

		Title: Test Generation and Refinement
		Version: 1.0 Date : 04/07/08 Pages : 35
		Author: Conformiq
		To: D-MINT Consortium
The D-MINT Consortium consists of: Nokia Siemens Networks, ABB, Åbo Akademi, Conformiq, Daimler, DATAPIXEL, ELIKO, Elvior, Fraunhofer IESE, Fraunhofer FOKUS, IDEKO, Innovalia Association, INSPIRE, iXtronics GmbH, Nethawk Oyj, PikeTec GmbH, SQS, SORALUCE, Tallinn University of Technology, Testing Technologies IST GmbH, TRIMEK, VTT Technical Research Centre of Finland, ETSI, TANDBERG, Simula Research		Printed on: 12-11-2009 12:38:00
Status: <input type="checkbox"/> Draft <input type="checkbox"/> To be reviewed <input type="checkbox"/> Proposal <input checked="" type="checkbox"/> Final / Released	Confidentiality: <input checked="" type="checkbox"/> Public - Intended for public use <input type="checkbox"/> Restricted - Intended for D-MINT consortium only <input type="checkbox"/> Confidential - Intended for individual partner only	
Deliverable ID: D.3.1.v1.0		
Title: <h2 style="text-align: center;">Test Generation and Refinement</h2>		
Summary / Contents: Architectures for test case and test harness generation. Finite and infinite state space generation driven mechanisms for test case generation. Pattern driven mechanisms for test case generation. Generation of test harness. Iterative maintenance of test cases and test harnesses. Human-computer interaction and usability issues.		



	Test Generation and Refinement Deliverable ID: D.3.1.v1.0	Page : 2 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

TABLE OF CONTENTS

1. Introduction.....	5
1.1 Abbreviations.....	5
2. Architectures for Test Case Generation Systems	6
2.1 Pattern driven Offline Test Generation	6
2.2 System Model Driven Offline Test Generation	7
2.2.1 Case: Connecting Conformiq Qtronic with EAST	7
2.3 System Model Driven Online Testing.....	11
2.4 System Model Driven Reactive Online Testing	12
3. Test Case Generation Mechanisms.....	12
3.1 State Space Search.....	12
3.1.1 Finite State Approaches.....	12
3.1.2 Infinite State Approaches	14
3.2 Pattern Driven Approaches: PTML	18
3.2.1 Extending the UTP: The PTML Metamodel	19
3.2.2 Pattern driven Test Generation Architecture.....	26
3.2.3 Requirements coverage	27
3.3 Offline Test case Generation for Control Systems: TPT.....	29
3.4 Test Harness Generation.....	32
3.4.1 NModel	32
4. Mechanisms for Iterative Test Refinement	32
4.1 Maintenance of Test Cases	32
4.2 Maintenance of Test Harnesses	32
4.2.1 NMODEL	32
5. Usability and Human-Computer Interaction	33
5.1 NModel.....	33
5.2 Conformiq Qtronic	33
5.3 Qtronic EAST SCRIPTER plug-in configuration.....	34
6. Conclusions	34
7. References	35


	<p>Test Generation and Refinement</p> <p>Deliverable ID: D.3.1.v1.0</p>	Page : 3 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

CHANGE LOG

Vers.	Date	Author	Description
0.1	Feb 2008	Antti Huima	Document created
0.2	Jun 2008	Antti Huima	Submitted for review
0.7	Jul 2008	Antti Huima	Proposed document after review round
1.0	Jul 2008		Final

APPLICABLE DOCUMENT LIST

Ref.	Title, author, source, date, status	Identification


	<p>Test Generation and Refinement</p> <p>Deliverable ID: D.3.1.v1.0</p>	Page : 4 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

EXECUTIVE SUMMARY

Four architectures for model-based test generation are presented: pattern driven test generation, offline test generation from system models, online test generation from system models, and online test generation from system models with precomputation.

For pattern driven test generation, the PTML language for telecommunication systems developed with D-MINT and TPT for control systems and the related methodology are reported. For system model driven approaches, Conformiq Qtronic, developed within D-MINT, as well as third-party tools like Nmodels are considered as concrete incarnations of the architectures.

It is concluded that all the architectures are feasible to implement and provide complementary benefits.

	Test Generation and Refinement Deliverable ID: D.3.1.v1.0	Page : 5 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

1. INTRODUCTION

The Task 3.1 of the D-MINT project focuses on defining and creating tools for test generation (i.e. generating tests based on models “from scratch”) and test refinement (i.e. updating, maintaining and improving tests based on previously generated testing assets). A tools-oriented task, 3.1 depends heavily on the results from WP2.

It is assumed that the reader is familiar with the basic concepts of model-based testing, and is knowledgeable of the results of WP2 in particular.

This document is structured as follows.

Sec. 2 looks at conceived architectures for test generation systems, i.e. on the question how to organize test case generation software itself and how to connect it to the various resources it needs, e.g. users, models, test case repositories, requirements management etc.


Sec. 3 focuses on mechanisms for “initial” test case generation, i.e. how to construct test cases based on models alone. Both state space search driven and pattern driven approaches are considered. There is also a section for the generation of test harnesses.

Sec. 4 is about mechanisms for iterative (or incremental or continuous) test refinement, i.e. how to update both test cases and test harnesses when models, testing requirements and testing interfaces evolve.

Sec. 5 is about the human perspective: how to take usability into account and how to provide smooth and intuitive human-computer interaction for the operator of the test case and test harness generation and maintenance tools.

1.1 ABBREVIATIONS

EFSM	Extended Finite State Machine
EMF	Eclipse Modeling Framework
ETSI	European Telecommunications Standards Institute
IDE	Integrated Development Environment
IUT	Implementation Under Test
MDA	Model Driven Architecture
MSC	Message Sequence Chart
NSN	Nokia Siemens Networks
OCL	Object Constraint Language
PTML	Pattern Driven Test Modeling Language
SUT	System Under Test
TPT	Time Partition Testing
TTCN-3	Test and Test Control Notation v. 3
U2TP	UML 2 Testing Profile
UML	Unified Modeling Language

	Test Generation and Refinement Deliverable ID: D.3.1.v1.0	Page : 6 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

WP Work Package
XMI XML Model Interchange

2. ARCHITECTURES FOR TEST CASE GENERATION SYSTEMS

In this section we cover different promising architectures for test case generation.

The **pattern driven** approach enables the user to create high-level, graphical definitions of test architectures and test cases and, significantly, to employ **predefined test patterns** to further speed up test definition. The user can also create **domain-specific test patterns**, enabling one to create reusable test assets with the specific domain that can be leveraged in multiple test design projects within that domain.

The **system model driven offline test generation** approach enables the user to create concise, expressive models of the **intended behaviour** of the system under test, and then generate **test cases** automatically from that model, which can be then later executed either manually or through an automatic test execution system.

The **system model driven online testing** approach enables the user to create similar, expressive models of the **intended behaviour** of the system under test, but instead of generating external test cases, to execute test cases **immediately and on-the-fly** against the system under test. Because the test cases are never actually exported, this methodology allows for full support for **nondeterministic models** which are important in practice especially if the system under test is concurrent, distributed, under underspecified.

The **system model driven reactive online testing** approach aims to combine the benefits of the offline and online testing approaches by performing first **precomputation**, and then executing **online testing** based on the precomputed assets. This increases the quality of the online tests conducted compared to the vanilla online testing approach but still retains full support for nondeterministic models.

2.1 PATTERN DRIVEN OFFLINE TEST GENERATION

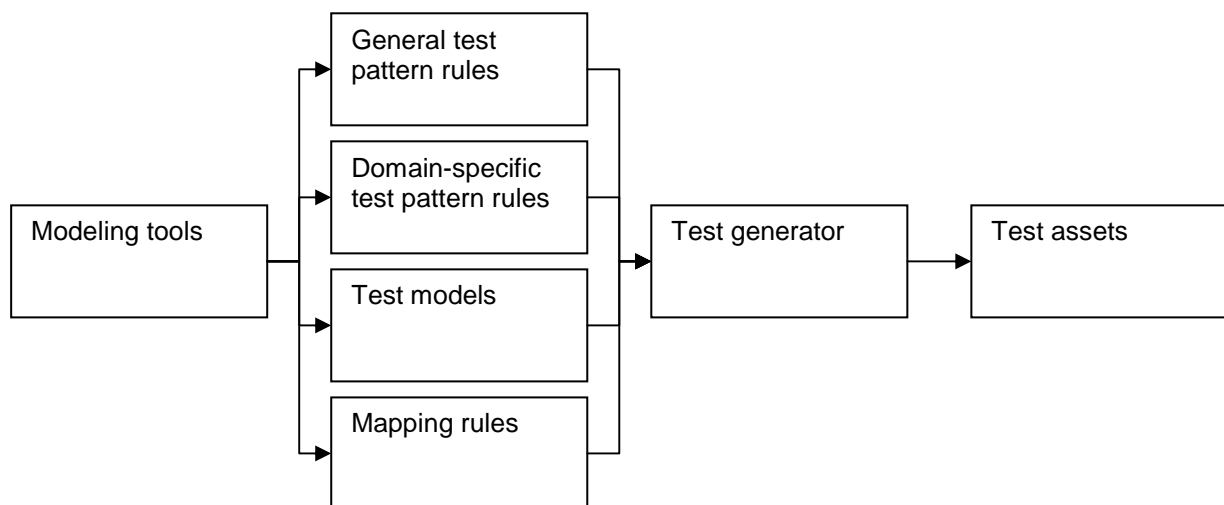



Figure 1 Pattern Driven Test Generation Architecture

	Test Generation and Refinement Deliverable ID: D.3.1.v1.0	Page : 7 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

In the pattern driven test generation approach, test assets (e.g. executable test cases in TTCN-3, or human-readable test cases as message sequence charts, MSCs) are produced by a **pattern driven test generator** from:

- Test models, describing the actual SUT-specific testing logic on high-level.
- Test patterns and rules implementing them. The test models link to the test patterns in order to reuse the testing heuristics and architectures embodied in the test patterns.
- Mapping rules, which describe how the essence of the test cases is actually rendered and presented to the user. For example, there would be one set of mapping rules to produce TTCN-3, and another to produce MSCs.

2.2 SYSTEM MODEL DRIVEN OFFLINE TEST GENERATION

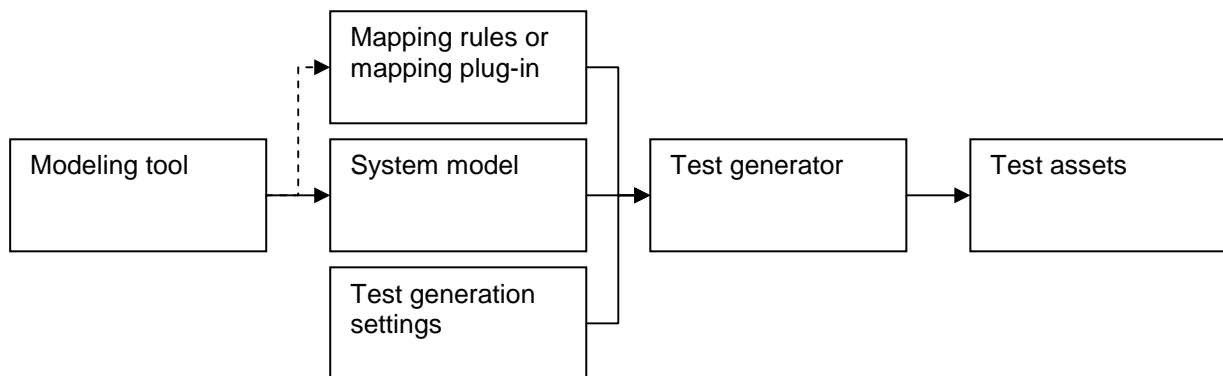


Figure 2 System Model Driven Offline Test Generation Architecture


In the system model driven offline test generation approach the user provides a behavioural model of the intended behaviour of the system under test. A test generator derives from the given model a set of test cases which are sequences of logical-level messages to and from the system under test. These test cases are rendered into test assets (e.g. test cases in TTCN-3 or as MSCs) either through mapping rules, or via a mapping plug-in if the test asset rendering function is not based on modelling approaches.

Importantly, because it is an open-ended question what is a “good” test suite to be generated from a given system model, as the system model itself defines a very large space of potential tests, the architecture needs to have provision for changing and optimizing test generation settings. The system model and the test generation settings drive the logical structure of the test cases, whereas the mapping rules or plug-in drives the actual external format of them.

2.2.1 Case: Connecting Conformiq Qtronic with EAST

Typically **NetHawk EAST** test cases are designed with set of graphical tools. These graphs are generated to run able test scripts. A script may be run in regression or load mode.

When using Conformiq Qtronic EAST scripiter plug-in for model based test case generation, test cases are generated only on EAST textual script language.

	Test Generation and Refinement Deliverable ID: D.3.1.v1.0	Page : 8 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

2.2.1.1 Connecting Qtronic Models to EAST Reference Libraries

To avoid the need to implement all complexity of telecom messaging in the model, we can utilize EAST reference libraries to hide most of parameters and lower level protocols from the model itself. See Figure 3 for EAST reference library editor.

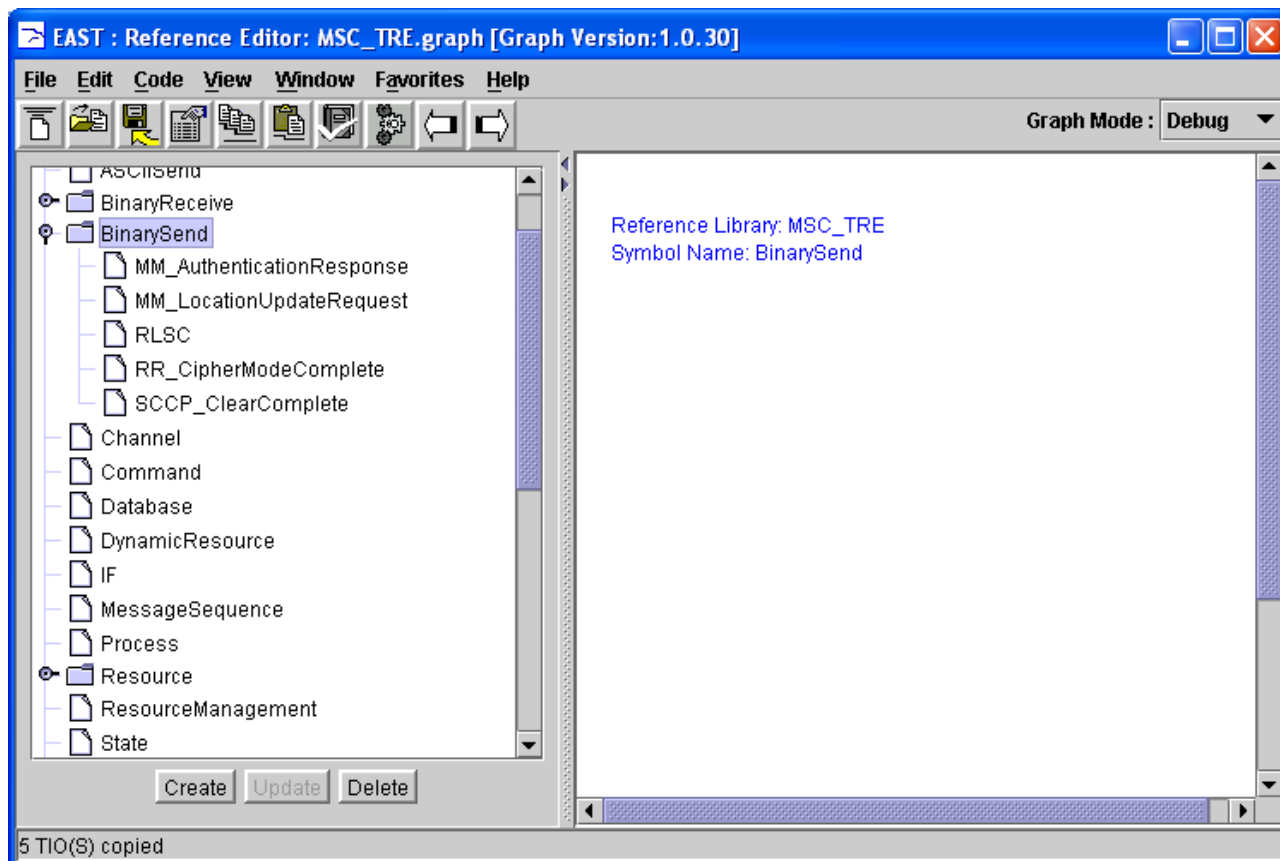


Figure 3 EAST reference library defines messages and variables and resources used by them

Messages and variables in EAST Reference libraries are mapped to messages and variables used in the Qtronic models. Variables used in the model are pre-populated to messages in the reference library. Other, uninteresting, parameters in reference messages are pre-populated by some known to work parameters. This way we can concentrate to generate cases with interesting parameters only. See Figure 4 for EAST message editor with variables and constants populated.

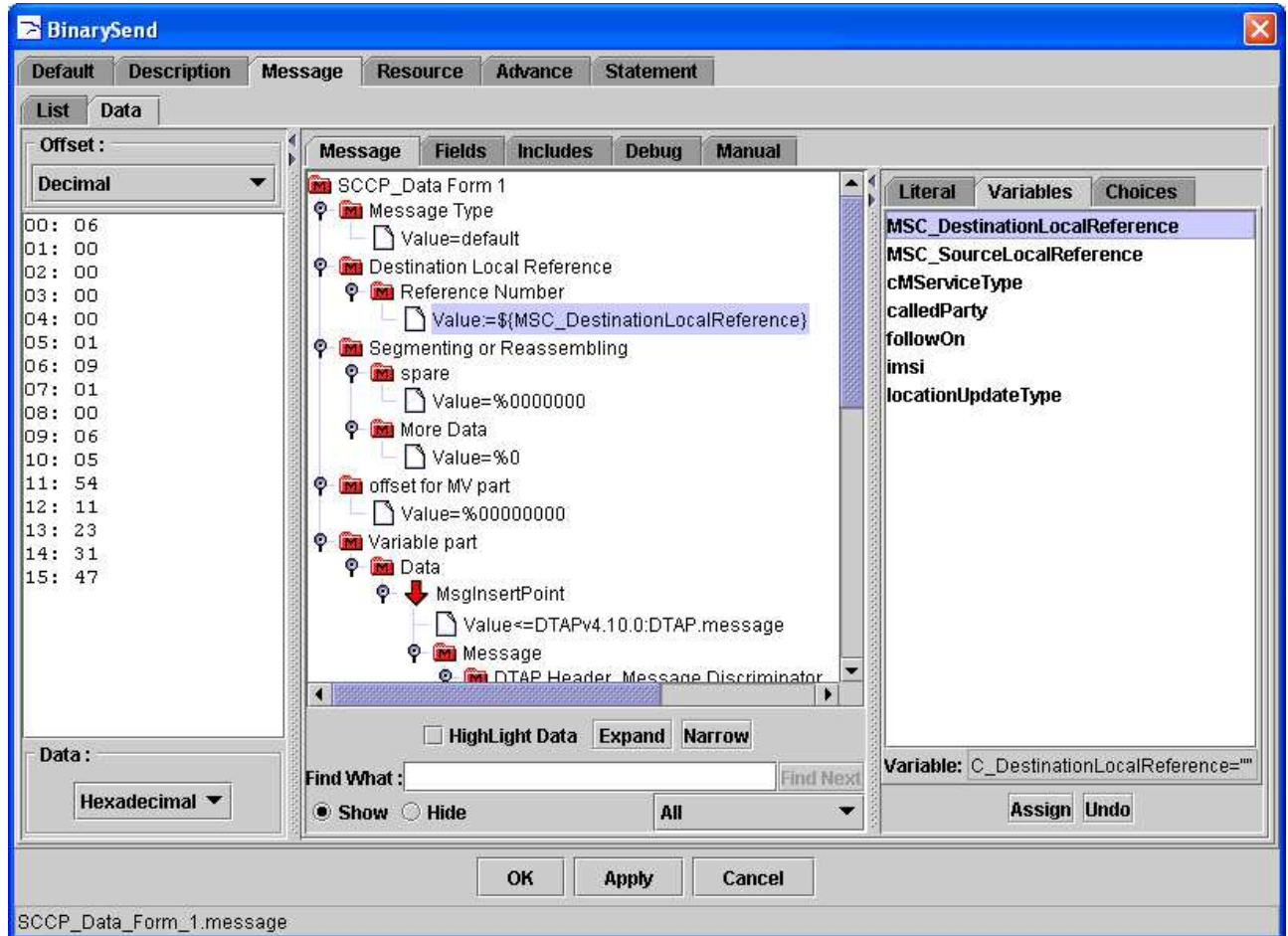


Figure 4 EAST Binary reference message structure. Some parameters have variables as value, some have hard coded values.

There are also definitions of resources used by messages in reference libraries. Resources define which lower level EAST protocol server message uses when communicating with SUT. This way we can exclude lower level protocols from the model itself. See Figure 5 for resource definition window.

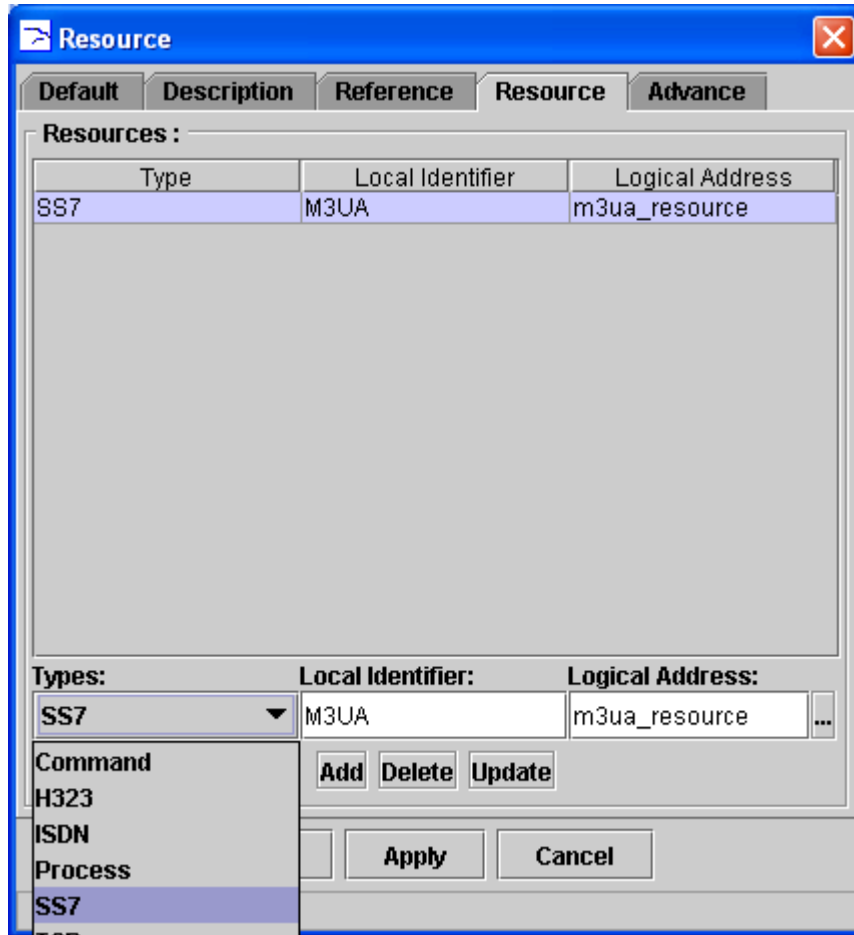


Figure 5 Resource definition

We can have different reference libraries for different technologies or for different message release versions. When generating test cases we can choose which reference library to use. This makes system very flexible without changes needed to the actual model. Actually reference library is a directory with reference message, variable and resource files inside.

If load testing is needed, it is possible to define resource specific decode state machine (called decode TLP in EAST). This decoder investigates messages going through it and decides to which test case message belongs to and directs it to there. Decode TLP is also defined as part of reference library.

All messages and parameters' names that are used in models must be defined exactly the way used in EAST reference libraries, or vice versa.

It is possible to generate EAST Test Suite out of test cases produced by Conformiq Qtronic and EAST scripser plug-in. Test suite is basically a collection of test cases. Test suite may be run with specific Test Suite runner. It is possible to run test cases in parallel or sequential mode.

EAST test cases maybe overridden or new cases with some date/time specific name may be always generated.

2.3 SYSTEM MODEL DRIVEN ONLINE TESTING

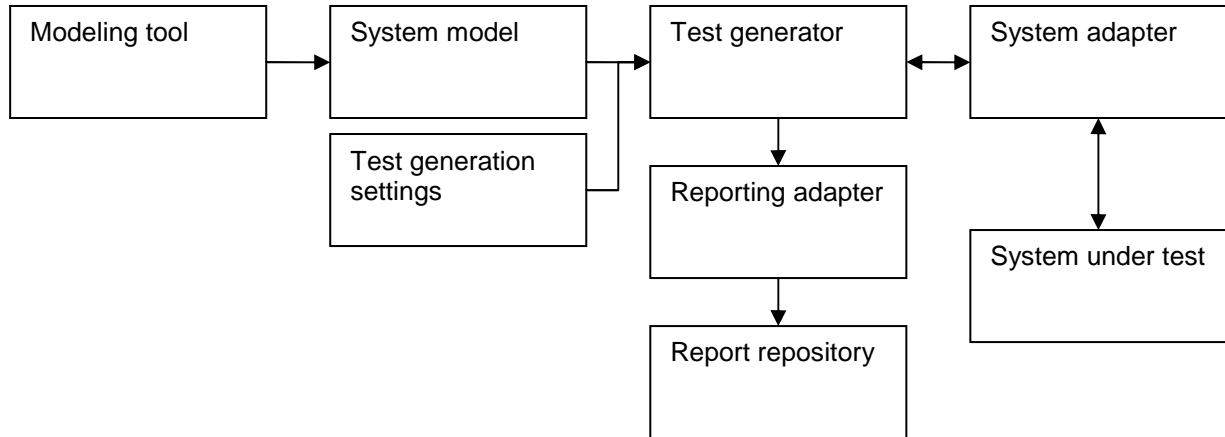


Figure 6 System Model Driven Online Testing Architecture

In the system model driven online testing approach the user provides a behavioral model as in the previous approach, but now the test generator is “directly” connected to the system under test and it executes tests on-the-fly against the system. Because there are not externalized test cases, the test generator can support **nondeterministic** systems, i.e. systems whose test cases could not be represented as linear message sequences.

In order to facilitate communication with the system under test, a component usually known as *system adapter* is used. It can be also called an [online] test harness.

Similarly, reporting functionality can be either provided directly by the test generator using fixed formats (e.g. XML), or, more generally, reporting functionality can be provided indirectly via a *reporting adapter* that is responsible for the physical representation of the test execution logs and reports that are provided on logical level by the test generator.

2.4 SYSTEM MODEL DRIVEN REACTIVE ONLINE TESTING

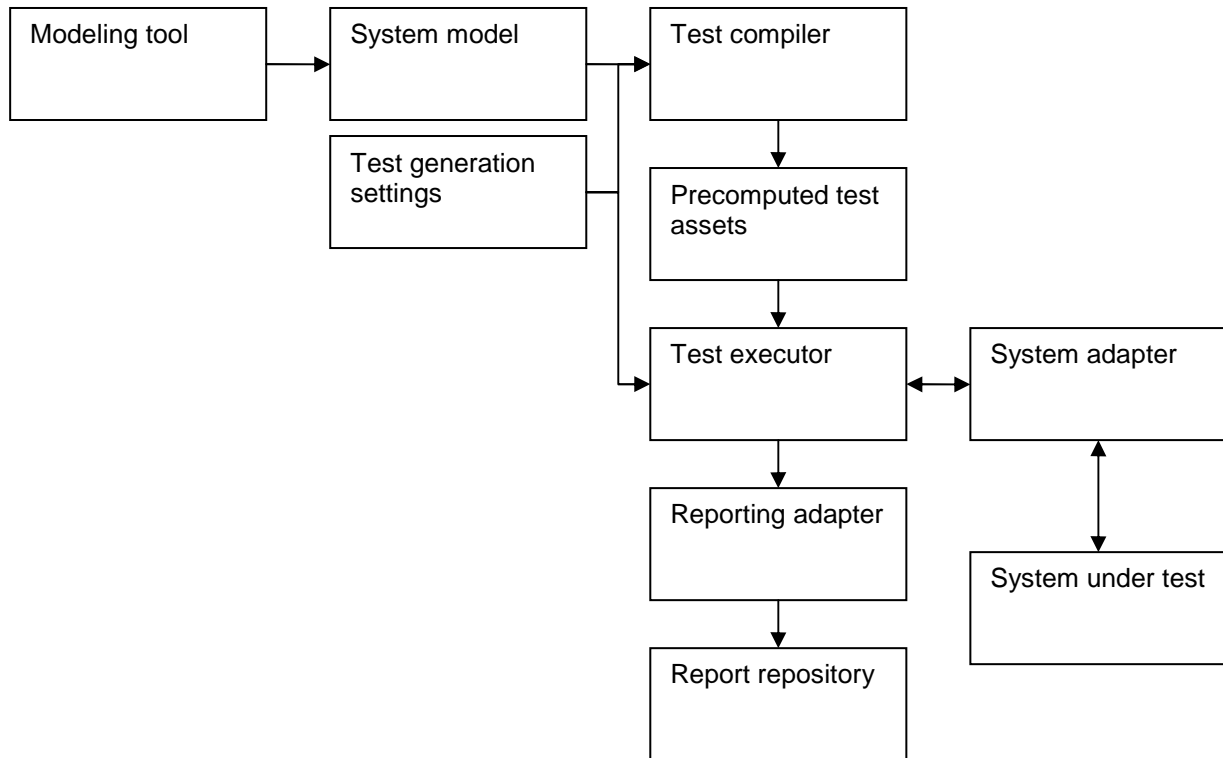


Figure 7 System Model Driven Online Testing Architecture

The reactive online testing approach extends the previous approach by adding a **precomputation step** that is carried out offline (i.e. before test execution). The precomputed test assets are then fed as input to a **test executor** that carries out the actual online testing. This architecture enables one to move a significant amount of the computational burden in online testing to the precomputation stage, making the actual test execution much faster and less resource intensive. This is especially important in the case of time-dependent systems like embedded controllers.


3. TEST CASE GENERATION MECHANISMS

3.1 STATE SPACE SEARCH

3.1.1 Finite State Approaches

3.1.1.1 MOTES

The model-based test generator MOTES is a tool for generating executable test code in TTCN-3 language from specification models using the Uppaal model checker. Test cases are generated offline from the SUT model before the tests are run. Therefore it is assumed that the SUT model is deterministic, i.e. the input and expected output sequences are preset. For specifying the observable behavior of the software under test a modeling language is defined. This language is based on extended finite state machines. A model in such a

	Test Generation and Refinement Deliverable ID: D.3.1.v1.0	Page : 13 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

language is transformed to an Uppaal model taking a structural coverage criterion as a parameter. Uppaal is used to find an abstract test sequence that is suboptimal in terms of length. The abstract test sequences are transformed to TTCN-3 language and the generated modules are executable in TTCN-3 environments.

The workflow of TTCN-3 test case generation comprises the following steps:


1. The test engineer defines the model of external behaviour of the SUT in the form of an EFSM. The SUT model is derived from the available specifications. In our approach the SUT model can be drawn by a CASE tool that is capable of drawing EFSMs (state charts) and exporting them into XMI. The currently supported tools include Poseidon 6.0 and Enterprise Architect 6.0.
2. The test engineer defines the interface of the SUT – input and output message types, message equivalence classes and interface ports. SUT interface is defined directly in TTCN-3.
3. The test engineer prepares the test data – instances of SUT input messages that the test cases will use to stimulate SUT during test execution. The test data is defined in TTCN-3, which allows the tool to import the data into the generated TTCN-3.
4. The test engineer defines the test configuration (test component definitions and port mappings in the case of more than one test component) directly in TTCN-3.
5. The test engineer defines a test coverage criterion for the test case generator.
6. The test case generator transforms the EFSM model into an Uppaal model and decorates the model with control information according to the selected coverage criterion. Guiding information for the model checker is added to the model.
7. The test case generator uses Uppaal Cora to find an abstract test sequence in the form of a sequence of transitions.
8. The test case generator converts the abstract test sequence to TTCN-3 modules. The generated TTCN-3 modules are executable in TTCN-3 test environments.

Thus MOTES generates test sequences from deterministic EFSM models using a guided model checker, Uppaal Cora. This approach allows the user to specify various structural test coverage criteria of EFSMs, for example, selected states/transitions, all transitions, all transition pairs etc. From the SUT model and a coverage criterion we construct an Uppaal model to achieve test sequences satisfying the criterion. The problem of generating test sequences is formulated as a bounded reachability problem and solved by model checking. When a model checker solves a reachability task it generates a witness trace that corresponds to an abstract test sequence. This abstract test sequence is further encoded to tester code in the TTCN-3 language.

The test generation tool support structural test coverage criteria of the model, for example, selected states, selected transitions, all transitions et al. For the purpose of test sequence generation we decorate the transitions of the model with trap variables according to the selected criterion. The trap variables are initially set to FALSE and they become TRUE one by one when the relevant transitions are passed by the model checker. The model checker solves the reachability task to find a sequence of transitions that makes all trap variables TRUE. The abstract test sequence is produced by Uppaal Cora as a witness trace.

The transformation of the EFSM model to Uppaal comprises of four steps (

Figure 8).

	Test Generation and Refinement Deliverable ID: D.3.1.v1.0	Page : 14 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

In the first step we reduce the search space of the model in a way that makes trace generation by model checking feasible. Thus, all receipt of input messages and transmittal of output messages are abstracted away from the model. Omitting them does not affect the resultant witness trace produced by Uppaal but reduces the search space. From the input messages we preserve only the input message fields that are used in the assignments or guards and therefore influence the control flow of the model. Such input parameters are transformed to the Uppaal model as initialized context variables. Note that the same message field values have to be used in creating the input message instances.

In the second step the model is decorated with trap variables to mark passing certain states or transitions. Trap variable declarations and assignments are added according to the selected coverage criterion. In the third step the reachability task is encoded into an observer automaton.

In the fourth step cost functions are added to each transition according to the selected coverage criterion. Cost functions are used in Uppaal Cora to guide the model checker to search for traces that minimise the total resultant cost. We are interested in finding a trace that is suboptimal in terms of its length. The length of the test sequence is important due to the time taken by executing the test cases.

The simplest way to achieve all transitions criterion is to set an equal cost for each transition. The overall cost function is the sum of costs over all transitions. Uppaal Cora tries to find a trace that keeps the overall cost to a minimum and in the best case it finds a trace that passes all transitions just once. It is possible to work out more sophisticated algorithms to guide model checking using cost functions but these algorithms require some analysis of the model.

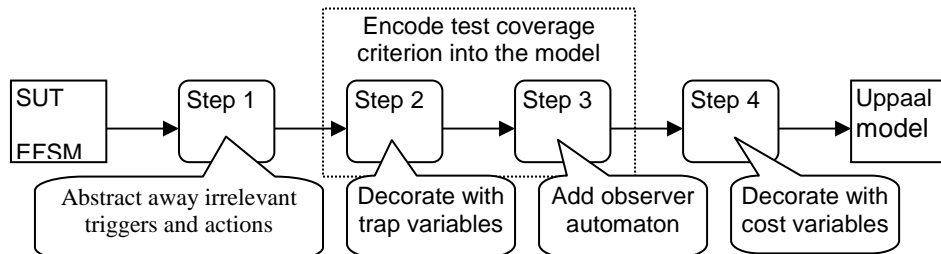



Figure 8. Procedure for generating Uppaal models from EFSM models of the SUT.

3.1.2 Infinite State Approaches

3.1.2.1 Conformiq Qtronic

Conformiq Qtronic [AH07, QTRONIC] is an infinite-state realization of both online and offline test generation approaches. Parts of this tool have been developed within the D-MINT project.

The tool uses extended Java and UML state charts, class diagrams and component diagrams as the notation for creating system models, and creates then test cases and conducts online testing. The tool follows closely the corresponding general architectures described in Sect. 2.

	Test Generation and Refinement Deliverable ID: D.3.1.v1.0	Page : 15 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

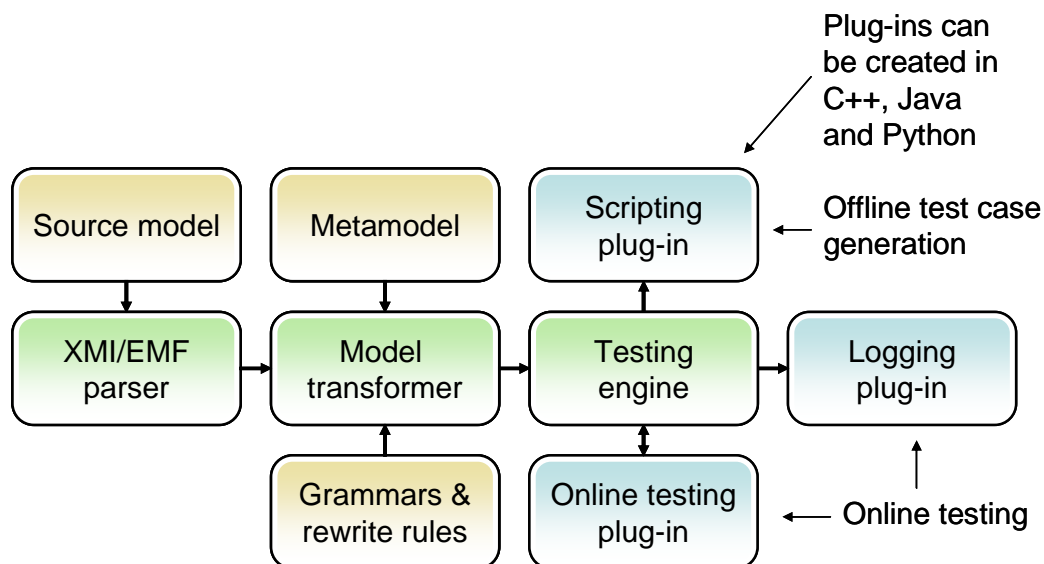


Figure 9 Conformiq Qtronic architecture, realizing online and offline test generation architectures


The test generation capability of Conformiq Qtronic is based on **symbolic execution**, which allows constructing test data efficiently without requiring users to predefine that data.

In addition to the test generation engine, a **modelling tool** has been developed to enable straightforward construction of state charts that comprise usually part of the total model given as input to Conformiq Qtronic. The other part is textual, extended Java code that is then automatically linked together with the state charts.

3.1.2.2 NModel

NModel is a model-based testing toolkit that has evolved from the experience with Spec Explorer in Microsoft Research. The models are written as model programs using C#. The contents of the toolkit are summarized in Figure 10. The toolkit provides the following:

- A modelling library that provides the abstract data structures and artefacts, such as attributes (known as annotations in Java community) that are used to designating member functions to being actions.
- A model program viewer, mpv.exe, a visualisation tool that is meant to be used in the model validation phase. It enables to view the state graph generated by the model program and to view the valuations of variables in each state. It can be used for producing graphical representations of the state graph and for exploring how the program behaves under different scenarios.
- Offline test generator, otg.exe, is an offline test generation tool. It produces test sequences to cover all actions in the model with all possible argument combinations, depending which kinds of coverage criteria are specified.
- Conformance tester, ct.exe, is a conformance tester which is meant for on-line testing, i.e. running the model in parallel with the actual implementation.
- The toolkit defines a number of interfaces for adding custom extensions. For example, we have created a tool to enumerate all states without visualising them to count the number of states and to check whether any state invariants can be violated in the model.

	Test Generation and Refinement Deliverable ID: D.3.1.v1.0	Page : 16 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

- NModel can be used in an environment that supports CLI 2.0. For compiling the model programs a C# compiler supporting C# 2.0 is required. The toolkit has been tested with Microsoft.Net 2.0 and Mono 1.2.6 (except model program viewer, which requires Windows graphics infrastructure).

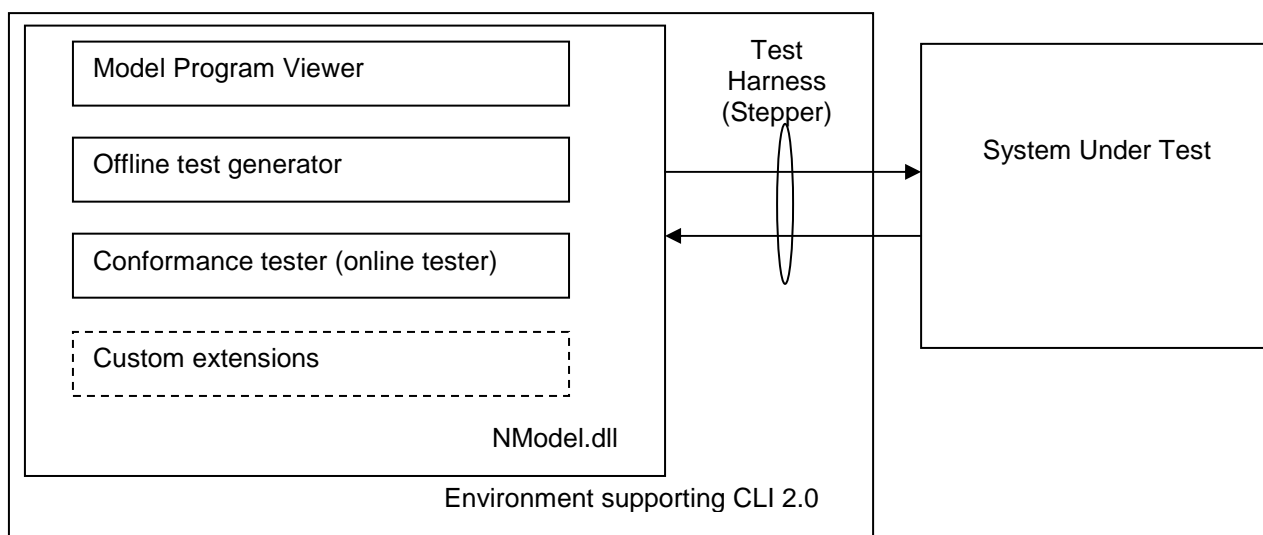


Figure 10 Architecture of the NModel toolkit

NModel enables modelling systems to use sets and objects. If we think of a system with one action where the action adds new objects to a set, we get an infinite state system, as a set can contain potentially infinite number of distinct objects.

NModel can be utilised in two different ways:

Monitoring mode


In monitoring mode NModel passively monitors the actions that occur in the system and raises an alarm when a violation of the specification is found. The test system produces a sequence of actions with predefined length that leads to the situation of specification violation.

Driver mode

In driver mode the model is an active party driving the SUT. This is typically the case when the model replaces some parts of the system, e.g. the user, some component of the system, or a combination of those. In driver mode the models are equipped with so called state filters which limit the number of objects in certain data structures.

3.1.2.3 Uppaal

Uppaal is an example of using zone abstractions for representing clocks which are thought of as real valued variables. Zone abstractions provide means to finitize the initially infinite domain and ultimately to reason about models of systems containing such variables.

	<p>Test Generation and Refinement</p> <p>Deliverable ID: D.3.1.v1.0</p>	Page : 17 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

3.1.2.4 Reactive Planner

The applicability and complexity of model-based testing and the test generation methods depend heavily on the IUT model characteristics. The most important border goes between deterministic and nondeterministic state models of the IUT.


The state model is deterministic if in each state the expected output and the next state are defined by the current state and the input. Test tool can drive the IUT from state to state by generating in the current state of the IUT the input according to the model. It can check the actual output against the expected one and it knows the next state of the IUT. Therefore a test sequence of inputs and expected outputs can be generated in advance of testing in case of deterministic model.

Nondeterministic state machine may have states where several alternative outputs are possible in reaction to an input. Each of the alternative outputs means different possible behaviors of the IUT that can lead the IUT into a different next state. The actual output in reaction to the given input and the next state depends on the IUT choice. In case of nondeterministic IUT the generated test is not a simple sequence of inputs and expected outputs anymore but it is a test tree that has branches due to nondeterministic choices of the IUT. When the IUT output is observable by the test tool then the test tool can observe the IUT actual output, compare it against to the one of the alternative expected outputs, and detect the next state of the IUT. Traditionally, test case generation systems are grouped as offline test generation systems and online (on-the-fly) test generation systems. Offline test generation means that the test cases are generated before the test cases execution. Online (on-the-fly) test generation means that the test case is generated at the same time when it is executed. Offline test generation suites best for generating test cases from deterministic models and online test generation suites best for generating test cases from nondeterministic models.

A significant measure of generated tests is their execution time efficiency. Tests execution time efficiency can be measured by the execution time needed for tests execution to satisfy the given test coverage. For simplification let's assume that all transitions on test sequence and test tree consume equal amount of execution time. Then we can say that a shorter test sequence has better execution time efficiency than a longer one and a test tree with the shorter paths has better execution time efficiency than a test tree with longer paths.

An offline generated test sequence or test tree is a complete test suite. For each deterministic model and given test purpose there exists always an efficient test sequence that is a shortest test sequence to cover the given test purpose. For each nondeterministic model there exists always an efficient test tree that consists all shortest paths to cover the given test purpose.

It can be efficient in means of test sequence length because it is possible to construct a test tree that consists of all the shortest paths towards the planned test coverage. But deriving such a test tree from the nondeterministic model that takes into account all alternative choices of the IUT is computationally more expensive than deriving a test sequence from the deterministic model because it requires the exploration of exponentially bigger state space. Therefore the common understanding in model-based testing community has been that the generation of test trees from nondeterministic models in advance (offline test generation) of the testing is infeasible for industry scale testing tasks. Instead, on-the-fly test generation is used in such cases. "On-the-fly" means that the test generation is performed simultaneously with executing it. Next subsequence of test is generated only after observing the actual output from alternative IUT expected outputs. The tests planning horizon in on-the-fly testing is significantly shorter than in offline test generation because on-the-fly test generation usually generates only next input and expected output pair or a sequence of input and expected output pairs until the next nondeterministic choice of the IUT. Step-by-step test generation with shortened test horizon consumes only fraction amount of memory compared to complete test tree and makes the test generation feasible for nondeterministic models. The benefit of on-the-fly testing over offline test generation is achieved by the cost of test planning accuracy. Lower test planning accuracy means longer test paths for meeting the specified test purpose because it is not possible to find the shortest path on the model (globally) by looking ahead with limited horizon (locally) only.

	Test Generation and Refinement Deliverable ID: D.3.1.v1.0	Page : 18 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

Reactive planning based test generation is kind a hybrid of offline and online test generation. In offline phase a planning tester is synthesized from the nondeterministic IUT state model. The synthesis bases on the reachability analysis of the model element. Based on the reachability analysis the tester state machine is constructed that can autonomously find the shortest path through the uncovered model elements during online phase.

3.2 PATTERN DRIVEN APPROACHES: PTML

The choice of an appropriate test modelling language or notation plays a very important role in the methods for test generation, since the selected language will define the input, based on which test cases will have to be generated. Therefore the selected notation should not only meet standard requirements on modelling notations such as expressivity, simplicity, comprehensibility etc., but should also support the specific requirements expressed in WP2 for test modelling notations and tools.


Pattern driven test case generation consists in automatically generating test skeletons based on test models following some previously defined and well-documented patterns.

The concept of test patterns has been around for a while. However, with the great number of different approaches on testing with regard to scope (unit-testing, sub-system-testing, system-testing), method (white-box, grey-box, black-box-testing) or purpose (conformance, interoperability, performance, reliability...), the notion of test patterns can be explained in many ways. Patterns can be defined at different levels of abstractions, ranging from high-level non-functional patterns describing good-practices for a certain domain to more functional (instantiable) ones like object-oriented design pattern. The FOKUS concept of test patterns aim at embodying the knowledge gathered by test designers in a way that, that knowledge could be used to drive intelligent test generation.

Test patterns cover all aspects of test modelling, i.e. test architecture, test data and test behaviour modelling. Therefore pattern driven test generation allows a more focussed generation of test assets (test configurations, test data, test behaviour), instead of the brute-force approaches based on emulating elements of the system under test (SUT). While existing test generation approaches consist in combining all possible input-output-state triplets for a given SUT based on its model, pattern-based test generation uses additional information stemming from domain-specific or well established and documented good practices to reduce the amount of test cases generated to those that have a higher potential in assessing the SUT's correctness. Patterns are best expressed at a conceptual level of abstraction, using MDA approaches. Therefore, it was a natural step to consider existing test modelling approaches for the purpose of expressing "instantiable" test patterns for integration in the test development process. The UML-2 Test Profile (U2TP or UTP) is the standard notation defined by the OMG for UML-based model driven testing. It defines a set of extensions to the UML 2.0 standard to enable test modelling with UML. The extensions defined in the UTP are grouped in 4 categories:

- Test architecture
- Test behaviour
- Test data
- Time concepts

Those four categories of extensions provided by the UTP match perfectly with the categories of test-patterns FOKUS defined in previous works, namely test configuration, test behaviour and test data patterns. Obviously, the only difference was that FOKUS integrated time concepts to behaviour patterns instead of defining them as a separate group of patterns

	Test Generation and Refinement Deliverable ID: D.3.1.v1.0	Page : 19 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

However, while the UTP in its current version provides most of the basic concepts required for pattern driven test modelling, these were not sufficient and hence some new concepts were needed, along with a refinement of some existing concepts. The extended UTP including those new concepts was then modelled as a meta-model for a notation FOKUS called PTML (Pattern driven Test Modelling Language).

FOKUS is developing a prototypical pattern driven test modelling and test generation tool towards U2TP(es), implementing selected aspects of the test generation methods developed in WP2. This section presents the specification for that tool and its relations to the requirements identified in the NSN case study and the ETSI case study respectively. At the current stage, the embedded system aspects identified in other cases studies presented in the D-MINT project are not covered yet and will be the object of later revisions to this document.

3.2.1 Extending the UTP: The PTML Metamodel

The PTML metamodel implements the concepts of pattern driven test-modelling. The elements of that metamodel are presented below, along with the corresponding UTP concepts, where applicable.

3.2.1.1 The PTML Metamodel

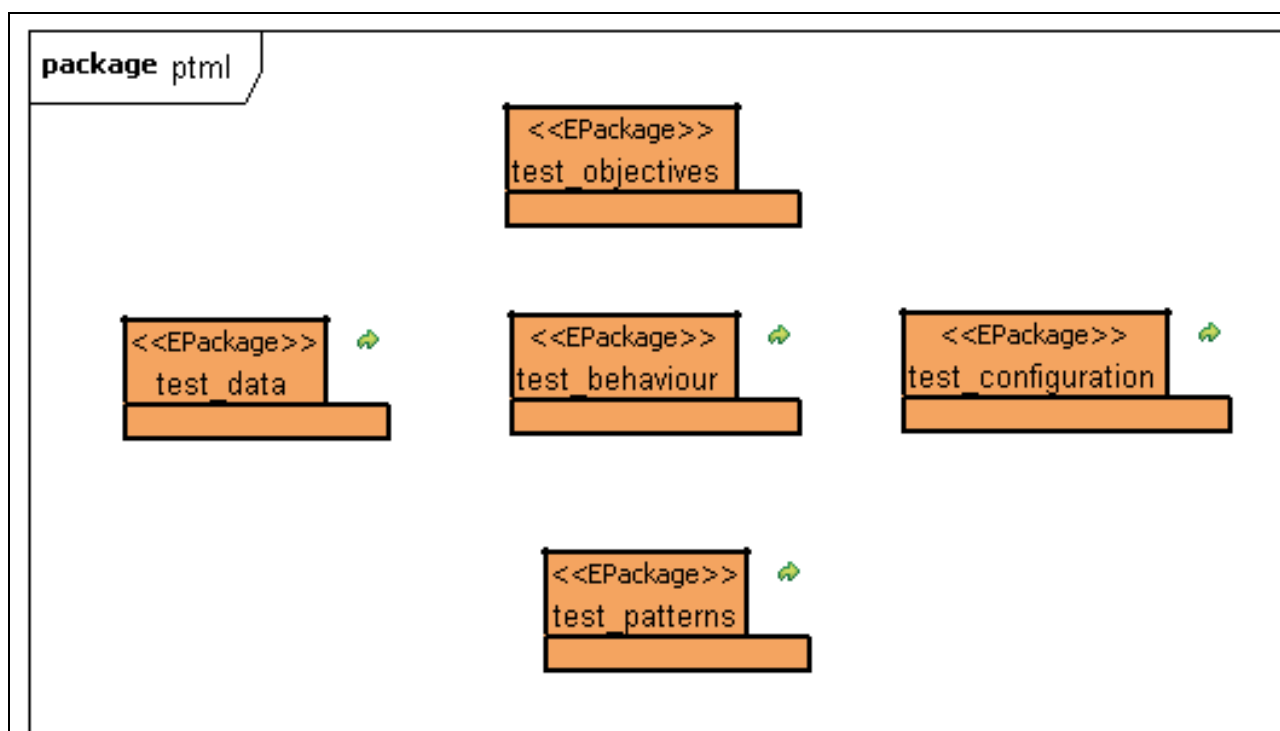


Figure 11 Overview of PTML Packages

Figure 11 displays the main packages of the PTML model reflecting its four main groups of concepts:

Additionally to the three groups of concepts already contained in the UTP, i.e. the test_data, test_behaviour and test_configuration packages, the PTML introduces two new groups of concepts as packages: the test_objectives package and the test_patterns package.

The test_objectives package contains all concepts related to the modelling of test objectives. The advantage of modelling the test objectives and the other aspects of testing using the same notation is that test cases

can be linked to test objectives to ensure a tracing back to those test objectives during test execution. Furthermore, provided a connection is made between user and system requirements and test objectives, executed test cases could even be traced back to those to evaluate coverage.

The test_patterns package defines concepts allowing the definition of test patterns, i.e. generic solutions for recurring testing problems for a specific domain or organization. An example of such a pattern could be one defining the composition of a test case as a preamble-test body-postamble triplet, or as a test body-postamble pair etc. Based on such test patterns wizards could be generated to assist the test designer in defining test cases following that pattern. An automatic validation of defined test cases could be provided as well.

3.2.1.2 Test Architecture Concepts

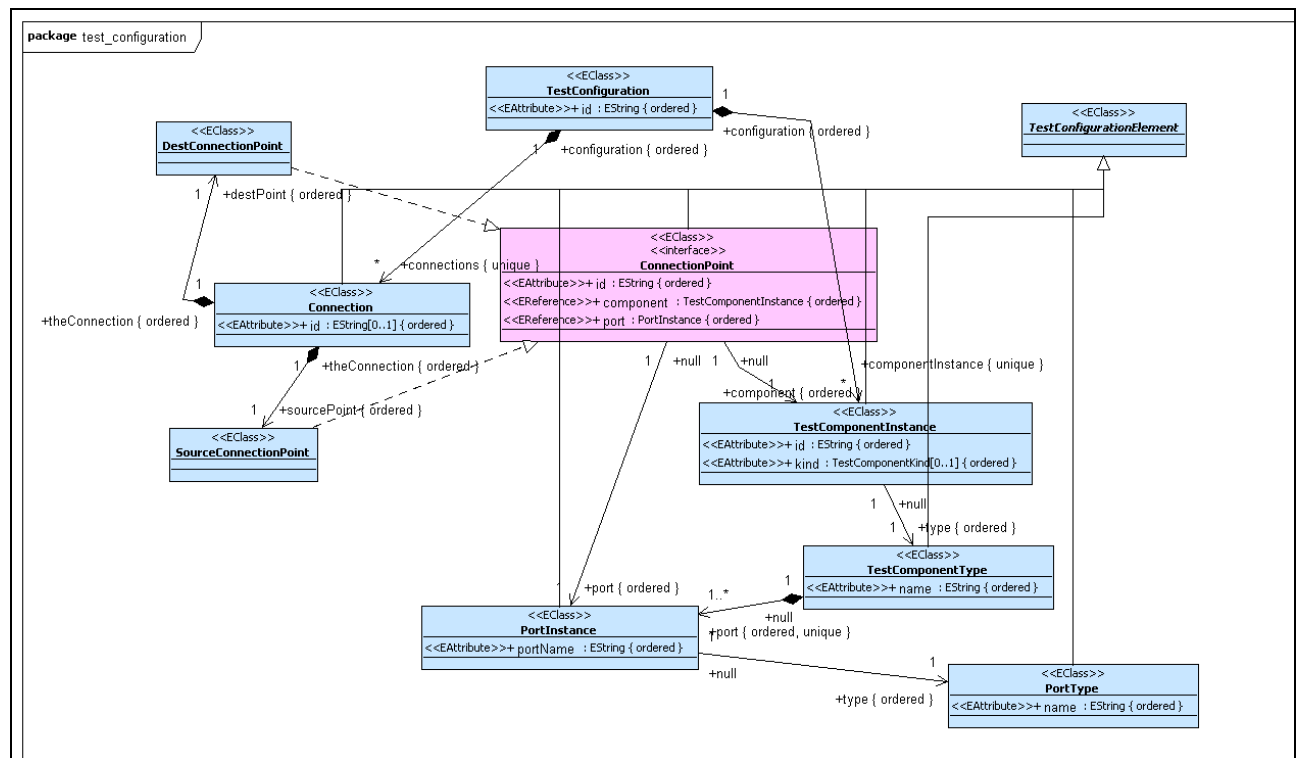


Figure 12 Test Architecture Elements of the PTML Metamodel

Figure 12 depicts the concepts defined in PTML for modelling test architectures. By test architecture, FOKUS means the various elements describing which components the test environment consist of and how they are inter-connected with each other and the components building the SUT. The central element of test architecture in PTML is the test configuration which is composed of test component instances and connections between those. A test component instance represents the instantiation of a previously defined test component type. As depicted on Figure 12, the PTML adopts most of the UTP concepts, however with a few differences.

In UTP descriptor- and instance-based concepts are mixed up without any indication of what the link between them is. E.g. for test architecture the UTP defines the test component concept, mapping it to TTCN-3 component type (descriptor). Therefore, it is not clear: If the UTP test component is descriptor concept, as it appears, then a concept for test component instance would be missing, otherwise the descriptor concept

would be missing, because test-modelling requires both descriptor- and instance-based concepts. For that reason, FOKUS separates the concept of test component type (**TestComponentType**) from that of test component instance (TestComponentInstance), to allow the usage of both in modelling the tests.

Instead of defining a separate concept of SUT, FOKUS chose to model the SUT functionality as an attribute of any component instance. Therefore, in defining a test configuration, any test component instance can be tagged as being part of the SUT.

3.2.1.3 Test Data Concepts

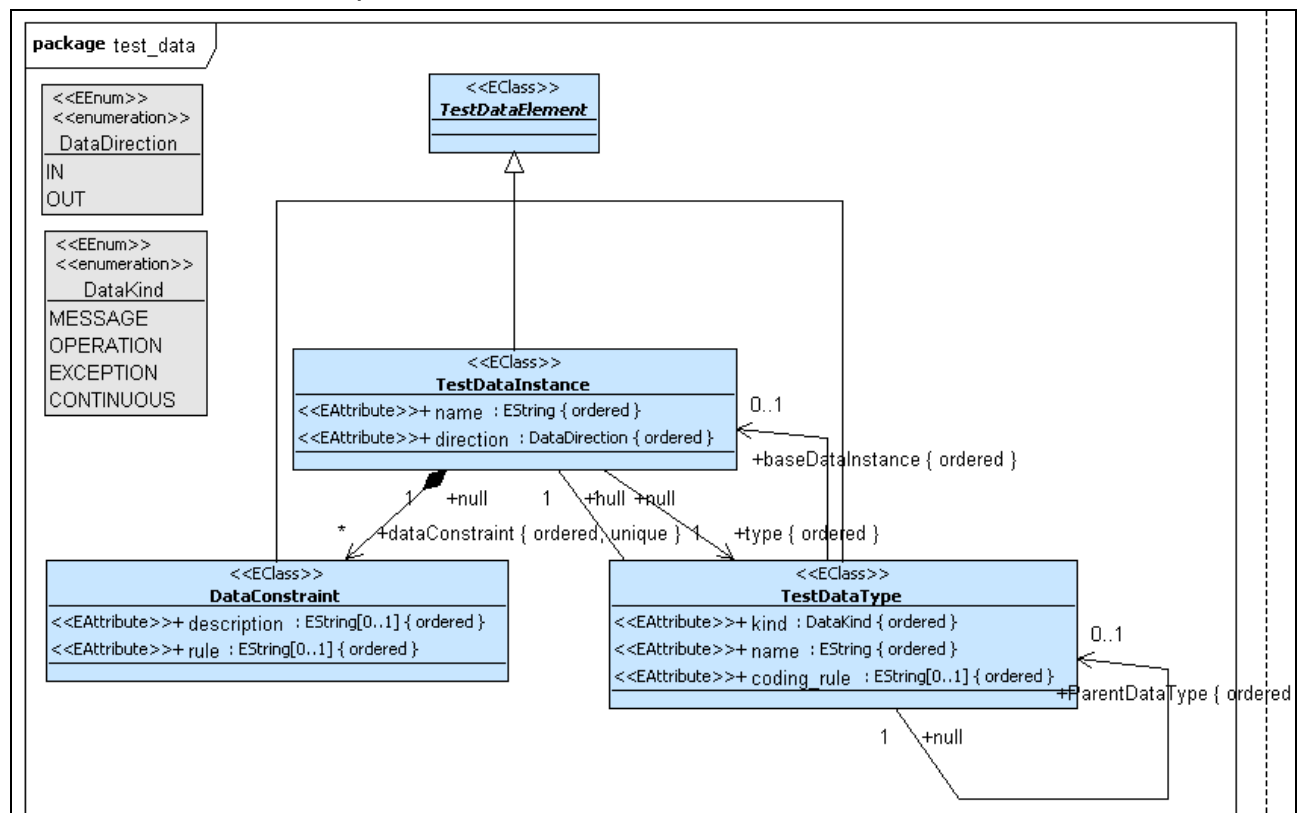


Figure 13 PTML Test Data Concepts

The UTP defines five concepts for test data modelling, i.e. wildcards, **datapool**, data partition and coding rules. As mentioned before for test architecture concepts, in this case no descriptor concept is defined. To fill that gap, PTML defines the concept of **TestDataInstance** as descriptor for a **TestDataInstance** concept. A test data instance (TestDataInstance) represents data that can be exchanged between test components, independent of whether they belong to the SUT or not. Each test data instance (TestDataInstance) has a direction (DataDirection) as attribute indicating whether it could be used as stimulus (OUT) or as observation (IN,OUT). Furthermore each TestDataInstance is associated to a test data type (TestDataInstance) and to a constraint (DataConstraint), defining its purpose. A constraint for an observation might be a wildcard, a random-value, a class-partition or any other data specification mechanism described as a character string (e.g. OCL).

3.2.1.4 Test Behaviour Concepts

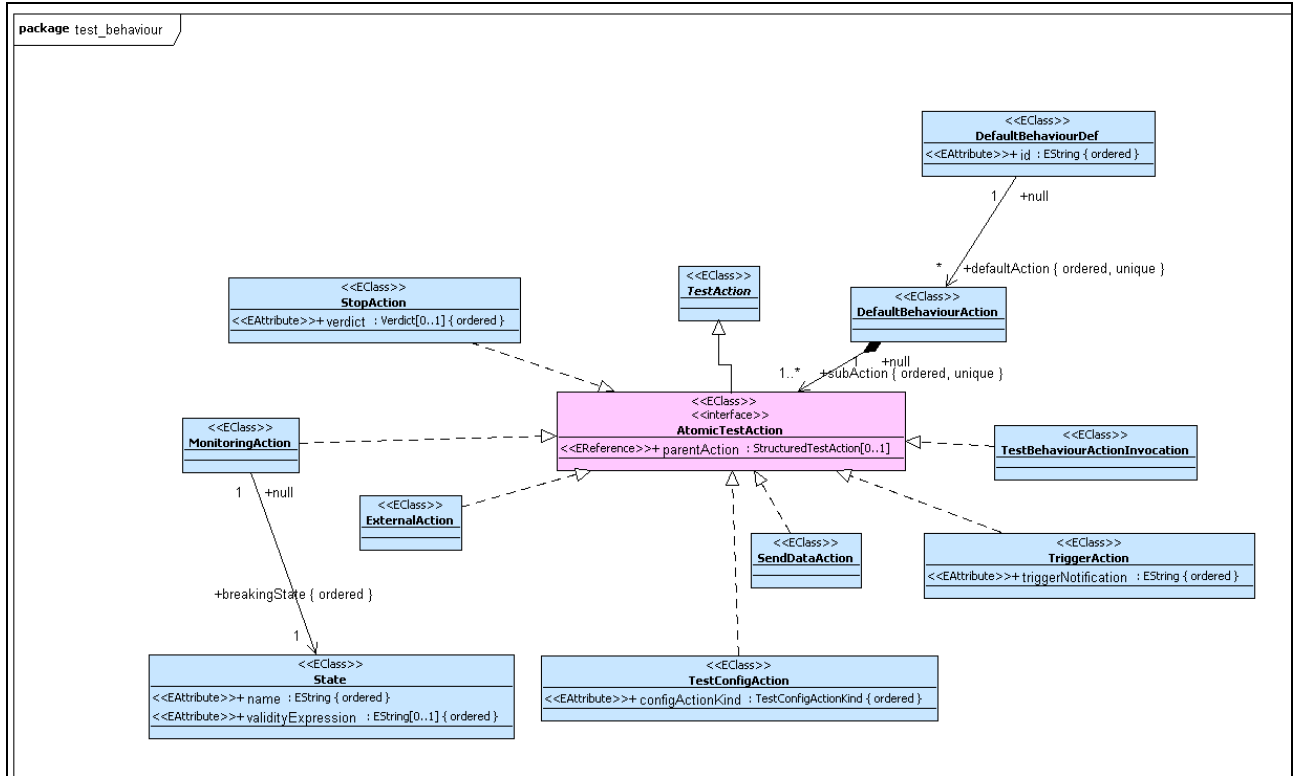


Figure 14 Atomic Test Actions Concepts

Test behaviour concepts in PTML can be grouped in two main categories: atomic test actions are the basic elements of test behaviour. They include state actions aiming at monitoring a specific state at the SUT (MonitoringAction), exchanging test data with the SUT (SendDataAction, ReceiveDataAction), invoking predefined behaviour etc. Additionally, UTP concepts are included as well. Figure 14 depicts the atomic test actions elements of the PTML meta model and how they relate to each other.

Additionally FOKUS introduces structured test actions that are compositions of atomic test actions, along with other test behaviour elements. The main elements of structured test actions are test cases and the TestBehaviourActionDef which is a descriptor for structured test actions (StructuredTestAction) and test cases (Testcase). As depicted on Figure 15, the Testcase meta-class is an extension of the TestBehaviorActionDef meta-class. In TTCN-3, the TestBehaviorActionDef meta-class would map to a function definition. Furthermore, as depicted on that figure, each test case can be linked to a Test Purpose, i.e. a precise Test objective as defined in a test specification document to ensure test case traceability.

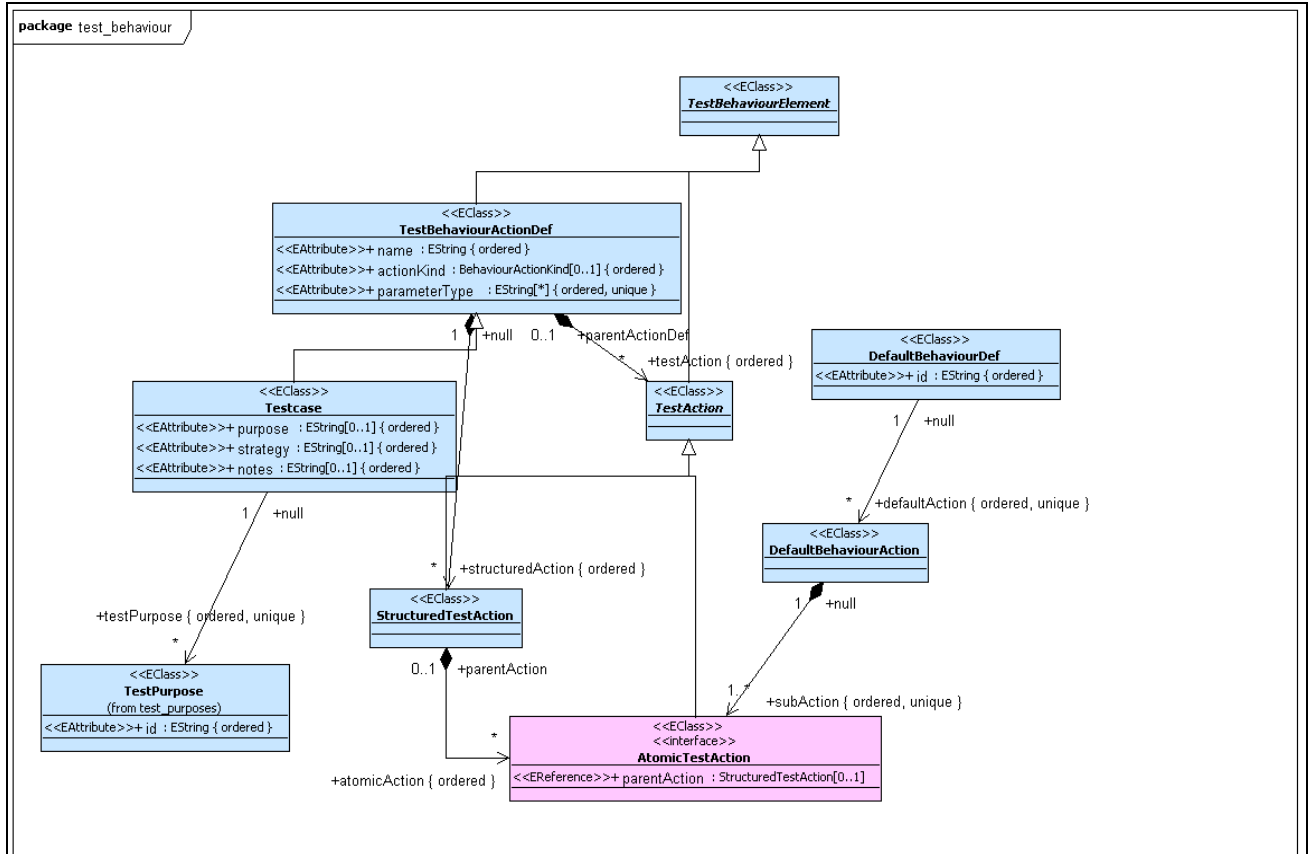


Figure 15 Structured Test Actions Concepts

3.2.1.5 Test Patterns Concepts

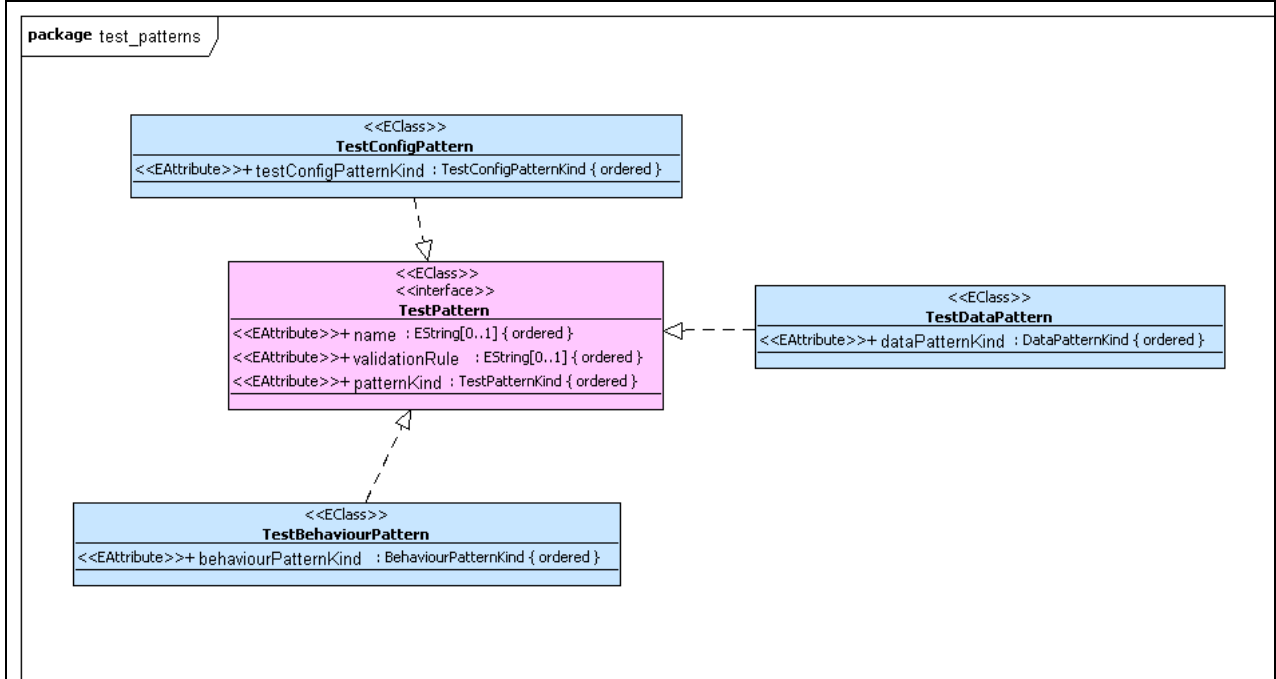


Figure 16 Test Pattern Concepts

Additionally to built-in test patterns covering test data, test behaviour and test architecture, PTML defines a collection of concepts to allow the modelling of test-patterns according to specific domain or test project requirements. Built-in patterns are pre-defined generic models instantiating given patterns. Examples of built-in patterns for test architecture can be used to model specific test configurations according to pre-defined topologies such as point-to-point (P2P), point-to-multipoint (PMP) or mesh. For those predefined patterns a set of OCL rules are included to validate the test configuration models created based on those patterns with regard to consistency.

As depicted on Figure 16, the generic test pattern meta model allows the specification of such a validation rule for each defined pattern. Each of the meta-classes embodying the categories of test patterns is defined as consisting of a name, a validation rule and a set of context elements. Context elements are instances of the base meta-class for each of the category. E.g. Figure 17 depicts the meta model for the test data patterns category. For each element being part of the pattern, a processable string (e.g. OCL) defining its role is required to ensure that during test modelling the test pattern can be validated on consistence.

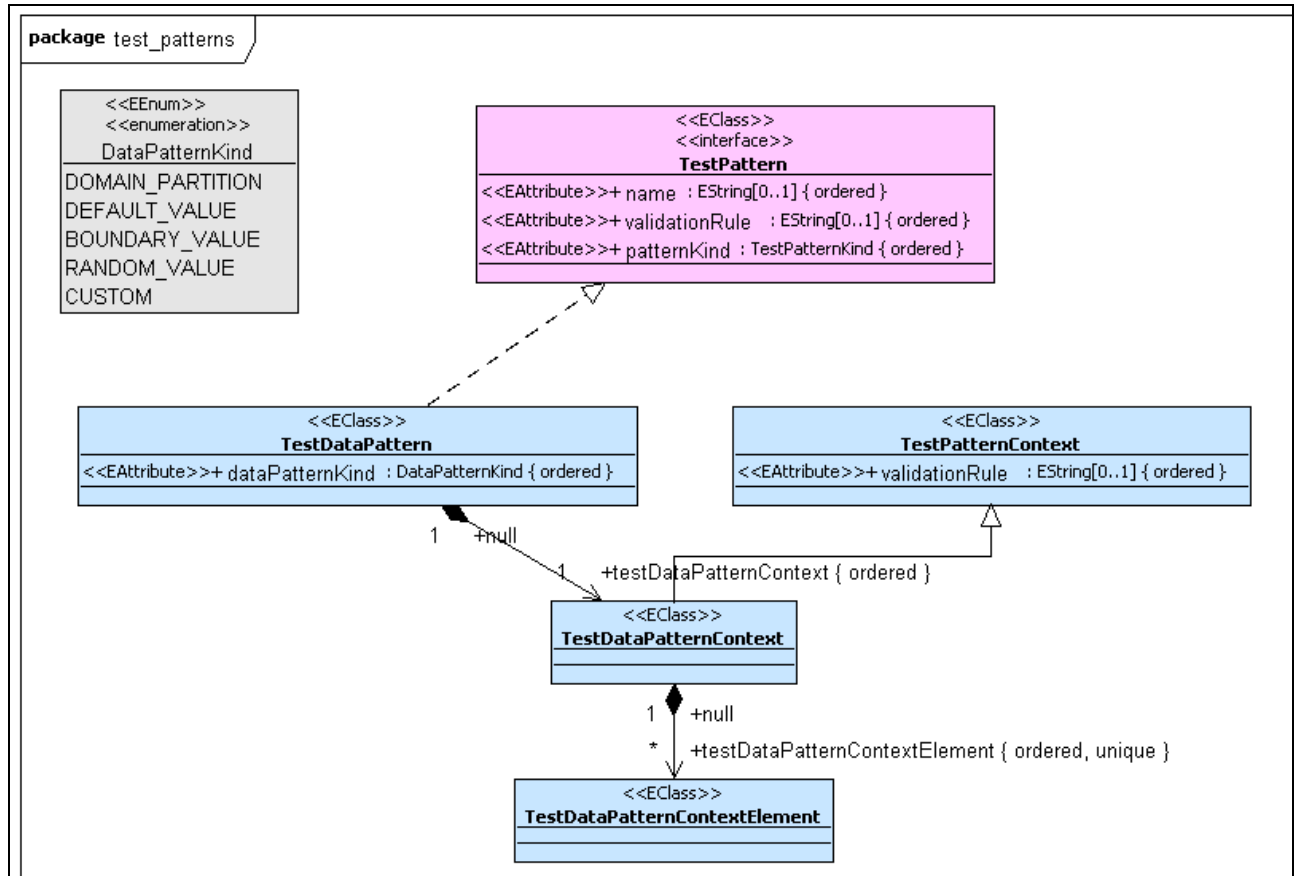


Figure 17 Test Data Patterns Meta Model

3.2.2 Pattern driven Test Generation Architecture

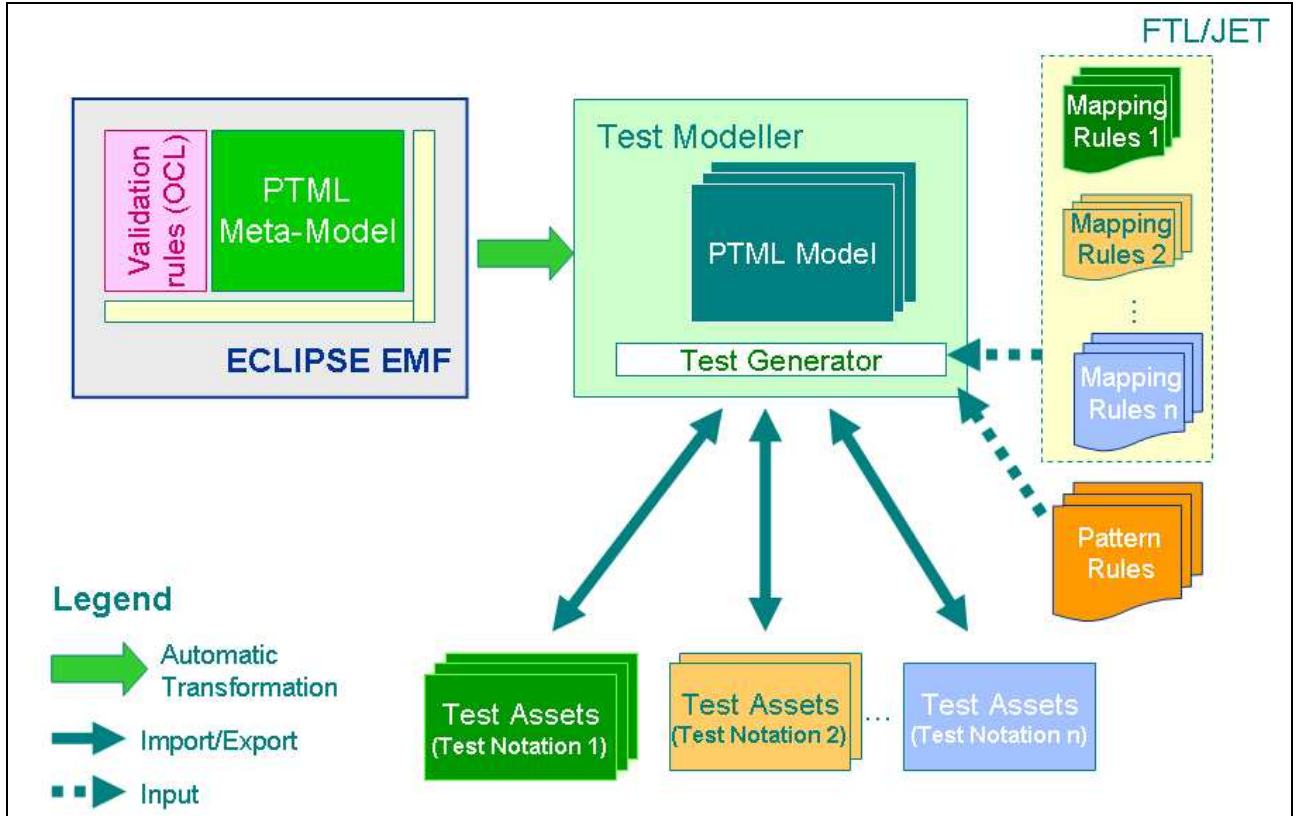



Figure 18 Architecture of the Pattern driven Test Modeller

As depicted in Figure 18, the architecture of the pattern driven test modelling tool follows a model driven approach as well. The pattern driven test modeller is based on a meta-model for the PTML notation defined as an Eclipse EMF ECore meta-model. The Eclipse Modelling Framework (EMF) is a modelling and data integration framework which exploits the facilities provided by the Eclipse IDE to generate code without losing user customizations (merging) and to facilitate tool development and integration into the Eclipse IDE. EMF is compatible to UML and supports the specification of validation rules for a meta-model using the OMG's Object Constraint Language (OCL). Based on the PTML Ecore meta-model and its associated set of OCL validation rules, the Eclipse EMF automatically generates an Editor for the PTML notation which is the pattern driven test modeller's main tool. Additionally a validation checker is generated automatically as well, which, in combination with the editor, can be used for static validation of the defined test models. The advantages of using such a model driven approach is mainly that, the focus of the effort can be laid on the conceptual aspects of test modelling, while the Eclipse EMF framework completely takes care of the tooling aspects through automated (JAVA) source code generation. Although the generated tool chain leaves room for improvements with regard to visualization and comfort in usage, it meets all basic requirements and provides mechanisms for customization to meet further user requirements, if any.

For test generation, a classical approach is used, consisting in defining a set of mapping rules for the target test notation, i.e. the notation in which the generated test cases would be expressed (e.g. TTCN-3, X-Unit, JAVA, etc.). As depicted on Figure 18, more than one mapping rule can be supported, with each of them defined as a plug-in to the test modeller. Additionally a set of test patterns rules is provided as well, as guidance for the test generator. The mapping rules could be specified using a template-notation such as the Freemaker Template Language (FTL), the Eclipse's JAVA Emitter Template (JET) notation combine with an

	Test Generation and Refinement Deliverable ID: D.3.1.v1.0	Page : 27 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public


XML-based description for the plug-in parameters. Providing a plug-in mechanism for the mapping rules and test patterns definitions ensures that, based on the same test model the generated test assets (test data, test architecture elements, test case skeletons) could be expressed in any language for which a mapping has been provided.

The prototype implementation will include a mapping to the TTCN-3 notation.


3.2.3 Requirements coverage

Table 1 below lists the requirements identified by the NSN case study and their coverage by the PTML Test Modeller. For each of the requirements that are covered by the tool, an indication of the feature covering that requirement is given as well. In a later update of this document, a similar table will be provided for the requirements identified by the ETSI case study as well.

Requirement ID	Description	Page	Support	Covering Feature
Requirement 1	Tool assisted transitions between the tasks of the development process.	13	<input checked="" type="checkbox"/>	IDE Integration
Requirement 2	Tool assisted feedback loops in case of problem fixing.	14	<input checked="" type="checkbox"/>	EMF-Built-in
Requirement 3	Debugging of tests.	14	<input type="checkbox"/>	N/A
Requirement 4	Traceability of requirements.	15	<input checked="" type="checkbox"/>	Test Objectives Modelling
Requirement 5	Modularity of models.	15	<input checked="" type="checkbox"/>	PTML Imported model attribute
Requirement 6	Reusability of models.	16	<input checked="" type="checkbox"/>	PTML Imported model attribute
Requirement 7	Elements of architectural models.	17	<input checked="" type="checkbox"/>	PTML
Requirement 8	Modelling static configurations.	18	<input checked="" type="checkbox"/>	PTML
Requirement 9	Modelling arbitrary data types.	18	<input checked="" type="checkbox"/>	PTML
Requirement 10	Support of ASN.1 type definitions.	19	<input type="checkbox"/>	
Requirement 11	Automatic value completion of value structures according to type definitions.	19	<input checked="" type="checkbox"/>	N/A but similar functionality with model content

	Test Generation and Refinement Deliverable ID: D.3.1.v1.0	Page : 28 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

				assistants
Requirement 12	Support for definition of message sequences and detailed behaviour.	20	<input checked="" type="checkbox"/>	PTML Behaviour concepts
Requirement 13	Support for multiple interfaces, protocols and connections.	20	<input type="checkbox"/>	
Requirement 14	Support for concurrency and co-ordination.	20	<input checked="" type="checkbox"/>	PTML behaviour concepts
Requirement 15	Support for the notion of time.	21	<input checked="" type="checkbox"/>	PTML
Requirement 16	Resolution of time.	21	<input type="checkbox"/>	
Requirement 17	Intuitive principle of integration of models	21	<input checked="" type="checkbox"/>	
Requirement 18	Syntax checking.	22	<input checked="" type="checkbox"/>	N/A but validation check
Requirement 19	Analysis of static semantics.	22	<input checked="" type="checkbox"/>	OCL-based validation support
Requirement 20	Intuitive and accurate error and warning reporting.	23	<input checked="" type="checkbox"/>	OCL-based validation support
Requirement 21	Fast model analysis feedback.	23	<input checked="" type="checkbox"/>	OCL-based validation support
Requirement 22	Analysis guidance.	23	<input checked="" type="checkbox"/>	OCL-based validation support
Requirement 23	Analysis of suspicious constructs.	24	<input checked="" type="checkbox"/>	OCL-based validation support
Requirement 24	Model re-factoring.	24	<input checked="" type="checkbox"/>	Built-in EMF
Requirement 25	Support for black-box testing.	25	<input checked="" type="checkbox"/>	PTML
Requirement 26	Automatic test generation.	25	<input checked="" type="checkbox"/>	TTCN-3 Export support
Requirement 27	Automated execution.	26	<input type="checkbox"/>	N/A

	Test Generation and Refinement Deliverable ID: D.3.1.v1.0	Page : 29 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

Requirement 28	Continued execution.	26	<input type="checkbox"/>	N/A
Requirement 29	Support of a notion of a test.	26	<input checked="" type="checkbox"/>	PTML
Requirement 30	Combining execution criteria.	27	<input checked="" type="checkbox"/>	PTML for Test Objectives
Requirement 31	Execute a specific scenario.	27	<input type="checkbox"/>	
Requirement 32	Reproduce the test based on saved information of a failed test.	28	<input type="checkbox"/>	
Requirement 33	Support of statistical testing.	28	<input type="checkbox"/>	
Requirement 34	Focus on fault-revealing parts of a model.	29	<input type="checkbox"/>	
Requirement 35	Test changed parts of a model.	29	<input type="checkbox"/>	
Requirement 36	Regression testing.	29	<input type="checkbox"/>	
Requirement 37	Combining coverage criteria.	29	<input type="checkbox"/>	
Requirement 38	Coverage control of evaluation criteria.	30	<input type="checkbox"/>	
Requirement 39	Optimised test generation.	30	<input checked="" type="checkbox"/>	Test Patterns Concepts
Requirement 40	Supported evaluation criteria types.	30	<input type="checkbox"/>	N/A
Requirement 41	Support for on-line execution mode.	31	<input type="checkbox"/>	N/A
Requirement 42	Support for off-line execution mode.	31	<input type="checkbox"/>	N/A
Requirement 43	Equality of on-line and off-line tests.	32	<input type="checkbox"/>	N/A
Requirement 44	Visualization.	32	<input type="checkbox"/>	Future
Requirement 45	Traceability to model.	32	<input type="checkbox"/>	N/A
Requirement 46	Test campaign information.	32	<input type="checkbox"/>	

Table 1 Coverage of NSN case study requirements for Pattern driven Test Modeller and Generator

3.3 OFFLINE TEST CASE GENERATION FOR CONTROL SYSTEMS: TPT

Test case generation using TPT is based on abstract test models or usage models. A TPT test/usage model is a “superset state machine” that can be seen as the union of all relevant test cases. With TPT each individual test case is not just a constant sequence of test steps but a full state machine that is part of the superset state machine modelled with TPT. TPT test models are hybrid state machines that allow the stimulation and observation of discrete and continuous input and output of the SUT. This concept allows the

definition of powerful *reactive tests* (also known as *closed-loop tests*) where the test case behaviour may depend on the system behaviour at runtime (online).

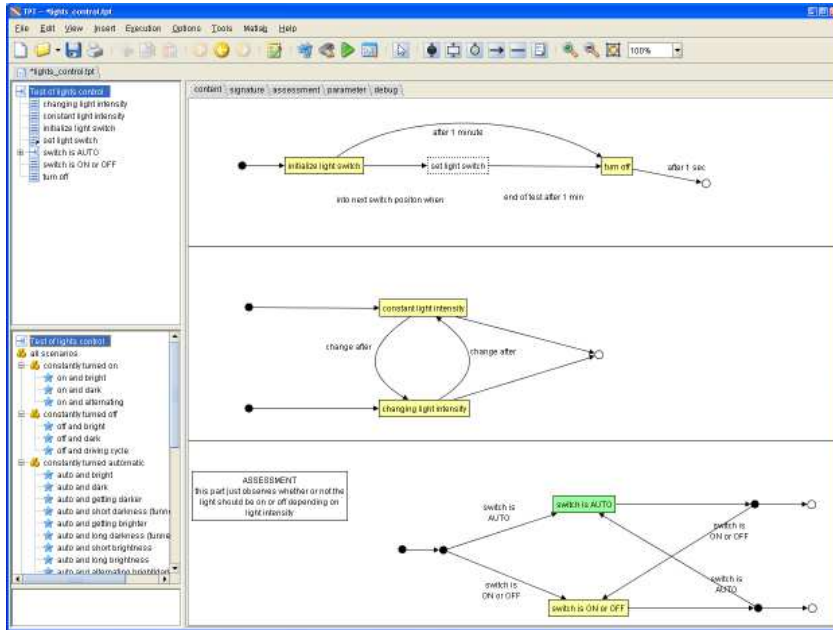


Figure 19 An example of a TPT state machine

Test cases with TPT are generated offline either manually or automatically using combination rules (test pattern rules). Test generation is the process of selecting, compiling, linking (with additional resources) and mapping test models to the actual SUT.

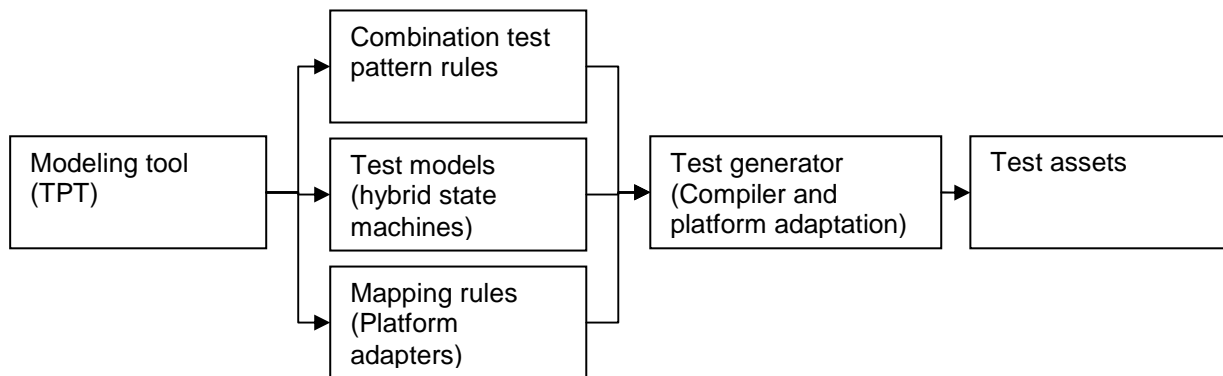


Figure 20 Pattern Driven Test Generation Architecture with TPT

TPT test cases describe the abstract semantics of the desired test behaviour. The mapping from the test models to the actual SUT is done by means of so called platform adapters that define the mapping rules between the test model and the SUT, i.e. how to represent test cases in the desired test environment and how to stimulate and observe the concrete SUT interface. With that the TPT test models can be shared and reused between different SUT implementations. This feature is highly relevant in the embedded domain where the system development goes through a couple of development phases – from initial models through the implemented software to the final embedded device.

In the domain of embedded control systems *system parameters* are crucial to be considered when testing. System parameters are configuration settings that allow an embedded system to be used in different environments. System parameters specify exactly the environment specific setup. In this sense an embedded control system constitutes a system family that has to be tested in a broader variety of scenarios. Usually the following categories of system parameters exist:

- **Enabling parameters:** Parameters that signal the existence/absence of physical or SW-features in the environment of the SUT. In dependence of those parameter settings some system algorithms/features etc. behave different.
- **Setup parameters:** Parameters that can be tuned to specific physical (e.g. electrical, mechanical) or legal ranges. Those parameters are usually used to adjust the system functionality to perfectly interact with the physical environment or to meet legal constraints of particular countries.
- **Adoptable parameters:** Parameters that can be tuned at runtime to adjust the system behaviour automatically depending on the online system observation functions and the expected, built-in obsolescence of mechanical parts.

When testing systems with parameters the test models must support parameter definitions and parameter variation. Since parameter setup can be rather complex in practice and there are many dependencies between the chosen parameter settings, parameters cannot be defined in the test model only but there on both sides, the test model *and* the SUT. This feature requires a handshake protocol of parameter settings between the test model and the SUT.

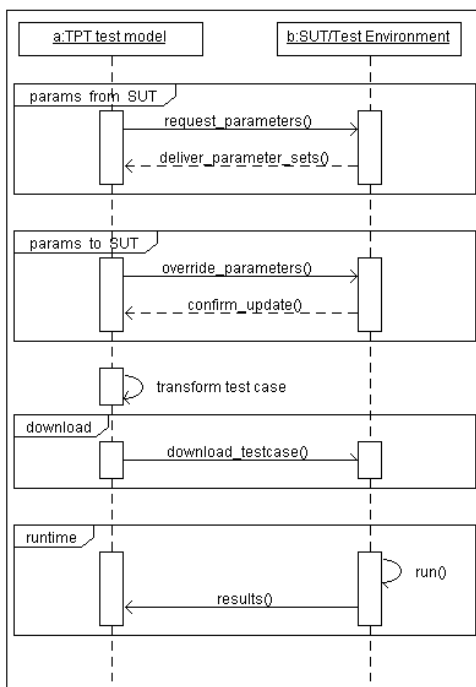



Figure 21 Parameter handshake protocol of parameter weaving into test cases

	<p>Test Generation and Refinement</p> <p>Deliverable ID: D.3.1.v1.0</p>	Page : 32 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

Before executing a test case parameter settings are requested from the SUT by TPT. Usually these parameters are defined in the simulation or test environment of the SUT – at least for a system parameter subset. As soon as TPT receives the parameter settings it will be checked which of these parameters are defined in the test model too and which parameters are not defined in the test environment in the first place. These parameters have to be overridden in the system setup. For that purpose TPT sends the parameter bindings to the SUT. After confirmation that parameter settings have been adjusted TPT transforms the abstract test cases together with the parameter bindings into executable test cases. Finally these generated test cases are downloaded and executed in the actual test environment.

3.4 TEST HARNESS GENERATION

3.4.1 NModel

Test harness glues the abstract model together with the actual system. Creating a test harness is an important stage in model-based testing for it is in the harness where the abstractions made during modelling need to be resolved into actual data and constructs that are there at the system level.

The toolkit NModel provides the means for creating the test harness in the native environment – C#. The code making up a test harness needs to fulfil the IStepper interface, figuratively named in NModel to mean that the stepper translates each “step” of the model into an equivalent step or a set of steps at the implementation level.

4. MECHANISMS FOR ITERATIVE TEST REFINEMENT

4.1 MAINTENANCE OF TEST CASES

When system specifications change incrementally (e.g. when a new increment or a new release is being specified), test cases need to be also maintained incrementally.


In the context of the pattern driven architecture this entails straightforward and manual modification of the test cases as modelled, and re-generation of test assets.

But in the system model driven approaches the situation is more complex because the user cannot manually modify the test cases themselves; they are an automatically generated set of assets and it is the input (the system model) that needs to be changed. But it becomes then a nontrivial problem how to explain to the user what has changed and what not in the test case domain, and, for example, should a newly generated test case be considered a “new version” of an older one, or a new, independent one.

4.2 MAINTENANCE OF TEST HARNESSES

4.2.1 NMODEL

When the model is modified, for example, some actions are added or the number of arguments of some actions has changed, it is necessary to update the test harness to accommodate such changes. The main benefit of maintaining test harnesses built using NModel is that NModel supports requirements-based modelling in a way that supports tracing a requirement to certain artefacts in the actual model. If some requirements of the system change, it is possible to trace which parts of the model implement these requirements. Only these parts of the model have to be modified.

	Test Generation and Refinement Deliverable ID: D.3.1.v1.0	Page : 33 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

5. USABILITY AND HUMAN-COMPUTER INTERACTION

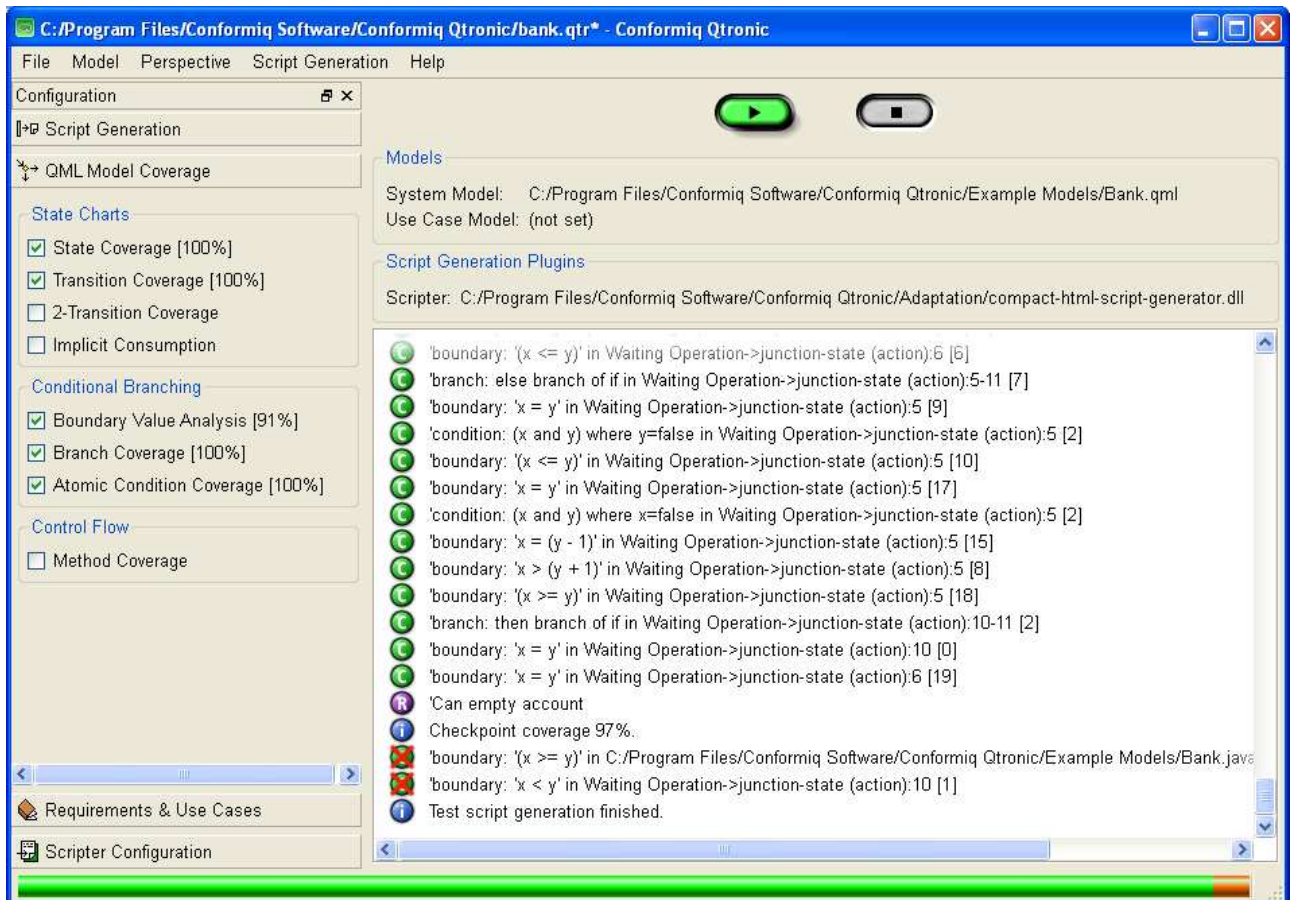
5.1 NMODEL


In NModel models are C# programs. As there are a fair amount of Java and C# programmers around, it can be said that understanding the semantics of the program code is fairly widespread. Still there is an important model validation phase after writing the model where it is possible to visualise the transition system generated by evaluating the model programs step-by-step. This is supported by the model program viewer, mpv.exe that is part of the NModel toolkit. In addition to viewing the state space of the model programs as a finite state machine, it is possible to visualise the state graphs of each state. This is important for understanding the structure of data and how it evolves in a program.

The general maintenance and versioning of model programs is meant to be done with industry standard software configuration management systems.

The current state of the art of visualisation tools needs to be developed further. For example, there should be tool support for building scenarios. It should also be possible to group variables and data structures according to the model features they belong to.

5.2 CONFORMIQ QTRONIC



	Test Generation and Refinement Deliverable ID: D.3.1.v1.0	Page : 34 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

The basic operation of the Conformiq Qtronic [AH07, QTRONIC] consists of loading model in, selecting test case generation criteria, and generating tests. As such, the human-computer interaction cycle is relatively simple. However, as part of the D-MINT efforts, Conformiq Software is working on a more interactive user experience that is based on the **Eclipse architecture**. As the writing of this report, this work is still ongoing.

5.3 QTRONIC EAST SCRIPTER PLUG-IN CONFIGURATION

EAST scripiter plug-in is configured in Qtronic Scripiter Configuration sub-window. All necessary parameter to generate variety of test cases and suites will be gathered here. See Figure 22 for initial version.




Figure 22 Qtronic EAST scripiter plug-in configuration window

6. CONCLUSIONS

In this document four main architectures for test generation were presented: pattern driven; offline; online; and online with precomputation. It is straightforward to argument that the benefits provided by these architectures are **complementary**:

The **pattern driven** approach provides a test-centric view with the promise of significantly improving test design and maintenance productivity by the provided capability to leverage and share **test patterns** that embody either general or testing-specific technical knowledge.

The **system model driven** approaches in whole, instead, provide a system-centric view and a completely automated approach to test design and selection. The test-centric view is useful in contexts where the specification and validation process is built heavily around the concept of a human-designed test case, whereas the system-centric view is appropriate in contexts where specification and validation are driven by functional requirements.

	<p>Test Generation and Refinement</p> <p>Deliverable ID: D.3.1.v1.0</p>	Page : 35 of 35
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

Within the system model approach, the **offline test generation** architecture provides the best run-time test execution performance as all computationally intensive tasks have been carried out during the offline generation. Also, it provides **maximal repeatability of tests** because every test case is deterministic.

On the other hand, the **online architecture** provides **support for nondeterministic models** (and hence tests) but has worse performance characteristics. However, because there is **no** offline calculation performed, the lead time to start test execution is very minimal. This can be extremely useful in exploratory and agile setups.

The **online with precomputation** approach provides a trade-off to improve the run-time performance of online testing but at the same time to impose a cost in terms of longer lead-time to **starting** test execution. Hence, it will be relatively more beneficial in those cases where the model is stable compared to those cases where the model is being created in a radically exploratory fashion.

7. REFERENCES

- [AH07] Antti Huima: *Implementing Conformiq Qtronic*, in Alexandre Petrenko, Margus Veanes, Jan Tretmans, Wolfgang Grieskamp (Eds.): *Testing of Software and Communicating Systems*, 19th IFIP TC6/WG6.1 International Conference, TestCom 2007, 7th International Workshop, FATES 2007, Tallinn, Estonia, June 26-29, 2007, Proceedings. Lecture Notes in Computer Science 4581 Springer 2007, ISBN 978-3-540-73065-1.
- [QTRONIC] <http://www.conformiq.com/>