Journal of Software Engineering and Applications, **2013, 6, 36-46**
http://dx.doi.org/10.4236/jsea.2013.610A005 Published Online October 2013 (http://www.scirp.org/journal/jsea)

Scientific Research

Traceability in Acceptance Testing

Jean-Pierre Corriveau[1], Wei Shi[2]

[1]School of Computer Science, Carleton University, Ottawa, Canada; [2]Business and Information Technology, University of Ontario Institute of Technology, Oshawa, Canada.
Email: jeanpier@scs.carleton.ca, wei.shi@uoit.ca

Received August 30th, 2013; revised September 28th, 2013; accepted October 6th, 2013

Copyright © 2013 Jean-Pierre Corriveau, Wei Shi. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

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 first goal 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. We then investigate whether such traceability is addressed in other relevant specification-based approaches.

Keywords: Validation; Acceptance Testing; Model-Based Testing; Traceability; Scenario Models

1. 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 [4]. 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]: errors originating in requirements have catastrophic economic consequences, as demonstrated by Jones and Bonsignour [4]. 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

[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!

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 then investigate whether such traceability is addressed in other relevant specification-based approaches.

Jones and Bonsignour [4] suggest that the validation of both functional and non-functional requirements can be decomposed into two steps: requirements analysis and requirements verification. They emphasize the importance of requirements analysis in order to obtain a specification (*i.e.*, a model) of a system's requirements in which defects (e.g., incompleteness and inconsistency) have been minimized. Then requirements verification checks that a product, service, or system (or portion thereof) meets a set of design requirements captured in a specification. In this paper, we only consider functional requirements and, following Jones and Bonsignour, postulate that requirements analysis is indeed a crucial first step for acceptance testing (without reviewing however the large body of literature that pertains to this task). We start by addressing code-based approaches to acceptance testing because they in fact reject this postulate.

2. 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, a specification of the requirements of a system captured separately from code). Consequently, there is no traceability [14] between a requirements model and the test cases exercising the code. But, in our opinion, 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. Let us consider, for illustration, the game of Yahtzee[2] (involving throwing 5 dice up to three times per round, holding some dice between each throw, to achieve the highest possible score according to a specific poker-like scoring algorithm). In an assignment given to more than a hundred students over several offerings of a 4th year undergraduate course in Software Quality Assurance at Carleton, students were first asked to develop a simple text-based implementation of this game using TDD. Despite familiarity with the game and widespread availability of the rules, it is most telling that only a few students had their implementation prevent the holding of all 5 dice for the second or third roll... The point to be grasped is that requirement analysis (which does not exist in TDD for it would require the production of a specification) would likely avoid this omission by checking the completeness of the requirements pertaining to holding dice.

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. For example, returning to our Yahtzee experiment, we observed that, even for such a small and quite simple application, the implementations of the students shared similar designs but vastly differed at the code level. Consequently, the test suites of students also vastly differed in their code. For example, some students handled the holding of dice through parameters of the procedure responsible for a single roll, some used a separate procedure, some created a data structure for the value and the hold value of each die, and some adopted much less intuitive approaches (e.g., involving the use of complex return values...) resulting in rather "obscure" test cases. In a follow-up assignment (before the TDD assignment was returned and students could see which tests they had missed), students were asked to develop a suite of implementation-independent tests (written in English) for the game. Students were told to refer to the "official" rules of the game to verify both consistency and completeness as much as they could (that is, without developing a more formal specification that would lend itself to a systematic method for verifying consistency and completeness). Not surprisingly, in this case, most test suites from students were quite similar.

Thus, in summary, the reuse potential of implementation-driven and implementation-specific test cases is quite limited: each change to the IUT may require several test cases to be updated. In contrast, the explicit capturing of a suite of implementation-independent tests generated from a requirements model offers two significant advantages:

1) It decouples requirements coverage [11] 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.*, executable 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. For example, how many spe-

[2]http://en.wikipedia.org/wiki/Yahtzee

cific test cases are devoted to holding dice or to scoring a (valid or invalid) full house in Yahtzee, can be completely decided before any code is written.

2) It enables reuse of a suite of tests across several IUTs, be they versions of a constantly evolving IUT or competing vendor-specific IUTs having to demonstrate compliance to some specification (e.g., in the domain of software radios). For example, as a third assignment pertaining to Yahtzee, students are asked to develop a graphical user interface (GUI) version of the game and demonstrate compliance of their implementation to the suite of tests (not test cases) we provide. Because performance and usability of the GUI are both evaluated, implementations can still vary (despite everyone essentially using the same "official" scoring sheet as the basis for the interface). However, a common suite of tests for compliance ensures all such submissions offer the same functionality, regardless of how differently this functionality is realized in code.

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 (see [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 as they do not involve the execution of code.

2) As stated previously, we postulate 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] (that is, intuitively, ones that view the system as a black box), not unit-level ones (*i.e.*, specific to particular procedures). Let us consider once more the issue of holding dice in the game of Yahtzee to illustrate this point. As mentioned earlier, there are several different ways of implementing this functionality, leading to very different code. Tests pertaining to the holding of dice are derived from a TRM and, intuitively, involve determining:

- how many tests are sufficient for the desired coverage of this functionality
- what the first roll of each test would be (fixed values or random ones)

and then for each test:

- what dice to hold after the first roll
- what the 2nd roll of each test would be (verifying whether holding was respected or not)
- whether a third roll occurs or not, and, if it does:
  a) what dice to hold after the second roll
  b) what the 3rd roll is (verifying whether holding was respected or not)

The resulting set of tests is implementation-independent and adopts a user perspective. It is a common mistake however to have the creators of tests wrongfully postulate the existence of specific procedures in an implementation (e.g., a hold procedure with five Boolean parameters). This error allows the set of tests for holding to be expressed in terms of sequences of calls to specific procedures, thus incorrectly linking system-level tests with procedures (*i.e.*, unit-level entities). In reality, automatically inferring traceability between system-level tests and unit-level test cases is still, to the best of our knowledge, an open problem (whereas manual traceability is entirely feasible but impractical due to an obvious lack of scalability, as discussed shortly). Furthermore, we remark that the decision as to how many tests are sufficient for the desired coverage of the holding functionality must be totally independent of the implementation. (For example, it cannot be based on assuming that there is a hold procedure with 5 Boolean parameters and that we merely have to "cover" a sufficient number of combinations of these parameters. Such a tactic clearly omits several facets of the set of tests suggested for the hold functionality.)

Thus, in summary, tools conceived for unit testing cannot directly be used for acceptance testing.

3) Similarly, 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

Copyright © 2013 SciRes.

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[3] to test cases (be they generated or merely captured in some code-based testing tool) currently reduces to manual cross-referencing. That is, there is currently no automated way of connecting requirements with test cases. But a scalable approach to acceptance testing requires such automated traceability. Without it, the initial manual linking of (e.g., hundreds 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, their learning curves, 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.

3. 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[4] reveals some significant hurdles:

First, Grieskamp [9], the creator of Spec Explorer [10],

[3]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.

[4]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.

Copyright © 2013 SciRes.

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 can require a huge state space if the 13 rounds of the game are to be modeled. Both tools (Conformiq and SpecExplorer) 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 tests. The use of "slicing" in Spec Explorer [10], via the specification of a scenario (see **Figures 1-3**), constitutes a much better solution to the problem of state explosion because 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). (**Figure 3** also conveys how tedious (and non-scalable) the task of verifying the generated state machine can be even for a very simple scenario...)

Second, in Conformiq, requirements coverage[5] is only possible if states and transitions are manually associated

```
// verify handling scoring "three of a kind" works
// correctly: it must return the total of the dice if 3 or
// more are identical.
// compute score for 36 end states with 3, 3, 3 as last dice
// (ie only 2 first dice are random)
// then compute score for the sole end state
// corresponding to roll 2, 2, 1, 1, 3.
// In that case, all dice are fixed and the game must
// score 0 if that roll is scored as a three-of-a-kind
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)
// This last line is the one carrying out the slicing by
// limiting a totally random roll of five dice to the
// sequence of two rolls (and scoring) specified above it.
}
```

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

[5]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].

```
// Sample hold test: we fix completely the first roll,
// then hold its first 3 dice and roll again only 4th and 5th
// dice.
// This test case 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 2. A Spec Explorer scenario for holding the first three dice.

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 elements 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. For example, in the case of Yahtzee, a sliced machine can be obtained quite easily for each of the 13 scoring categories of the game (see **Figures 1** and **3**). Traceability from these machines to the requirements of the game is quite straightforward (albeit not automated). Conversely, other aspects of the game (such as holding dice, ensuring no more than 3 rolls are allowed in a single round, ensuring that no category is used more than once per game, ensuring that exactly 13 rounds are played, etc.) require several machines in order to obtain sufficient coverage. In particular, the machine of **Figure 2** is not sufficient to test holding dice. Clearly, in such cases, traceability is not an isomorphism between sliced machines and requirements. Finally, there are aspects of

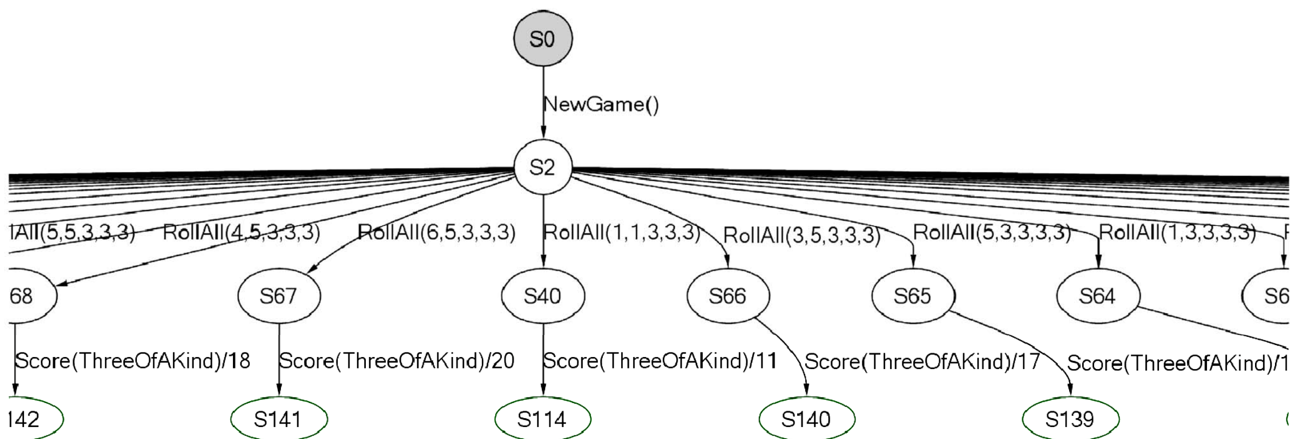

Figure 3. A part of the generated sliced state machine for scoring of three-of-a-kind rolls.

Yahtzee that are hard to address with state machines and/or scenarios. For example, a Yahtzee occurs when all five dice have the same value at the end of a round. Yahtzee is the most difficult combination to throw in a game and has the highest score of 50 points. Without going into details, if a player obtains more than one Yahtzee during a same game, these additional Yahtzees can be used as wild cards (*i.e.*, score full points in *other* categories). For example, a second Yahtzee could be used as a long straight! Such behavior (wild cards at any point in time) drastically complicates models (leading most who attempt to address this feature to later abandon it...). In fact, the resulting models are so much more complex that:

- getting slicing to work correctly is very challenging (read time-consuming, in terms of modeling and verification of the generated machines), especially given insufficient slicing will lead to state exploration failing upon reaching some upper bound (making it even more difficult to decide if the partially generated machine is correct or not). Such a situation typically leads to oversimplifications in the model and/or the slicing scenarios...
- traceability between such machines and the game requirements is not obvious. That is, even someone who is an expert with the game and with Spec Explorer will not necessarily readily know what a particular sliced machine is exactly testing. (This is particularly true when using some of the more powerful slicing operators whose behavior must be thoroughly understood in order to decide if the behavior they generate corresponds or not to what the tester intends.)

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 than the one Conformiq uses. Such an algorithm would not be rooted in state machines but in path sensitization using scenarios [11] and this would lead 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 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.

Copyright © 2013 SciRes.

In Spec Explorer [10], each test case corresponds to a specific path through a generated ‘sliced’ 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 (*i.e.*, 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 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 the rule (**Figure 4**) called RollAll (used in **Figures 1** and **2**) to capture the state change corresponding to a roll of the dice.

In the rule RollAll, 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 5 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.

More specifically:

1) the rule RollAll(int d1, int d2, int d3, int d4, int d5) has 65 = 7776 possible next states but

2) the rule RollAll(int d1, int d2, int d3, int d4, int d5, boolean d1Held, boolean d2Held, boolean d3Held, boolean d4Held, boolean d5Held) has 65 * 25 = 248,832 possible next states.

A round for a player may consist of up to 3 rolls, each one using RollAll to compute its possible next states. In the first version of this rule, if no constraints are used, each of the 7776 possible next states of the first roll has itself 7776 possible next states. That amounts to more

```
 [Rule]
 static void RollAll(int d1, int d2, int d3, int d4, int d5)
{
// We can roll if we haven't rolled 3 times already for this
// round and if we still have a round to play and score
    Condition.IsTrue(numRolls < 3);
    Condition.IsTrue(numRounds < 13);
// if this is the first roll for this round,
// then make sure no die is held
      if (numRolls == 0)
        { Condition.IsTrue(numHeld == 0);   }
      else
   {
// the state variables d_iVal hold the values of the dice
// from the previous roll
// if a dice is held then the new value di of dice i,
// which is a parameter to this rule must be the same as
// the previous value of this die.
      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 in the state variables*/
    d1Val = d1; d2Val = d2; d3Val = d3;
    d4Val = d4; d5Val = d5;
   }   // of else clause
// increment the state variable that keeps track
// of the number of rolls for this round.
 numRolls += 1;
}
```

Figure 4. Rule RollAll.

than 60 million states and we have yet to deal with a possible third roll. The explosion of states is obviously even worse with the second version of the RollAll rule: after two rolls there are 61 billion possible states... State exploration will quickly reach the specified maximum for the number of generated states, despite the sophisticated state-clustering algorithm of SpecExplorer. Furthermore, unfortunately, an alternative design for modeling the holding of dice is anything but intuitive as it requires using 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

Copyright © 2013 SciRes.

scores are still correctly recorded. But achieving this is notoriously challenging in SpecExplorer (unless it is explicitly programmed into the glue code that connects the test cases to the IUT; an approach that is less than ideal in the context of automated testing).

We discuss further the issue of semantics in the context of traceability for acceptance testing in the next section.

4. On Semantics for Acceptance Testing

There exists a large body of work on "specifications" for testing, as discussed at length in [38]. Not surprisingly, most frequently such work is rooted in state-based semantics[6]. For example, recently, Zhao and Rammig [40] discuss the use of a Büschi automaton for a state-oriented form online model checking. In the same vein, COMA [41], JavaMOP [42] and TOPL [43] offer implemented approaches to runtime verification. The latter differs from acceptance testing inasmuch as it is not concerned with the generation of tests but rather with the analysis of an execution in order to detect the violation of certain properties. Runtime verification specifications are typically expressed in trace predicate formalisms, such as finite state machines, regular expressions, context-free patterns, linear temporal logics, etc. (JavaMOP stands out for its ability to support several of these formalisms.) While "scenarios" are sometimes mentioned in such methods (e.g., [44]), they are often quite restricted semantically. For example, Li *et al.* [45] use UML sequence diagrams with no alternatives or loops. Ciraci *et al.* [46] explains that the intent is to have such "simplified" scenarios generate a graph of all possible sequences of executions. The difficulty with such strategy is that it generally does not scale up, as demonstrated at length by Briand and Labiche [47][7]. Similarly, in MBT, Cucumber is a tool rooted in BDD [48], a user-friendly language for expressing scenarios. But these scenarios are extremely simple (nay simplistic) compared to the ones expressible using slicing in SpecExplorer [10].

It must be emphasized that not all approaches to run-time verification that use scenario-based specifications depend on simplified semantics. In particular, Krüger, Meisinger and Menarini [49] rely on the rich semantics of Message Sequence Charts [50], which they extend! But, like many similar approaches, they limit themselves to monitoring sequences of procedures (without parameters). Also, they apply their state machine synthesis algorithm to obtain state machines representing the full communication behavior of individual components of the system. Such synthesized state machines are at the centre of their monitoring approach but are not easy to trace back to the requirements of a system's stakeholders.

Furthermore, all the approaches to runtime verification we have studied rely on specifications that are implementation (and often programming language) specific. For example, valid sequences are to be expressed using the actual names of the procedures of an implementation, or transitions of a state machine are to be triggered by events that belong to a set of method names. Thus, in summary, it appears most of this research bypasses the problem of traceability between an implementation-independent specification and implementation-specific executable tests, which is central to the task of acceptance testing. Requirements coverage may also be an issue depending on how many (or how few) execution traces are considered. Furthermore, as is the case for most MBT methods and tools, complex temporal scenario inter-relationships [20] are often ignored in runtime verification approaches (*i.e.*, temporal considerations are limited to the sequencing of procedures with little attention given to temporal scenario inter-relationships).

At this point of the discussion, we observe that traceability between implementation-independent specifications and executable IUT-specific test cases remains problematic in existing work on MBT and, more generally, in specifications for testing. Hierons [38], amongst others, comes to the same conclusion. Therefore, it may be useful to consider modeling approaches not specifically targeted towards acceptance testing but that appear to address traceability.

First, consider the work of Cristia *et al.* [51] on a language for test refinements rooted in (a subset of) the Z notation (which has been investigated considerably for MBT [*Ibid*.]). A refinement requires:

- "Identifying the SUT's [System Under Test] state variables and input parameters that correspond to the specification variables
- Initializing the implementation variables as specified in each abstract test case
- Initializing implementation variables used by the SUT but not considered in the specification
- Performing a sound refinement of the values of the abstract test cases into values for the implementation variables."

A quick look at the refinement rule found in Figure 3 of [51] demonstrates eloquently how implementation-specific such a rule is. Thus, our traceability problem remains.

In the same vein, Microsoft's FORMULA (Formal Modeling Using Logic Programming and Analysis) [52] is:

[6]Non state-based approaches do exist but are quite remote from acceptance testing. For example, Stoller *et al.* [39] rely on Hidden Markov Models to propose a particular type of runtime verification rooted in computing the probability of satisfying an aspect of a specification.

[7]Imposing severe semantic restrictions on scenarios serves the purpose of trying to limit this graph of all possible sequences of execution. But if loops, alternatives and interleaving are tackled, then the number of possible sequences explodes.

Copyright © 2013 SciRes.

"A modern formal specification language targeting model-based development (MBD). It is based on algebraic data types (ADTs) and strongly-typed constraint logic programming (CLP), which support concise specifications of abstractions and model transformations. Around this core is a set of composition operators for composing specifications in the style of MBD." [*Ibid*.]

The problem is that the traceability of such specifications to a) a requirements model understandable by stakeholders and b) to an IUT remains a hurdle.

In contrast, the philosophy of model-driven design (MDD) [53] that "the model is the code" seems to eliminate the traceability issue between models and code: code can be easily regenerated every time the model changes[8]. And since, in MDD tools (e.g., [54]), 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 purpose 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 a new 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. 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 [53] 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 [55] and use case maps [56,57], and generating test cases out of sequence diagrams [58,59]. Path sensitization [11,12] 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 challenging. As is the issue of path coverage [Ibid.] (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 [11] remains an impediment and an automated solution for it appears to be quite unlikely. However, despite these observations, we remark simple implementations of this technique already exist (e.g., [56] for Use Case Maps).

2) Partial, if not ideally fully automated, traceability between use cases, use case maps and sequence diagrams can certainly be envisioned given their semantic closeness, each one in fact refining the previous one.

3) Traceability between sequence diagrams (such as Message Sequence Charts [50]) 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 [57]). 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 (e.g., via type inference as proposed in [60]) between tests and test cases and between test cases and IUT seems realizable.

5. Acknowledgements

Support from the Natural Sciences and Engineering Research Council of Canada is gratefully acknowledged.

REFERENCES

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

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

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

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

[5] R. Johnson, "Avoiding the Classic Catastrophic Computer Science Failure Mode," *Proceedings of the* 18*th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Santa Fe, 7-11 November 2010,

[8]As one of the original creators of the ObjecTime toolset, which has evolved in Rational Rose Technical Developer [54], 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.

[6] M. Surhone, M. Tennoe and S. Henssonow, "Cisq," Betascript Publishing, New York, 2010.

[7] P. Ammann and J. Offutt, "Introduction to Software Testing," Cambridge University Press, Cambridge, 2008. http://dx.doi.org/10.1017/CBO9780511809163

[8] A. Bertolino, "Software Testing Research: Achievements, Challenges and Dreams," *Proceedings of Future of Software Engineering* (*FOSE* 07), Minneapolis, 23-25 May 2007, pp. 85-103.

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

[10] Microsoft, "Spec Explorer Visual Studio Power Tool," 2013. http://visualstudiogallery.msdn.microsoft.com/271d0904-f178-4ce9-956b-d9bfa4902745

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

[12] J.-P. Corriveau, "Testable Requirements for Offshore Outsourcing," *Proceedings of Software Engineering Approaches for Offshore and Outsourced Development* (*SEAFOOD*), Springer, Berlin, 2007, pp. 27-43. http://dx.doi.org/10.1007/978-3-540-75542-5_3

[13] B. Meyer, "The Unspoken Revolution in Software Engineering," *IEEE Computer*, Vol. 39, No. 1, 2006, pp. 121-123.

[14] J.-P. Corriveau, "Traceability Process for Large OO Projects," *IEEE Computer*, Vol. 29, No. 9, 1996, pp. 63-68. http://dx.doi.org/10.1109/2.536785

[15] "List of Testing Tools," 2013. http://www.softwaretestingclass.com/software-testing-tools-list

[16] Wikipedia, "Second List of Testing Tools," 2013. http://en.wikipedia.org/wiki/Category:Software_testing_tools

[17] "Testing Tools for Web QA," 2013. http://www.aptest.com/webresources.html

[18] "JUnit," 2013. http://www.junit.org

[19] B. Meyer, *et al*., "Programs that Test Themselves," *IEEE Computer*, Vol. 42, No. 9, 2009, pp. 46-55. http://dx.doi.org/10.1109/MC.2009.296

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

[21] D. Arnold, J.-P. Corriveau and W. Shi, "Validation against Actual Behavior: Still a Challenge for Testing Tools," *Proceedings of Software Engineering Research and Practice* (*SERP*), Las Vegas, 12-15 July 2010.

[22] B. Meyer, "Design by Contract," *IEEE Computer*, Vol. 25, No. 10, 1992, pp. 40-51. http://dx.doi.org/10.1109/2.161279

[23] "IBM Rational Robot," 2013. http://www-01.ibm.com/software/awdtools/tester/robot

[24] "HP Quality Centre," 2013. http://www8.hp.com/ca/en/software-solutions/software.html?compURI=1172141#.UkDyk79AiHk

[25] "Team Foudation Server," 2013. http://msdn.microsoft.com/en-us/vstudio/ff637362.aspx

[26] "Blueprint," 2013. https://documentation.blueprintcloud.com/Blueprint5.1/Default.htm#Help/Project%20Administration/Tasks/Managing%20ALM%20targets/Creating%20ALM%20targets.htm

[27] Object Management Group (OMG), "UML Superstructure Specification v2.3," 2013. http://www.omg.org/spec/UML/2.3

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

[29] "Special Issue on Model-Based Testing," *Testing Experience*, Vol. 17, 2012.

[30] M. Prasanna, *et al*., "A Survey on Automatic Test Case Generation," *Academic Open Internet Journal*, Vol. 15, No. 6, 2005.

[31] A. Neto, R. Subramanyan, M. Vieira and G. H. Travassos, "A Survey of Model-Based Testing Approaches," *Proceedings of the* 1*st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies* (*WEASELTech* 07), Atlanta, 5 November 2007, pp. 31-36.

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

[33] S. Bukhari and T. Waheed, "Model Driven Transformation between Design Models to System Test Models Using UML: A Survey," *Proceedings of the* 2010 *National S/w Engineering Conference*, Rawalpindi, 4-5 October 2010, Article 08.

[34] "Seppmed," 2013. http://wiki.eclipse.org/EclipseTestingDay2010_Talk_Seppmed

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

[36] "Conformiq Tool Suite," 2013. http://www.verifysoft.com/en_conformiq_automatic_test_generation.html

[37] "Conformiq Manual," 2013. http://www.verifysoft.com/ConformiqManual.pdf

[38] R. Hierons, *et al*., "Using Formal Specifications to Support Testing," *ACM Computing Surveys*, Vol. 41, No. 2, 2009, pp. 1-76. http://dx.doi.org/10.1145/1459352.1459354

[39] S. D. Stoller, *et al*., "Runtime Verification with State Estimation," *Proceedings of* 11*th International Workshop on Runtime Verification* (*RV'11*), Springer, Berlin, 2011, pp. 193-207.

[40] Y. Zhao and F. Rammig, "Online Model Checking for Dependable Real-Time Systems," *Proceedings of the IEEE* 15*th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, Shenzhen, 11-13 April 2012, pp. 154-161.

Copyright © 2013 SciRes.

[41] P. Arcaini, A. Gargantini and E. Riccobene, "CoMA: Conformance Monitoring of Java Programs by Abstract State Machines," *Proceedings of* 11*th International Workshop on Runtime Verification* (*RV*'11), Springer, Berlin, 2011, pp. 223-238.

[42] D. Jin, P. Meredith, C. Lee and G. Rosu, "JavaMOP: Efficient Parametric Runtime Monitoring Framework," *Proceedings of the* 34*th International Conference on Software Engineering* (*ICSE*), Zurich, 2-9 June 2012, pp. 1427-1430.

[43] R. Grigor, *et al*., "Runtime Verification Based on Register Automata," *Proceedings of the* 19*th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS*), Springer, Berlin, 2013, pp. 260-276.

[44] Z. Zhou, *et al.*, "Jasmine: A Tool for Model-Driven Runtime Verification with UML Behavioral Models," *Proceedings of the* 11*th IEEE High Assurance Systems Engineering Symposium* (*HASE*), Nanjing, 3-5 December 2008, pp. 487-490.

[45] X. Li, *et al*., "UML Interaction Model-Driven Runtime Verification of Java Programs," *Software*, *IET*, Vol. 5, No. 2, 2011, pp. 142-156.

[46] S. Ciraci, S. Malakuti, S. Katz, and M. Aksit, "Checking the Correspondence between UML Models and Implementation," *Proceedings of* 10*th International Workshop on Runtime Verification* (*RV*'10), Springer, Berlin, 2011, pp. 198-213.

[47] L. Briand and Y. Labiche, "A UML-Based Approach to System Testing," *Software and Systems Modeling*, Vol. 1, No. 1, 2002, pp. 10-42.
http://dx.doi.org/10.1007/s10270-002-0004-8

[48] D. Chelimsky, *et al.* "The RSpec Book: Behaviour Driven Development with Rspec, Cucumber and Friends," Pragmatic Bookshelf, New York, 2010.

[49] I. H. Krüger, M. Meisinger and M. Menarini: "Runtime Verification of Interactions: From MSCs to Aspects," *Proceedings of* 7*th International Workshop on Runtime Verification* (*RV*'07), Springer, Berlin, 2007, pp. 63-74.

[50] International Telecommunication Union (ITU), "Message Sequence Charts, ITU Z.120," 2013.
http://www.itu.int/rec/T-REC-Z.120

[51] M. Cristiá, P. Rodríguez Monetti, and P. Albertengo, "The Fastest 1.3.6 User's Guide," 2013.
http://www.flowgate.net/pdf/userGuide.pdf

[52] Microsoft, "FORMULA," 2013.
http://research.microsoft.com/en-us/projects/formula/

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

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

[55] C. Nebut, *et al*., "Automatic Test Generation: A Use Case Driven Approach," *IEEE Transactions on Software Engineering*, Vol. 32, No. 3, 2006, pp. 140-155.
http://dx.doi.org/10.1109/TSE.2006.22

[56] A. Miga, "Applications of Use Case Maps to System Design with Tool Support," Master's Thesis, Carleton University, Ottawa, 1998.

[57] D. Amyot and G. Mussbacher, "User Requirements Notation: The First Ten Years", *Journal of Software*, Vol. 6, No. 5, 2011, pp. 747-768.

[58] J. Zander, *et al.*, "From U2TP Models to Executable Tests with TTCN-3—An Approach to Model Driven Testing," *Proceedings of the* 17*th International Conference on Testing Communicating Systems*, Montreal, 31 May-2 June 2005, pp. 289-303.
http://dx.doi.org/10.1007/11430230_20

[59] P. Baker and C. Jervis, "Testing UML 2.0 Models using TTCN-3 and the UML 2.0 Testing Profile," Springer, Berlin, 2007, pp. 86-100.

[60] D. Arnold, J.-P. Corriveau and W. Shi, "Modeling and Validating Requirements Using Executable Contracts and Scenarios," *Proceedings of Software Engineering Research*, *Management & Applications* (*SERA* 2010), Montreal, 24-26 May 2010, pp. 311-320.

Copyright © 2013 SciRes.