UML for Validation:

Experimenting automatic test generation for flight software validation

A. Philippe HYOUNET[1], B. Jérémie POULY[2]

1: ASTRIUM SAS, 31 rue des cosmonautes 31402 Toulouse Cedex 4
philippe.hyounet@astrium.eads.net
Phone : +33 (0)5 62 19 5125
Fax : +33 (0)5 62 19 71 58
2: *CNES, 18 avenue Edouard Belin 31401 Toulouse Cedex 9*
jeremie.pouly@cnes.fr
Phone : +33 (0)5 61 28 23 67
Fax : +33 (0)5 61 27 45 5

Abstract:

UML for validation is a CNES study that aims at prototyping and experimenting automatic test generation technologies in the context of a model-based approach applied to on-board software development and tests. Starting from real test cases and test procedures taken from state-of-the-art onboard software, we first applied a reverse engineering methodology to obtain an augmented software specification model, i.e. ready to support automated test generation. In parallel, we defined and prototyped a test generation tool using innovative model-based technologies based on EMF (Eclipse Modeling Framework). Finally, a representative end-to-end experiment was performed to evaluate the benefit of such technologies.

1. Model driven Engineering

To deal with the increasing complexity of space systems while maintaining flight software high validation level, software engineering techniques must evolve accordingly. Model-based engineering aims at making complexity management easier by constructing virtual representations that enable early prediction of behaviour and performance of a system, as well as documentation and code generation. Among the various modelling techniques, UML is the one that fits best the on-board software domain.

1.1 Interest of model-based software management

In Model-Driven Engineering; the model is the reference for all activities. The use of Domain Specific Languages (DSL) and model transformations benefits the whole software development life cycle by allowing a refinement-based approach.

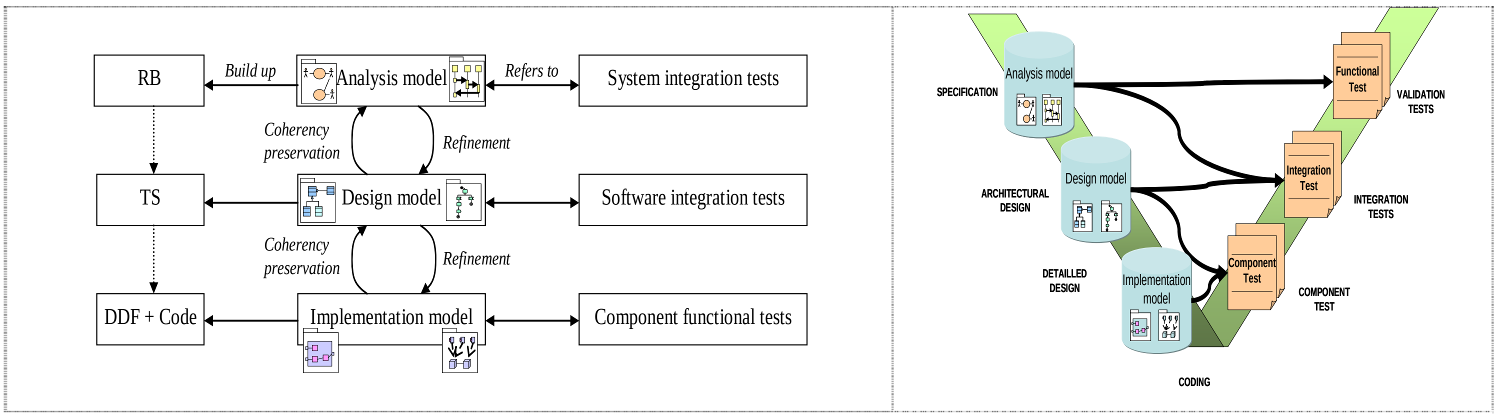


Figure 1: Model-drive engineering in the V-cycle

During specification phase, the model supports requirements capture and analysis, and provides a communication ground between all contributors to the developed product. Thus the model allows a better understanding of the future software itself and makes documentation reading and reviews easier.
During architectural design phase, the use of models facilitates standardisation and reuse through the design of configurable software building blocks. Patterns and framework contribute to know-how and knowledge capitalisation for maintenance and reuse. Executable models enable early verification and validation (V&V) through simulation. Expected properties can also be modelled and verified during software specification and design phases. Impact analysis should be done automatically when modifying an existing product.
During detailed design phase, model provides automatic code generation facilities, which reduce iteration time allowing rapid prototyping on real target.
Finally, automatic test generation produces test procedures automatically from test case specification, with optimal coverage of requirements and design in the test plan definition.

1.1 Automatic test generation

Consistency and traceability between development and validation is improved by including the formal definition of the test cases in the software UML model. Test case definition is based on object oriented concepts in order to factorize common elements and improve validation productivity. Automatic generation enables to better separate test cases and the test environment: this improves test portability and thus software reuse efficiency.

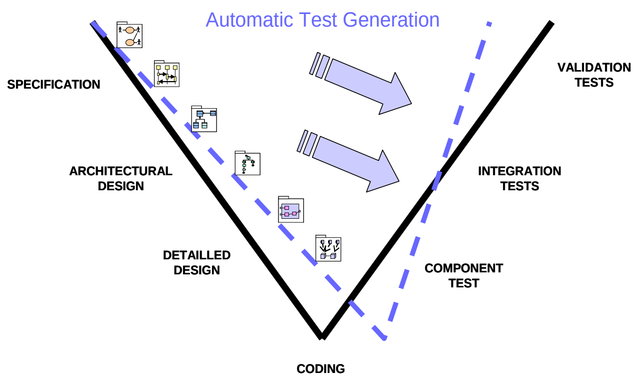


Figure 2: Impact of automatic test generation on the software development cycle

The additional modelling effort required during the specification and design phases is recovered during validation phases through automatic code and test generation. Maintenance effort and in particular regression testing are significantly reduced.

2. Test generator prototype implementation

During the UML for validation study, we have implemented a model-based process at architectural design phase level. Whenever it was possible we used innovative open source model-based technologies, such as model transformation and code generation. Thus we managed to reduce time and cost for the development of the test generator prototype, while ensuring reuse capability.

2.1 Actors and relationships

Implementing model-based engineering methodology involves different actors due to the numerous skills required. Relationships between actors has been identified and illustrated on the figure below for integration test phase relative to architectural design validation. Similar process should be applicable for validation tests phase.

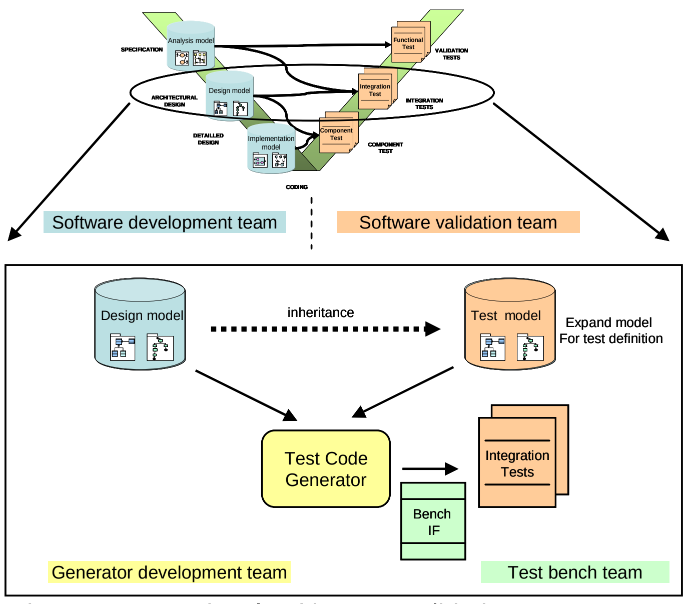


Figure 3: Actors involved in UML validation process

UML validation process involves the following actors:

- The software development team which provides UML design models to test (corresponding to Architectural Design document) to validation team.
- The software validation team which implements UML test models corresponding to test specification document and specifies test code generator to generator development team. Then the validation team gets the environment to automatically generate test procedures and performs them on the test bench.
- The generator development team which implements the test code generator tool according validation team specifications.

- The test bench team which implements test bench interface specific code.

Note: specifications documents, such as Architectural Design and test Specification, are part of the UML model and will be automatically generated by the UML modeller tool. For the study use case, this part is out of scope and specifications document are used for model implementation.

2.2 Modelling

A model-based engineering approach has been applied. The software architecture design model, which substitutes to software specification, has been implemented in UML. Selected test objectives have been added to the UML model using Object Constraint Language. Observability and commandability, involved in OCL constraints, has been specified by adding stereotypes in the UML model. Finally, test procedures have been added using UML activity diagrams.

Stereotypes

Stereotypes are used to clarify the model with label information added in the graphical description and tag useful information on UML elements (class, attributes, and operations) for code generation. Two profiles have been defined for the study:

- Testing profile allow tagging model classes to identify design and test models and to specify test bench interface.
- Tmtc profile allow tagging model class properties and operations to specify observability and commandability attributes.

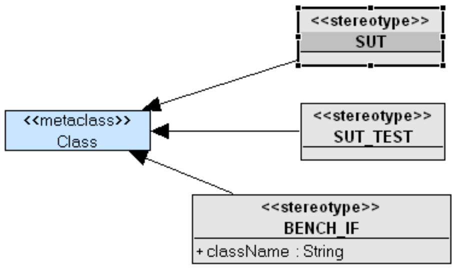


Figure 4: Testing profile stereotypes

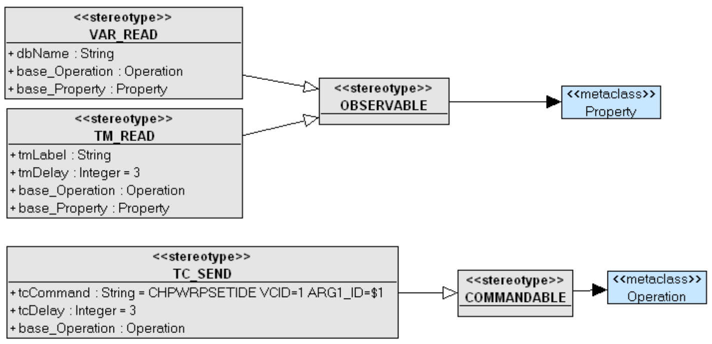


Figure 5: TM TC stereotypes

OCL expressions

OCL constraints are used to specify unitary test on the system under test. Two kinds of constraints have been implemented for the study.

Invariant constraints are applied to <SUT> class property and allow specifying a rule which shall be always verified to ensure the correct behavior of the class instance.

Pre-post operation constraints are applied to <SUT_TEST> class operation and allow specifying system conditions to verify before and after class operation call.

Activity diagrams

Activity diagrams are used to specify test procedures relative to a <SUT_TEST> class.

They allow describing sequential class operation call, class instances and operation parameter values.

Architecture

The model architecture is packaged to separate different actors' contribution.

The system package, designed by the software team, contains the system model classes to test tagged with <SUT> stereotype. Observable class properties, used in OCL expressions, are tagged with <VAR_READ> or <TM_READ> stereotypes.

The validation package, designed by the validation team contains classes tagged with <SUT_TEST> stereotype which inherit from System model package <SUT> classes. OCL constraints applied on validation model classes properties and operations allow specifying static and dynamic unitary test to perform on the <SUT>. Validation class operations can be tagged with <TC_SEND>

stereotype to specify telecommand attributes to invoke <SUT> operation to test.

The bench Facilities package, designed by the validation team, allows specifying test bench facilities TM/TC interfaces.

2.3 Generating code

From the UML model of the software including tests specification, the test generation tool implements three java code generators as illustrated below.
Acceleo generator is an EPL open source “model to code” generator which is user configurable threw templates.
OCL and Activity generators make used of a java Platform Specific Model developed by Astrium.
The Generated java code is target independent, the target code interface, relative to observability and commandability is hand coded in the bench interface instance specified in <BENCH_IF> stereotype.

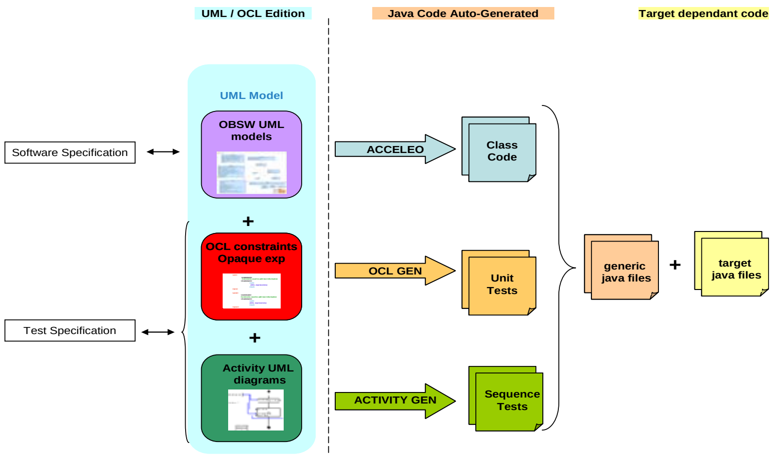


Figure 6: Test generator prototype implementation

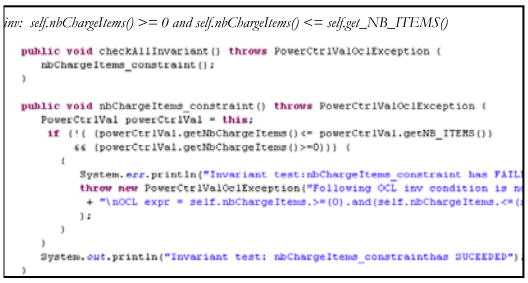
Acceleo generator

Acceleo generator chain generates the code relative to UML Class diagram. It is customised to implement body functions according stereotypes defined in the model.<VAR_READ>, <TM_READ> and <TC_SEND> stereotypes attributes allow implementing observability and commandability functions based on <BENCH_IF> interface.

OCL generator

OCL generator generates the code relative to OCL expressions defined in the model. When a class owns OCL constraints a new class is generated which contain all produced code relative to these constraints.

All invariant constraints are merged in a common function which is call in each pre-post operation constraint.

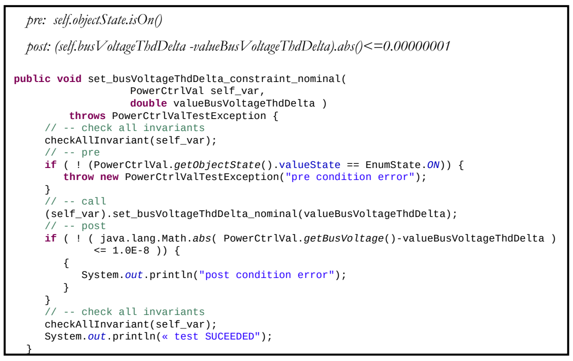


inv: self.nbChargeItems() >= 0 and self.nbChargeItems() <= self.get_NB_ITEMS()

```
public void checkAllInvariant() throws PowerCtrlValOclException {
   nbChargeItems_constraint();
}

public void nbChargeItems_constraint() throws PowerCtrlValOclException {
   PowerCtrlVal powerCtrlVal = this;
    if (!( (powerCtrlVal.getNbChargeItems()<= powerCtrlVal.getNB_ITEMS())
        && (powerCtrlVal.getNbChargeItems()>=0))) {
      {
        System.err.println("Invariant test:nbChargeItems_constraint has FAIL
        throw new PowerCtrlValOclException("Following OCL inv condition is n
         + "\nOCL expr = self.nbChargeItems.>=(0).and(self.nbChargeItems.<=(
        );
      }
   }
   System.out.println("Invariant test: nbChargeItems_constrainthas SUCEEDED");
}
```

Figure 7: OCL invariant constraint code



pre: self.objectState.isOn()

post: (self.busVoltageThdDelta -valueBusVoltageThdDelta).abs()<=0.00000001

```
public void set_busVoltageThdDelta_constraint_nominal(
                   PowerCtrlVal self_var,
                   double valueBusVoltageThdDelta )
         throws PowerCtrlValTestException {
     // -- check all invariants
     checkAllInvariant(self_var);
     // -- pre
     if ( ! (PowerCtrlVal.getObjectState().valueState == EnumState.ON)) {
        throw new PowerCtrlValTestException("pre condition error");
     }
     // -- call
     (self_var).set_busVoltageThdDelta_nominal(valueBusVoltageThdDelta);
     // -- post
     if ( ! ( java.lang.Math.abs( PowerCtrlVal.getBusVoltage()-valueBusVoltageThdDelta )
             <= 1.0E-8 )) {
        {
           System.out.println("post condition error");
        }
     }
     // -- check all invariants
     checkAllInvariant(self_var);
     System.out.println(« test SUCEEDED");
  }
```

Figure 8: OCL pre-post constraint code

Activity generator

Activity generator generates the code relative to activity diagrams defined in the model. When a class owns activity diagrams a new class is generated which contains all produce code relative to these diagrams.

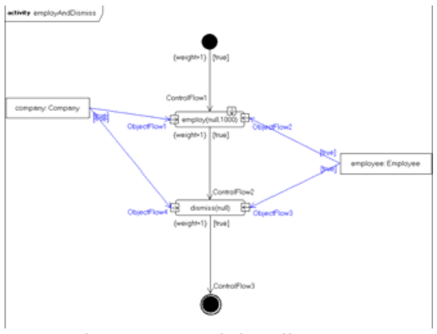
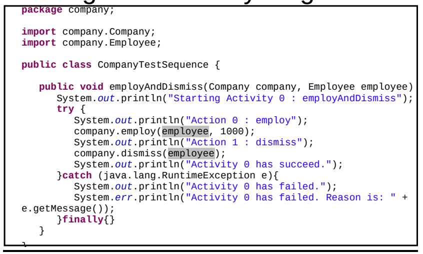


Figure 9: Activity diagram



```
package company;

import company.Company;
import company.Employee;

public class CompanyTestSequence {

   public void employAndDismiss(Company company, Employee employee)
      System.out.println("Starting Activity 0 : employAndDismiss");
      try {
         System.out.println("Action 0 : employ");
         company.employ(employee, 1000);
         System.out.println("Action 1 : dismiss");
         company.dismiss(employee);
         System.out.println("Activity 0 has succeed.");
      }catch (java.lang.RuntimeException e){
         System.out.println("Activity 0 has failed.");
         System.err.println("Activity 0 has failed. Reason is: " +
e.getMessage());
      }finally{}
   }
}
```

Figure 10: Activity diagram code

3. Use case experiment

The experimentation is built on a subset of Pleiades on-board software. It is based on two major equipments:

- Modelling and test code generation have been performed using Topcased software environment toolkit.
- Test procedures have been executed on the Simops simulator (internal product of Astrium).

3.1 Topcased modelling tool

Topcased is an eclipse platform integrated tool dedicated to the realisation of critical embedded systems which promotes model-driven engineering facilities. During the study Topcased has been improved to integrate UML Validation model design requirements. Customized Activity diagram editor and test code generator chain has been integrated in Topcased as Eclipse plugins.
Starting from Pleiades software specification and associated test specification, the UML/OCL model has been implemented in Topcased environment.

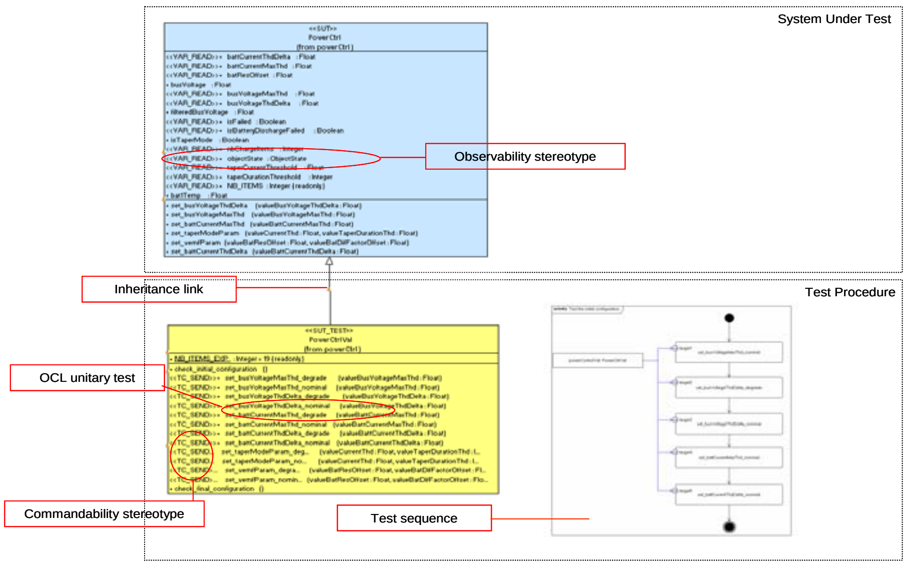


Figure 10: Use case UML model

Test procedures were automatically generated from the model. Stubbed bench interface have been produced to integrate OCL model implementation under eclipse native java execution platform.

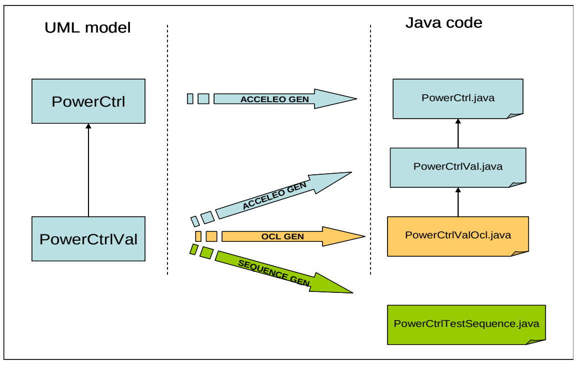


Figure 11: Test java code architecture

3.1 Simops test environment

Simops test environment is connected to a numerical simulation of the equipments and the on-board computer. The connection is made of TM/TC interface and processor emulator services such as read/write of memory symbols. It allows executing the real On-Board Software in a representative configuration.
Java test procedures, automatically generated under Topcased tool, have been executed on Simops test control environment.

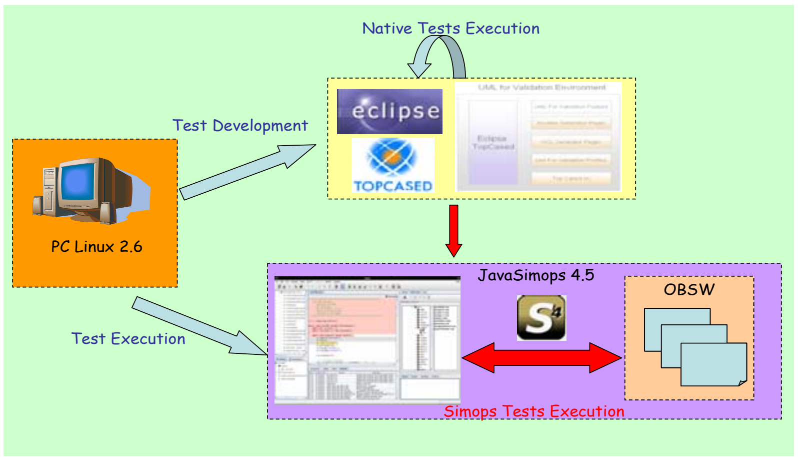


Figure 6: Test experiment configuration

4. Compliance with existing standards

If we take ECSS-E-40 standard, the main requirements on software testing and validation are coming from the “Software design and implementation engineering” and “Software validation” processes defined in the standard. They require that Unit, Integration and Validation test are developed to produce test plans, test procedures test data and reports to provide evidence that the software is working according to its requirements and design.
The technologies investigated in this study integrate quite well in the existing software development life cycle and do not change the objectives of software testing and validation. They just provide new ways of implementing them: by adding test related elements to the software models and by automating the generation of test procedures from the software models. The most challenging point, that is general to the use of model based engineering, is to organize efficient reviews of the produced UML models.
Today’s ECSS standard is focussed on documents and it is quite difficult to produce and easy to read documentation from models. Using direct access to models offers the possibility to navigate and follow the links between model entities but modelling tools do not support and formalized model review process to ensure completeness of the review. Using modelling also puts strong requirements on the people involved in the review: mastering UML and OCL language requires some specific training compared to natural language documents.

4. Conclusion

The UML for validation study demonstrated the feasibility of using automatic test generation to produce test scripts that can be executed in a real onboard software environment. It also allowed benchmarking a new methodology for future validation process, which combines software development and validation teams.
A consequent effort should be performed by actors to define and implement the test behaviour model, but the quality relative to the coherence in the test engineery process is improved as all information's are defined in the model. During maintenance phase, the evaluation impact of software modification and corresponding test regression is also improved.
The test generator prototype implementation highlight that if OCL is adapted to specify unitary test, its syntax is complex and real type is limited as it not support both floating and double types.
Model based methodology is innovative for space applications, which get very specific quality constraints. Applying such process required a robust and high quality model design tool. Topcased demonstrated enough robustness to perform test modelisation on dimensioning models, but an industrial development is mandatory to fulfil space quality standards.

7. References

[EPL]	Eclipse Public License.
[UML]	Meta Model UML 2.0 Specification, October 2003.
[OCL]	Meta Model UML 2.0 OCL Specification, October 2003.
[MDA]	Model Driven Architecture Guide, Object management group (OMG), June 2003.
[Acceleo]	http://www.acceleo.org.
[Topcased]	http://topcased.gforge.enseeiht.fr.

8. Glossary

EMF:	Eclipse Modelling Framework
OCL:	Object Constraint Language, declarative language for describing rules, which is part of the UML standard, and provides constraint and object query expressions on any model
SUT:	System Under Test