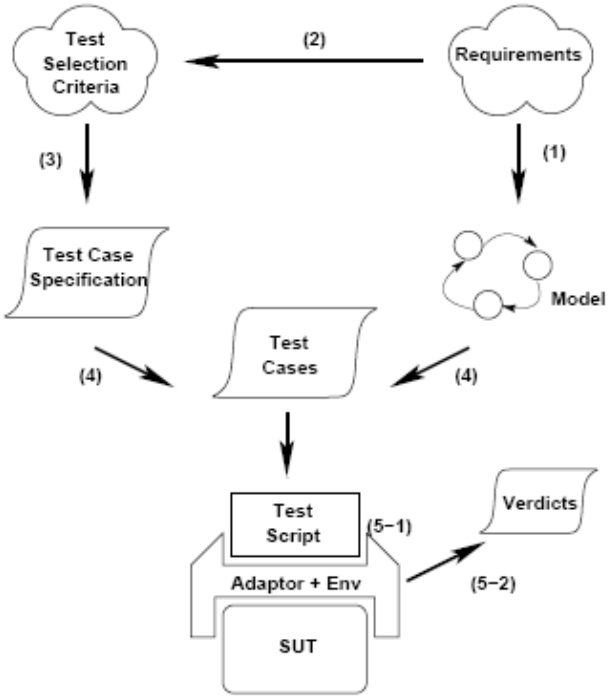

Figure 2-4: The Process of Model based Testing [9]

Model-based testing involves the following major activities [9]: building the model, defining test selection criteria and transforming them into operational test case specifications, generating tests, conceiving and setting up the adaptor component and executing the tests on the SUT.

- Step 1. A model of the SUT is built on the grounds of requirements or existing specification documents. This model encodes the intended behavior, and it can reside at various levels of abstraction.
- Step 2. Test selection criteria are defined. In general, test selection criteria can relate to a given functionality of the system (requirements based test selection criteria), to the structure of the model (state coverage, transition coverage, def-use coverage), to stochastic characterizations such as pure randomness or user profiles, and they can also relate to a well-defined set of faults.
- Step 3. Test selection criteria are then transformed into test case specifications. Test case specifications formalize the notion of test selection criteria and render them operational: given a model and a test case specification, some automatic test case generator must be capable of deriving a test suite (see step 4).
- Step 4. Once the model and the test case specification are defined, a test suite is generated. The set of test cases that satisfy a test case specification

can be empty. Usually, however, there are many test cases that satisfy it. Test case generators then tend to pick some at random.

- Step 5. Once the test suite has been generated, the test cases are run. Running a test case includes two stages.
    - Step 5-1. The test model and SUT reside at different levels of abstraction, and that these different levels must be bridged. Executing a test case then denotes the activity of applying the concretized input part of a test case to the SUT and recording the SUT's output. Concretization of the input part of a test case is performed by a component called the adaptor. The adaptor also takes care of abstracting the output.
    - Step 5-2. A verdict is the result of the comparison of the output of the SUT with the expected output as provided by the test case. To this end, the output of the SUT must have been abstracted.

The verdict can take the outcomes pass, fail, and inconclusive. A test passes if expected and actual outputs conform. It fails if they do not, and it is inconclusive when this decision cannot be made.

A test script is some executable code that executes a test case, abstracts the output of the SUT, and then builds the verdict. The adaptor is a concept and not necessarily a separate software component—it may be integrated within the test scripts.

2.5 Taxonomy of Model Based Testing

In [30], a the taxonomy of model-based testing is presented, this taxonomy is an enrichment of the taxonomy presented in [9].

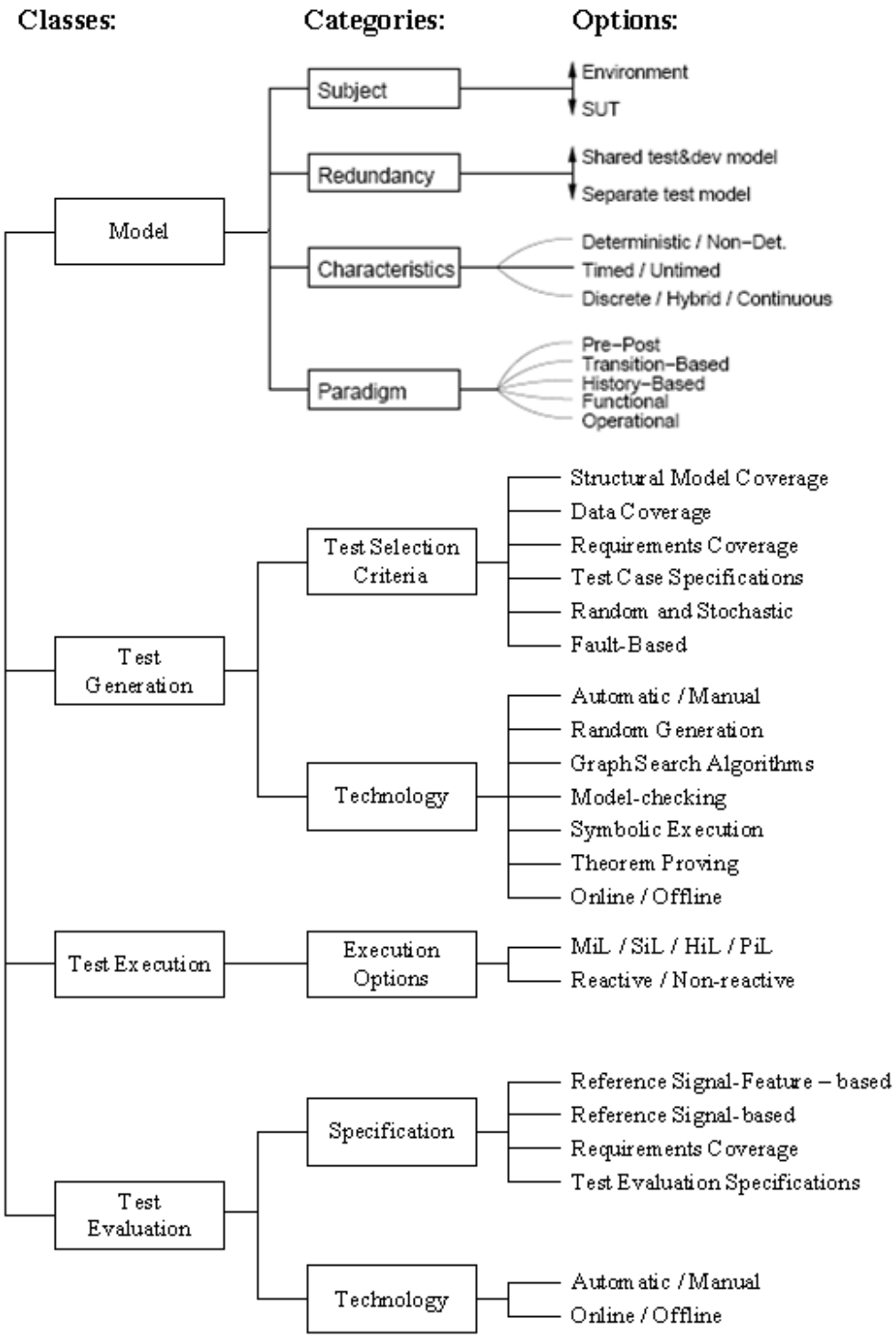

Figure 2-5: A taxonomy of model-based testing [9]

The first dimension is the subject of the model, namely the intended behavior of the SUT or the possible behavior of the environment of the SUT. The model can describe the behavior of the SUT or the external environment of the SUT or both.

Redundancy / Independence reflects the source of the test model. The test model can be a testing-specific model that is built from the specification documents or it can be generated from another model that is used to generate both test cases and code.

Model characteristics relate to nondeterminism, to the incorporation of timing issues, and to the continuous or event-discrete nature of the model.

The fourth dimension is what paradigm and notation are used to describe the model.

The fifth dimension defines the facilities that are used to control the generation of tests. Accordingly, tools can be classified according to which kinds of test selection criteria they support.

The sixth dimension is the technology that is used during test generation.

The seventh dimension is about the execution options for the execution of a test.

The eighth and last dimension are concerned with the test evaluation, also called the test assessment, is the process that exploits the test oracle. It is a mechanism for analyzing the SUT output and deciding about the test result.

2.6 Model based Testing Approaches and Tools

There are several model based testing approaches that can be applied at different testing level: system testing, integration testing, regression testing, component testing, unit testing...

In [10] a survey of model based testing approaches can be found. The approaches are classified depending on the models that use, the testing level, the level of automation, the level of complexity, support tools... Although usage and structural models can be also useful as complement, most of the approaches use behaviour models for generating test cases such as finite state models or state charts.

There are also several model-based testing tools, in the Table 1 some of the most known are mentioned.

Table 1: Model-based testing tools

Tool	Licensing	Modeling Language
Conformiq Test Generator	Commercial	UML Statecharts
LEIRIOS Test Generator – LTG/UML	Commercial	UML 2.0

mbt.tigris.org	Open source	Directed graph
ModelJUnit	Open source	FSM and EFSM in java
NModel	Open source	FSM in c#
ParteG	Open source	State machine
Reactis	Commercial	Mathlab Simulink Stateflow
SpecExplorer	Free	Spec#, Asml
Statemate Automatic Test Generator / Rhapsody ATG	Commercial	Statemate statecharts and UML state machine
TAU Tester	Commercial	TTCN-3
TestOptimal	Commercial, free Community Edition	State diagram
TTModeler (TTWorkbench)	Commercial	The UML Testing Profile (UTP)
T-Vec Tester for simulink	Commercial	Simulink and MATRIXx
ZigmaTEST	Commercial	Finite state machine (FSM)

The previous tools are specific tools for generating test cases from models. However, model based testing will happen in a MDD environment where the SUT is modelled and transformed from higher abstraction models to lower abstraction models. The generation of test cases can be also understood as a kind of transformation. So more MDD generic tools that are used for modelling can also be useful such as Eclipse based tools: Papyrus, TOPCASED, IBM Rational Software Architect, MDDi, etc. as well as transformation tools such as MOFScript, OAW, ATL and Acceleo.

2.7 Model-Based Testing and Agile Methods

Model-Based Testing can fit into agile processes as Extreme Programming (XP) or SCRUM.

Test-Driven Development (TDD) is one of the most important testing practices of agile methods. TDD consists on writing unit tests for a component before writing the implementation code of the component, then running those tests frequently as the code is developed and refactored (the test, code, refactor cycle) [4]. TDD is used for unit testing.

Model-Based Testing can also be used for Unit Testing. MBT offers the possibility of generating a suite of unit tests from a small model, which may reduce the cost of developing the unit tests, give deeper understanding of the desired behaviour, and allow more rapid response to evolving requirements [4].

TDD is one of the approaches that is proposed for unit testing in eDIANA (in section 4.1). And MBT can be performed in TDD way [33] or in a complementary way with TDD [34].

3. Test Modelling

In a Model Driven Development approach (see *Figure 3-1*), the System Under Test must be modelled using models, those models will be of different levels of abstraction and transformations will be used from going from one level to another. Those models will be the input for generating the test cases (this is another kind of transformation: from SUT model to test model). Those test models must be also modelled.

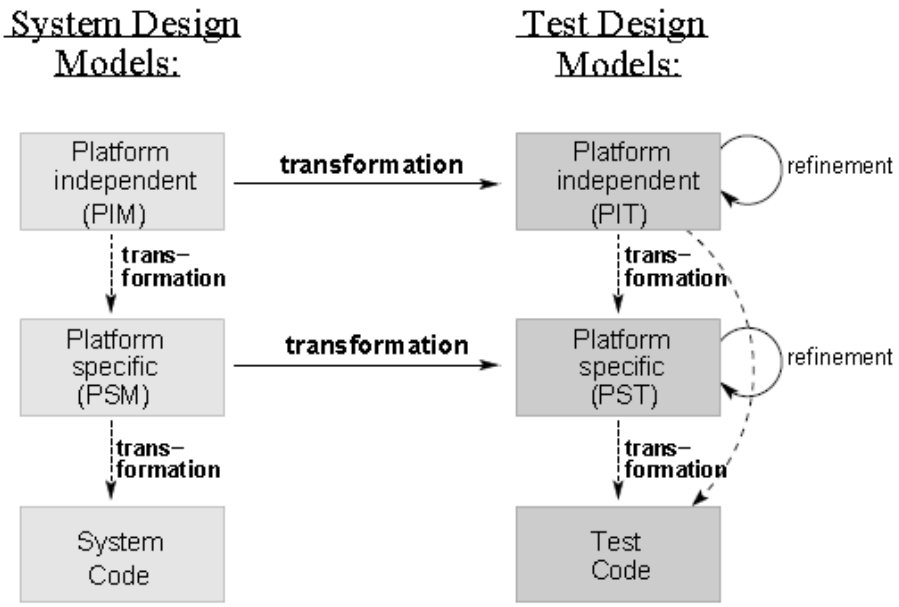

Figure 3-1: System Design Models vs. Test Design Models (from [27])

As mentioned previously, one of the most common software development processes is the well-known V model. This model can be extended for the development of the test system: it is called W-model [25], see Figure 3-2. As mentioned in introduction, Model-based design enables new techniques for verification and validation throughout the development process instead of applying the traditional V-diagram process. However, the V-model and W-model are used with the purpose of explaining the role of test modelling in the software development process.

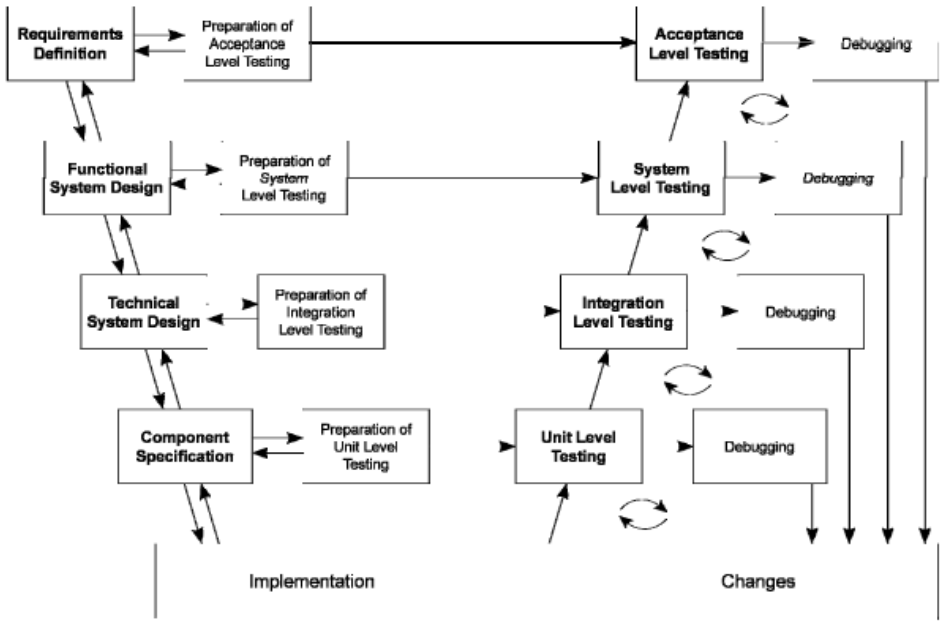

Figure 3-2: The W-model (from [26])

SUT models can be described using different languages and notations. For example (see Figure 3-3), UML can be used for SUT models, for specifying the test model the UML testing Profile (UTP) can be used and for execution the TTCN-3 (Testing and Test Control Notation).

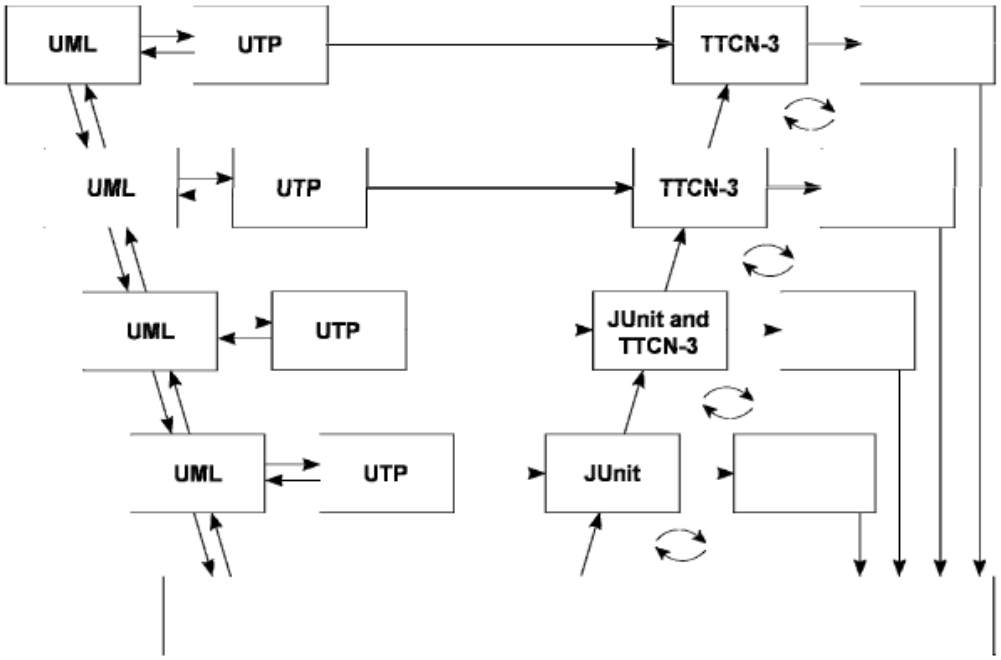

Figure 3-3: UML focused W-model (from [26])

In this chapter, two of the most used formal test notations for specifying test cases are described:

- U2TP, UML 2.0 test profile, which provides means to use UML both for system modelling as well as test case specification.
- TTCN-3, the Testing and Test Control Notation, which is a standardized language to formulate tests and to control their execution.

3.1 UML 2.0 Testing Profile (U2TP)

Not only is the Unified Modelling Language (UML) one of most used software development technique, but since its version 2.0 it can also be applied for testing.

U2TP (UML 2.0 Testing Profile) [17] provides the definition of a testing profile to capture all information that would be needed to specify test goals, test procedures and test assessments for system components as well as for complete systems.

With U2TP system models and test models can be developed and aligned in all system development phases. This profile addresses the classic testing concepts such as test cases, test configuration or test results. Moreover, this testing profile enables:

- Static test definition and test generation based on structural aspects of UML models
- Dynamic test definition and test generation based on behavioural aspects of UML models
- The inter-operation with test technologies for black-box testing, where the internal structure of the SUT remains hidden.

So, these functionalities may be divided in four building blocks:

- **Test architecture**: Specifying test structure and test configuration.
- **Test data**: Specifying types and values involved in a test.
- **Test behaviour**: Specifying test cases and their associated behaviours.
- **Test time**: Specifying concepts for a time quantified definition of test procedures.

3.1.1 Test architecture

This aspect specifies test components and classes to provide test configurations. The concepts needed to describe the elements that the test cases defined using the profile are shown in the next figure, and are explained below:

- **SUT**: system under test. The SUT is not specified as part of the test model, and the test architecture package imports the UML model of the SUT. From the point of view of black-box testing, internal information is not available for use in the specification of test cases using the Testing Profile. So, the SUT provides only a set of operations via publicly available interfaces.

- **Test Configuration**: This element contains all the test component objects and their connections to the SUT. It defines both the initial test configuration and the maximum number of test components objects and connections that might be added during the test execution.

- **Test context**: It contains a test configuration and a collection of test cases being executed on the test configuration. Test components interact with the SUT to realize the test cases defined in this test context and fulfil the test objectives.

- **Arbiter**: It is a predefined interface provided with the Testing Profile. Its objective is to determine the final verdict for a test case. This determination is done according to a particular arbitration strategy, which is provided in the implementation of the arbiter interface. It is a passive component, except for reporting the test case verdict at the conclusion of each test case. Every test context must have an implementation of the arbiter interface.

- **Scheduler**: It is a predefined interface provided with the Testing Profile. Its objective is to control the execution of the different test components. The scheduler will keep information about which test components exist at any point in time, and it will collaborate with the arbiter to inform it when it is time to issue the final verdict. It keeps control over he creation and destruction of test components and it knows which test components take part in each test case. Every test context must have an implementation of a scheduler.

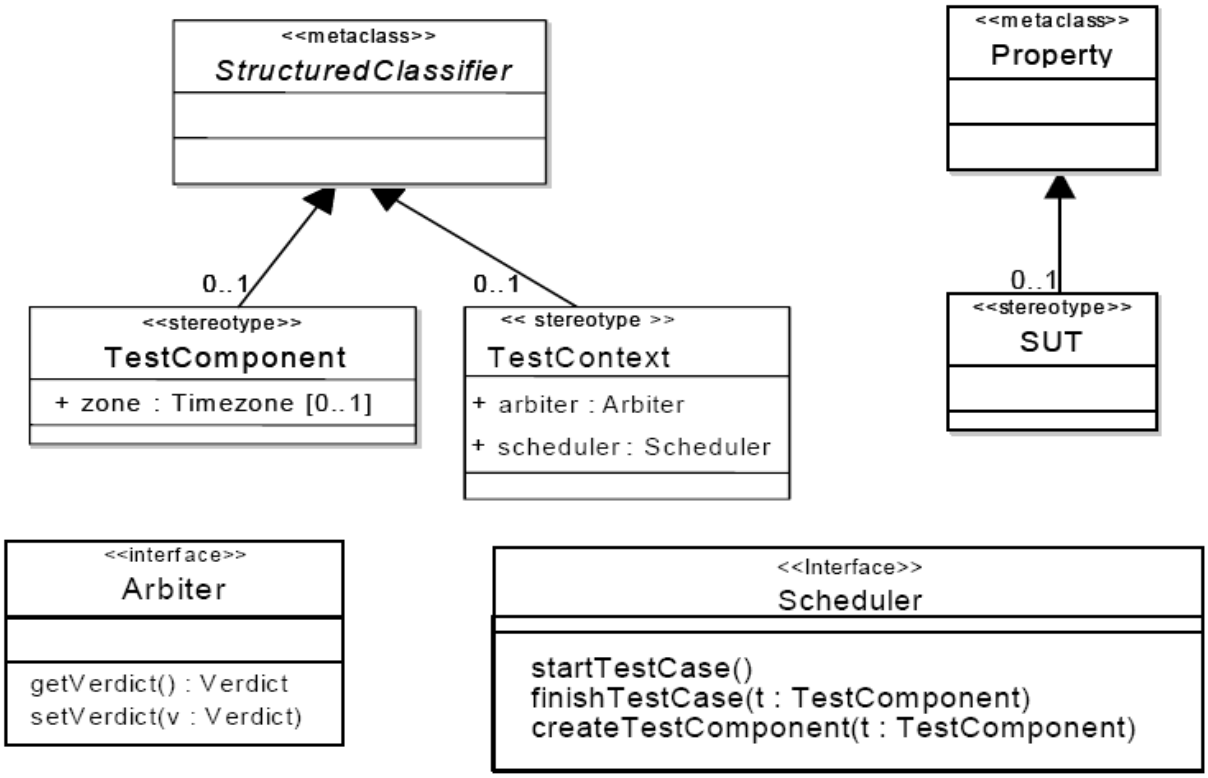

Figure 3-4 U2TP Test architecture

3.1.2 Test behaviour

The area of test behaviour describes the set of concepts required to specify test behaviours, their objectives, and the evaluation of SUT. It might be described in different forms, such as sequence diagrams, sequence activity graphs or state machine

The main elements of test behaviour are:

- **Test objective**, which describes what should be tested.
- **Test case**, which is a complete technical specification of how the SUT should be tested for a given test objective.

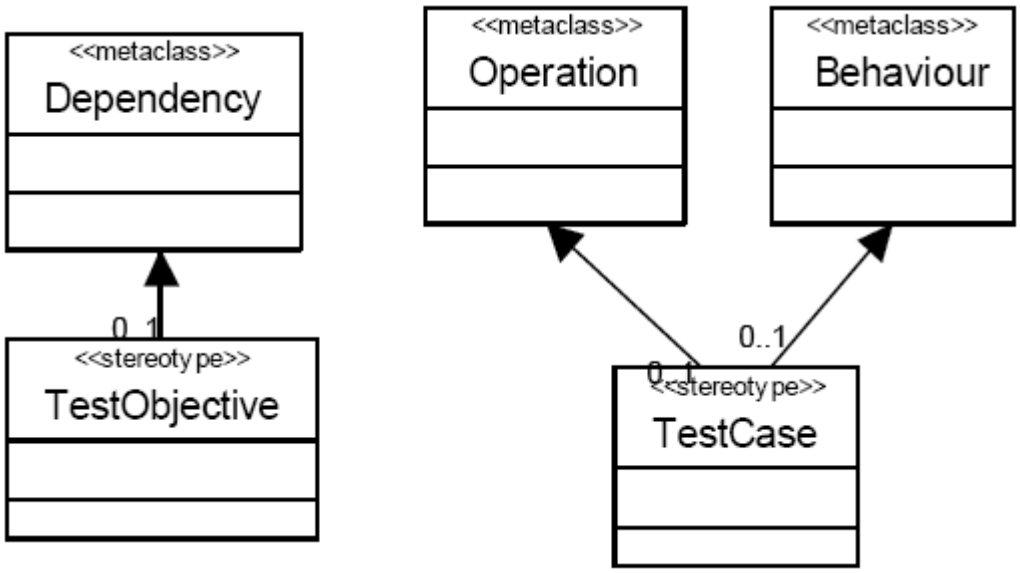

Figure 3-5 Main elements of Test Behaviour

Apart from the scope driven by the test objective, the test case must include inputs, results and test conditions, and it may invoke other test cases. So, it is defined in terms of sequences, alternatives, loops, data sent to the SUT (stimulus) and data receive from the SUT (observation).

As UML does not necessarily specify every possible trace of execution, and there is a need to have complete definitions in the area of testing, the concept of default is introduced. It is a behaviour triggered by a test observation that is not handled by the behaviour of the test case per se. Defaults are executed by test components. The reason for designing with defaults rather than making sure that the main description is complete, is to separate the most common and normal situations from the more esoteric and exceptional. The distinction between the main part and the default is up to the designer and the test strategies.

Moreover, the test case is a property of a test context, so three more concepts should be included here:

- **Test Control**: it is a technical specification for the invocation of test cases. Its objective is to know how the SUT should be tested with the given test context.

- **Test Invocation**: a test case can be invoked with specific parameters and within a specific context. The test invocation leads to the execution of the test case. The test invocation is denoted in the test log.

- **Test Log**: Traces from test context and test cases can be recorded as test logs, becoming part of the test specification.

On other hand, a test case uses an arbiter, described in the previous section, in order to evaluate the outcome of its test behaviour. The arbiter determines a **verdict.** The verdict indicates how has been performed the test case. It must include at least the following values:

- A pass verdict indicates that the test case is successful and that the SUT has behaved according to what should be expected.
- A fail verdict shows that the SUT is not behaving according to the specification.
- An inconclusive verdict means that the test execution cannot determine whether the SUT performs well or not.
- An error verdict tells that the test system itself and not the SUT fails.

Finally, the Testing Profile has defined a few action utilities to help in the definition of test behaviour:

- **FinishAction** completes the test case for one test component. The action has no implicit effect on other test components involved in the same test case, but it has recognized the need for other test components to be notified of the finish such that they may no longer expect messages from the finished test component. This must be specified explicitly.
- **LogAction** indicates that information about the test should be recorded for the test component performing the action.
- **determAlt** is an interaction operator and is an alternative where the operands are evaluated in sequence such that it is deterministic which operand is chosen given the value of the guards, regardless of the fact that the guard for more than one operand may be true.

3.1.3 Test data

This section contains concepts additional to UML data concepts needed to describe test data. In other words, test data specifies the types and values sent to or received from the SUT. It covers:

- **Wildcards**: They are literals and denote an omitted value, any value, or any value or omit. They are typically used for a loose specification of data exchanged with the SUT.

- **Data Partition**: It is a stereotyped classifier used to define an equivalence class for a given type. Its purpose is to provide a more visible differentiation of data. A data partition must be associated with a data pool.

- **Data Pool**: It provides a means for associating data sets with test contexts and test cases. So, a data pool is a classifier containing either data partitions or explicit values; and can only be associated with a test context or test components.

- **Data Selector**: They are operations that performance over the contained values or value sets. They may be related to a data pool or a data partition in order to make easier the different data selection strategies.

- **Coding Rules**: They specify how values are encoded and/or decoded. Coding rules are shown as strings referencing coding rules defined outside the Testing Profile such as by ASN.1, CORBA, or XML**.**

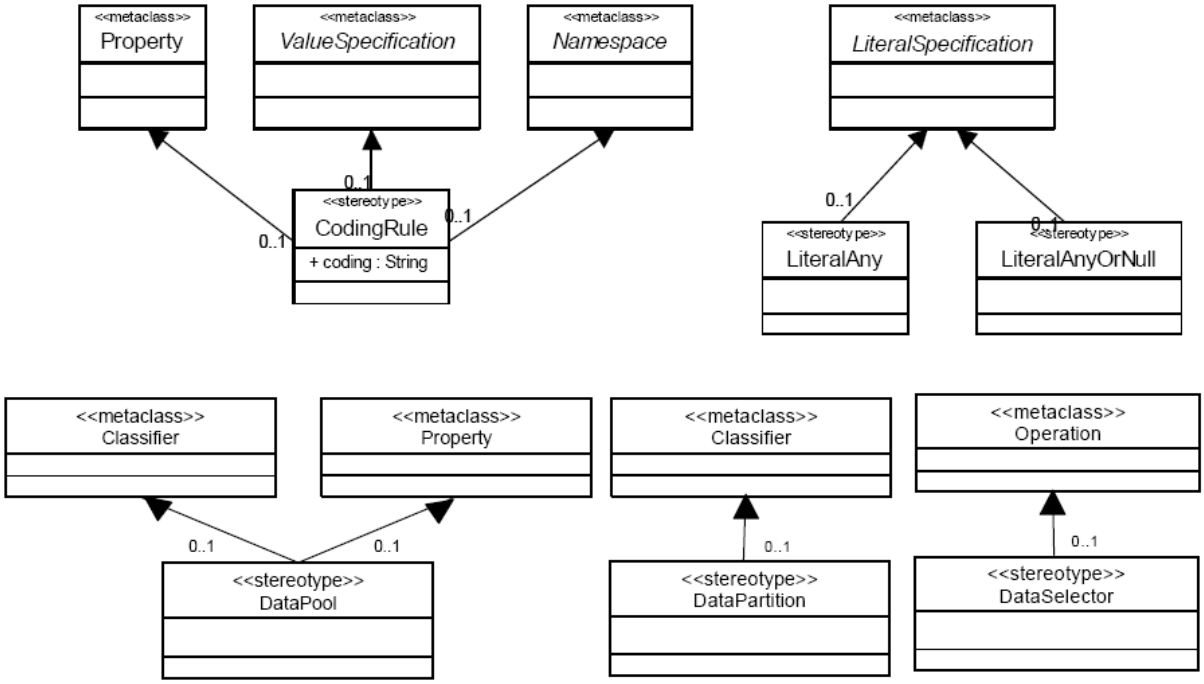

Figure 3-6 Test data

3.1.4 Test time

As UML time concepts do not fulfil the requirements of a test specification, the U2TP provides a small set of useful time concepts. The main time concepts are timers and timezones.

On the one hand, timers are predefined interfaces which cope with manipulate and control test behaviour, or even assure that a test case terminates successfully. The timer interface defines operations such as

- "Start" specifies the value to star a timer.
- "Stop" is able to stop an active timer.
- "Read" provides the expiration time of an active timer.

On the other hand, timezones are grouping mechanisms within a distributed test system. Each test component belongs at most to one timezone. Besides, test components belonging to the same timezone are considered to be time synchronized. The time zone of a test component can be accessed both in the model and in run-time.

3.1.5 Test implementation

Once the building blocks of U2TP have been explained, a test implementation environment is needed. U2TP provides two mappings towards test execution environments: JUnit and TTCN-3.

Junit is an open source unit testing framework for the Java programming language. It has been popular in the development of test-driven development.

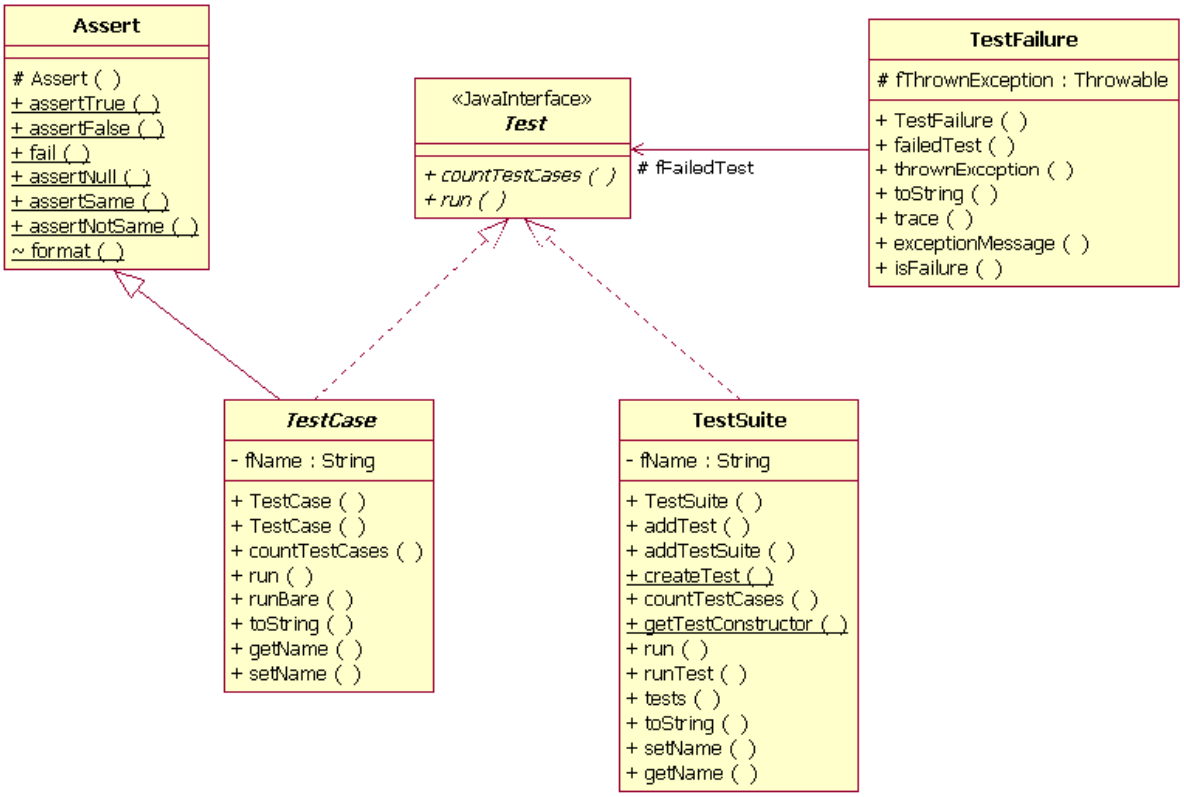

Figure 3-7 JUnit framework

TTCN-3 is another standard for test system development, although it is usually focused to telecommunication and data communication area. It will be detailed in the following section. The great majority of UML 2.0 Testing Profile specifications can be represented by TTCN-3 modules and executed on TTCN-3 test platforms as stated in *Model-based testing with UTP and TTCN-3 and its application to HL7* [19].

3.2 Testing and Test Control Notation version 3 (TTCN3)

TTCN-3 *(Testing and Test Control Notation version 3)* [18] is a strongly typed test scripting language used in the translation of test specification into an executable representation. This powerful test specification and implementation language has been developed by ETSI (European Telecommunications Standards Institute).

It is manly used to define test procedures for black-box testing of distributed systems, because of the following advantages:

- Because of its well defined syntax, TTCN-3 provides an unambiguous specification and execution of tests. It enables completely automated test execution.
- It is easy to learn, because it looks like a regular programming language.

- It is very flexible, it is not tied to particular application, test execution environment, compiler or operation system.
- It is harmonized with ASN.1, and future developments are supposed to include XML and IDL.

The TTCN-3 test suite may be divided into these four building blocks, described below:

- Test Data types
- Actual Test Data
- Test Configuration
- Test Behaviour

3.2.1 Test Data types

TTCN-3 provides attributes for encoding, display or user-defined information. There are data and signature templates with powerful matching mechanisms, including regular expression.These data types specify

- Structure of messages or calls and their information elements (fields, parameters)
- Internal data structures that may be used for several purposes, for instance, computation
- Possibly encoding or display information

Moreover, it includes the following built-in types:

- Built in basic types such as integer, boolean, float, bitstring, hexstring, octetstring, charstring or universal charsrting
- Built-in structured types such as record, record of, set, set of, union or enumerated
- Built-in special types such as component, port, verdict type, or default

3.2.2 Actual Test Data

Not only data types are provided, but values of actual test data are available during testing, such as constants or templates for specific message or call parameter values. It enables using also template decomposition, test suite parameterization and modification.

Some matching expressions for allowing multiple messages or call parameter values are included:

- Value range and value list
- Wildcards
- Presence
- Length and size
- Permutation
- Regular expressions

3.2.3 Test Configuration

The test configuration block includes some static aspects related to test component and port types.

But the interesting aspect of this block is the possibility of dynamic concurrent testing configurations:

- Dynamic instantiation and management of test components
- Mappings of test components to abstract test system interfaces
- Connections between test component interfaces
- Management of test components

3.2.4 Test Behaviour

TTCN-3 focus only on implementation to be tested and it ensures the control of complex test configurations. The concept of verdict and verdict resolution mechanism

is included in the control of Test Case execution and selection mechanisms. The main aspects of test cases are:

- They specify sending/receiving messages, computation (e.g., checksums), and verdict assignment and handling
- Decomposition with functions and altsteps
- Reuse of default behaviour
- Use of timers and timeouts

Optionally, test execution control is available, in order to specify order, repetitions or conditions. Besides, various communication mechanisms are contemplated, synchronous as well as asynchronous.

3.2.5 TTCN-3 test system architecture

TTCN-3 specifies a test but a test system is needed for test execution. The typical TTCN-3 test system architecture consists of:

- TTCN-3 Executable (TE): execution core that runs test cases. This compiler and execution environment is general for every system.
- TTCN-3 Runtime Interface (TRI).
- SUT Adapter (SA): implementing TRI SA interface that is responsible for network interface code.
- Platform Adapter (PA): implementing TRI PA interface that is responsible for timers and external functions.
- TTCN-3 Control Interface (TCI) between Test System Executor and Test Management.
- TTCN-3 Test Management including Test Control, Logging and Coding and Decoding.

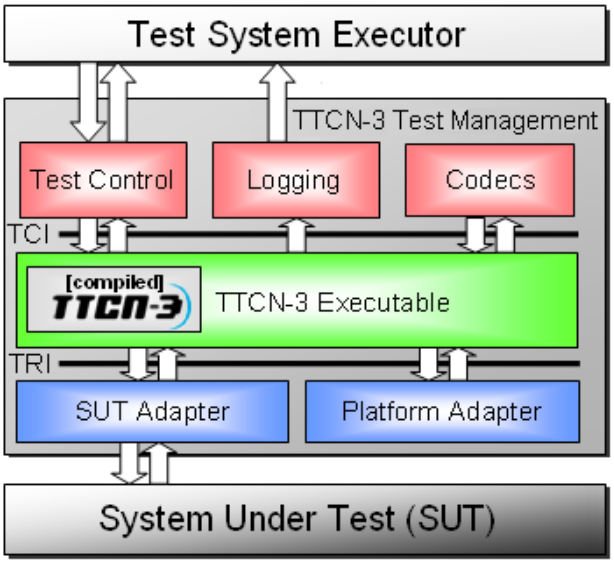

Figure 3-8 TTCN-3 typical architecture

3.2.6 eDiana Platforms TTCN-3 Compliance

As introduced in previous chapters TTCN-3 testing protocol is focused in automation and quality assurance compliant communications environments. The eDiana platform, covering a very heterogeneous device integration process, is a perfect case in which the TTCN-3 standard could be fully applicable.

The reference architecture describes all the communication interfaces present in the whole eDiana platform.

The picture below, extracted from the eDiana Reference architecture, summarizes the most relevant components and their communication interfaces.

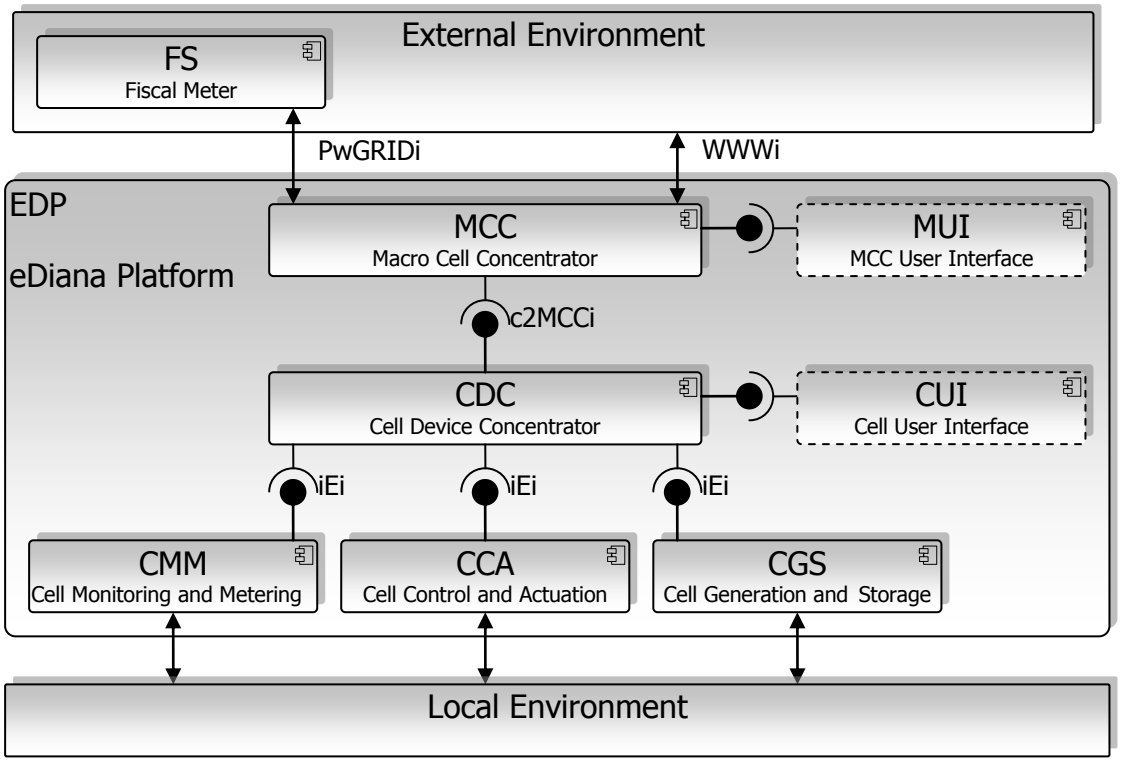

Figure 6 eDiana Platform Communication Interfaces

The most relevant interfaces in the eDiana platform regarding to the specific eDiana developments are:

- iEi Interface
- c2MCCi Interface
- PwGRIDi Interface

3.2.6.1 iEi Interface

The iEi interface is the generic interface among the Cell level concentrator and the field devices. The full functionalities for the iEi interface are described in WP03 which in charge of the development of the cell level and its components.

The mapping among the iEi interface to test and the TTCN-3 standard suggested testing architecture main components would be:

- SUT: The field device including the iEi interface.

- TTCN-3 Executable: Main executable developed to act as the cell level concentrator and communicating with the field devices.
- SUT Adapter (SA): Communication layer or API to be used by the TTCN-3 Executable.
- Platform Adapter (PA): Run time engine developed to act as communication scheduler.

3.2.6.2 C2MCCi Interface

The c2MCCi interface is the generic interface among the Cell level concentrator and the Macrocell. The full functionalities for the c2MCCi interface are described in WP04 data gathering component.

The mapping among the c2MCCi interface to test and the TTCN-3 standard suggested testing architecture main components would be:

- SUT: The hardware device acting as Cell level concentrator.
- TTCN-3 Executable: Main executable developed to act as the Macrocell level concentrator and communicating with the cell concentrator devices.
- SUT Adapter (SA): Communication layer or API to be used by the TTCN-3 Executable.
- Platform Adapter (PA): Run time engine developed to act as communication scheduler.

To test the interface in its fully operation the reverse approach would have to be done, this is, the SUT would be the Macrocell level and the TTCN-3 executable would simulate the Cell level layer behavior.

3.2.6.3 PwGRIDi Interface

The PwGRID interface is the generic interface among the Macrocell level concentrator and the power grid domain. The full functionalities for the PwGRID interface are described in WP04 data gathering component.

The mapping among the PwGRID interface to test and the TTCN-3 standard suggested testing architecture main components would be:

- SUT: The hardware device acting as MacroCell level concentrator.
- TTCN-3 Executable: Main executable developed to act as the Power Grid domain.

- SUT Adapter (SA): Communication layer or API to be used by the TTCN-3 Executable.

- Platform Adapter (PA): Run time engine developed to act as communication scheduler.

The TTCN-3 layer and eDiana components mapping, in the current document, has been done only for illustrative purpose. More detailed approach could be done, for the above mentioned three interfaces, but it is out of the scope of the present document.

The Verification and Validation mechanisms to adopt by the eDiana platform will described in the verification and validation procedures.

4. Generation of test scenarios from models

In this section, the generation of test scenarios from models for different test scopes is addressed.

As mentioned in section 2, software testing can be performed at different levels/scopes along the development: Unit Testing (the target of the test is a single module), Integration testing (the target of the test is a group of modules) and System testing (the target of the test is the whole system) [11].

- Unit Testing: Unit testing verifies the functioning in isolation of software pieces which are separately testable. Depending on the context, these could be the individual subprograms or a larger component made of tightly related units.

- Integration Testing: Integration testing is the process of verifying the interaction between software components.

- System Testing: System testing is concerned with the behaviour of a whole system.

4.1 Unit testing

The Unit Testing is lowest testing level regarding to functional complexity involved, due to it is focused in testing individual functionalities and no the integration or interaction among them.

Before go into the details of subject topic, it is convenient to discuss a little about software engineering, SDLC (Software Development Life Cycle) and OOSD (Object Oriented Software Development)[1][2].

SDCL, also known as "Macro Process", shall be divided into different phases logically and carefully. These SDLC phases broadly include:

- Vision – Conceptualization of the software domain;

- Define – Detailing the requirements along with specification – Use cases;

- Design – Realization of the Use Cases;

- Develop – Realization of the design leads to software development / Coding;

- Test – Enforcing the use cases while Testing.

The iterative and incremental behaviour of the software process is reflected in the following picture.

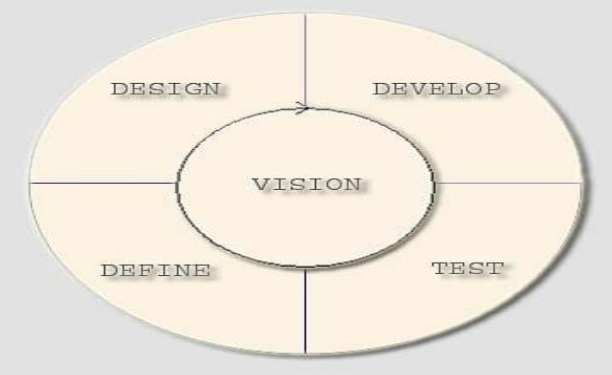

Figure 4-1: Phases of SDCL

In the OOSD (Object Oriented Software Development) or Object Oriented Paradigm, when analyzing a problem domain or analyzing requirements, actually is looking for objects/abstractions that will solve some particular problem in the software. To introduce the usage of MDD or TDD regarding to the different Early V&V processes under the eDiana project domain, both will be discussed in the OOSD.

The first approach for MDD is to take into account the following division:

- objects/abstractions will be names/nouns
- behaviour will be represented by verbs/actions.

Next step is fill the gaps of how the abstraction will look like, and go deeper in analyzing the requirements, and come up with some handful of use cases. Those actually give an idea of functional aspects of the objects, or provide interfaces, called object contracts.

So, the outcome of the use cases are the interfaces, provide the means of testing the contracts. This practice along with scenarios leads to find interactions between objects, in order to design the solution. This process leads a static design which shows how the interfaces, and then the concrete implementations, would look like and what the interdependencies among those objects are.

This approach was bottom up: it is necessary to find the objects, then the components and then the domains. During this categorization process, some new and unknown areas appear. Those areas come across right in the beginning of the objects / components discovery and we'll realize that we might have incomplete

picture of the design. It will become much more obvious when we discover the rest of the domains.

At this point then, Test Driven Development (TDD) approach becomes feasible and reasonable to consider for Early V&V process.

TDD, also known as Agile Process or RAD Process, is both Iterative and Incremental. As stated before, the unit testing is the lowest testing level. Under this domain, the TDD becomes the strategy to follow to face the Early V&V process in conjunction with the development. It does not replace traditional testing, instead it defines a proven way to ensure effective unit testing: instead of writing functional code first and then testing the code as an afterthought, development staking a TDD approach refuses to write a new function until a failed test exists for it

The steps of TDD are overviewed in the UML activity diagram shown below. The first step is to quickly add a test, basically just enough code to fail, run tests, then, and update the functional code to make it pass the new tests.

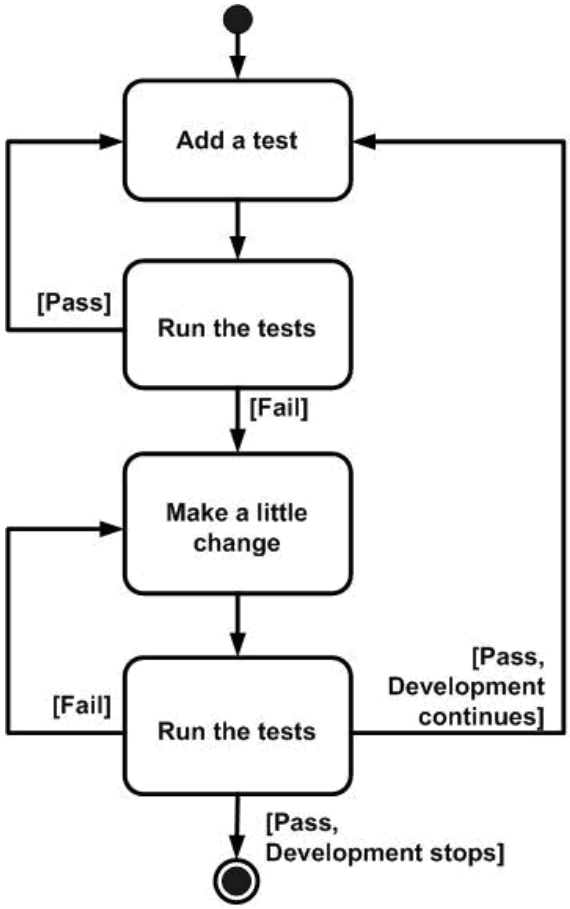

Figure 4-2: TDD steps

It is important to integrate model based testing with development processes and to reuse models from the design processes where possible. The facilities provided by UML are ideally placed to capture functional requirements. Tests generated from low level models can work directly on the implementation, such as JUnit or NUnit. However, the tests must be expressed in terms of implementation detail. The essential idea of model based testing is to compare an abstract specification to a concrete implementation. Tests generated from models that describe high-level functional requirements and associated information structures change much more slowly than design models.

The information content of a system is often expressed as UML class models. Given a relationship between the information model (logical system view) and the implementation (physical system view) then the information model can be used to generate individual operation tests in terms of the correct changes to the information states expressed as pairs of snapshots.

Furthermore, given behavioural models, such as state machines, it is possible to construct tests in terms of sequences of operations and the required information states. Sequences of snapshots produced by operations in this way will be referred to as filmstrips.

The unit testing modelling for the eDiana platform based on the technique of the filmstrips relaying on the TDD methodology, is foreseen as the most quality assurance compliant and flexible way to face the platform development in a collaborative development scenario in which the development, testing and deployment task are very loosely coupled.

4.2 Integration and System testing

Unit testing focuses on testing a unit of the code whereas Integration testing is the next level of testing, it focuses on testing the integration of units of code or components and System testing is concerned with the behaviour of a whole system.

Integration testing is also known as integration and testing (I&T). Classical integration testing strategies, such as top-down or bottom-up, are used with traditional, hierarchically structured software. Modern systematic integration strategies are rather architecture-driven, which implies integrating the software components or subsystems based on identified functional threads. Integration testing is a continuous activity. Except for small, simple software, systematic, incremental integration testing strategies are usually preferred to putting all the components together at once, which is pictorially called "big bang" testing [11].

In embedded systems, software can be deployed in a complex real environment. And to test the software in this real environment can be very expensive. Moreover, early validation of a part or the whole system is necessary in order to detect problems as soon as possible and before the final deployment.

This is the case of eDiana, where each of the eDiana installations (at houses, flats, offices) will involve different devices, configurations and software.

Simulation using simulators or emulators of real-world devices can allow early software validation. Simulink [12], developed by The MathWorks, is a commercial tool for multidomain simulation and Model-Based Design for dynamic and embedded systems. It must be mentioned the existing synergy among tool suppliers in embedded system sector, which forms a "de facto" standard that is very used: Matlab/Simulink-dSPACE-Autosar [28]. A similar tools combination is described at [29], mentioning among others:

- Matlab's Simulink and Stateflow (http://www.matlab.com) for designing the system ant its components.
- DSpace's TargetLink (http://www.dspace.com) used for reusing and automatic code generation from Matlab models.
- Tessy (of Hitex supplier) used for automated unit testing.
- CTE (Classification Tree Editor, Hitex) for configuring testing based on the system's input domain.
- Time Partition Testing, for generating test cases with continuous data flows.
- DSpace's MTest for automatic generation of test cases from Simulink and TargetLink (http://www.dspace.com) models.
- QA-C/Misra can be used for analyzing the resultant C code (http://www.qasystems.de).
- PolySpace, tool that can detect potential errors in run-time during compilation time (http://www.polyspace.de)
- Mercury's Quality Center, which is a global set of tools for assuring the quality of software (http://www.mercury.com/).

The model-based development process of embedded systems usually occurs on at least three different levels. First a model of the system is built. It simulates the required system behavior and usually represents an abstraction of the system. When the model is revealed to be correct, code is generated from the model. This is the software level. Eventually, hardware including the software is the product of the development [30]. The multiple V-model [32], based on the traditional V-Model, takes this phenomenon into account. In the multiple V-model, each specification

level (e.g., model, software, final product) follows a complete V-development cycle, including design, build, and test activities (see Figure 4-3).

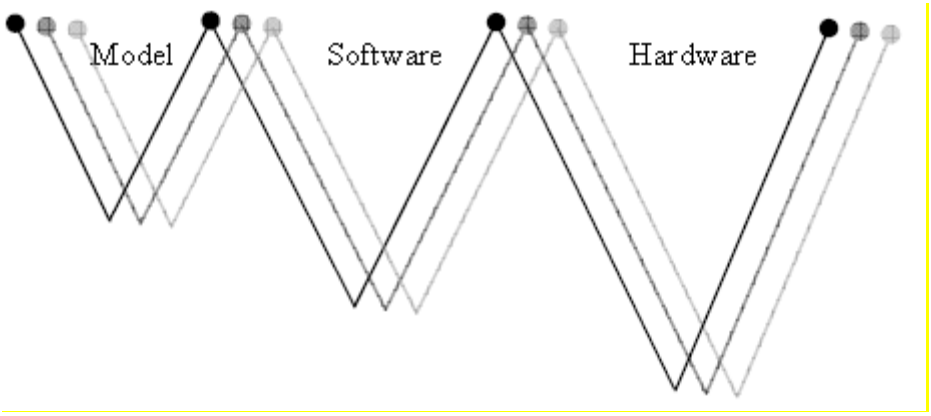

Figure 4-3: The multiple V-model

Related to those different levels, different test platforms are required [30]:

- Model-in-the-Loop (MiL): The first integration level, MiL, is based on the model of the system itself. In this platform the SUT is a functional model or implementation model that is tested. Model exists entirely in native simulation tool (e.g., Simulink / Stateflow). The test purpose is basically functional testing in early development phases in simulation environments.
- Software-in-the-Loop (SiL): During SiL the SUT is software tested. The software components under test are usually implemented in C and are either hand-written or generated by code generators based on implementation models. Part of the model exists in native simulation tool (e.g., Simulink / Stateflow), and part as executable C-code (e.g., S-function). The test purpose in SiL is mainly functional testing.
- Processor-in-the-Loop (PiL): In PiL embedded controllers are integrated into embedded devices with proprietary hardware (i.e., ECU). Testing on PiL level is similar to SiL tests, but the embedded software runs on a target board with the target processor or on a target processor emulator. It is the last integration level which allows debugging during tests in a cheap and manageable way.
- Hardware-in-the-Loop (HiL): When testing the embedded system on HiL level the software runs on the final ECU. However the environment around the ECU is still a simulated one.
- System: Finally, the last integration level is obviously the system itself.

4.2.1 System Testing Example using Simulink

Using Simulink it is possible to validate the software before deploying it in its real environment. With Simulink it is possible to create and model block diagrams of the system. These blocks can be mechanical/physical/hardware elements (devices) or software that is embedded in a block, so simulation model include the simulation of both mechanicals and hardware elements or devices and software that manages these elements.

For being able to perform system validation (at simulation level), the following phases are performed:

1. Generate the input (test cases) for the simulation
2. Create the simulation model (.mdl).
3. Simulate and analyze the results

1. Generation of test cases

The ModelJUnit [16] tool can be used. It is an open source tool, it allows time annotations, it uses a transition-Based notation (FSM) and it has test generation algorithms for random generation (random walk) and metrics for structural coverage. And it can be used for online and offline testing.

ModelJUnit is a java library that extends JUnit to support Model Based Testing. ModelJUnit allows writing simple finite state machine (FSM) models or extended finite state machine (EFSM) that can take into account time aspects (TimedFsmModel).

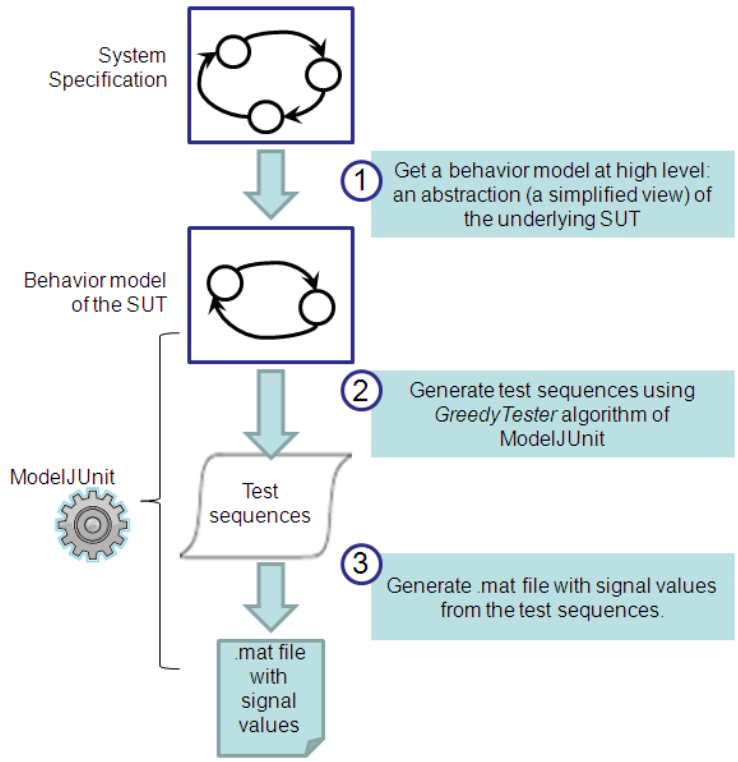

Figure 4-4: Test generation process

The steps for generating test/simulation sequences for simulink are the next ones (see Figure 4-4):

1. Define EFSM model of the behaviour of the SUT starting from the system specification. This model reflects the behaviour of the system at high level; it is an abstraction (a simplified view) of the underlying SUT.
    a. The transitions in this model can be signal driven (as it is an embedded system).
    b. Time annotations and timeouts that trigger transitions must be needed. So TimedFsmModel can be used and timeouts specified.

    For obtaining the behaviour model (EFSM) at high abstraction level, the specification models of the software (state machine models) can be the basis and starting from them, the models are abstracted and simplified in order to represent the behaviour of the systems (including transitions triggered by signals).

2. The *greedyTester* algorithm can be selected and applied to generate test sequences offline. This algorithm tests a system by making greedy walks through an EFSM model of the system. A greedy random walk gives preference to transitions that have never been taken before. Once all

transitions out of a state have been taken, it behaves the same as a random walk. This way, high transition coverage is obtained. Metrics for coverage may be also obtained.

3. Code must be added to write .mat file with signal values from the generated test sequence messages. This way, the input for simulink simulation scenario is obtained.

2. Create the simulation model (.mdl)

For creating the simulation model: both software and devices must be considered:

- Software-Under-Test is transformed into SFunction blocks to integrate in the simulation model
- And the blocks that simulate the devices are modelled.

3. Simulate and analyze the results

The execution of simulation and analysis can be manual or automatic. For performing automatic testing, it is necessary to create an oracle that provides a reference output, for checking test results, in a given test, and accordingly produces a verdict of "pass" or "fail" [11].

5. Traceability between test scenarios and requirements

Taking into account the definition of the "requirements traceability" term [35], onefold:

Requirements traceability *refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases)*

The requirements traceability offers a clear vision of how high-level requirements, objectives, goals and the stakeholder's needs, are transformed into low-level requirements and furthermore, how the relationships within the other related information layers are foreseen [21][23][24].

Requirements traceability might cover the following constrains:

- Ensure traceability for each level of decomposition performed on the project. In particular:
    - Ensure that every low level requirement can be traced to a high level requirement or original source
    - Ensure that every design, implementation, and test element can be traced into a requirement
    - Ensure that every requirement is represented in design and implementation
    - Ensure that every requirement is represented in testing/verification
- Ensure that traceability is used in conducting impact assessments of requirements changes on project plans or activities.
- Be maintained and updated whenever changes occur.
- Be consulted during the preparation of Impact Assessments for every proposed change to the project
- Be maintained as an electronic document

Traceability relationships are usually many-to-many, that is, one low-level requirement may be linked to several high-level requirements and vice versa [23].

Figure 5.1. depicts the assurance that the above constrains mentioned are covered.

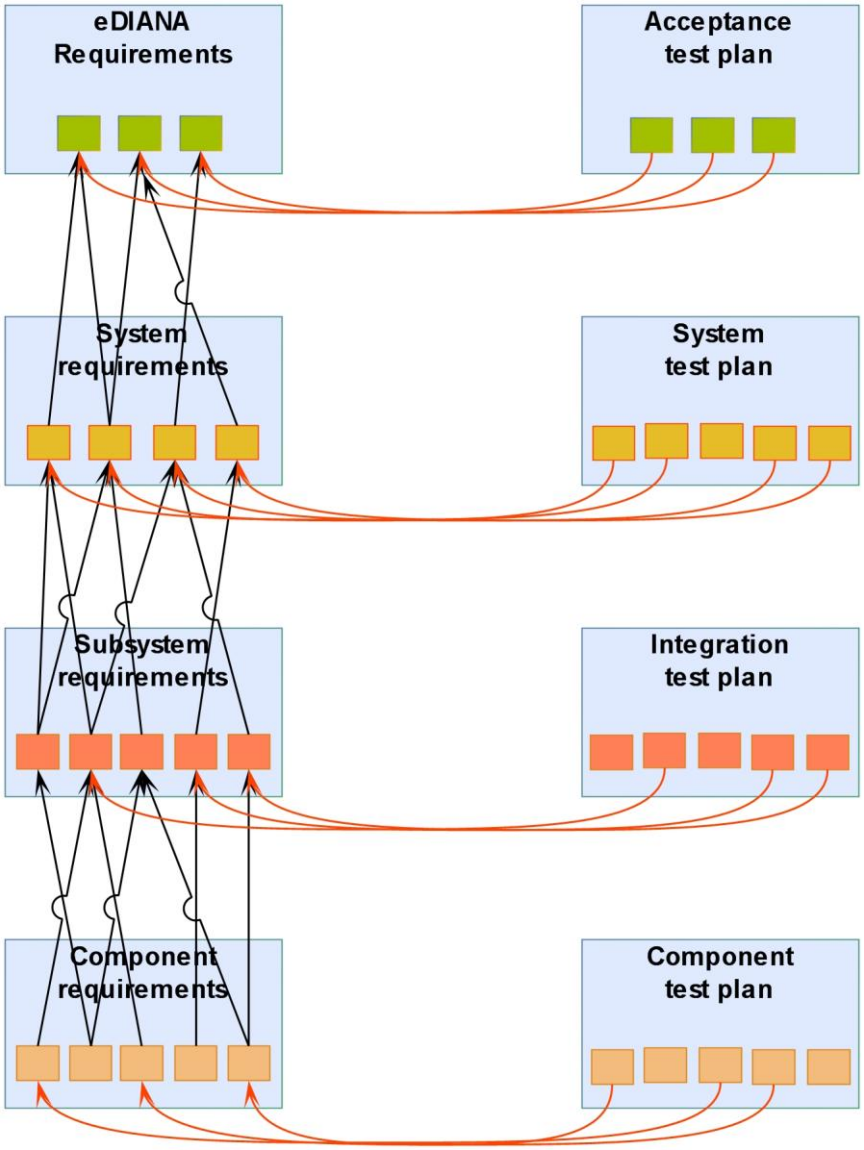

Figure 5-1: Requirements traceability

5.1 Requirements Traceability analysis

Three types of analysis are covered in function of the requirements changes or to follow-up the relationship within the other layers [23]:

- **Impact Analysis**, this kind of analysis is used to determine what others artefacts might be affected if a selected artefact changes.
- **Derivation Analysis**, this analysis works in opposite direction to impact analysis, getting a low level artefact as requisite, design element or test, the traceability links are used to determine what higher level requirements have given rise to it.
- **Coverage analysis**, can be used to determine that all requirements do trace downwards to lower layers and across to tests, if it does not exist trace in it should be an indicator that probably the requisite will not be meet.

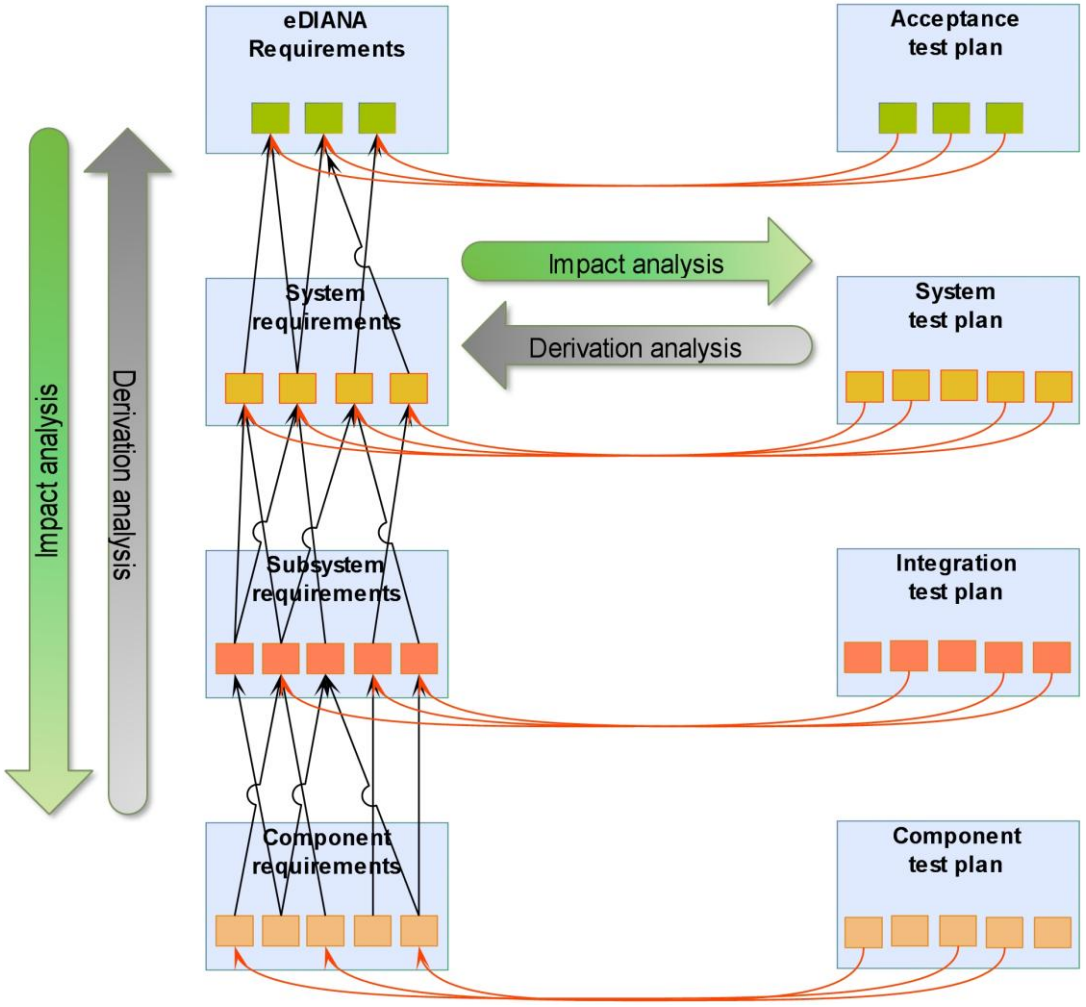

Figure 5-2: Requirements traceability analysis

5.2 Requirements Traceability techniques

A lot of requirement traceability techniques are identified, many of them within the research scope and other implemented in the industrial sector:

- Value Based Requirement Traceability (VBRT)
- Feature Oriented Requirements Tracing (FORT)
- Pre-RS Requirements Tracing
- Event Based Traceability (EBT)
- Information Retrieval (IR)
- Rule Based (RB) Approach
- Hyper-text Based Approach (HB)
- Feature-Model Based Approach (FB)
- Scenario-Based Approach (SB)
- Process Centered Environments
- Design Patterns
- Traceability Matrices
- Keywords and Ontology
- Aspect Weaving
- Goal Centric Traceability (GCT)
- Analysis

All of them could be classified into two types based on two key aspects of traceability.

1. Techniques facilitating pre-RS traceability.

This type includes those traceability techniques which help to describe the life of requirements when they are not included in the requirements specification.

2. Techniques facilitating post-RS traceability.

This type includes those techniques which help to trace the life of requirements when they are included in the requirements specification and forward. The techniques supporting post-RS traceability are subdivided in three types: those who favouring traceability of functional requirements, techniques favouring non-functional requirements and finally techniques that favouring both, functional and non-functional requirements. The latter is most suitable to the eDIANA features, because support traceability of both functional and non-functional requirements. The techniques included in this category of the above mentioned are EBT, IR, Hyper text based approach, Feature model based approach, Scenario based traceability, Process centric environment, Matrices and Aspect weaving.

The two techniques selected in eDIANA are:

- Scenario-Based Approach (SB), with this technique scenarios are used to model system functionality and to generate functional test cases. Scenarios-based test cases create a mapping between requirements and other artefacts like design and code. The traceability is established by mapping scenarios with the design elements. Scenarios are created to trace only the interesting cases therefore they might not provide complete coverage. [20]

- Traceability Matrices, this technique are commonly used in industry to define relationships between requirements document and other type of artefacts [20]. The other artefacts include design modules, code modules and test cases. In traceability matrices the links are manually created between requirements and other artefacts.

Currently different products are in the market as **RETRO**, **DesignTrack**, **TRAM**, **Scenario Advisor Tool**, **DOORS** or **ARTS**.

Nevertheless the tool used currently in eDiana for requisite management is **Rational *RequisitePro***, which is a requirements management tool developed by IBM that provides support to save software requirements specification (SRS) document, link requirements to use case diagrams, and test cases. When change to requirements occur Rational *RequisitePro* identifies the corresponding software artefacts that are affected. But the most important feature in this context is that also provides *traceability* support for the requirements.

Requisite Pro offer the possibility to display and manage the requirements, their attributes, and their relationships with other requirements in views, and this information should present in a table/matrix or in an outline tree. In eDIANA Project

would have to be created several views to show up the relationships among requirements.

Two different views are allowed when using *RequisitePro*:

- Attribute Matrix, type of view that lets view all the requirements of a particular type, which allows sort and prioritize the requirements.
- Traceability Matrix, reflects the relationships between two different types of requirements. A traceability matrix view will be used to create, modify, and delete traceability relationships so that requirements can be traced throughout the development life cycle. The Traceability Matrix view will allow the display of both direct and indirect traceability relationships between two types of requirements or requirements of the same type. A traceability relationship is direct when it traces from one requirement to another. A traceability relationship is indirect when a requirement traces to an intermediate requirement, which in turn traces to another requirement.

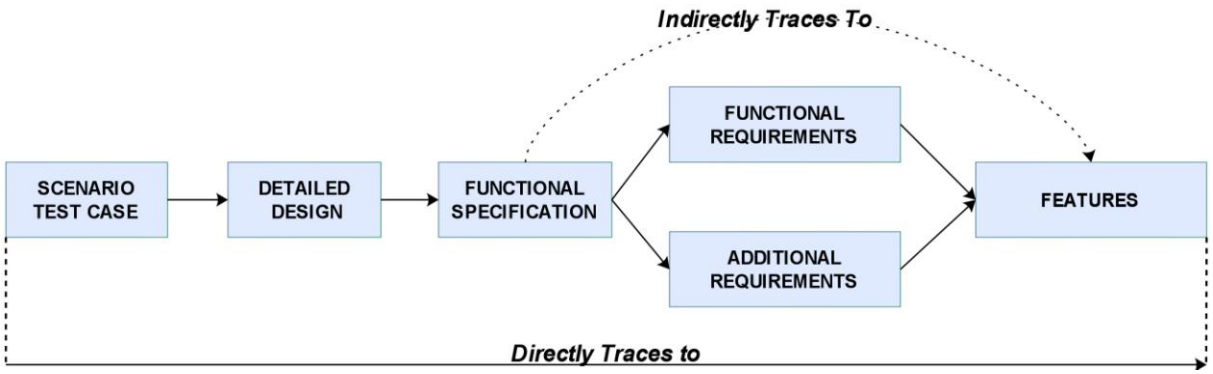

Figure 5-3: Direct and Indirect Relationships

Then *RequisitePro* tool will be used in order to create the Systems Requirements Traceability Matrix.

5.3 Test Management

To ensure the Quality of the system, and verify that all the requisites are covered, a test management tool called ***HP Quality Center*** [22] will be used in eDIANA. This tool has a total integration within *RequisitePro.* And the integration is focused in the synchronization of the requirements between both tools.

5.3.1 HP Quality Center

HP Quality Center, is a web based Test management tool, composed by five main modules for management of testing processes.

Even though just three modules will be used in eDIANA:

- "Requirements", which is used for requirements management and requirements traceability through test cases stored in the HP-QC repository.

- "Test Plan", which is used for creating or updating different Test Cases. The Test Cases are contained in different folders which are displayed in a tree like structure. It can store both Manual as well as automated test cases. Manual Test Cases can be written locally or imported from Excel Sheets. With each 'Test Step' having Expected Result and ActualResult section. QC supports automated script developed for different Automation Tools like QTP, LoadRunner, WinRunner etc. These scripts can be saved directly from the Tool into the Test Plan tab of QC. However, prior to this, appropriate QC Add-in needs to be installed to support an Automation Tool.

- "Test Lab", this module is for execution of the test cases stored in the Test Plan module which can be imported locally to the Test Lab screen and Run. When Manual Test case is executed, it opens up a pop up listing all the Test Steps and the user is supposed to update status of each step with Passed, Failed or Not Complete. When automated test case is run, QC invokes the Automation Tool which in turn executes the script and stores back the result into QC repository and displays on the UI.

With HP Quality Center it is possible link and trace requirements to one another to highlight dependency relationships and verify that no requirements are inadvertently overlooked during the development and testing process.

5.3.2 HP Quality Center Synchronism

The synchronism is the way to have the test references inside the Requirement Environment or to import the requirements inside HP Quality Center so that we can manage the tests and their results inside HP Quality Center. The following figure show how the synchronism works.

Figure 5-4: Requirements sync HP QC with RequisitePro

The only aspect that we need to have into account is how mapping the fields between RequisitePro and HP QC. For default the following rules apply:

- The Synchroniser maps the content of free text field
- In case of list value fields, it maps the value from Requisite Pro with the ones in HP QC

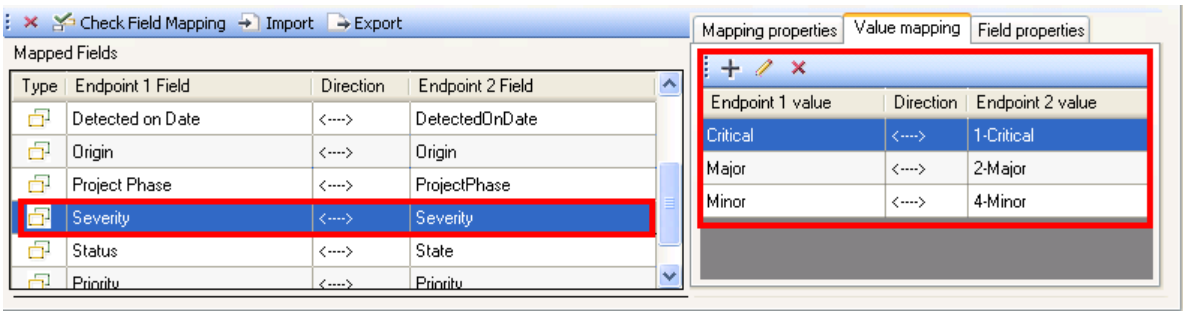

Figure 5-5: HP QC Mapping rules

To summarize, three steps are needed to synchronize the requirements and manage all the lifecycle of the requirements:

1. Sync RequisitePro with HP QC.
2. Define tests Plan. Should cover all the test specifications.
3. Verify the test plans in the test Lab.

6. Variability management in testing

In most of the systems, there are different configurations that impact on testing; this section will present an approach to manage the variability in testing context. This is the case of eDiana, where installations can differ very much one from another: the number and type of devices can be different: some devices will be present in some installations but not in others, the communication protocols can be different, the number of cells and macrocell can differ, etc.

For validating it is also necessary to consider this variability and possible scenarios. To validate a software with multiple validation context is like a validation product line.

In order to identify and model the environments/contexts in which software should be validated, a feature model can be used. One of the most adequate model for representing variability among products.

A feature model is an and/or tree of different features. A feature as "a prominent or distinctive and user-visible aspect, quality, or characteristic of a software system or systems" [14]. Feature modelling was first proposed as part of the FODA (Feature Oriented Domain Analysis) method [14]. Features can be mandatory, optional or alternative. And composition rules are used to define the semantics existing between features that are not expressed in the feature diagram: Mutual dependency (Requires) and mutual exclusion (Mutex-with) relationships. Features are an effective way of identifying the variability (and the communality) among different products in a domain. Moreover, they are a natural and intuitive way of expressing the variability [13]; feature model plays a central role, not only in the development of the reusable assets, but also in the management and configuration of multiple products in a domain.

For validating or testing software in different contexts or environments, the feature model helps to identify the variability in the execution context. In eDIANA installations the feature model of validation is the following one:

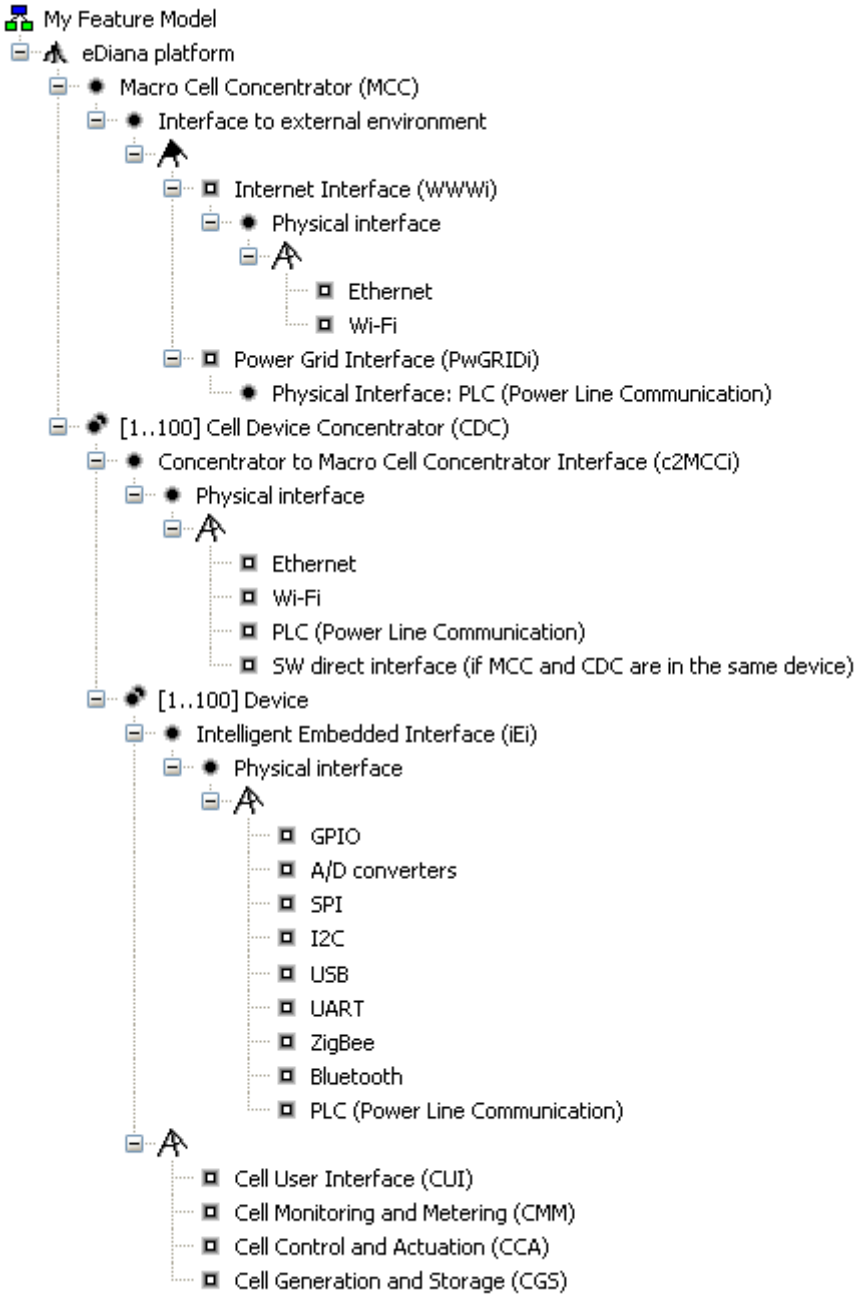

Figure 6-1: eDiana validation feature model

In this feature model, we can see that it is necessary to consider different validation context depending on the number and type of devices and protocols that will be included in an installation.

- Different number of cells

- Different number and type of devices in a cell
- Different physical interfaces

To validate the software at system level, it is necessary to manage the variability of the context. This variability impacts on the next aspects:

- Variability in Software-Under-Test.
- Variability in the validation environment.
- Variability en testing scenarios.

To be able to validate the software in an appropriate way, it is necessary to manage variability in those three aspects, in order to be able to validate the software in different contexts. Each of the aspects is described below.

Variability in the Software under test

The embedded software can be developed following different development paradigms in order to be adaptable and has variability.

One option can be to use a software product line engineering approach. Using this paradigm each product or configuration will only have the specific software that needs.

Another option is to use a configurable product. In this case, an unique software is developed but this software is able to run in every configuration. This is the case of eDIANA, the software that will be installed in the installations is the same but it will be configured according to the devices of the installation. For validating this software, it is necessary to configure it for being able to analyze the answer of the software in the different situation under it can be run.

Variability in the validation environment

The embedded software often runs under different configurations. It can be connected to a different number of devices, run under different processors... It is necessary to manage this variability to be able to create the adequate environment to validate/test each of the configuration of the software.

In the validation environment, the variability can come from:

- Number and type of sensors
- Number and type of actuators

- Communication mechanisms
- Processors

If we are using Simulink model to test the system, variability can be introduced in Simulink models in relations and blocks that are required for simulation. Simulation elements can also be contained in a library and be connected automatically in order to create the simulation model for a concrete installation.

Variability in the testing scenarios

Not all the configurations or installations have the same requirements regarding testing. Depending on the configuration, some functionalities may be no active or change slightly. For this reason, variability must also be considered in the testing sequences or test cases.

6.1 Tools for managing variability

To be able to manage variability in all these aspects is necessary to have tool support. Pure::variants is a commercial tool for managing variability and developing software product lines that have a Simulink plug-in [15] that allows to create and maintain reusable models with Simulink. It provides variability management for Simulink. It allows to maintain and configure all model variants within a single master model structure by feature selection in pure::variants.

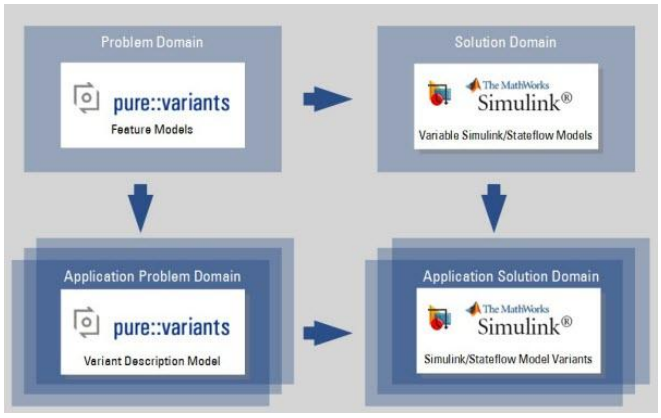

Figure 6-2: Pure::variants for Simulink

Conclusion

In this deliverable, model based testing approach and its application in eDIANA has been analyzed. An introduction to Model based testing: benefits, process, taxonomy, approaches and tools has been provided. And the generation of test scenarios in eDIANA project has been addressed for Unit testing using the Test Driven Development (TDD) approach and for early system testing via simulations.

For modelling test related artefacts, U2TP (UML 2.0 Testing Profile) and TTCN-3 (Testing and Test Control Notation version 3) has been studied.

Traceability between test scenarios and requirements in eDIANA has been also addressed using the tool RequisitePro for managing requirements and HP Quality Center for test management. As well as, variability management of the different configurations of eDIANA that can affect testing.

Acknowledgements

The eDIANA Consortium would like to acknowledge the financial support of the European Commission and National Public Authorities from Spain, Netherlands, Germany, Finland and Italy under the ARTEMIS Joint Technology Initiative.

References

[1] Booch, Brady. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2007.

[2] de Champeaux, Dennis; Lea, Douglas; Faure, Penelope. *Object-Oriented System Development.* Addison-Wesley, 1993

[3] Hans-Gerhard Gross, Component-Based Software Testing with UML, Springer, 2005

[4] M. Utting, B.Legeard, Practical Model-Based Testing: A Tools Approach, Morgan-Kaufmann 2007.

[5] A.C. Dias, R. Subramanyan, M. Vieira, G. H. Travassos, A Survey on Model-based Testing Approaches: A systematic Review, Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2007, pp 31-36

[6] Model-based Testing, http://en.wikipedia.org/wiki/Model-based_testing

[7] Harry Robinson (Google), Model-Based Testing tutorial, *STARWEST* - Software Quality Engineering - Software Testing, 2006

[8] Software Acquisition Gold Practice: Model-Based Testing, http://www.goldpractices.com/practices/mbt/, 2010

[9] M. Utting, A. Pretschner, B. Legeard, A taxonomy of Model-Based Testing, Technical report 04/2006, Department of Computer Science, University of Waikato, April, 2006.

[10] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, Guilherme H. Travassos, A survey on model-based testing approaches: a systematic review, Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 31-36, 2007

[11] A. Bertolino. Software Testing. In SWEBOK: Guide to the Software Engineering Body of Knowledge, IEEE

[12] Simulink, http://www.mathworks.com/products/simulink/
[13] Kwanwoo Lee, Kyo Chul Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *ICSR-7: Proceedings of the 7th International Conference on Software Reuse*, pages 62–77, London, UK, 2002. Springer-Verlag.
[14] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, November 1990.
[15] Pure::variants for Simulink. http://www.pure-systems.com/pure_variants_for_Simulink.164.0.html
[16] ModelJUnit webpage, http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/, 2009
[17] *UML Testing Profile*, version 1.0. Object Management Group, July 2005.
[18] TTCN-3 Homepage: http://www.ttcn-3.org/
[19] Pietsch, Stephan; Stanca-Kaposta, Bogdan. *Model-based testing with UTP and TTCN-3 and its application to HL7*. Testing Technologies IST GmbH, 2008
[20] J.Cleland-Huang, "Toward Improved Traceability of Non-Functional Requirements", Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering TEFSE'05, ACM, 2005, pp. 14-19.
[21] Wiegers, Karl E., "Software Requirements", Second Edition, Microsoft Press, 2003
[22] HP Quality Center, http://h50281.www5.hp.com/software/index.html
[23] Prof. Elizabeth Hull, Prof. Ken Jackson, Dr Jeremy Dick , "Requirements Engineering" 2004, Springer, ISBN 1852338792
[24] Daryl Kulak, Eamonn Guiney, "Use Cases: Requirements in Context, Second Edition", 2003, Addison Wesley, ISBN 0-321-15498-3
[25] A. Spillner: The W-Model Strengthening the Bond Between Development and Test. Orlando, 2002
[26] Paul Baker, Zhen Ru Dai, Jens Grabowski, Øystein Haugen, Ina Schieferdecker, Clay Williams, Model-Driven Testing: Using the UML Testing Profile, Springer, 2008
[27] Z. Dai, Model-Driven Testing with UML 2.0, in Proc. of the 2nd European Workshop on Model Driven Architecture, 2004
[28] Sandmann, G., Thompson, R., 2008. Development of AUTOSAR Software Components within Model-Based Design. The Mathworks
[29] Ridderhof, Gross, Doerr, 2007. Establishing Evidence for Safety Cases in Automotive Systems. Report TUD-SERG-2007-008. Delft University of Technology.
[30] Justyna Zander-Nowicka, Model-based Testing of Real-Time Embedded Systems in the Automotive Domain, phD Thesis, Technischen Universität Berlin, Berlin, 2009
[31] Dijkstra, E. W.: Notes on Structured Programming. In Structured Programming, Volume 8 of A.P.I.C. Studies in Data Processing, Part 1, Editor: Hoare C. A. R., Pages: 1 – 82. Academic Press, London/New York, 1972.

[32] Brökman, B., Notenboom, E.: Testing Embedded Software. ISBN: 978-0-3211-5986-1. Addison-Wesley International, 2002.

[33] Olli-Pekka Puolitaival, Adapting model-based testing to agile context, VTT PUBLICATIONS 694

[34] Sebastian Wieczorek, Alin Stefanescu, Mathias Fritzsche, Joachim Schnitter, Enhancing Test Driven Development with Model Based Testing and Performance Analysis, Proceedings of the Testing: Academic & Industrial Conference - Practice and Research Techniques table of contents, pp 82-86, IEEE Computer Society, 2008

[35] O. Gotel, A. Finkelstein, "Extended Requirements Traceability: Results of an Industrial Case Study", Proceedings of the Third IEEE International Symposium on Requirements Engineering, IEEE, 1997, pp.169-178.