Improving Model Quality by Validating Constraints with Model Unit Tests

Lars Hamann and Martin Gogolla
Computer Science Department - Database Systems Group
University of Bremen
D-28334 Bremen, Germany
{lhamann|mg}@informatik.uni-bremen.de

***Abstract*—A central part of modern development methods is the use of tests. A well-defined test suite is usually the basis for code refactoring because changes to the system under test can be easily validated against the test suite. In model-based development tests can be derived from the model but possibilities to test the originally specified model and therefore to improve the quality of model refactorings are rare. We propose a method for defining model unit tests which allows a developer to define test suites similar to the well-known xUnit testing frameworks. This gives the developer the possibility to easily check and assess model changes against valid and invalid scenarios.**

***Keywords*-Validation, Unit Tests, Test Suite, OCL, Quality**

I. Introduction

Like in traditional development methods the reuse of software components as libraries is more and more used in model-based development. In the context of the Unified Modeling Language (UML) [1] profiles may be regarded as libraries which are delivered as a set of UML elements, e. g., special base classes, stereotypes and constraints which restrict the usage of the delivered elements. These constraints can be formulated with the Object Constraint Language (OCL) [2], as it is done in the UML specification to define the well-formedness rules. To be useful, profiles and software libraries must be of a high quality. In the context of a modeled UML profile one quality aspect is for example that the included invariants constrain the usage of the profile as expected. For example they have to fail when invalid applications of a stereotype are modeled. We are partner in a German government project called XOEV which provides a UML profile for transforming UML models into XML schema definitions and DocBook [3] documentation fragments. Details about the architecture of this model transformation process can be found in [4]. The provided profile comes with a set of well-defined naming and design rules in order to enforce the correct transformation of the specified models [5]. Because a lot of other projects rely on this profile, these naming and design rules which are mostly defined as OCL constraints need to be well-tested before a new version of the profile is released. In our view, the usage of OCL constraints as design guidelines for published profiles will increase, because more and more modeling tools support the definition and validation of OCL constraints during modeling time. Two examples are the commercial tool MagicDraw[6] and the open source tool Papyrus UML[7]. Especially when central parts of the defined constraints change, e. g., heavily used helper operations, the developer needs assistance to validate that the constraints still behave as expected.

Unit tests, as initially introduced by Beck for Smalltalk [8] (later called SUnit) and their adaption to various programming languages, e. g., JUnit for Java[9], NUnit[10] for the .NET platform and Test::Unit[11] for Ruby, are a central part of agile software development, because they provide stability assistance [12]. Moreover, test first development seems to increase productivity, too [13].

We argue that the idea of programming language unit tests used in the context of model design as model unit tests can increase the quality of models, both during the early stages of model design and during the maintenance phase [14]. When developing and maintaining a UML profile well-defined and easy to run model unit tests are essential to support the developer in order to improve the quality of the delivered product because it is unusual to translate a single profile to executable program code that could be tested with programming language unit tests. When applied to models, there are particular aspects of model unit tests which are different to programming language unit tests. One of these aspects is the presence of pre- and postconditions (further called prepos) for operations in UML models. In this paper, we highlight these aspects and discuss how one can benefit from them.

The rest of this paper is structured as follows: Section II motivates the idea of model unit tests by an example. In Sect. III we introduce our framework by first describing the general structure independent from a concrete implementation. After this abstract description, we picture our done integration into an existing design tool. Section IV shows how our implementation can be applied to the previously introduced example. Before we end with a conclusion and future work in Sect. VI we give an overview of related work and how it differentiates from our approach in Sect. V.

II. Motivating Example

To focus on the general idea of our approach, we provide a simple example motivating our idea. However, the overall approach is independent from a concrete model and the meta level where it is placed. Our example is a simple model of companies, jobs and workers as shown in Fig. 1. A `Company`

978-0-7695-4384-0/10 $26.00 © 2010 IEEE
DOI 10.1109/MoDeVVa.2010.18

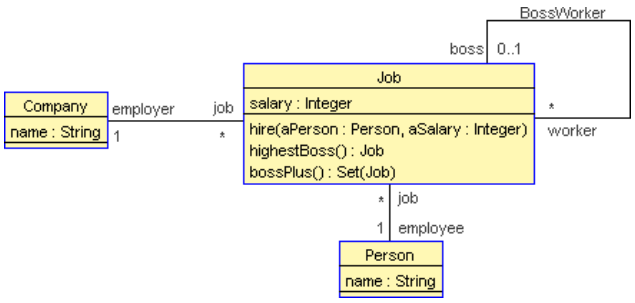

Fig. 1. Class structure of our example

has job descriptions (`Job`) which can be arranged hierarchically with the association `BossWorker`. Each job is linked to a person fulfilling the job. For several invariants, a helper method `highestBoss()` is provided which calculates the topmost Job in a hierarchy. One invariant constraints the system so that persons can hold top level positions in only one company to avoid conflicts of interest.

```
context Job inv noTwoLeadingPositions:
  let boss = self.highestBoss().employee in
    Job.allInstances()->forAll(j : Job |
      boss = j.highestBoss().employee implies
        self.employer = j.employer)
```

As one can see, the shown invariant uses the helper method `highestBoss()`. In OCL only side effect free operations can be used by OCL expressions. The body of such an operation can be defined using OCL. A first implementation of this operation might look like this:

```
highestBoss() : Job =
 if   self.boss.isUndefined() then self
 else self.boss.highestBoss() endif
```

Because this implementation would not terminate if the BossWorker hierarchy contains cycles and because of the fact that such cycles would not make sense, another invariant (`Job::bossWorkerIsHierarchy` which is explained in [15]) ensures that the links between jobs form a hierarchy. The multiplicities of the reflexive association BossWorker in combination with this invariant ensure that there is only one job without a boss in any hierarchy. During the ongoing development a developer might refactor the implementation to a simpler one, which uses the operation `bossPlus()`. This operation calculates the transitive closure of bosses for a given job. The job without a boss is returned.

```
highestBoss() : Job =
 self.bossPlus()->any(boss.isUndefined)
```

On a first look both operations are doing the same. But for one boundary case the result of both implementations differ. When called on the topmost job the first implementation returns the same job on which the operation was called, whereas the second one returns the undefined value, because `bossPlus()` returns an empty set of jobs. With this implementation the previously shown invariant will fail on system states that should be valid, because when called on a topmost job it forces each other topmost job to be in the same company.

Systematic tests would report this failure and the developer can react on it. As stated in [12], often called tests narrow down the development period in which an error was introduced because the error must have been made between the last test run and the last successful run which allows a developer to fix an error in shorter time. Therefore, test runs should be easy to execute and should be efficient. To allow this, we propose model unit tests, which can be called during modeling without much effort. Before we show a concrete solution for the previously shown problem, we describe in the next section our framework and how it is embedded into the UML meta model, in order to allow a general adaption by other tools. After that, we explain our implementation.

III. A Testing Framework for Models

The well-known unit test frameworks use assert methods to check a constructed system state against a given value. While this is also useful in model unit tests, several other aspects of model unit tests are unique in comparison to programming language testing frameworks. First, a model can have additional constraints specified by some constraint language, e. g., OCL. Second, operations can be extended with pre- and postconditions to specify additional requirements and to support contract-based design. Our proposed framework takes these additional constraints into account. Therefore, one is not only interested in building positive test cases which fulfill the given constraints but also in building counter examples which show the usefulness and the correct behavior of the defined constraints. To support these counter examples, our framework does not force a test to only build up valid system states. Indeed, a test can be useful if one can show that a given constraint is violated under a given system state. Furthermore, a user only has to build small system states with instances required for the specific test. To allow our framework to be used in various scenarios, we don't make any assumptions about the runtime environment and the construction of system states. This can be done for example by some kind of execution language for UML, see for example [16], or by manually designed model instances.

A. Abstract Syntax

The abstract syntax of our framework and its relations to the UML and OCL meta model is shown in Fig. 2. Classes used from the meta models are shown with a white background to distinguish them from the framework classes. Note that all classes from the meta models are used by delegation and not by inheritance. This ensures that the used languages are not modified. In contrast if we inherited for example `Assert` from `Expression` an assert would be legal at any place where an expression is expected. This would leave to unwanted changes to the used meta models. The framework provides the meta class `TestSuite` as a container for different `TestCases`. To allow a common system state for all test cases in one test suite as a setup, `TestSuite` is linked to

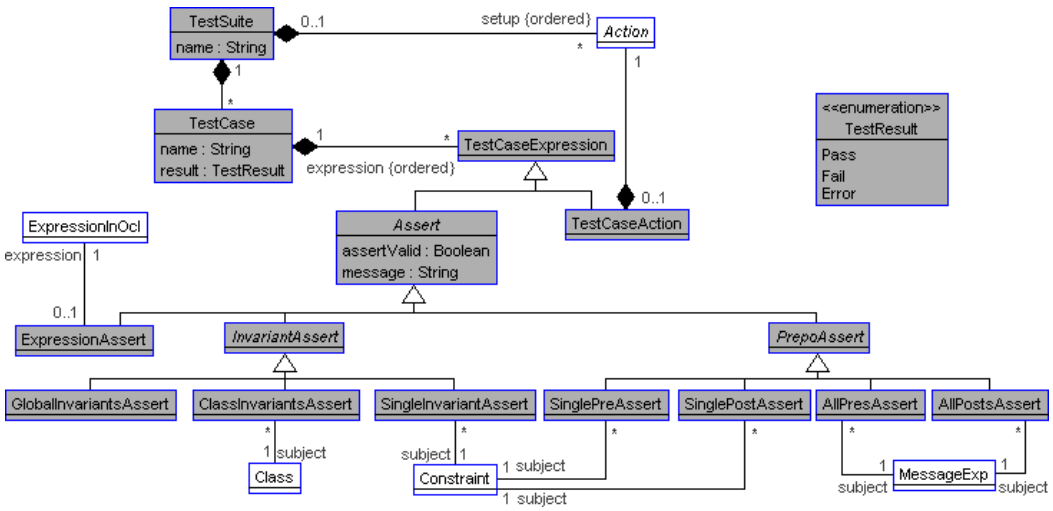

Fig. 2. Abstract Syntax of the Testing Framework

the UML meta class `Action`. These actions are evaluated before each single text case. A central part of unit tests are assertions done inside a test case. While writing a test, the user compares calculated results against expected values. We provide the abstract class `Assert` for such assertions. To allow state manipulation during test cases a test case consists of `TestCaseExpressions` which are either assertions or `TestCaseActions`. `TestCaseAction` delegates its evaluation to the concrete action linked to it. The result of a test case is defined by the enumeration `TestResult` which consists of the three values Pass, Fail and Error. If all assertions during a test case run are fulfilled and no unexpected error is encountered, the test result is Pass. If an assertion fails, the result is Fail which signals the user that the model is not working as expected. The result Error is returned if an unexpected runtime error occurs, e. g., by trying to set an attribute of a non existing object.

The aforementioned abstract class `Assert` as a basis for test assertions is specialized for concrete model assertions. We defined eight different types of assertions for our model unit test framework to cover the special features of UML- and OCL-based models, which are not present in traditional xUnit frameworks. Seven of them are divided into two different groups, which are denoted by the abstract classes `InvariantAssert` and `PrepoAssert`. The remaining assertion type is `ExpressionInOcl` which is the most general statement and nearest to the known unit test asserts. The subtypes of `InvariantAssert` relate to asserts about class invariants, whereas the subtypes of `PrepoAssert` are used for asserts about operation pre- and postconditions. The base class `Assert` defines the attributes *assertValid* and *message*. The attribute *assertValid* determines whether the concrete assertion is a positive assertion when the value of *assertValid* is true or a negative one in the other case. It enables the definition of negative tests without changing (negating) constraints, e. g., to test if a given invariant behaves as expected and fails on constructed invalid system states. The attribute *message* can store additional information, that can be used to display detailed information about a failed assertion. This information could be provided as a fixed string literal or could be calculated by an OCL expression during the test execution to provide contextual messages. The meaning and requirements of each of the eight concrete assertion types are as follows[1]:

1) `ExpressionAssert`
   An expression assert evaluates a boolean OCL expression wrt. the current system state and compares the result to the expectation indicated by the inherited attribute *assertValid*. That is, if *assertValid* is true the expression has also to return true to be a successful assertion. If *assertValid* is false the return value of the given OCL expression has to be false, otherwise the assertion fails. Note that the boolean OCL expression may use the OCL keyword `@pre` to access, for example, the value of an object attribute before an operation has been called; the keyword `@pre` is allowed only in postconditions and the respective assert must directly follow the closing of an operation call.
2) `GlobalInvariantsAssert`
   This assertion evaluates all defined invariants in the current system state. The influence of the inherited attribute *assertValid* is likely the same as before. To be successful when *assertValid* is true, all invariants must hold, whereas when *assertValid* is false, at least one invariant has to fail. Since the value of *assertValid* can always change the meaning in the same way for single expressions, e. g., a single invariant, and expressions on a set of constraints, e. g., all invariants, we skip this description for the next elements (3.–8.).

[1]Due to space limitations we do not present a formal description of the well-formedness-rules of our unit test language.

3) `ClassInvariantsAssert`
   A ClassInvariantAssert evaluates all invariants defined for a given class. Therefore it is linked with the UML meta class `Class`. To make it easy for a developer to find the source of a test failure, only the invariants defined for this class are taken into account but not inherited invariants from superclasses.
4) `SingleInvariantAssert`
   This assertion tests a single invariant. Therefore it is linked to the UML meta class `Constraint`. For the abstract syntax it is enough to link this assertion class to the corresponding constraint because the owning class can be retrieved by the UML meta model.
5) `AllPresAssert`
   This assertion and the next three ones are required for testing the dynamic behavior of a model. `AllPresAssert` validates all preconditions for a given operation call. This is indicated by the association between `AllPresAssert` and `MessageExp` from the OCL meta model. `MessageExp` provides the information needed for an operation call or a signal to send, e. g., the target of an operation call and its parameters. In other words this assertion can validate whether a given operation call is currently valid or invalid.
6) `SinglePreAssert`
   The `SinglePreAssert` statement evaluates a single precondition. As with `AllPreAssert` it requires all information of a message expression. In contrast to `AllPresAssert`, it cannot be used to determine whether an operation call is currently valid but evaluates the result of a single precondition to allow finer tests.
7) `AllPostsAssert`
   In UML, behavioral features can be constraint by postconditions. To test them, `AllPostsAssert` can be employed. It can be used after an operation call and evaluates all postconditions of the linked subject. The usage is restricted so that the assert has to follow immediately after the handling of the received message, otherwise another operation could have changed the system state in the meantime.
8) `SinglePostAssert`
   As before, this assertion is the detailed version of `AllPostAssert` and allows finer grained test cases.

As already mentioned, one advantage of UML in combination with OCL is the possibility to define pre- and postconditions to allow contract-based development. We take this into account not only for assertions about the states of pre- and postconditions, but also by allowing test cases to access the system state that was present before the last operation started. Among other things, this enables a test writer to validate, that a given operation does not change any unrelated attribute. While such assurances could also be done in postconditions this kind of test should not be contained inside the model to keep the model specification small and precise. In OCL it is allowed to access the system state that was present before an operation in a postcondition via the keyword `@pre`. We extend this usage to any expression inside a test case. When the keyword `@pre` is used in a context without a previously executed operation call, an error is raised.

B. Concrete Syntax

To validate our approach, we integrated a first version of the described testing framework in our UML and OCL tool USE [17]. USE employs an own command language to create and modify system states and uses an easy to understand textual syntax for defining models. In order to follow the USE philosophy, we implemented the abstract syntax as a simple textual language. Each test suite is defined in a single file, which can contain a setup sequence of commands to build up a system state common to all tests defined in a test suite. Test cases can be defined after the setup part. Test cases can be a mixed sequence of USE commands and assert statements proposed before. The following shows the concrete syntax of all assert statements in the same order as in the previous section with a sample call and its description.

1) `assert (valid|invalid) <expression>`
   Example: `assert valid Person.allInstances ->forAll(p:Person|p.name = p.name@pre)`
   The example verifies that the name of each person was not changed by the last operation call. Note, that the access to all instances of a class is a built-in feature of OCL.
2) `assert (valid|invalid) invs`
   Example: `assert valid invs`
   This statement asserts, that all invariants defined in the model are valid.
3) `assert (valid|invalid) invs <classname>`
   Example: `assert valid invs Company`
   This statement checks all invariants of the class `Company` for validity.
4) `assert (valid|invalid) inv <classname> ::<invariantname>`
   Example: `assert invalid inv Job::bossWorkerIsHierarchy`
   The given example asserts that the invariant `bossWorkerIsHierarchy` constraining instances of the class `Job` is invalid. Thus, the current system state contains a cycle in at least one boss worker relation.
5) `assert (valid|invalid) pre <objexp>. <opname>(<params>)`
   Example:
   `assert invalid pre aCompany.hire(ada)`
   The example is successfully executed, if a call to the method `hire` with the parameter `bob` on the instance `ada` is invalid in the current system state, thus at least one of the preconditions of `hire()` fail.
6) `assert (valid|invalid) pre <objexp>. <opname>(<params>)::<prename>`
   Example: `assert invalid pre aCompany. hire(ada)::aPersonIsDefined`

Asserts that the single precondition `aPersonIsDefined` fails.

7) `assert (valid|invalid) post`
   Example: `assert valid post`
   Asserts that all postconditions of the last called operation hold.
8) `assert (valid|invalid) post <postname>`
   Example: `assert valid post jobDefined`
   Verifies that the single postcondition `jobDefined` for the last operation call returns true.

One might be wondering why the `assert post` statements do not require additional information about the operation call. As stated in the last subsection, a test on postconditions only makes sense immediately after the execution of an operation. So it is easy to keep track of the last called operation which must be the target of the assertion. In our implementation we raise an error, if an expression inside an `ExpressionAssert` accesses the pre-state before any operation was called. This supports a test writer in defining good tests. Also we do not allow a method call, if a precondition of the operation fails. Such a method call would be of no use, because the operation implementation could rely on the preconditions to hold. We might change this behavior in the future or make it configurable to validate the robustness of a specified method with invalid input.

IV. Application to the Example

Next, we show how our model testing approach is applied in USE to the motivating example given in Sect. II. The test suite shown below defines three test cases and a base system state setup. The setup process creates two companies with related jobs and linked employees. The first shown test case *testHighestBoss* simply validates the result of the operation `highestBoss()` by comparing the result values of operation calls to expected values. While the first test case is similar to JUnit tests, the second test case *testOnlyOneHighestPosition* uses features which are unique to our framework. First, it is checked that the invariant `Job::noTwoLeadingPositions` holds in the base state created by the setup process. After that assert, the system state is manipulated so that one person fulfills two topmost jobs. After the system state has been built, an assert statement checks for the invalidity of the invariant `Job::noTwoLeadingPositions`. The last test case `testHire` shows the usage of dynamic aspects. It validates, that a call to the operation `hire()` is not valid, because the job appleDesigner is taken by another person and the provided salary would be greater than the salary of the higher position appleCeo. The test uses the detailed version of the precondition asserts because when one assertion fails, the developer can directly see which precondition fails. After testing that the preconditions behave as expected, a valid call to `hire()` is constructed and an execution is simulated between the commands `!openter` and `!opexit`. After the operation call the following statement asserts that the operation has not changed the salary of the higher ordered job. This is done by the usage of the keyword `@pre` following the attribute *salary* of the object appleCeo. It retrieves the value of salary before the last operation call. The value is then compared to the actual value of salary.

```
testsuite PerCom for model percom.use
setup
 -- setup job hierarchies
 !create apple : Company
 !create appleCeo, appleDesigner : Job
 !insert (appleCeo, appleDesigner) into BossWorker
 !insert (apple, appleCeo) into CompanyJob
 !insert (apple, appleDesigner) into CompanyJob
 !create ibm : Company
 !create ibmCeo:Job
 !insert (ibm, ibmCeo) into CompanyJob
 !create ada, bob, cyd : Person
 --   link persons to jobs
 !insert (ada, ibmCeo) into PersonJob
 !insert (bob, appleCeo) into PersonJob
 !insert (cyd, appleDesigner) into PersonJob
end
testcase testHighestBoss
 assert valid appleCeo.highestBoss()=appleCeo
 assert valid appleDesigner.highestBoss()=appleCeo
end
testcase testOnlyOneHighestPosition
 assert valid inv Job::noTwoLeadingPositions
 !create ibmCio : Job
 !insert (ibm, ibmCio) into CompanyJob
 !insert (bob, ibmCio) into PersonJob
 --Bob already boss at Apple
 assert invalid inv Job::noTwoLeadingPositions
end
testcase testHire
  !create dan : Person
  !set appleCeo.salary := 3000
  assert invalid pre appleDesigner
    hire(dan, 3500)::bossBetterPaidThanWorker
  assert invalid pre appleDesigner
    hire(dan, 3500)::jobIsAvailable
  !delete (cyd, appleDesigner) from PersonJob
  !openter appleDesigner hire(dan, 2500)
    !insert (dan, appleDesigner) into PersonJob
    !set self.salary := theSalary
  !opexit
  assert valid post
  assert valid appleCeo.salary=appleCeo.salary@pre
end
```

When executed with the first implementation of the operation `highestBoss()`, which uses recursion, the tests `testHighestBoss` and `testOnlyOneHighestPosition` run successful. With the modified implementation, which uses the transitive closure, both tests fail. When executed in USE, the test suite shown afore results in the following output.

```
Test suite 'PerCom' with 3 test cases
Executing test 1/3 'testHighestBoss'... failure
Line 23: Assertion
  'appleCeo.highestBoss()=appleCeo' failed.
Executing test 2/3 'testOnlyOneHighestPosition'... failure
Line 28: Assertion
  'valid inv Job::noTwoLeadingPositions' failed.
Executing test 3/3 'testHire'... success
### 2 FAILURES ###
```

First, general information about the test suite is provided, e. g., the name and the amount of included test cases. Each single execution and the result of a test case is shown in one line. When a test case fails detailed information about the failed assertion is shown, as can be seen for the first and second test case. The last line shows the overall result of the test suite in order to prevent the user to browse through all single test case results. The developer can use the provided information to react on the failures for example by reverting the changes

or modifying the failing constraints. The test itself could also be modified. But one has to pay attention to this modification.

V. Related Work

The UML Testing Profile [18] published by the Object Management Group (OMG) defines a complex profile for designing and specifying test systems as a whole. The runtime behavior of the system under test can be specified by UML sequence, activity and state automate diagrams. In contrast to our work, the test focus lies on the resulting systems rather then on the designed models. The work in [19], [20] can be viewed as domain specific testing with focus on specific application domains, e. g., security testing. [21], [22] use models with constraints as a basis for test generation. Test generation can be supported by our approach, because it can increase the quality of the defined constraints which results in tests of higher quality. Test generation is not adequate to test meta models or profiles. An approach to test meta models is presented in [23] which is somehow similar to our work. It tests meta models against positive and negative meta model instances, dynamic aspects of models are not covered. In [24] a testing language for conceptual schemas is presented which is similar to our framework, but it does not cover the UML/OCL features like accessing previous states or asserts on single constraints, yet. UMLAnT [25] is an Eclipse based testing framework for testing UML designs and implementations. It utilizes USE to validate pre- and postconditions during animation of the designed model. While UMLAnt, like USE, allows to check the whole system state for validity, finer grained automated checks, e. g., the failure of a single invariant are not supported.

VI. Conclusion

In this paper, we have presented an approach for combining agile development techniques with precise, formal UML and OCL requirements for system invariants and operation pre- and postconditions. We have proposed a language for the formulation of test suites of models which takes up ideas from the xUnit testing frameworks and extends xUnit assertions with OCL specifics for global invariants, pre- and postconditions of operations and access to all objects of a particular class and to previous values after operation calls.

Future work will include to build a graphical user interface in order to provide easy access to the method. Further, test generation from modeled snapshots to reduce the manual scripting task will be considered. Apart from the assertions which we have introduced already, it might be useful to offer checks for the validity of all invariants on a single object or a group of objects. Queries asking for all operations, whose preconditions are satisfied, are in our scope as well. A further point of work concerns the refactoring of tests after the model has been refactored in a way such that the tests are not applicable any more, for example, after multiplicity changes. More examples and involved case studies as well as feedback from developers working with OCL or USE will check and verify the usefulness of the proposed features.

References

[1] *UML Superstructure 2.3*. Object Management Group (OMG), Feb. 2010. [Online]. Available: http://www.omg.org/spec/UML/2.3/

[2] *Object Constraint Language 2.2*. Object Management Group (OMG), Feb. 2010. [Online]. Available: http://www.omg.org/spec/OCL/2.2/

[3] OASIS, "DocBook." [Online]. Available: http://www.docbook.org

[4] F. Büttner et. al., "MDA Employed in a Joint eGovernment Strategy: An Experience Report," in *Proc. 3rd ECMDA Workshop "From Code Centric To Model Centric Software Engineering"*, T. Bailey, Ed. European Software Institute, 2008. [Online]. Available: http://www.esi.es/modelplex/c2m/program.php

[5] F. Büttner, N. Cordes, C. Crome, S. Drees, A. Franke, L. Hamann, J. Heins, C. Karich, Krolczyk, M. Kuhlmann, K. Lahmann, C. Lange, D. Lopes, Y. Rabenstein, C. Senf, F. Steimke, A. Stosiek, H. Weber, and W. Zimmer, *Handbuch zur Entwicklung XÖV-konformer IT-Standards*. OSCI-Leitstelle Bremen, Die Beauftragte der Bundesregierung für Informationstechnik, Mar. 2010. [Online]. Available: http://www1.osci.de/sixcms/media.php/13/spezifikation.4921.pdf

[6] No Magic, Inc., "MagicDraw." [Online]. Available: http://www.magicdraw.com

[7] Papyrus UML. [Online]. Available: http://www.papyrusuml.org

[8] K. Beck, "Simple Smalltalk Testing: With Patterns," *The Smalltalk Report*, vol. 4, no. 2, pp. 16–18, 1994.

[9] JUnit. [Online]. Available: http://www.junit.org

[10] NUnit. [Online]. Available: http://www.nunit.org

[11] Test::Unit, "Ruby Unit Testing." [Online]. Available: http://rubyforge.org/projects/test-unit

[12] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[13] H. Erdogmus, M. Morisio, and M. Torchiano, "On the Effectiveness of the Test-First Approach to Programming," *IEEE Transactions on Software Engineering*, vol. 31, pp. 226–237, 2005.

[14] S. W. Ambler, *The Object Primer: Agile Model-Driven Development with UML 2.0*, 3rd ed. Cambridge, UK: Cambridge University Press, 2004.

[15] M. Gogolla, J. Bohling, and M. Richters, "Validating UML and OCL Models in USE by Automatic Snapshot Generation," *SoSyM*, 2005.

[16] F. Büttner and H. Bauerdick, "Realizing UML Model Transformations with USE," in *UML/MoDELS Workshop on OCL (OCLApps'2006)*, pp. 96–110.

[17] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-Based Specification Environment for Validating UML and OCL," *Science of Computer Programming*, vol. 69, pp. 27–34, 2007.

[18] *UML Testing Profile 1.0*. Object Management Group (OMG), Jul. 2005. [Online]. Available: http://www.omg.org/technology/documents/formal/test_profile.htm

[19] O. Pilskalns and A. A. Andrews, "Using UML Designs to Generate OCL for Security Testing," in *SEKE*, 2006, pp. 505–510.

[20] S. Noikajana and T. Suwannasart, "An Improved Test Case Generation Method for Web Service Testing from WSDL-S and OCL with Pair-Wise Testing Technique," in *COMPSAC '09. 33rd Annual IEEE International*, vol. 1, Jul. 2009, pp. 115 –123.

[21] S. Weißleder and B.-H. Schlingloff, "Quality of Automatically Generated Test Cases based on OCL Expressions," in *ICST*. IEEE Computer Society, 2008, pp. 517–520.

[22] G. Engels, B. Güldali, and M. Lohmann, "Towards Model-Driven Unit Testing," in *MoDELS Workshops*, ser. LNCS, T. Kühne, Ed., vol. 4364. Springer, 2006, pp. 182–192.

[23] D. A. Sadilek and S. Weißleder, "Testing Metamodels," in *ECMDA-FA*, ser. LNCS, I. Schieferdecker and A. Hartman, Eds., vol. 5095. Springer, 2008, pp. 294–309.

[24] A. Tort and A. Olivé, "First Steps Towards Conceptual Schema Testing," in *Proceedings of the Forum at the 21st Conference on Advanced Information Systems Engineering (CAiSE 2009).*, vol. 453. CEUR, 2009, pp. 1–6.

[25] T. T. Dinh-Trong, S. Ghosh, R. B. France, M. Hamilton, and B. Wilkins, "UMLAnT: an Eclipse plugin for animating and testing UML designs," in *ETX*. ACM, 2005, pp. 120–124.