Model-Driven Testing with UML 2.0

Zhen Ru Dai

Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany
`dai@fokus.fraunhofer.de`

Abstract. The UML 2.0 Testing Profile provides support for UML 2.0 based model-driven testing. This paper introduces a methodology of how to use the profile in order to transform an existing UML system design model for tests. For the formalization of the proposed methodology, the QVT transformation rules defined by CBOP/IBM/DSTC are considered.

1 Introduction

The Model-Driven Architecture (MDA) is not only about system modelling throughout the abstraction levels in terms of platform independent system modelling, platform specific system modelling and system code generation [1, 2]. The MDA abstraction levels can also be applied to test modelling [3].

Due to increasing complexity of today's software systems, the early integration of testing into the development process becomes more and more important. By doing so, design mistakes and implementation faults can be detected in an early stage of the design process. This allows reduction of time and costs. Additionally, the developed tests can be executed against the developed system after it has been released to the customer in order to check its correct behavior in the customer's target environment.

The Unified Modeling Language (UML) is a visual language to support the design and development of complex object-oriented systems. With the growing system complexity the need for solid testing increases. But UML itself, even the newest version 2.0 [4, 5], provides no means to describe a test model. Thus, a UML 2.0 profile for the testing, called the *UML 2.0 Testing Profile* (U2TP) [6], has been defined which has become an official OMG standard since March 2004. U2TP bridges the gap between designers and testers by providing a means for using UML for both system modeling and test specification. This allows a reuse of UML design documents for testing and enables test development in an early system development phase.

According to the philosophy of MDA, the same modelling mechanism can be re-used for multiple targets [7]. Strict distinguishment should be made between platform independent and platform specific system models before generating executable system codes. Within these three abstraction levels, transformation techniques are applied. Similarly, test models can be specified platform independently and platform specific before generating executable test codes. Researches have been made on transformation between the different system or test development abstraction levels (vertical arrows in Figure 2) [8–10]. But only few research has been done for the transformation between system models and test models (horizontal arrows in Figure 2).

In this paper, we introduce a methodology of how to apply U2TP concepts to an existing UML system design model effectively in order to retrieve a test design model. The methodology is concretized by transformation rules which are formalized in Query/View/Transformation (QVT) rules defined by CBOP/IBM/DSTC [11].

The paper is structured as follows: After a short introduction about model-driven testing and UML 2.0 Testing Profile in Sections 2 and 3, the methodology is provided in Section 4, where different test aspects of UML 2.0 Testing Profile are discussed. In Section 5, a transformation example is outlined. Section 6 summarizes and concludes this paper.

2 Approaches to Model-Driven Testing

The philosophy of MDA can be applied both on system modelling and test modelling. As shown in Figure 1, platform independent system design models (PIM) can be transformed into platform specific system design models (PSM). While PIMs focus on describing the pure functioning of a system independently from potential platforms that may be used to realize and execute the system, the relating PSMs contain a lot of information on the underlying platform. In another transformation step, system code may be derived from the PSM. Certainly, the completeness of the code depends on the completeness of the system design model.

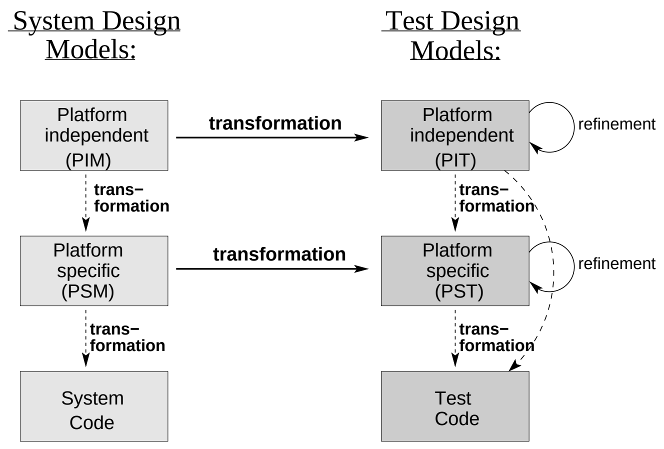

Fig. 1. System Design Models vs. Test Design Models

The same abstraction in terms of platform independent, platform specific modelling and system code generation can be applied to test design models.

Furthermore, test design models might be transformed from system design models directly. This enables the early integration of test development into the overall development process. Once the system design model is defined at PIM level, a platform independent test design model (PIT) can be derived. This model can be transformed either directly to test code or to a platform specific test design model (PST) [12]. The same transformation technology can be used for deriving PSTs from the PSM. After each transformation step, the test design model can be refined and enriched with test specific properties. Although the transformed test design model may already contain static and dynamic aspects, the behavior has to be completed in order to cover unexpected system behavior as well. Also, test issues such as e.g. test control and deployment information has to be manually added to the test design model. At last, the test design model can be finally transformed into executable test code from either PST or PIT.

3 The *UML 2.0 Testing Profile* (U2TP)

The UML 2.0 Testing Profile provides concepts to develop test specifications and test models for black-box testing [13]. The profile introduces four logical concept groups covering the aspects [6]: *test architecture, test behavior, test data* and *time*. Together, these concepts define a modeling language for visualizing, specifying, analyzing, constructing and documenting a test system. In the following, the U2TP concepts are introduced (Figure 2).

Test Architecture Concepts One or more objects can be identified as the *System Under Test* (SUT). *Test components* are objects within a test system which can communicate with the SUT or other components to realize the test behavior. The *test context* allows users to group test cases, to describe a corresponding *test configuration*, i.e. the connection between test components and the SUT, and to define the *test control*, i.e. the required execution order of the test cases. Arbitration is a means for evaluating an overall verdict for a test context. A tester can either use the default arbitration or define their own arbitration scheme using an *arbiter*. The *scheduler* controls the test execution and test components. It is responsible for the creation of test components, a synchronized start of the different test components, and the detection of test case termination.

Test Behavior Concepts A *test objective* defines the aim of a test. Herefore, UML Interaction Diagrams, such as State Machines and Activity Diagrams can be used to define test stimuli, observations, test control/invocations, coordination and actions. The normative test behavior is specified in a *test case*, which is an operation of the test context specifying how a set of co-operating components interact with the SUT to realize a test objective. When normative test behavior is defined, focus is given to the definition of unexpected behaviors which is achieved through specification of *defaults*. A *validation action* is performed by a local test component to inform the arbiter about its local test verdict. A test *verdict* shows the result of the executed test. Possible test verdicts are `pass`, `inconclusive`, `fail`, and `error`.

Test Data Concepts In the UML 2.0 Testing Profile, *wildcards* are used to handle unexpected events, or events containing many different values. The profile introduces wildcards allowing the specification of: (1) Any value and (2) Any or omitted values. *Data pools* are associated with test context and include concrete test data. *Data selectors* are operations to retrieve test data from the data pool or data partitions. The notion of *coding rules* allows the tester to define the encoding and decoding of test data when communicating with the SUT.

Time Concepts The time concept group defines concepts to constrain and control test behavior with regard to time. *Timers* are needed to manipulate and control test behavior as well as to ensure the termination of test cases. *Time zones* are used to group components within a distributed system, allowing the comparison of time events within the same time zone.

Test Architecture Concepts	Test Behavior Concepts	Test Data Concepts	Time Concepts
▧ SUT	▓ Test objective	▓ Wildcards	▦ Timer
▦ Test component	▧ Test case	▦ Data pool	▓ Time zone
░ Test context	▦ Defaults	▦ Data partition	
▦ Test configuration	▓ Validation action	▓ Data selector	
▦ Test control	░ Verdicts	▓ Coding rules	
Arbiter			
Scheduler			

░ Mandatories	▧ Derivable Mandatories	▓ Optionals	▦ Derivable Optionals

Fig. 2. U2TP Concepts & A Methodology on Test Design Model Development

4 A Methodology on Model-Driven Test Development

In this section, we introduce a methodology for using the UML 2.0 Testing Profile effectively after having received a detailed system design model which is to be tested [14]. In the following, we determine *system design model* to be the UML 2.0 system model in UML and the *test design model* to be the UML 2.0 model using U2TP concepts.

Having a system design model, a tester may have to specify tests for the system. This can be done by extending the system design model with U2TP concepts. The following aspects must be considered when transforming a system design model into a test design model:

First of all, define a new UML package as the test package of the system. Import the classes and interfaces from the system design package in order to get access to messages and data types in the test specification. Next, start with the specification of the *test architecture* and continue with *test behavior* specifications. Test data and time are mostly already comprised in either the test architecture (e.g. timezone or data pool) or test behavior (e.g. timer or data partitioning) specifications.

Below, issues regarding test architecture and test behavior specifications are listed. They are subdivided into two categories: mandatory issues and optional issues (Figure 2 on page 4). *Mandatories* are issues which are essential for a test design model with U2TP. The most important mandatory issues are e.g. SUT and test components. *Optional* issues are specific to test requirements and are therefore not always needed for the test design model specification. Optional issues are e.g. test control and timers. Additionally, there are both mandatory and optional concepts which can be derived directly from existing system design diagrams[1].

In the following, the mandatories and optionals are listed and possible derivations outlined. A test design model based on U2TP may use all UML diagram types for test specification. Depending on the given system design diagram types, different test design diagram types can be transfered. Therefore, in the methodology, we also point to the diagram type feasable for the derivations. These derivations are used for the test design model transformation in Section 5:

1. Test architecture:
   i. Mandatory:
      - Assign the classes (in a Class Diagram) or objects (in an Object Diagram) you would like to test to *SUT* class/object.
      - Specify a *test context* class listing the test attributes and test cases, also possible test control and test configuration.

   ii. Optional:
      - Depend on their functionalities, test components have to be defined. Group the classes/objects (except the SUT) to *test component* classes/objects. Test components are not needed in unit tests.
      - In order to define the ordering of test case execution, specify the *test control*. If there are Activity Diagrams given in the system design model, each activity illustrates one test case and the activity flow describes the test flow in the test control specification. If there are Use Case Diagrams provided, each use case depicts one test case which should be stringed together for the test control specification. If neither Activity Diagram nor Use Case Diagram exist in the system design model, string the test cases together for the test control specification. In a more complex test control specification, loops and conditions should also be used.
      - *Test configuration* are easily retrieved by means of existing Interaction Diagrams. Whenever two components exchange messages with

[1] A detailed case study on the methodology can be found in [14].

each other, assign a communication channel between the components. If there is no Interaction Diagram provided, connect the test components and SUT to an appropriate *test configuration* so that the configuration is relevant for all test cases included in the test suite.
   - Assign *timezones* to the components if the test system is a distributed system.
   - Provide *coding rule* information.

2. Test behavior:
   i. Mandatory:
      - For the specification of *test cases*, take given Interaction Diagrams from the system design model. Change (i.e. rename or group) the instances and assign them with stereotypes according to their roles (i.e. test component or SUT). If there are Use Case Diagrams or Activity Diagrams provided in the system design model, the use cases and activities are specified in additional Interaction Diagrams. Thus, for each use case or activity, a test case should be specified.
      - Assign *verdicts* at the end of each test case specification. Usually, the verdict in a test case is set to pass.

   ii. Optional:
      - Define *test objectives* for each test case that is to be specified.
      - System behavior which are not used for the tests should be taken for *default* specifications. Herefore, Interaction Diagrams like Sequence Diagrams, State Machines or Activity Diagrams should be used. Use *wildcards* to catch unexpected behavior. Verdict settings in a default are either fail or inconclusive.
      - *Timers* should be derived from time constraint specifications within a Sequence Diagram or State Machine.

UML 2.0 Testing Profile provides default arbitration and scheduling mechanisms which by default should be implemented by the tool vendor. Additionally, the profile also provides the tester the means to specify his own arbiter and scheduler. To do so, the tester needs additional diagrams in order to describe the behavior of the arbiter and the scheduler. Furthermore, the tester should also consider the modification in the whole test architecture.

5 Test Design Model Transformation

Figure 3 shows the meta-model based transformation for the test design model transformation. Herein, the source meta-model is the UML meta-model and the target meta-model is the U2TP meta-model. In the methodology (Section 4), classes and objects are grouped together in order to define test components or SUT. Such mechanisms cannot be performed by transformations. Thus, for our transformation approach, we have to define those mechanisms in order to provide the tester a means to group or delete elements[2], reference test behavior fragments etc. These mechanisms are called test directives and its meta-model is the *Test Directive Meta-Model.* Transformation rules are applied on both the UML meta-model and the Test Directive Meta-Model to create an instance of the U2TP Meta-Model. All three meta-models are based on MOF.

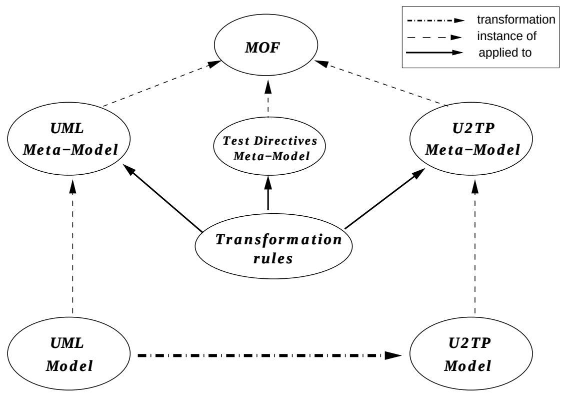

Fig. 3. Meta-Model Based Transformation

The transformation of the UML model to U2TP model is specified by a set of rules defined in the transformation meta-models [15] according to the QVC specification from CBOP/IBM/DSTC[11]. The introduced transformation language is aspect-oriented, declarative and pattern-based. It shows concepts for specification of rules, patterns and tracking relationships. *Transformation rules* are used to describe a correspondence between patterns of elements in the source model(s) and the elements to be created in a target model. *Patterns* are reusable definitions. When used in the source of a rule, a pattern is a query. When it is used in the target, it acts as a template for model elements. *Tracking relationships* associate the source model elements with the target model elements.

In the following, we will show a small example of our test design model transformation: Let us assume that we have an existing Object Diagram from the system design model and want to perform system test on this model. For the transformation for test components, the methodology says (in the test architecture optionals in Section 4): *Depend on their functionalities, test components have to be defined. Group the objects (except the SUT) to* test component *objects.* Thus, besides the Object Diagram, we also need a grouping mechanism, which should be provided by the Test Directives Meta-Model. A grouping mechanism is applied to at least two objects in the diagram.

Figure 4 shows how the transformation can be performed on instance level. On the left upper corner, a UML package with three objects is shown. In the left lower corner, the relationship between the objects which should be grouped

[2] Elements are UML elements such as classes, objects, instances etc.

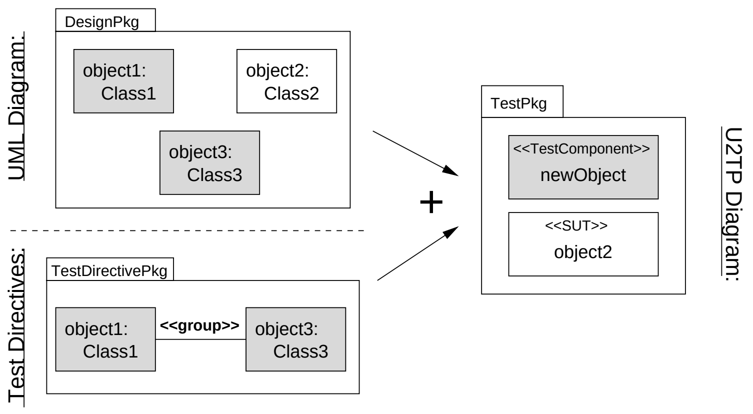

Fig. 4. Test Component Transformation

to a test component is specified in a test directives model. The grouping notation is an association between the objects with the stereotype <<group>>. In this example, only object1 and object3 should be grouped into one test component. Therefore, after the transformation in this example, the output test model consists of one test component and one SUT instance. Of course, two test components could also be specified, depending on the choice of the transformation rules. The stereotypes <<TestComponent>> and <<SUT>> are U2TP notations. By performing appropriate transformation rules on the different system design diagrams, test architecture and behavior can be specified for the test design model.

6 Summary and Outlook

In this paper, we have presented a methodology of how to derive a U2TP test design model from an existing UML system design model. Furthermore, the methodology can be formalized by defining transformation rules from system design diagrams to test design diagrams. For the transformation, we chose the QVT specification from CBOP/IBM/DSTC.

The definition of the transformation rules is not fully completed. Thus, we shall complete this work first. Unfortunately, due to lacking tool support for UML 2.0 and U2TP at time, we are not able to proof our model transformation rules. Thus, in our future work, we plan to investigate in tools which support the U2TP concepts and automated derivation of test design models from system design models.

Acknowledgement

Many thanks to Mr. Keith Duddy from DSTC for the various intensive discussion sessions.

References

1. OMG: (Model-Driven Architecture (MDA)) http://www.omg.org/mda/.
2. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture–Practice and Promise. Addison-Wesley Pub Co (2003)
3. Gross, H.: Testing and the uml – a perfect fit. Technical report, Fraunhofer IESE Report 110.03E (2003)
4. http://www.omg.org/uml.
5. Born, M., Holz, E., Kath, O.: Softwareentwicklung mit UML 2. Addison-Wesley (2004)
6. U2TP Consortium: UML 2.0 Testing Profile. (2004) Final Adopted Specification at OMG (ptc/04-04-02).
7. Siegel, J., the OMG Staff Strategy Group: Developing in omg's model-driven architecture. OMG white paper (2001)
8. OMG: MDA Guide Version 1.0. (2003)
9. Bézivin, J.: From object composition to model transformation with the mda. In: IEEE TOOLS-39, Santa Barbara, USA, TOOLS (2001)
10. Born, M., Schieferdecker, I., Gross, H.G., Santos, P.: Model-driven development and testing – case study (2003) http://www.fokus.fraunhofer.de/mdts/.
11. CBOP/DSTC/IBM: MOF Query/Views/Transformations, 2nd Revised Submission (ad/04-01-06). OMG. (2004)
12. Schieferdecker, I., Din, G.: A meta-model for ttcn-3. 1st International Workshop on Integration of Testing Methodologies (ITM 2004) (2004)
13. B.Beizer: Black-Box Testing. John Wiley & Sons, Inc (1995)
14. Dai, Z.R., Grabowski, J., Neukirchen, H., Pals, H.: From Design to Test with UML. Testing of Communicating Systems (Editors: R. Groz and R. Hierons) – 16th IFIP International Conference, TestCom2004, Oxford, Proceedings. Lecture Notes in Computer Science (LNCS) 2644, Springer, pp. 33-49 (2004)
15. Duddy, K., Gerber, A., Lawley, M., Raymond, K., Steel, J.: Model Transformation: A declarative, reusable patterns approach. (In: 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003)) pp. 174–185