On Acceptance Testing

Conference Paper · January 2013

CITATION
1

READS
632

2 authors:

Wei Shi
Carleton University
97 PUBLICATIONS **574** CITATIONS
SEE PROFILE

Jean-Pierre Corriveau
Carleton University
94 PUBLICATIONS **466** CITATIONS
SEE PROFILE

On Acceptance Testing

Jean-Pierre Corriveau
School of Computer Science
Carleton University
Ottawa, CANADA
jeanpier@scs.carleton.ca

Wei Shi
Faculty of Business and Information Technology
University of Ontario Institute of Technology
Oshawa, CANADA
Wei.shi@uoit.ca

Abstract— **Regardless of which (model-centric or code-centric) development process is adopted, industrial software production ultimately and necessarily requires the delivery of an executable implementation. It is generally accepted that the quality of such an implementation is of utmost importance. Yet current verification techniques, including software testing, remain problematic. In this paper, we focus on acceptance testing, that is, on the validation of the actual behavior of the implementation under test against the requirements of stakeholder(s). This task must be as objective and automated as possible. Our goal here is to review existing code-based and model-based tools for testing in light of what such an objective and automated approach to acceptance testing entails. Our contention is that the difficulties we identify originate mainly in a lack of traceability between a testable model of the requirements of the stakeholder(s) and the test cases used to validate these requirements.**

Keywords— Validation, Acceptance Testing, Model-Based Testing, Traceability, Scenario Models

Contact author for SERP 2013 paper: J-Pierre Corriveau

I. INTRODUCTION

The use and role of models in the production of software systems vary considerably across industry. Whereas some development processes rely extensively on a diversity of semantic-rich UML models [1], proponents of Agile methods instead minimize [2], if not essentially eliminate [3] the need for models. However, regardless of which model-centric or code-centric development process is adopted, industrial software production ultimately and necessarily requires the delivery of an executable implementation. Furthermore, it is generally accepted that the quality of such an implementation is of utmost importance [4]. That is, except for the few who adopt 'hit-and-run' software production [1], the importance of software verification within the software development lifecycle is widely acknowledged. Yet, despite recent advancements in program verification, automatic debugging, assertion deduction and *model-based testing* (hereafter MBT), Ralph Johnson [5] and many others still view software verification as a "catastrophic computer science failure". Indeed, the recent CISQ initiative [6] proceeds from such remarks and similar ones such as: "The current quality of IT application software exposes businesses and government agencies to unacceptable levels of risk and loss." [*Ibid*.]. In summary, software verification remains problematic. In particular, software testing, that is evaluating software by observing its executions on actual valued inputs [7], is "a widespread validation approach in industry, but it is still largely ad hoc, expensive, and unpredictably effective" [8]. Grieskamp [9], the main architect of Microsoft's MBT tool Spec Explorer [10], indeed confirms that current testing practices "are not only laborious and expensive but often unsystematic, lacking an engineering methodology and discipline and adequate tool support".

In this paper, we focus on one specific aspect of software testing, namely the validation [11] of the actual behavior of an *implementation under test* (hereafter IUT) against the requirements of stakeholder(s) of that system. This task, which Bertolino refers to as "acceptance testing" [8], must be as objective and automated as possible [12]. Our goal here is to survey existing tools for testing in light of what such an "objective and automated" approach to acceptance testing entails. To do so, we first discuss in section 2 existing code-based and, in section 3, existing model-based approaches to acceptance testing. We contend that the current challenges inherent to acceptance testing originate first and foremost in a lack of traceability between a testable model of the requirements of the stakeholder(s) and the test cases (i.e., code artifacts) used to validate the IUT against these requirements. We conclude by considering whether Model-Driven Development may offer an avenue of solution.

[1] according to which one develops and releases quickly in order to grab a market share, with little consideration for quality assurance and no commitment to maintenance and customer satisfaction!

II. CODE-BASED ACCEPTANCE TESTING?

Testing constitutes one of the most expensive aspects of software development and software is often not tested as thoroughly as it should be [8, 9, 11, 13]. As mentioned earlier, one possible standpoint is to view current approaches to testing as belonging to one of two categories: code-centric and model-centric. In this section, we briefly discuss the first of these two categories.

A code-centric approach, such as Test-Driven Design (TDD) [3] proceeds from the viewpoint that, for 'true agility', the design must be expressed once and only once, in code. In other words, there is no requirements model per se (that is, captured separately from code). Consequently, there is no traceability [14] between a requirements model and the test cases exercising the code. But such traceability is an essential facet of acceptance testing: without traceability of a suite of test cases 'back to' an explicitly-captured requirements model, there is no objective way of measuring how much of this requirements model is covered [11] by this test suite.

A further difficulty with TDD and similar approaches is that tests cases (in contrast to more abstract *tests* [11]) are code artifacts that are implementation-driven and implementation-specific. Consequently, the reuse potential of such test cases is quite limited: each change to the IUT may require several test cases to be updated. The explicit capturing of a suite of implementation-independent tests generated from a requirements model offers two significant advantages:

1) It decouples requirements coverage from the IUT: a suite of tests is generated from a requirements model according to some coverage criterion. Then, and only then, are tests somehow transformed into test cases proper (*i.e.*, code artifacts specific to the IUT). Such test cases must be kept in sync with a constantly evolving IUT, but this can be done totally independently of requirements coverage.

2) It enables reuse of a suite of tests across several IUTs, be they versions of a constantly-evolving IUT or, more interestingly, competing vendor-specific IUTs having to demonstrate compliance to some specification (e.g., in the domain of software radios).

Beyond such methodological issues faced by code-based approaches to acceptance testing, because the latter requires automation (e.g., [11, 12]), we must also consider tool support for such approaches.

Put simply, there is a multitude of tools for software testing (e.g., [15, 16]), even for specific domains such as Web quality assurance [17]. Bertolino [8] remarks, in her seminal review of the state-of-the-art in software testing, that most focus on functional testing, that is, check "that the observed behavior complies with the logic of the specifications". From this perspective, it appears these tools are relevant to acceptance testing. A closer look reveals most of these tools are code-based testing tools (e.g., JAVA's JUnit [18] and AutoTest [19]) that mainly focus on unit testing [11], that is, on testing individual procedures of an IUT (as opposed to scenario testing [20]). A few observations are in order:

1) There are many types of code-based *verification* tools. They include a plethora of static analyzers, as well as many other types of tools (see [21] for a short review). For example, some tackle design-by-contract [22], some metrics, some different forms of testing (e.g., regression testing [11]). According to the commonly accepted definition of software testing as "the evaluation of software by observing its executions on actual valued inputs" [7], many such tools (in particular, static analyzers) are not testing tools per se.

2) As argued previously, acceptance testing requires an implementation-independent requirements model. While possibly feasible, it is unlikely this testable requirements model (hereafter TRM) would be at a level of details that would enable traceability between it and unit-level tests and/or test cases. That is, typically the tests proceeding from a TRM are system-level ones [11], not unit-level ones.

3) Integration testing tools (such as Fit/Fitness, EasyMock and jMock, etc.) do not address acceptance testing proper. In particular, they do not capture a TRM per se. The same conclusion holds for test automation frameworks (e.g., IBM's Rational Robot [23]) and test management tools (such as HP Quality Centre [24] and Microsoft Team Foundation Server [25]).

One possible avenue to remedy the absence of a TRM in existing code-based testing tools may consist in trying to connect such a tool with a requirements capture tool, that is, with a tool that captures a requirements model but does not generate tests or test cases from it. However, our ongoing collaboration with Blueprint [26] to attempt to link their software to code-based testing tools has revealed a fundamental hurdle with such a multi-tool approach: Given there is no generation of test cases in Blueprint, traceability from Blueprint requirements[2] to test cases (be they generated or merely captured in some code-based testing tool) reduces to *manual* cross-referencing. That is, there is currently no automated way of connecting requirements with test cases. But a scalable approach to

[2] Blueprint offers user stories (which are a simple form of UML Use Cases [11, 27]), UI Mockups and free-form text to capture requirements. The latter are by far the most popular but the hardest to semantically process in an automated way.

acceptance testing requires such automated traceability. Without it, the initial manual linking of (e.g., hundred of) requirements to (e.g., possibly thousands of) test cases (e.g., in the case of a medium-size system of a few tens of thousands lines of code) is simply unfeasible. (From this viewpoint, whether either or both tools at hand support change impact analysis is irrelevant as it is the initial connecting of requirements to test cases that is most problematic.) At this point in time, the only observation we can add is that current experimentation with Blueprint suggests an eventual solution will require that a 'semantic bridge' between this tool and a code-based testing tool be constructed. But this is possible only if both requirements and test cases are captured in such a way that they enable their own semantic analysis. That is, unless we can first have algorithms and tools that can 'understand' requirements and test cases (by accessing and analyzing their underlying representations), we cannot hope to develop a semantic bridge between requirements and test cases. However, such 'understanding' is *extremely* tool specific, which leads us to conclude that a multi-tool approach to acceptance testing is unlikely in the short term (especially if one also has to 'fight' a frequent unfavorable bias of users towards multi-tool solutions, due to their over-specificity, their cost, etc.).

The need for an automated approach to traceability between requirements and test cases suggests the latter be somehow generated from the former. And thus we now turn to model-based approaches to acceptance testing.

III. MODEL-BASED TESTING

In her review of software testing, Bertolino [8] remarks: "A great deal of research focuses nowadays on model-based testing. The leading idea is to use models defined in software construction to drive the testing process, in particular to automatically generate the test cases. The pragmatic approach that testing research takes is that of following what is the current trend in modeling: whichever be the notation used, say e.g., UML or Z, we try to adapt to it a testing technique as effectively as possible [.]"

Model-Based Testing (MBT) [10, 28, 29] involves the derivation of tests and/or test cases from a model that describes at least some of the aspects of the IUT. More precisely, an MBT method uses various algorithms and strategies to generate tests (sometimes equivalently called 'test purposes') and/or test cases from a behavioral model of the IUT. Such a model is usually a partial representation of the IUT's behavior, 'partial' because the model abstracts away some of the implementation details.

Several survey papers (e.g., [8, 30, 31) and special issues (e.g., [29]) have addressed such model-based approaches, as well as the more specific model driven ones (e.g., [32, 33]). Some have specifically targeted MBT tools (e.g., [28]). While some MBT methods use models other than UML state machines (e.g., [34]), most rely on test case generation from such state machines (see [35] for a survey).

Here we will focus on state-based MBT tools that generate executable test cases. Thus we will not consider MBT contributions that instead only address the generation of tests (and thus do not tackle the difficult issue of transforming such tests into executable IUT-specific test cases). Nor will we consider MBT methods that are not supported by a tool (since, tool support is absolutely required in order to demonstrate the executability of the generated test cases).

We start by discussing Conformiq's Tool Suite [36, 37], formerly known as Conformiq Qtronic (as referred to in [35]). This tool requires that a system's requirements be captured in UML statecharts (using Conformiq's Modeler or third party tools). It "generates software tests [...] without user intervention, complete with test plan documentation and executable test scripts in industry standard formats like Python, TCL, TTCN-3, C, C++, Visual Basic, Java, JUnit, Perl, Excel, HTML, Word, Shell Scripts and others." [37]. This includes the automatic generation of test inputs (including structural data), expected test outputs, executable test suites, test case dependency information and traceability matrix, as well as support for boundary value analysis, atomic condition coverage, and other black-box test design heuristics" [*Ibid*.].

While such a description may give the impression acceptance testing has been successfully completely automated, extensive experimentation[3] reveals some significant hurdles:

First, Grieskamp [9], the creator of Spec Explorer [10], another state-based MBT tool, explains at length the problems inherent to test case generation from state machines. In particular, he makes it clear that the *state explosion problem* remains a daunting challenge for all state-based MBT tools (contrary to the impression one may get from reading the few paragraphs devoted to it in the 360-page User Manual from Conformiq [37]). Indeed, even the modeling of a simple game like Yahtzee (involving throwing 5 dice up to three times per round, holding some dice between each throw, to

[3] by the authors and 100+ senior undergraduate and graduate students in the context of offerings of a 4th year undergraduate course in Quality Assurance and a graduate course in Object Oriented Software Engineering twice over the last two years.

achieve the highest possible score according to a specific poker-like scoring algorithm) can require a huge state space if the 13-rounds of the game are to be modeled. Both tools offer a simple mechanism to constrain the state 'exploration' (or search) algorithm by setting bounds (e.g., on the maximum number of states to consider, or the "look ahead depth"). But then the onus is on the user to fix such bounds through trial and error. And such constraining is likely to hinder the completeness of the generated test suite. The use of 'slicing' in Spec Explorer [10], via the specification of a scenario (see Figures 1a and 1b), constitutes a much better solution (to the problem of state explosion) for it emphasizes the importance of *equivalence partitioning* [11] and rightfully places on the user the onus of determining which scenarios are equivalent (a task that, as Binder explains [*Ibid*.], is *unlikely* to be fully automatable).

```
// score 36 end states with 3, 3, 3 (as last dices)
// then score one end state for 2, 2, 1, 1, 3: must score 0
machine ScoreThreeOfAKind() : RollConstraint
{ (    NewGame;
       (RollAll(_, _, 3, 3, 3);
       Score(ScoreType.ThreeOfAKind)
   |   RollAll(2, 2, 1, 1, 3);
       Score(ScoreType.ThreeOfAKind)))
   || (construct model program from RollConstraint)
}
```

Figure 1.a A Spec Explorer scenario for exploring scoring of three-of-a-kind rolls

```
//Sample hold test: should vary only 4th and 5th dice
// Gives 36 possible end states
machine hold1() : RollConstraint
{      (NewGame; RollAll(1,1,1,1,1);
       hold(1); hold(2); hold(3); RollAll)
   || (construct model program from RollConstraint)
}
```

Figure 1.b: A Spec Explorer scenario for holding the first three dice

Second, in Conformiq, requirements coverage[4] is only possible if states and transitions are manually associated with requirements (which are thus merely annotations superimposed on a state machine)! Clearly, such a task lacks automation and scalability. Also, it points to an even more fundamental problem: requirements traceability, that is, the ability to link requirements to test cases. Shafique and Labiche [35, table 4.b] equate "requirements traceability" with "integration with a requirements engineering tool". Consequently, they consider that both Spec Explorer and Conformiq offer only "partial" support for this problem. For example, in Conformiq, the abovementioned requirements annotations can be manually connected to requirements captured in a tool such as IBM RequisitePro or IBM Rational DOORS [37, chapter 7]. However, we believe this operational view of requirements traceability downplays a more fundamental semantic problem identified by Grieskamp [9]: a system's stakeholders are much more inclined to associate requirements to scenarios [20] (such as UML use cases [27]) than to parts of a state machine... From this viewpoint:

1) Spec Explorer implicitly supports the notion of scenarios via the use of "sliced machines", as previously illustrated. But slicing is a sophisticated technique drawing on semantically complex operators [10]. Thus, the state space generated by a sliced machine often may not correspond to the expectations of the user. This makes it all-the-more difficult to conceptually and then manually link the requirements of stakeholder's to such scenarios.

2) Conformiq *does* support use cases, which can be linked to requirements and can play a role in test case generation [37, p.58]. Thus, instead of having the user manually connect requirements to elements of a state machine, a scenario-based approach to requirements traceability could be envisioned. Intuitively this approach would associated a) requirements with use cases and b) paths of use cases with series of test cases. But, unfortunately, this would require a totally different algorithm for test case generation, one not rooted in state machines, leading to a totally different tool.

Third, test case executability may not be as readily available as what the user of an MBT tool expects. Consider for example, the notion of a "scripting backend" in Conformiq Designer. For example [37, p.131]: "The TTCN-3 scripting backend publishes tests generated by Conformiq Designer automatically in TTCN-3 and saves them in TTCN-3 files. TTCN-3 test cases are executed against a real system under test with a TTCN-3 runtime environment and necessary adapters." The point to be grasped is (what is often referred to as) 'glue code' is required to connect the generated tests to an actual IUT. Though less obvious from the documentation, the same observation holds for the other formats (e.g., C++, Perl, etc.) for which Conformiq offers such backends. For example, we first read [37, p.136]: "With Perl script backend, Perl test cases can be derived automatically from a functional design model and be executed against a real system." And then find out on the next page that this in fact

[4] not to be confused with state machine coverage, nor with test suite coverage, both of these being directly and quite adequately addressed by Conformiq and Spec Explorer [35, tables 2 and 3].

requires "the location of the Perl test harness module, i.e., the Perl module which contains the implementation of the routines that the scripting backend generates." In other words, Conformiq does provide not only test cases but also offers a (possibly 3rd party) test harness [*Ibid*.] that enables their execution against an IUT. But its user is left to create glue code to bridge between these test cases and the IUT. This manual task is not only time-consuming but potentially error-prone [11]. Also, this glue code is implementation-specific and thus, both its reusability across IUTs and its maintainability are problematic.

In Spec Explorer [10], each test case corresponds to a specific path through a generated state machine. One alternative is to have each test case connected to the IUT by having the rules of the specification (which are used to control state exploration, as illustrated shortly) explicitly refer to procedures of the IUT. Alternatively, an adapter, that is, glue code, can be written to link these test cases with the IUT. That is, once again, traceability to the IUT is a manual task. Furthermore, in this tool, test case execution (which is completely neatly integrated into Visual Studio) relies on the IUT inputting test case specific data (captured as parameter *values* of a transition of the generated state machine) and outputting the expected results (captured in the model as return *values* of these transitions). As often emphasized in the associated tutorial videos (especially, session 3 part 2), the state variables used in the Spec Explorer rules are only relevant to state machine exploration, not to test case execution. Thus any probing into the state of the IUT must be explicitly addressed through the use of such parameters and return values. The challenge of such an approach can be illustrated by returning to our Yahtzee example. Consider a rule called RollAll to capture the state change corresponding to a roll of the dice:

```
    [Rule]
     static void RollAll(int d1, int d2, int d3, int d4, int d5)
{           Condition.IsTrue(numRolls < 3);
            Condition.IsTrue(numRounds < 13);
            if (numRolls == 0)   {
                        Condition.IsTrue(numHeld == 0);   }
             else   {  Condition.IsTrue(!d1Held || d1 == d1Val);
                        Condition.IsTrue(!d2Held || d2 == d2Val);
                        Condition.IsTrue(!d3Held || d3 == d3Val);
                        Condition.IsTrue(!d4Held || d4 == d4Val);
                        Condition.IsTrue(!d5Held || d5 == d5Val);
                    }
           /* store values from this roll */
            d1Val = d1;          d2Val = d2;     d3Val = d3;
            d4Val = d4;          d5Val = d5;     numRolls += 1;
}
```

Here numRolls, numRounds, numHeld, d_iHeld and d_iVal are all state variables. Without going in details, this rule enables all valid rolls (with respect to the number of rounds, the number of rolls and which dice are to be held) to be potential next states. So, if before firing this rules the values for d_iVal were {1, 2, 3, 4, 5} and those of the d_iHeld were {true, true, true, true, false}, then only rolls that have the first 4 dice (which are held) as {1, 2, 3, 4} are valid as next rolls. The problem is that {1, 2, 3, 4, 5} is valid as a next roll. But, when testing against an IUT, this rule makes it impossible to verify whether the last dice was held by mistake or actually rerolled and still gave 5. The solution attempted by students given this exercise generally consists in adding 6 more Boolean parameters to RollAll: each Boolean indicating if a die is held or not. The problem with such a solution is that it leads to state explosion (especially if the scenario under test addresses the 3 throws of a round!). One alternative, which is far less obvious, is to use the return value of this rule to indicate for each die if it was held or not...

The key point to be grasped from this example is that, beyond issues of scalability and traceability, one fundamental reality of all MBT tools is that their semantic intricacies can significantly impact on what acceptance testing can and cannot address. For example, in Yahtzee, given a game consists of 13 rounds to be each scored once into one of the 13 categories of the scoring sheet, a tester would ideally want to see this scoring sheet after each roll in order to ensure not only that the most recent roll has been scored correctly but also that previous scores are still correctly recorded. But achieving this is notoriously challenging unless it is explicitly programmed into the glue code that connects the test cases to the IUT; an approach that is quite distant from the goals of automated testing.

Finally, on the topic of semantics, it is important to emphasize the wide spectrum of semantics found in MBT tools. Consider, for example, Cucumber rooted in BDD [38], a user-friendly language for expressing scenarios. But these scenarios are extremely simple (nay simplistic) compared to the ones expressible using slicing in Spec Explorer [10]. In fact, most MBT tools cannot adequately address the semantic complexities (e.g., temporal scenario inter-relationships [20]) of a scenario-driven approach to test case derivation [*Ibid*.])[5]. The question then is to ask how relevant to acceptance testing other semantic approaches may be. We consider this issue next.

[5] despite, we repeat, Grieskamp's [9] crucial observation that the stakeholders of a software system are much likelier to express their requirements using scenarios than state machines!

IV. DISCUSSION

There exists a large body of work on modeling 'specifications' in vacuum, that is, with no connection to an executable system. From Büschi automata to Formula [39], researchers have explored formalisms whose semantics enable objective (and possibly automated) 'model checking', which consists in deciding if a model is well-formed or not. The lack of traceability to an IUT disqualifies such work from immediate use for acceptance testing. In fact, because the semantic gap between such approaches and what can be observed from the execution of a system under test is so significant, it is unlikely such approaches will be reconcilable in the short or medium term with the demands of practical acceptance testing, especially with respect to traceability from requirements to test cases.

Because the lack of traceability between models and code is widely acknowledged as a common problem, we should consider modeling approaches not specifically targeted towards acceptance testing but that address traceability. More to the point, we must now ask if *model-driven design* (MDD) [40] may be the foundations on which to build a scalable traceable approach to acceptance testing. MDD's philosophy that "the model is the code" [*Ibid.*] certainly seems to eliminate the traceability issue between models and code: code can be easily regenerated every time the model changes[6]. And since, in MDD tools (e.g., [41]), code generation is based on state machines, there appears to be an opportunity to reuse these state machines not just for code generation but also for test case generation. This is indeed feasible with Conformiq Designer [36], which allows the reuse of state machines from third party tools. But there is a major stumbling block: while both code and test cases can be generated (albeit by different tools) from the same state machines, they are totally independent. In other words, the existence of a full code generator does not readily help with the problem of traceability from requirements to test cases. In fact, because the code is generated, it is extremely difficult to reuse it for the construction of the scriptends that would allow Conformiq's user to connect test cases to this generated IUT. Moreover, such a strategy defeats the intention of full code generation in MDD, which is to have the users of an MDD tool never have to deal with code directly (except for defining the actions of transitions in state machines).

One possible avenue of solution would be to develop an integrated generator that would use state machines to generate code and test cases for this code. But traceability of such test cases back to a requirements models (especially a scenario-driven one, as advocated by Grieskamp [9]), still remains unaddressed by this proposal. Thus, at this point in time, the traceability offered in MDD tools by virtue of full code generation does not appear to help with the issue of traceability between requirements and test cases for acceptance testing. Furthermore, one must also acknowledge Selic's [40] concerns about the relatively low level of adoption of MDD tools in industry.

In the end, despite the dominant trend in MBT of adopting state-based test and test case generation, it may be necessary to consider some sort of scenario-driven generation of test cases from requirements for acceptance testing. This seems eventually feasible given the following concluding observations:

1) There is already work on generating tests out of use cases [11, 42] and use case maps [43], and generating test cases out of sequence diagrams [44, 45]. Path sensitization [11] is the key technique typically used in these proposals. There are still open problems with path sensitization [*Ibid.*]. In particular, automating the identification of the variables to be used for path selection is problematic. As is the issue of path coverage (in light of a potential explosion of the number of possible paths in a scenario model). In other words, the fundamental problem of equivalence partitioning [*Ibid.*] remains and an automated solution for it appears to be quite unlikely. However, despite all of this, we remark simple implementations of this technique already exist (e.g., [43] for use case maps).

2) (Partial if not ideally fully) automated traceability between these three models can certainly be envisioned given their semantic closeness, each one in fact refining the previous one.

3) Traceability between sequence diagrams and an IUT appears quite straightforward given the low-level of abstraction of such models.

4) Within the semantic context of path sensitization, tests can be thought of as paths (i.e., sequences) of observable responsibilities (i.e., small testable functional requirements). Thus, because tests from use cases, use case maps and sequence diagrams are all essentially paths of responsibilities, and because responsibilities ultimately map onto procedures of the IUT, automated traceability between tests and test cases and between test cases and IUT seems realizable.

[6] As one of the original creators of the ObjecTime toolset, which has evolved in Rational Rose Technical Developer [41], the first author of this paper is well aware of the semantic and scalability issues facing existing MDD tools. But solutions to these issues are not as relevant to acceptance testing as the problem of traceability.

REFERENCES

[1] Kruchten, P.: The Rational Unified Process, Addison-Wesley, Reading, 2003.

[2] Rosemberg, D. and Stephens, M.: Use Case Driven Object Modeling with UML, APress, New York, 2007.

[3] Beck, K.: Test-Driven Development: By Example. Addison-Wesley Professional, Reading, 2002.

[4] Jones, C. and Bonsignour, O.: The Economics of Software Quality, Addison-Wesley Professional, 2011.

[5] Johnson, R.: Avoiding the classic catastrophic computer science failure mode, 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010.

[6] Surhone, M., Tennoe, M. and Henssonow, S.: Cisq, Betascript Publishing, 2010.

[7] Ammann P. and Offutt J.: Introduction to Software Testing, Cambridge University Press, 2008.

[8] Bertolino, A.: Software Testing Research: Achievements, Challenges and Dreams, Future of Software Engineering (FOSE '07), pp.85-103, IEEE Press, Minneapolis, May 2007.

[9] Grieskamp, W.: Multi-Paradigmatic Model-Based Testing, Technical Report, pp.1-20, Microsoft Research, August 2006.

[10] Spec Explorer Visual Studio Power Tool, http://visualstudiogallery.msdn.microsoft.com/271d0904-f178-4ce9-956b-d9bfa4902745

[11] Binder, R.: Testing Object-Oriented Systems, Addison-Wesley Professional, Reading, 2000.

[12] Corriveau, J.-P.: Testable Requirements for Offshore Outsourcing, SEAFOOD, Zurich, February 2007.

[13] Meyer, B.: The Unspoken Revolution in Software Engineering, IEEE Computer 39(1), pp.121-123, 2006.

[14] Corriveau, J.-P.: Traceability Process for Large OO Projects, IEEE Computer 29(9), pp.63-68, 1996.

[15] First list of testing tools: http://www.info.com/Tools%20Software%20Testing?cb=22&cmp=316574&gclid=CO7R4eeH1LICFaR9OgodYBsAmw

[16] Second list of testing tools: http://en.wikipedia.org/wiki/Category:Software_testing_tools

[17] Testing tools for Web QA: http://www.aptest.com/webresources.html

[18] JUnit, http://www.junit.org/

[19] Meyer, B. et al.: Programs that test themselves, IEEE Computer 42(9), pp.46-55, 2009.

[20] Ryser, J. and Glinz, M.: SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test., Technical Report. University of Zurich, 2003.

[21] Arnold, D., Corriveau, J.-P. and Shi, W.: Validation against Actual Behavior: Still a Challenge for Testing Tools, SERP, Las Vegas, July, 2010.

[22] Meyer, B.: Design by Contract. IEEE Computer 25(10), pp. 40-51, 1992.

[23] IBM: Rational Robot, http://www-01.ibm.com/software/awdtools/tester/robot/

[24] HP Quality Centre http://www8.hp.com/us/en/software-solutions/software.html?compURI=1172141&jumpid=ex_r11374_us/en/large/eb/go_qualitycenter#.UTEXSI76TJo

[25] TFS http://msdn.microsoft.com/en-us/vstudio/ff637362.aspx

[26] Blueprint https://documentation.blueprintcloud.com/Blueprint5.1/Default.htm#Help/Project%20Administration/Tasks/Managing%20ALM%20targets/Creating%20ALM%20targets.htm

[27] UML Superstructure Specification, v2.3, http://www.omg.org/spec/UML/2.3/

[28] Utting, M. and Legeard, B. 2007: Practical Model-Based Testing: A Tools Approach, Morgan Kauffmann

[29] Testing Experience, Model-Based Testing, March 2012, Díaz & Hilterscheid GmbH, Germany

[30] Prasanna, M. et al.: A survey on Automatic Test Case Generation, Academic Open Internet Journal, Volume 15, part 6, 2005

[31] Neto, A. et al.: A survey of Model-based Testing Approaches, WEASELTech'07, Atlanta, November 2007.

[32] Baker, P., Dai, Z.R., Grabowski, J., Schieferdecker, I. and Williams, C.: Model-Driven Testing: Using the UML Profile, Springer, 2007.

[33] Bukhari, S. and Waheed, T.: Model driven transformation between design models to system test models using UML: A survey, Proceedings of the 2010 National S/w Engineering Conference, article 08, Rawalpindi, Pakistan, October 2010.

[34] http://wiki.eclipse.org/EclipseTestingDay2010_Talk_Seppmed

[35] Shafique, M. and Labiche, Y.: A Systematic Review of Model Based Testing Tool Support, Technical Report, SCE-10-04, Carleton University, 2010

[36] Conformiq Tool Suite, http://www.verifysoft.com/en_conformiq_automatic_test_generation.html

[37] Conformiq Manual, http://www.verifysoft.com/ConformiqManual.pdf

[38] Chelimsky, D. et al.: The RSpec Book: Behaviour Driven Development with Rspec, Cucumber and Friends, Pragmatic Bookshelf , 2010.

[39] FORMULA, http://research.microsoft.com/en-us/projects/formula/

[40] Selic, B.: Filling in the Whitespace, http://lmo08.iro.umontreal.ca/Bran%20Selic.pdf

[41] Rational Technical Developer, http://www-01.ibm.com/software/awdtools/developer/technical/

[42] Nebut C., Fleury F., Le Traon Y., and Jézéquel J. M.: Automatic Test Generation: A Use Case Driven Approach. IEEE Transactions on Software Engineering, Vol. 32, 2006.

[43] A. Miga, Applications of Use Case Maps to System Design with Tool Support, M.Eng. Thesis, Dept. of Systems and Computer. Engineering, Carleton University, 1998.

[44] Zander, J. et al: From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing. 17th International Conf. on Testing Communicating Systems TestCom 2005, Montreal, Canada, ISBN: 3-540-26054-4, May 2005.

[45] Baker, P. and Jervis, C.: Testing UML 2.0 Models using TTCN-3 and the UML 2.0 Testing Profile. LNCS 4745, pp. 86-100, 2007.