ISSN: 2186-8476, ISSN: 2186-8468 Print
Vol. 2 No. 2, June 2013

A STUDY OF REQUIREMENTS VALIDATION WITH UML

Pin Ng

Hong Kong Community College,
Hong Kong Polytechnic University, HONGKONG.

ccpng@hkcc-polyu.edu.hk

ABSTRACT

The major concern of requirements validation is to evaluate software system at the end of the software development process to ensure compliance with the software requirements. Among the various bases for specifying software requirements, graphical notations are the most suitable means to be used in requirements validation. Many practitioners and researchers have advocated model-based testing for improving the efficiency and effectiveness of test cases generation. The behavior models in Unified Modeling Language (UML) are good candidates for such purposes; in particular, UML state machine model is a useful basis for deriving test scenarios. By traversing the state machine model, feasible transition sequences can be obtained. Each feasible transition sequence represents an operational scenario that describes the desired behavior of the target system. Therefore the feasible transition sequences derived from the state machine model can form a set of test scenarios for requirements validation purposes.

Keywords: Requirements validation, Model-based testing, UML

INTRODUCTION

All engineering projects begin with the definition of requirements. The requirements specification acts as an important contract agreement [18]. It also forms the basis of the design and implementation of the target system. In the context of software development, requirements engineering [15] is one of the early and important phases in the software development life cycle. Requirements engineering consists of several interrelated activities [21][22], including requirements elicitation, requirements specification, requirements validation, and requirements management. In particular, requirements validation [26] plays an important role in both the initial development and the ongoing maintenance of a software system with evolving requirements. The major concern of requirements validation is to evaluate software system at the end of the software development process to ensure compliance with the software requirements [6].

Figure 1 shows a typical requirements validation process. With reference to a complete set of requirements specification, test cases and the associated expected results can be derived. After the software system has been implemented based on the requirements specification, the system is tested with the pre-defined test cases. If the running results conform to the expected results, that demonstrates the system behaves according to the software requirements.

Among the various bases for specifying software requirements [26][7], graphical notations are the most suitable means to be used in requirements validation. Many practitioners and researchers [1][25] have advocated model-based testing for improving the efficiency and effectiveness of test cases generation. Model-based testing is a system testing technique that derives a suite of test cases from a model representing the behavior of a software system. By executing the set of model-based test cases, the conformance of the target system to its specification can be validated.

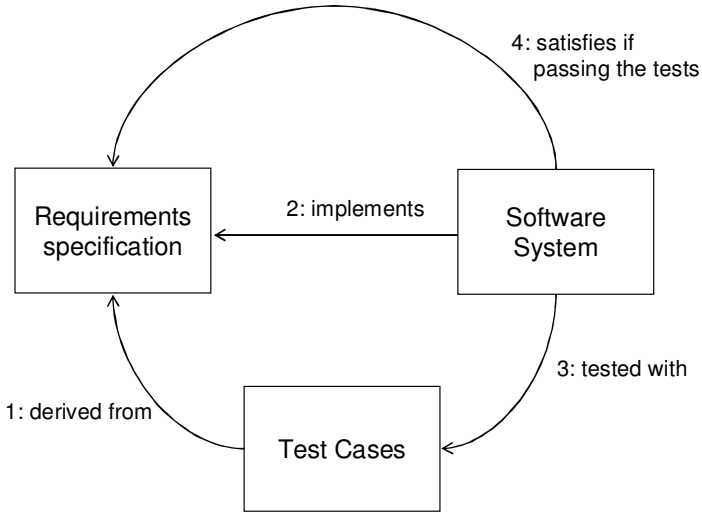

Figure 1. Requirements validation process

CURRENT PRACTICES IN REQUIREMENTS VALIDATION

Verification and validation are two closely related concepts in the field of software testing. Verification answers the question: "Are we building the product right?" whereas validation answers the question: "Are we building the right product?" The ultimate goal of verification and validation is to establish confidence that the software system is 'fit for purpose' [23]. In practice, the verification and validation of a typical software development project involve four stages of testing: unit, integration, system, and acceptance level testing. The four stages of testing are depicted in form of a V-model [13] as shown in Figure 2. Our research work focuses on requirements validation, which is about evaluating software system at the end of the software development process to ensure compliance with the software requirements [6].

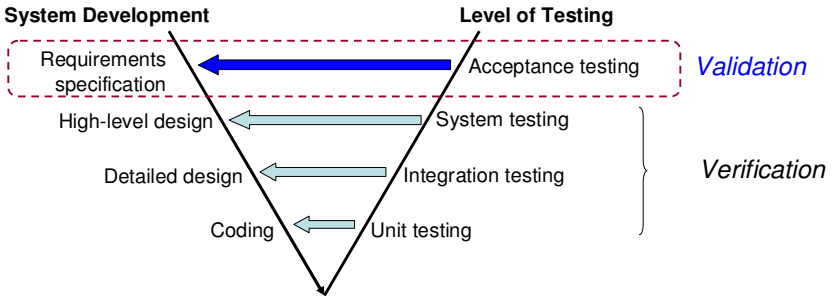

Figure 2. The V-model

In relation to the current practices in requirements validation [26] [7], there are some commonly used bases with which the requirements are specified and the validation test cases are derived.

- *Natural language*: Natural language is often used in documenting software requirements because of flexibility and ease of understanding. However, natural language may introduce ambiguity because different stakeholders may have different interpretations of the statements written in natural language.

- *Design description languages*: This approach uses a language style similar to a programming language but with more abstract features to specify the requirements [23]. One advantage of using a design description language for requirements specification is that the specification can be transformed directly to some programming languages. However, not all stakeholders are familiar with the specific design description language so that there may exist some communication problems.
- *Graphical notations*: A graphical language, supplemented by text annotations, can be used to define the software requirements for the system [23]. Some common examples are: Entity-Relationship Diagram (ERD), Data Flow Diagram (DFD), and Unified Modeling Language (UML). The latest version of UML comprises 13 types of diagrams [20]. The UML diagrams have been widely accepted in the software development industry [11] for visualizing the requirements for the ease of communicating requirements among stakeholders.
- *Mathematical specification languages*: These specifications are derived based on mathematical expressions, such as the Z notation [24]. The specifications are very precise and concise. Validation of the requirements can be achieved through mathematical proofing mechanisms. However, a major obstacle for adopting mathematical expressions is that it is difficult for the common users to fully understand the mathematical specifications of the software requirements.

Among these bases for specifying software requirements, graphical notations are the most suitable means to be used in requirements validation and that is why model-based testing becomes a popular validation technique in the communities of software testing practitioners and researchers [13]. Model-based testing [1] is a system testing technique that derives a suite of test cases from a model representing the behavior of a software system. By executing the set of model-based test cases, the conformance of the target system to its specification can be validated. The typical activities involved in model-based testing [25] are listed as follows:

(1) *System modeling*: This step involves the modeling of the software requirements by using some system models.

(2) *Selecting Test Suite*: Based on the system models, test cases can be derived. Test coverage criteria are used for the guiding the selection of a suite of test cases and determining the adequacy of the test suite.

(3) *Executing Test Suite*: The implemented target system is executed with the selected test cases.

(4) *Test Review*: The running results of executing the target system with the test cases are checked with the expected results. If the running results match with the expected results, it implies the behavior of the target system conforms to the requirements. Otherwise, that implies there are some faults in the implementation of the target system. Debugging activities will be carried out if faults are found. The test cases and the corresponding system model will be useful means for locating the sources of faults.

Testing with model-based specifications has several advantages. Firstly, the specifications in form of a model can be used as a basis for checking the running results. This will reduce the effort of deriving the testing oracles which are essential in deciding whether the running results conform to the expected system behavior. Secondly, the process of deriving the test cases from the model-based specification can help the testing team to discover and resolve the problems in the specification at the early stage of software development. Thirdly, the test scenarios derived based on the model-based specification are implementation independent. That means the test cases are applicable to test any implementations of the software system with different programming languages or software components.

REQUIREMENTS VALIDATION WITH UML

A system model gives an abstract view of a system, highlighting certain important aspects of its design [11]. A 'good' system model can act a driving force for the requirements engineering process [26] by:

(1) System models provide precise and concise descriptions of important aspects of a system that may be too complex to be handled as a whole;

(2) System models provide a valuable means of communication between the members of the development team, and also between the development team and the users.

As software development often involves collaborative work among multiple developers, having a standard modeling language is crucial for a successful software development project. Since the early 1990s, there have been a number of different object-oriented analysis and design methods. Unified Modeling Language (UML) was evolved from the work of Grady Booch, James Rumbaugh, and Ivar Jacobson [20]. It has gone through a standardization process with the OMG (Object Management Group) and become an OMG standard [17] is now a main stream in software development [10] and it provides a standard way for developing system blueprints [19]. The UML models address a number of design issues from different perspectives through a variety of diagrams [11]. In the latest version of UML, it comprises 13 types of diagrams as shown in Figure 3.

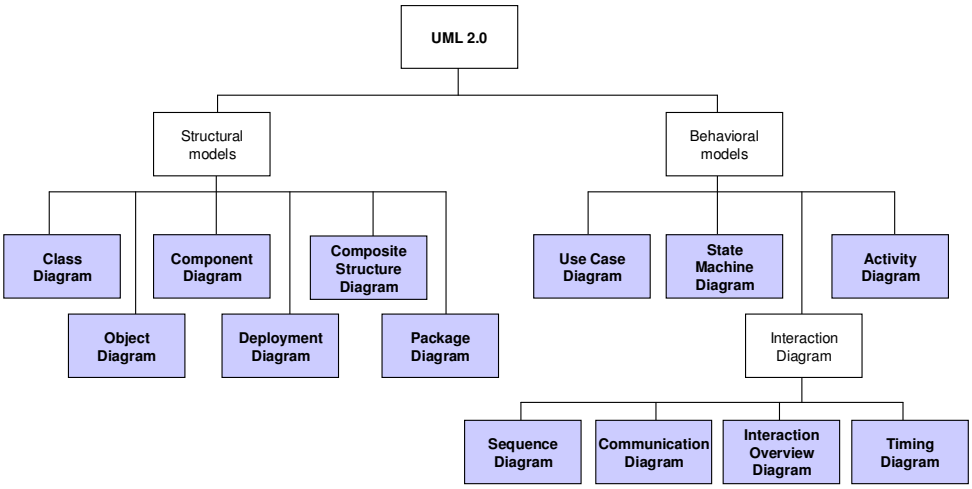

Figure 3. UML 2.0 diagrams

These diagrams are broadly classified into two main categories: structural models and behavioral models.

Structural models are used to model the overall structure of a system and the organization of the system elements [20]. They include:

- *Class diagram* describes the static view of a system. It shows a set of object classes and the relationships among them. Class diagram is an essential model in modeling object-oriented systems.
- *Object diagram* represents a snapshot of object instances found in a class diagram. It shows a set of object instances and their relationships, and addresses the static view of a system from the perspective of some typical cases.
- *Component diagram* models the static implementation view of a system. It shows the dependencies among a set of system components. Each system component could be one or more object classes for implementation.
- *Composite structure diagram* depicts the internal structure of a class, component, or use case.
- *Deployment diagram* represents the static deployment view of a system's physical architecture. It shows the configuration of internal processing units. Deployment diagrams are related to component diagrams in that each processing unit typically encompasses several system components.
- *Package diagram* shows how model elements are organized into packages and the dependencies among the packages.

Behavioral models describe the interactions among the system elements and their runtime behaviors [20]. Usually these interactions are modeled in terms of message passing whereas the run-time behaviors are described through the changes of states and flow of control. Behavioral models include:

- *Use case diagram* models a set of use cases, actors and their relationships. Each use case represents a major service that can be provided by a system. Use case diagrams are useful in organizing and modeling the overall functionality of a system.
- *Sequence diagram* is a kind of interaction diagram. An interaction diagram represents the dynamic view of a system, and models the interactions in form of the messages passed among the objects. A sequence diagram emphasizes on the time ordering of message passing.
- *Communication diagram* is also a kind of interaction diagram. It emphasizes the structural organization of the objects which interact with each other through message passing.
- *State machine diagram* models the dynamic behavior of a system in response to external stimuli. It is particular useful for modeling the lifetime behavior of a system, or its components, whose states are triggered by some specific events.
- *Activity diagram* is considered as a special kind of a state machine diagram. It addresses the dynamic view of a system, and focuses on the

flow of activities within a system. It is useful in modeling flows of control.
Interaction overview diagram is a hybrid of activity diagram and sequence diagram for modeling the control flow of a system or business process.

- *Timing diagram* depicts the change in state of an object over time in response to external events.

Among the various UML diagrams, use case diagram, sequence diagram, and state machine diagram are most widely used in model-based testing [2] [9] [12] [16] at system level, particularly for deriving test cases for testing the conformance of a system to its functional requirements.

- *Use case diagram* represents the high level functionalities provided by the system to the user. A use case or scenario shows how the system interacts with actors [4]. An actor may be a user or an external system with which the target system is communicating. However, since a use case is not something easily measurable, there is no coverage criterion defined for use case testing. Therefore, one limitation of using use case in model-based testing is that it is not possible to determine the test coverage and thus, it is hard to determine the adequacy of testing.
- *Sequence diagram*, in general, captures the time dependent sequences of interactions between objects. Message sequences are used in sequence diagram to model the interactions [8]. In UML, a message is a request for a service from one object to another. Each sequence diagram represents a complete trace of messages during the execution of a user-level operation. A path of message sequence can form a trace of system level interaction. Message sequence path coverage is a coverage criterion for model-based testing with sequence diagram. With that criterion, for each sequence diagram in the requirements specification, there exist some test cases that can exercise each message sequence path at least once. Although sequence diagram is useful in deriving test scenarios, it only represents a partial view of the interactions between multiple system objects.
- *State machine diagram* describes software behaviors with states and transitions and defines the dynamic behavior of software system (or its components) in terms of how it responds to external stimuli [8]. The transition from one state to another is initiated by an event. An event will cause an action and the system will change to another. State machine model is especially useful for modeling reactive systems whose states are triggered by specific events. It is often used for modeling embedded software [3] and user interface design [5]. Being the most formalized model in UML, state machine model [16] provides a natural basis for test case generation. Several coverage criteria have been proposed for test case selection from state machine model. Some of the well established criteria include all transitions, full predicates, and all transition pairs [1].

Nobe and Warner [14] reported their industrial experience at Boeing Commercial Airplane Group and identified the following major advantages of modeling with state

machine model:

- State machine model can be readily used by system domain experts to express and analyze behavioral requirements.
- State machine model allowed designers to simplify the requirements specification.
- State machine model provided engineers with a means for early validation of requirements.
- State machine model facilitated clear communication among project engineers.

These facts exhibit that state machine model is a useful means for requirements analysis, specification, and validation. Naik and Tripathy [13] defined model-based testing with state machine-based specification as follows:

Given a state machine model M of the requirements of a system and an implementation IM of M, model-based testing is to conform that the implementation IM behaves as prescribed by M.

The basic steps are:

(1) Derive sequences of state transitions from *M*.

(2) Turn each sequence of state transition into a test sequence.

(3) Test *IM* with a set of test sequences and observe whether or not *IM* possesses the corresponding sequences of state transitions.

(4) The conformance of *IM* to the requirements can be tested by choosing enough state transition sequences from *M*.

In particular, being the most formalized model in UML, state machine model provides a natural basis for test case generation. By traversing the state machine model, feasible transition sequences can be obtained. Each feasible transition sequence represents an operational scenario that describes the desired behavior of the target system. Therefore the feasible transition sequences derived from the state machine model can form a set of test scenarios for requirements validation purposes.

CONCLUSION

The UML diagrams have been widely accepted in the software development industry for visualizing the requirements for the ease of communicating requirements among stakeholders. The models are used as the blueprints for constructing the software system. They can also be used as the basis for testing for the conformance of the target system to the software requirements. UML state machine model is a popular modeling tool for specifying dynamic perspective of a system and its interactions with the users through sequences of transitions. Each sequence of transitions derived from the state machine model can form a scenario which represents a set of situations of common characteristics that might reasonably occur. The set of feasible transition sequences can serve as test scenarios in requirements validation.

REFERENCES

[1] Binder, R.V. (2000). *Testing Object-Oriented Systems-Models*, Patterns, and Tools, Object Technology. Addison-Wesley.

[2] Briand, L. C., Labiche, Y., & Cui, J. (2005). Automated support for deriving test requirements from UML statecharts. *Software and Systems modeling*, 4(4), 2005, pp.399–423.

[3] Broekman, B. & Notenboom, E. (2003). *Testing embedded software*, Addison-Wesley.

[4] Hass, A. M. J. (2008). *Guide to advanced software testing*. Boston: Artech House, 2008.

[5] Horrocks, I. (1999). *Constructing the user interface with statecharts*. Addison-Wesley.

[6] IEEE (2004). *Guide to the software engineering body of knowledge (SWEBOK)*, Professional Practices Committee, and IEEE Computer Society.

[7] Jones, C. (2010). *Software engineering best practice*s: lessons from successful projects in the top companies, New York: McGraw-Hill.

[8] Kansomkeat, S., Offutt, J., Abdurazik, A., & Baldini, A. (2008). A Comparative Evaluation of Tests Generated from Different UML Diagrams. Proceedings of the 9th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2008), Phuket Thailand, August 2008, pp.867–872.

[9] Korel, B., Singh, I., Tahat, L., & Vaysburg, B. (2003). *Slicing of state-based models.* Proceedings of International Conference on Software Maintenance, ICSM 2003, 22-26 September 2003, pp.34–43.

[10] Lange, C. F. J., Chaudron, M. R. V., & Muskens, J. (2006). In practice: UML software architecture and design description. *IEEE Software*, 2006, 23(2), pp.40–46.

[11] Medvidovic, N., Rosenblum, D. S., Redmiles, D. F., & Robbins, J. E. (2002). Modeling software architectures in the Unified Modeling Language. ACM *Transactions on Software Engineering and Methodology (TOSEM),* 11(1), January 2002, pp.2–57.

[12] Murthy, P. V. R., Anitha, P. C., Mahesh, M., & Subramanyan, R. (2006). Test ready UML statechart models. *Proceedings of the 2006 international workshop on scenarios and state machines*: models, algorithms, and tools SCESM '06, May 2006, pp.75–81.

[13] Naik, K. & Tripathy, P. (2008). *Software testing and quality assurance*: theory and practice. John Wiley & Sons.

[14] Nobe, C. R. & Warner, W. E. (1996). Lessons learned from a trial application of requirements modeling using statecharts. *Proceedings of the Second International Conference on Requirements Engineering*, 15-18 April 1996, pp.86–93.

[15] Nuseibeh, B. & Easterbrook, S. (2000). *Requirements engineering: a roadmap. Proceedings of the International Conference on Software Engineering* (ICSE-2000), pp.35–46.

[16] Offutt, J., Liu, S., Abdurazik, A. & Ammann, P. (2003). Generating Test Data from State-based Specifications. *Software Testing, Verification and Reliability*, 13(1), 2003, pp.25–53.

[17] OMG. (2005). *Unified Modeling Language (UML)*: Superstructure, version 2.0, Object Management Group (OMG), http://www.omg.org, 2005-11-08, August, 2005.

[18] Pressman, R. S. (2010). *Software Engineering*: A Practitioner's Approach, 7th ed., McGraw-Hill.

[19] Priestley, M. (2003). *Practical object-oriented design with UML*, 2nd ed. McGraw-Hill.

[20] Rumbaugh, J., Jacobson, I. & Booch G. (2005). *The Unified Modeling Language Reference Manual,* 2nd ed., Addison-Wesley.

[21] Sawyer, P. & Kotonya, G. (2004). *Software Requirements. Guide to the software engineering body of knowledge (SWEBOK),* Professional Practices Committee, and IEEE.

[22] SEI (1990). *Software Requirements SEI Curriculum Module SEI-CM-19-1.2*, Software Engineering Institute, Carnegie Mellon University, January 1990.

[23] Sommerville, I. (2011). *Software engineering*, 9th ed., Pearson/Addison-Wesley.

[24] Spivey, J. M. (1992). *The Z notation: a reference manual*, 2nd ed., Prentice Hall.

[25] Utting, M. & Legeard, B. (2007). *Practical Model-Based Testing*: A Tools Approach, Morgan Kaufmann.

[26] Van Lamsweerde, A. (2009). *Requirements engineering: from system goals to UML models to software specifications*, John Wiley.