JZUS

Testing agile requirements models

BOTASCHANJAN Jewgenij[†1], PISTER Markus[†1], RUMPE Bernhard[2]

(*[1]Software & Systems Engineering, Technische Universität München, Boltzmannstr. 3, D-84758 Garching/Munich, Germany*)

(*[2]Software Systems Engineering, Technische Universität Braunschweig, Mühlenpfordtstr. 23, D-38023 Braunschweig, Germany*)

[†]E-mail: botascha@cs.tum.edu; pister@cs.tum.edu

Abstract: This paper discusses a model-based approach to validate software requirements in agile development processes by simulation and in particular automated testing. The use of models as central development artifact needs to be added to the portfolio of software engineering techniques, to further increase efficiency and flexibility of the development beginning already early in the requirements definition phase. Testing requirements are some of the most important techniques to give feedback and to increase the quality of the result. Therefore testing of artifacts should be introduced as early as possible, even in the requirements definition phase.

Key words: Requirements, UML, Model-based testing, Requirements evolution

Document code: A   **CLC number:**   TP31

INTRODUCTION

One main purpose of requirements is to describe the functionality of software. Thus requirements often serve as a basis for contracts as well as for communication between customers and developers. However, they are usually captured in natural language accompanied by a few top-level informal drawings like use cases or activity diagrams that denote the structure of the functionalities in an abstract way. One disadvantage of natural language is that the developer has to cope with its ambiguity. The usage of precise or even formal descriptions for requirements helps getting along with this problem, because it allows increasing the degree of tool support. Especially in innovative environments, where the requirements as well as the design and the implementation rapidly change, the probability for inconsistencies between formulated requirements and the implemented system is very high. Simulations of the behavior described in the specifications and automatisms to synchronize the requirements with the design and the im-

plementation increase greatly the quality of the result and efficiency of the development.

The approach taken here consists of a number of partially well proven techniques, applied in a new area. The key idea is to combine the advantages of these techniques to gain additional synergy effects. The idea of upfront testing of the design came from the Agile Methods Community, namely Extreme Programming, the use of modeling techniques from object-oriented software development methods (or from software engineering best practices in general), and the intensive use of code generators including behavior generators from embedded software development and in particular automotive industry, where simulation and lately also production code generation from high level, state based modeling techniques are already in use. The goal of this paper is to combine these concepts and transfer them to the earlier phases of requirements modeling.

The usage of the Unified Modeling Language (UML) (OMG, 2002) as a formal requirements description language is the main topic of Section 2.

In Section 3 the automated code generation to cope with evolution of systems is sketched. As one of the primary technical elements of model-based development, the form of model-based tests for the production code is discussed in Section 4. In Section 5 the validation of the requirements models is considered, which aims to leverage the quality of the specification. Section 6 gives the conclusion.

REQUIREMENTS MODELLING WITH UML

UML undoubtedly has become the most popular modeling language for software intensive systems used today. Its precision increases based on the ongoing standardization process. Thus with some adaptations and interpretation guidelines for the language an unambiguous description can be created with relatively small effort. An example for an adaptation can be found in Rumpe (2003).

The UML consists of as many as nine kinds of diagrams usable for the description of the architecture and the design of the software. This variety of diagrams can be also used for requirements models as well. Therewith the developers do not have to handle different and incompatible modeling languages within the same project.

Though using the same description language, there are some distinctions between design and requirements models:

(1) Requirements models are usually less detailed than design models.

(2) Requirements models describe properties of the system as a black box and do not describe the internal structure.

(3) Requirements models often refer to the system as well as to the environment (neighbor systems as well as user behavior), whereas design and implementation models concentrate on the system under development.

Regarding Point 1, there is a general misconception about the formality and precision of languages and statements: A requirement can and should be captured using precise formal language. But even in formal language it is possible to formulate abstract statements that only describe the details really needed in a precise way and not anything more (Kiczales *et al.*, 1997). Although the UML has some deficits in allowing an abstract specificational modeling style, it is well suited to model abstract requirements.

Regarding Point 2, it is clear today, that abstract behavioral specifications work well for algorithms, such as sorting. In business information systems instead, we observe the data structure and accompanying functionality to be an integral part of the requirements model even so it serves also as part of the design. Thus most requirements engineering approaches distinguish between system requirements describing the black box view and constraints influencing the design, architecture and implementation (Wiegers, 2003; Gabb, 1998). The models of these constraints, which will be overtaken from the specification into the architecture and design of the system, have to be explicitly distinguished from the regular design decisions, because they may not be changed arbitrarily without consulting the stakeholders who formulated them. One possible solution is to assign the stereotype «requirement» to such models.

Requirements models are usually built from the user's perspective on the system. Though having vague ideas of the design of the application, the user often describes the system by formulating exemplary working steps that should be supported. By this, the software is described as a service to support working situations. These exemplary situations and the interaction with the software system can be described using sequence and object diagrams, but also using OCL constraints and class diagrams for data structures and invariants (Fig.1).

The task of design models however is to define interfaces and precisely describe the behavior of the system in a white box manner. The black-box description provided by requirements is refined into the "white-box" architecture and design of the system. This motivates the notion of refinement, defined as a mapping between an abstract system interface and a set of concrete design elements (e.g. interfaces or classes). Approaches for the refinement of specifications based on sequence diagrams can be found in Krüger (2000).
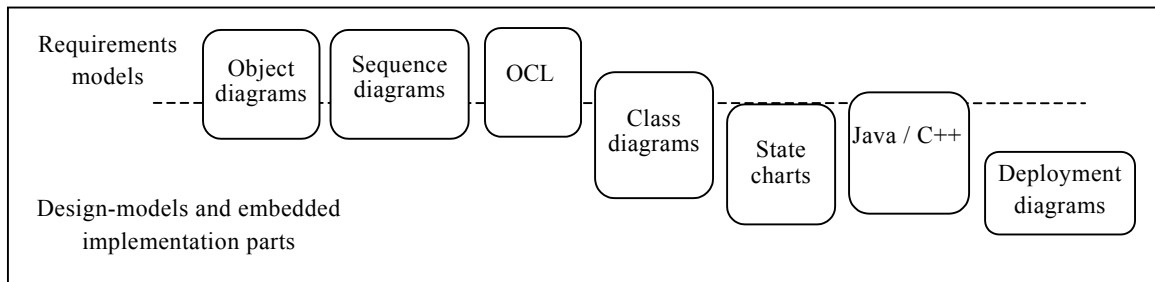
Fig.1 Using UML for requirements and design models

The need for design models in requirements engineering is determined by the need to consider design constraints given by the customer. Usually these constraints are based on an "architecture" model in the form of one or more class diagrams. The behavior is captured with general descriptions in OCL and state charts and exemplary descriptions in the form of sequence and object diagrams.

EXECUTABLE UML

Today, we experience continuously evolving systems. Requirements, design and implementation evolve and the conformance between them has to be ensured continuously.

This task needs methodological and tool support. Some UML-based tools today offer functionality to directly simulate models or at least generate parts of the test code for the software. The continuous improvement of this feature by tool vendors means that a sublanguage of UML will become a high-level programming language and modeling at this level becomes identical to programming. This raises a number of interesting questions mainly dealing with the implications of using an executable UML:

(1) Is it critical for a modeling language to be also used as programming language? For example analysis and design models may become overloaded with details that are not of interest in an early phase, because modelers are addicted to executability.

(2) Is the UML expressive enough to describe systems completely or will it always be accompanied by conventional languages? How well are these integrated?

(3) How will the toolset of the future look like and how will it overcome round trip engineering (i.e. mapping code and diagrams in both directions)? What are the implications of an executable UML on the development process?

In Rumpe (2003; 2002) we partly discussed these and other issues and demonstrated how UML in combination with Java may be used as a high-level programming language (Fig.2). The expression 'executable' in this context means the possibility to generate executable code, either in the form of system code or of executable test case code. Executable models are usually less abstract than design models, but are more compact and abstract than the implementation. Therefore, having an executable modeling language for requirements definition is not a contradiction, but instead an important tool for analysis of requirements. Among other uses, the UML can be used for modeling tests at various levels (class, integration, and system tests). Thus it can be used to describe tests for verifying the requirements at many abstraction levels.

MODEL-BASED TESTING

The use of requirements models for the definition of tests and production code can be manifold:

(1) Code or at least code frames stuffed with default behavior can be generated from a requirements model.
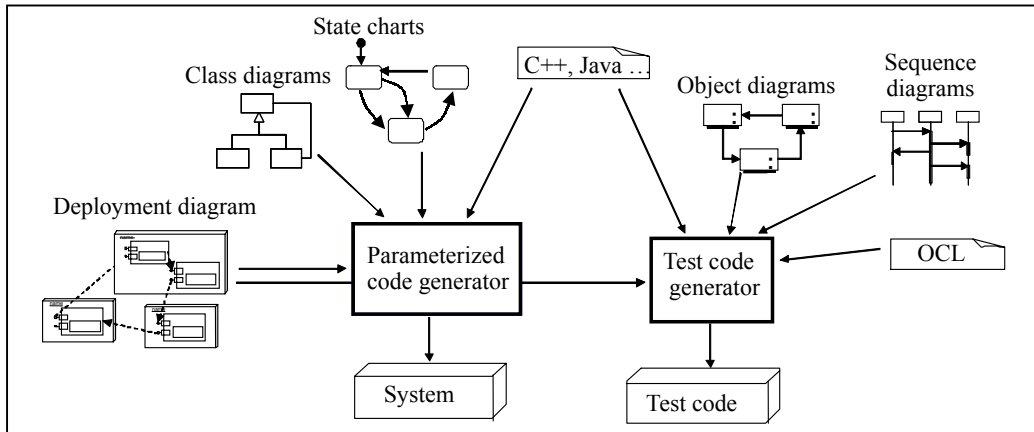
(2) Test cases can be derived from requirements

Fig.2 Input of UML-models for code generation

models that are not used for constructive generation of production code. For example behavioral models, such as statecharts, can be used to derive test cases that cover states, transitions or even paths.

(3) Models can be used for an explicit description of a test case or a part thereof. For example a sequence diagram incorporates a sequence of inputs and expected outputs and thus can be directly used as a test case.

The first two applications are for example discussed in (Briand and Labiche, 2001). Since the nature of requirements models is mainly descriptive, this section concentrates on the test case generation from requirement models and the use of models to describe tests. There already exists a huge variety of testing strategies (Binder, 1999; Briand and Labiche, 2001). A typical test, as shown in Fig.3 consists of a description of the test data, the test driver and an oracle characterizing the desired test result.

In object-oriented environments, the test data can be described by an object diagram (OD). It shows the necessary objects as well as concrete values for their attributes and the linking structure. The test driver can be modeled using a simple method call or, if more complex, a sequence diagram (SD). An SD has the considerable advantage that not only the triggering method calls can be described, but it is possible to model desired interactions and check object states during the test run.

For this purpose, the Object Constraint Language (OCL) (Warmer and Kleppe, 1998) is used to support the description of properties during and after the test run. It has proven efficient for modelling test oracles using a combination of object diagrams and OCL properties. An object diagram in this case serves as a fine grained property description and can therefore be rather incomplete, just focusing on the desired effects. The OCL constraints
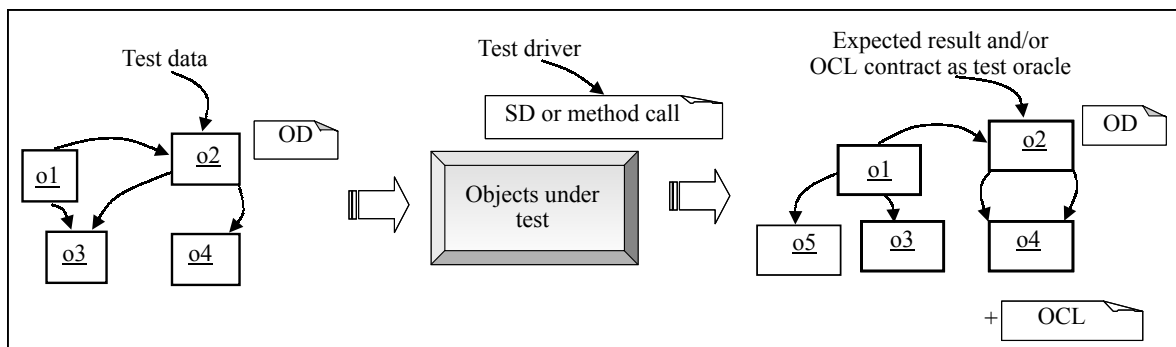

Fig.3 Structure of a test modeled with object diagrams (OD) and sequence diagrams

used can also be general invariants or specific property descriptions. The advantage of using OCL for the property description is the possibility to formulate general predicates for values of attributes. Thus, the same result description or at least parts of it can be reused for several test inputs.

As already mentioned, being able to use the same, coherent language for modeling the production system and the tests give us a good integration between both tasks. It allows the developer to immediately define tests for the production system developed. It is imaginable that in a kind of "test-first modeling approach" (Link and Fröhlich, 2002; Beck, 2001) the test data in the form of possible object structures are developed before the current implementation. This test-first approach perfectly fits the modeling of requirements. The models mostly describe the behavior of the system at the interfaces almost in a form that these requirements can be used as test drivers as well. For the creation of tests, the developers mainly have to create representative test inputs.

Fig.4 shows such a sequence diagram that is typically used for a test case validation. The initial part (marked with the stereotype «trigger») acts as driver for the test; the other interactions of the sequence diagram are observations to be made during a successful test run. This sequence diagram was derived almost directly from a requirements definition and only little extra effort was necessary to transform it into a test (namely adding the stereo-

type). The underlying data structure was reused from exemplaric requirements models, namely an object diagram. Furthermore, the OCL property description at the end of the sequence diagram is perfectly suited for a post-condition check.

VALIDATION OF REQUIREMENTS

In addition to the validation of the implementation to ensure conformity with the modeled requirements, the quality of the requirement definitions themselves has to be validated. It is of strong interest to gain feedback on the formulated requirements as early as possible to prevent errors that are discovered late and therefore expensive at the end. This can be obtained through reviews, but even better through simulation of the requirements models, in order to explore the specified behavior manually (Heymans and Dubois, 1998) or to run automated tests on it.

As explained before, sequence and object diagrams are perfectly suited for defining tests, assuming they are detailed enough. This imposes some additional work on the requirements model, but results in a highly valuable early feedback already in the requirements definition phase. The major problem with this approach is usually the missing complete behavioral description for elements that participate in a test. For example it may be that the behavior of a component is only defined
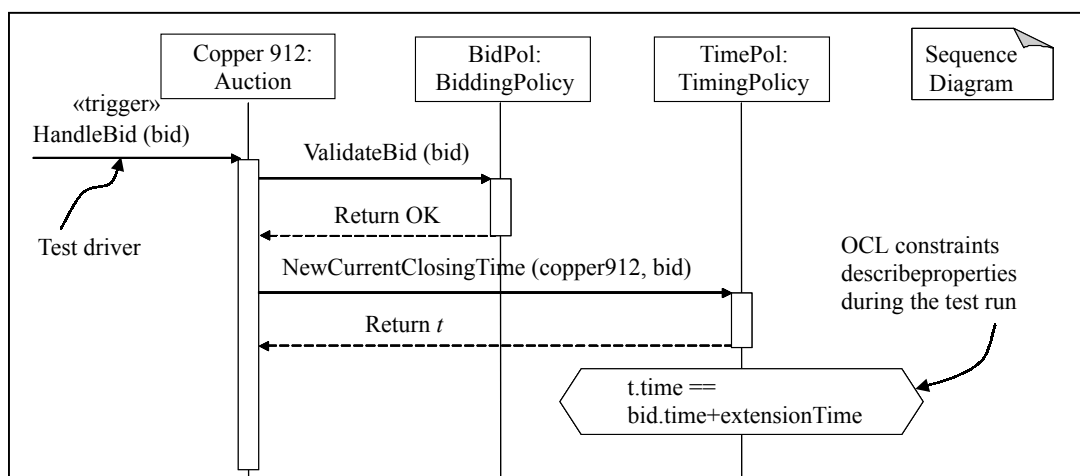

Fig.4 Test case description with a sequence diagram

in terms of a finite number of sequence diagrams, but no implementation or state chart is given. In this case, the implementation has to be simulated according to the given information. We can for example use an approach similar to (Krüger, 2000) or the Play-In/Play-Out Approach (Harel and Rami, 2003) to construct the overall behavior from the given sequence diagrams. For example Fig.5 demonstrates a method to check the consistency between requirements formulated with sequence diagrams and OCL conditions.

This technique is also useful if interaction occurs between components and the environment or neighbor systems that actively participate in tests, but will not become a part of the implementation. The technique allows simulating the environment for testing purposes. It is noticeable that if tests and production code are generated from the same models, both tests and code are consistent with each other (provided that the generators are correct) and therefore errors in the models cannot be detected. It is therefore important to generate tests and implementation from different models.

The idea of automatically creating a running system from the requirements model is an extension of the concept of rapid prototyping. The automatic generation delivers a prototype that checks the desired behavior of the system. During the process of generation, an initial analysis step (type or data flow checking, etc.) may reveal inconsistencies between requirements models. Furthermore, by recei-

ving fast feedback through simulation, the developers and customers can early on get an idea of the running system and adapt the specification before having designed the implementation. In addition to checking the consistency, this technique can be especially useful for validating the accuratacy of a specification by presenting the customer the whole defined functionality. This decreases the effort to be spent for changes and therefore leads to higher software quality and reduced development costs.

However, the generation of automated tests goes considerably beyond the mere prototyping approach. First, the automated tests can be rerun and used in regression testing not only during the requirements modeling, but also during design and implementation. Second, automated tests can be run by everyone in the project, not only by the experts who know about the requirements and thus about desired behavior. Third, experiments with the testing approach have shown that in the long run automated tests pay off, even though it is initially more time consuming to develop tests. Using requirements models for that purpose also increases the efficiency of test development.

CONCLUSIONS

In this paper we have described a pragmatic approach to introducing early feedback in the requirements definition through model-based testing.
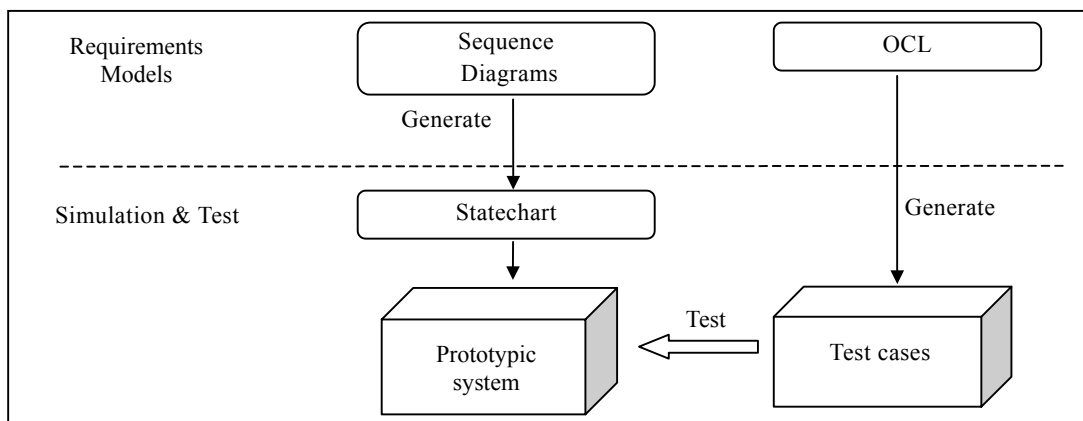

Fig.5 Testing requirements described with sequence diagrams against requirements described with OCL

The approach uses models as primary artifact for requirements and design documentation, code generation and test case development. The validation of the requirements by executable UML models allows the developers to detect errors in the specification earlier. The test case generation from requirements models also allows checking the correctness of the implementation and thus enables validation of the correctness of transformations of the code.

One might argue that requirements elicitation is nowadays usually a step where very informal techniques are used. Mostly, requirements elicitation results in a specification written in natural language. Only after their approval are the requirements mapped into high level analysis models. However, the approach taken here, where the understood requirements are translated into a machine understandable form, ideally also in executable form, has some advantages. It greatly facilitates the feedback of the elicitation process with the user, allows automated consistency checks and tests, as well as analysis for completeness of requirements. This, however, is not applicable in every project, as there may be obstacles like user demands, legal requirements, or very large groups of project members. But we are sure that an increasingly higher portion of projects will be able to use an approach that includes the concepts described here.

There already exist approaches that are working on this topic. For example, the Albert II-approach (Heymans and Dubois, 1998) maps extended MSCs to agent-systems and offers mechanisms for analysis. The Play-In/Play-Out Approach uses extended MSCs for requirements elicitation and validation. However these approaches are not very well integrated with the UML in its current version so the integration with the design task is not as efficient as proposed here.

Model based requirements engineering in evolutional systems will become successful only if well assisted by tools. This includes parameterized code generators for the system as well as for executable test drivers, analysis tools and comfortable help for systematic transformations on models. Fur-

thermore, an important prerequisite for the presented approach is the support of tracing and refinement techniques which allow relating requirements models with their design and/or implementation counterparts in order to automate the reuse of test cases on different abstraction levels.

References

Beck, K., 2001. Aim, Fire (Column on the Test-First Approach). IEEE Software.

Binder, R., 1999. Testing Object-Oriented Systems. Models, Patterns, and Tools. Addison-Wesley.

Briand, L., Labiche, Y., 2001. A UML-based Approach to System Testing. *In*: M. Gogolla and C. Kobryn (eds): «UML»−The Unified Modeling Language, 4th Intl. Conference, LNCS 2185. Springer, p.194-208.

Gabb, A., 1998. The Requirements Spectrum. Proceedings of the first regional Symposium of the Systems Engineering Society of Australia.

Harel, D., Rami, M., 2003. Specifying and executing behavioral requirements: the play-in/play-out approach. *Journal on Software and Systems Modeling*, SOSYM, **2**(2): 82-107.

Heymans, P., Dubois, E., 1998. Scenario-based techniques for supporting the elaboration and the validation of formal requirements. *Requirements Engineering Journal*, **3**:202-218.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopez, C., Loingtier, J.M., Irwin, J., 1997. Aspect-Oriented Programming. ECOOP'97−Object Oriented Programming, 11th European Conference, Jyväskylä, Finnland, LNCS 1241.

Krüger, I., 2000. Distributed System Design with Message Sequence Charts. Ph.D. Thesis, Technische Universität München.

Link, J., Fröhlich, P., 2002. Unit Tests mit Java. Der Test-First-Ansatz. dpunkt.verlag.

OMG, 2002. Unified Modeling Language Specification. V1.5.

Rumpe, B., 2003. Agiles Modellieren mit der UML. Habilitation Thesis. Technische Universität München, Institut für Informatik.

Rumpe, B., 2002. Executable Modeling with UML. A Vision or A Nightmare? *In*: Issues & Trends of Information Technology Management in Contemporary Associations, Seattle. Idea Group Publishing, Hershey, London, p.697-701.

Warmer, J., Kleppe, A., 1998. The Object Constraint Language. Addison-Wesley.

Wiegers, K.E., 2003. Software Requirements. Microsoft Press.