Reconciling Offshore Outsourcing with Model Based Testing

David Arnold[1], Jean-Pierre Corriveau[1], and Wei Shi[2]

[1] School of Computer Science, Carleton University, Ottawa, Canada
{darnold,jeanpier}@scs.carleton.ca
[2] Faculty of Business and IT, UOIT, Oshawa, Canada
wei.shi@uoit.ca

Abstract. In the context of offshore outsourcing, in order to validate the requirements of a stakeholder (the contractor) against the *actual* behavior of an implementation under test (IUT) developed by the contracted, it is necessary to have a requirement specification language from which test cases can be generated *and* executed on an *actual* IUT. Code-based testing tools are limited to unit testing and do not tackle validation *per se*. Conversely, model-based testing tools support the derivation of test cases from a requirements model. But such test cases are not necessarily executable and those tools that claim 'test execution' in fact offer *symbolic execution*, which is carried out using a model of the IUT, *not* the actual IUT. Here, we overview a requirements specification language and its corresponding fully implemented framework, that support the validation of IUT-independent requirements against actual IUT behavior, as required by offshore outsourcing.

Keywords: validation, requirements specification, model-based testing, scenarios, test case generation, executable test cases, contracts.

1 Introduction

In the context of software offshore outsourcing, Meyer [1] observes, "quality is indeed the central issue". A quality-driven approach to software development and testing demands that, ultimately, the requirements of stakeholders be *validated* against the *actual behavior* of an implementation under test (IUT). This is particularly true in the context of software offshore outsourcing. Let us elaborate. As argued elsewhere [2], we view outsourcing as a business relationship and thus assume that a *contract* is required in order to define a) what services are requested from the *contracted* party and b) how these services are to be delivered to the satisfaction of the *contractor*. More precisely, a software offshore outsourcing contract must include, among its quality assurance facets, the specification of a systematic (ideally *objective* and *automated*) approach to the validation of the requirements of the stakeholder (i.e., the contractor) against the *actual behavior* of an implementation under test (IUT) delivered by the contracted.

Unfortunately, typically, there is no such contract, and no systematic and automated approach to validation. Quite on the contrary, generally, the contracted is assumed to

M. Nordio et al. (Eds.): SEAFOOD 2010, LNBIP 54, pp. 6–22, 2010.

mostly carry out extensive unit testing [3]. Depending on the scope of the project, integration testing [3] may also be completely left to the contracted, or, involve both the contractor and the contracted in projects in which the outsourced software is merely a component of a larger system. In parallel to such low-level testing, extensive quality assurance activities may take place. The latter involve gathering metrics pertaining to code coverage, test suite coverage, frequency of build errors and of their resolution time, bug frequency and resolution time, etc. But neither unit/integration testing, nor metrics, tackle the problem we focus on in this paper, namely the *automated* validation of the requirements of the contractor against the *actual behavior* of an IUT delivered by the contracted. In fact, in the context of outsourcing, current practice with respect to validation often consists in offloading it to a separate group of testers. That is, generally, system-level validation is not performed by the developers, nor by the clients (i.e., the stakeholders). Instead, it is left to a separate team of testers, which will treat the IUT as a black box and merely verify that 'representative' use-cases [3] are handled correctly (in terms of system inputs and outputs). In the context of offshore outsourcing, such an approach leads to two observations: First, a separate team of testers entails one more participant in the already complex communication puzzle inherent to offshore outsourcing. Second, regardless of whether such testers report to the contracted, or work for the contractor, it is crucial that the contractor and the contracted agree *a priori* (ideally via a contract taking the form of a testable requirements model [2]) on what constitutes successful validation.

In this paper, we do not focus on organizational or process aspects of validation[1]. Instead, we present a tool that does support the *automated* validation of the requirements of the contractor against the *actual behavior* of an IUT delivered by the contracted. This is achieved via the automatic generation of executable test cases from a testable IUT-independent model of the requirements.

Before introducing our proposal, it is important to remark that modeling tools are mainly used to create models of (some aspects of) an IUT. Such tools provide support for model specification, analysis, and maintenance. While requirements can be modeled within (e.g., as use cases in the UML), and even possibly semantically linked to other models supported by, such tools, the latter generally do not have the ability to generate test cases, let alone run and monitor them against an actual (even possibly generated) IUT. Consequently, such modeling tools are not relevant to this paper, as they do not tackle the specific kind of validation we address.

Similarly, a *test automation framework* does not support validation *per se*. Such a framework generally accepts executable test cases that *already* have been manually created, automatically generated, or pre-recorded. The automation framework then executes the test sequences without human interaction. With respect to validation, such frameworks present at least two specific problems: First, requirements are not captured anywhere (but instead embedded implicitly in test cases) and thus traceability between requirements and test cases is inexistent. Second, test cases in such

[1] We insist however that any “state-of-the-art development process” and “vendor management practice” that does not *explicitly* tackle the automated validation of the requirements of the contractor against the actual behavior of an IUT is, from our viewpoint, inadequate in the context of offshore outsourcing [2]. In other words, we place this very specific kind of validation at the heart of offshore outsourcing.

frameworks are generally limited to unit testing (i.e., testing of procedures [3]) and very simple scenario testing [3] (e.g., exercising a graphical user interface). Semantically, the expression of requirements for industrial software systems is far more complex [4]. Consequently, we will not discuss further such frameworks.

Also, we must emphasize that, in the context of this paper, testing notations *in vacuo* are of no interest. For example, TTCN-3 (the Tree and Tabular Combined Notation) [5] is an international standard that is widely used to capture functional test cases. But its syntax and semantics are of little importance here: TTCN is purely descriptive and offers no operational semantics *per se*. Similarly, the UML's use cases and interaction diagrams [3] are *modeling* notations used to capture scenarios. Capturing a scenario can be semantically challenging [6] yet remains a much simpler task than building a tool capable of setting up this scenario in an IUT, monitoring its multiple (possibly concurrent) executions at run time, and reporting on the outcomes of the later! In the same vein, we remark that the UML's Object Constraint Language (OCL) does not support semantically the notion of a scenario and that, consequently, the few tools capable of generating executable test cases from OCL statements generally have such test cases limited to the scope of a class[2].

Not surprisingly, given the previous observations, Grieskamp [4] from Microsoft Research, comments that current testing practices are not only laborious and expensive but often unsystematic, lacking methodology and adequate tool support.

Current approaches to validation and testing fall into two categories: code-centric and model-centric. A code-centric approach to validation and testing, such as Test-Driven Design (TDD) [7], typically uses *test cases* (as opposed to more abstract *tests* [3][3]) written at the implementation level in order to guide development. A TDD approach begins by creating a test case addressing one or more requirements of a software system being developed. The test case is executed against this system, usually resulting in a failure. Code is then added to the system until this test case succeeds. The process repeats until all relevant requirements have been satisfied, but there is little effort to scope and capture such requirements in a model! Test cases are purely implementation-driven and implementation-specific (as opposed to proceeding from the requirements of stakeholders): validation *per se* is not addressed [8].

Model-Based Testing (MBT) involves the derivation of tests from a model that describes some aspects of an IUT. More precisely, an MBT tool uses various algorithms and strategies to generate tests 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. Tests derived from such a model are functional ones expressed at the same level of abstraction as the model. Such tests can then grouped together to form an abstract test suite. Such an abstract test suite *cannot* be directly executed against an IUT because the tests are not at the same level of abstraction as the code. In general, such tools use a Finite State Machine (FSM) or FSM-like representation to create a behavioral model of the IUT. The model is then traversed

---

[2] In fact, in OCL, constraints can be associated not only with a class's attributes and operations, but also with use cases, interaction diagrams, etc. However, currently, test case generation from OCL statements is limited to class scope and requires that such statements be first embedded in a specific implementation (an approach quite far from the model-based derivation and execution of test cases we aim for).

[3] A test case can in fact be viewed as an *instantiation* of a test for a specific IUT [3].

according to some test 'purposes' [3] in order to generate test cases. Several MBT tools claim to offer test case generation and execution. But it is important to understand that such 'execution' typically consists in *symbolic execution*, that is, is carried out using a (typically state-based) *model* of the IUT, *not* the actual IUT. The latter (which is taken to be a black-box binary) is merely to be stimulated in order for it to generate the outputs that act as triggers for the transitions of this state-based model.

Most interestingly, a survey of existing MBT tools [8] reveals that there are significant semantic differences between the specification languages used in such tools. For example, one can contrast the "business-readable (i.e., stakeholder-friendly)" specification language of Cucumber [9], which is utterly simplistic, with the semantic richness of Spec# [10] and the subtle complexities of expressing intricate scenario interrelationships [6]. Indeed, in our opinion, Spec Explorer [11] constitutes the state-of-art in MBT tools because Spec#, its specification language, is able to capture a detailed model of an IUT's behavior, which can be exercised via the symbolic execution of user-specified scenarios. Spec# is a textual programming language that includes and extends C#. In addition to the functionality provided by C#, Spec# adds pre- and post-conditions [12], high-level data types, logical quantifiers (e.g., *forAll* and *Exists*), and a simple form of scenarios (which is generally absent in other state-based approaches). Thus, a Spec# model can be viewed as a program. But, as with most other existing MBT approaches, even in Spec Explorer requirements are *not* tested against an actual IUT, but rather against a *model* of the behavior of this IUT. More precisely, the *executability* of test cases is defined with respect to a state-based model and is rooted in the concept of state exploration (which faces the difficult problem of state explosion [4, 11]).

Ultimately if existing MBT tools are to tackle the validation of the requirements of the stakeholder (i.e., the contractor) against the *actual behavior* of an IUT delivered by the contracted, then the generated test cases must be 'transformed' into an executable test suite that can run against an IUT. Such a transformation is performed via the use of 'glue code'. That is, test cases obtained from a model are subsequently coded, typically manually. The resulting code (which includes not only executable test cases but also test drivers and oracles [3]) must be *handcrafted* and may end up *not* corresponding to the model-derived test cases. Moreover, this glue code is implementation-specific: both its reusability across several IUTs and its maintainability are highly problematic [13]. In other words, the creation of glue code is a non-automated endeavor that is time-consuming and just as error-prone as the development of the original IUT. Consequently, in the context of offshore outsourcing, the use of glue code is likely to greatly hinder the ability to carry out validation in a systematic, ideally automated way.

In the rest of this paper, we overview the specification language and the fully implemented framework we have built to support the validation approach we believe is required by offshore outsourcing.

2 An Alternative Approach to MBT

2.1 Semantic Foundations

In order to validate the requirements of a stakeholder (the contractor) against the actual behavior of a candidate IUT submitted by the contracted, it is necessary to have a

specification language from which test cases can be generated *and* executed on an actual IUT (as opposed to a model of the latter). This position statement leads to three questions that we will address in the rest of this section: First, what are to be the semantics of this specification language? Second, how is traceability between a model expressed in this language and an IUT to be specified and maintained? And, third, how are test cases to be generated and executed against an IUT?

With respect to semantics, Spec# [10] aims at reducing the gap between the specification language and an implementation language in order to improve correspondence between model and IUT. While Spec# can be easily picked up by someone with C# programming experience, in the context of validation, it is difficult for a stakeholder to understand it, as it resides at a low-level of abstraction. Yet it is desirable to offer an *IUT-independent* specification language that stakeholders can master. For offshore outsourcing, this is particularly true when considering that, in addition to the contractor, it is often the case that stakeholders will include the separate team of testers mentioned earlier. Much like the contractor, such testers want to treat the IUT as a black box whose behavior is to be used to validate the *IUT-independent* requirements.

On the topic of semantics, Grieskamp's [4] observes that industrial users of MBT approaches prefer scenario-based (as opposed to state-based) semantics for the specification language of an MBT tool. Consequently, we have rooted the semantics of our proposed approach to MBT in the notions of *responsibilities* and *scenarios* [6, 14]: scenarios are conceptualized as grammars of responsibilities [14]. Each responsibility represents a simple action or task. Intuitively, a responsibility is either to be bound to a procedure within an IUT, or the responsibility is to be decomposed into a sub-grammar of responsibilities (as illustrated later). In addition to responsibilities and scenarios, our semantics offer a set of *Design-by-Contract* [12] elements such as pre- and post- conditions, and invariants (which have been shown to be very useful for unit testing [15] and which will be illustrated in the next section).

Our proposed specification language (called ACL for Another Contract Language), which is to be used to model IUT-independent requirements, is not limited to functional dynamic testing (in contrast to AutoTest [15] and similar tools): it also supports static testing, as well as the evaluation of metrics (as explained shortly).

Most importantly, in contrast to existing MBT approaches and tools, we root validation not only in the generation of IUT-independent tests but also in the execution of the *corresponding* test cases on an *actual* IUT (not a model of it). In the context of offshore outsourcing, this decision is crucial. Let us elaborate. Outsourcing is often motivated by cost reduction considerations. The use of any specification language to obtain a requirements model represents a significant investment of time and money. This investment is leveraged if the requirements model is IUT-independent: only in this case will changes to the IUT *not* entail changes to the requirements model. (In fact, an IUT-independent model can be reused across a set of candidate IUTs.) For the same reason, we aim for IUT-independent tests (which are thus non-executable on an IUT). If symbolic execution is then used, the specific test cases generated from the IUT-independent tests will be executed on a behavioral model of the IUT. But an outsourcing contract [2] will *not* be concerned with validation against a behavioral model of an IUT, but rather against the actual behavior of an actual IUT! And without such validation, in our opinion, there is very little possibility of formalizing an outsourcing contract.

Insisting on the executability of test cases against an actual IUT requires that we now address how such test cases can be derived from IUT-independent tests, themselves generated from an IUT-independent requirements model. An answer requires a) a transformation from such generated tests to test cases executable on a particular IUT and b) the instrumentation of such executable test cases, that is, the addition to the IUT of run-time monitoring code in order to observe/evaluate the outcomes of the executable test cases. As previously mentioned, a manual transformation from IUT-independent tests to test cases executable on an IUT is undesirable: it is not only time-consuming but often error-prone, and the resulting executable test cases may not correspond to the tests generated from the requirements model. Thus, we advocate an automated transformation, as well as automatic instrumentation of the IUT. This has a crucial consequence for the semantics of ACL: our position is that *only* semantics that enable their automated transformation to IUT-executable test cases and automatic instrumentation are to be included in ACL.

The question then is: how does such transformation and instrumentation work? The answer requires that we introduce the validation framework (VF) we have developed.

2.2 Using the Validation Framework

Our VF [16] operates on three input elements. The first element is the Testable Requirements Model (hereafter TRM). This model is expressed in ACL, a high-level general-purpose requirements *contract* language. We use here the word 'contract' because a TRM is formed of a set of contracts, as will be illustrated in the next section. ACL is closely tied to requirements by defining constructs for the representation of scenarios, and design-by-contract constructs [12]. Additional domain-specific constructs can also be added to the ACL, via modules known as *plug-ins* (as explained elsewhere [16]). The second input element is the candidate IUT against which the TRM will be executed. Currently, this IUT is a .NET executable (for which we do not require the source code). *Bindings* represent the third and final input element required by the VF. Before a TRM can be executed, the types, responsibilities, and *observability* [3] requirements of the TRM must be bound to concrete implementation artifacts located within the IUT. A structural representation of the IUT is first obtained automatically. Our binding tool, which is part of the VF, uses this structural representation to map elements from the TRM to types and procedures defined within the candidate IUT. Most importantly, our binding tool is able to automatically infer most of the bindings required between a TRM and an IUT [13]. Such bindings are crucial for three reasons. First, they allow the TRM to be independent of implementation details, as specific type and procedure names used with the candidate IUT *do not* have to exist within the TRM. Second, because each IUT has its own bindings to a TRM, several candidate IUTs can be tested against a single TRM. Finally, bindings provide explicit traceability between a TRM and IUT.

Users of our VF can bind contract elements to procedures and types of the IUT manually, or use the Automated Binding Engine (ABE) we provide. ABE supports an open approach to the automation of binding creation: different algorithms for finding bindings are separately implemented in different *binding modules*. We have implemented two such binding modules as part of the current release of our VF [16]. The first binding module takes into account the names of types and procedures in order to

find matches, whereas the second module uses only structural information such as return type and parameter type/ordering to infer a binding. Each of our two implemented binding modules have correctly bound approximately 95% of the required bindings found in the five case studies we have developed so far (approx. 200 bindings) [16]. Missing bindings are specified manually.

ACL provides the user of our VF with built-in (and user-defined) static checks, dynamic checks, and metric evaluators. (The Container ACL model found in [16] provides a simple example that includes static and dynamic checks, as well as metric evaluators.) A static check performs a check on the structure of an IUT. Such check is accomplished without execution. Examples of currently supported static checks include checks involving inheritance (e.g., type A must be a descendant of type B), and checks on types (e.g., type A must contain a variable of type B). A static check can be viewed as an operation: each check has a return type and may accept a fixed number of parameters. All static checks are guaranteed to be side effect free. The point is that should a contractor have requirements pertaining to the structure of an IUT, our proposed tool is able to validate such requirements.

A dynamic check is used to perform a check on the IUT during execution. That is, a dynamic check can only be evaluated while the IUT is being executed. The evaluation of pre- and post-conditions, and of invariants constitutes one category of dynamic checks, particularly relevant to unit testing. Other examples of dynamic checks include: testing the value of a variable at a given point, ensuring a given state exists within an object (with respect to the values of that object's instance variables), and validating data sent between two different objects. As with static checks, dynamic checks can be viewed as an operation with a return type and parameter set. The execution of a dynamic check is also guaranteed to be side effect free.

Metric evaluators are used to analyze and report on the metrics gathered while the candidate IUT was executing. Metric gathering is automatically performed by the VF. Once metric gathering is complete and the IUT has concluded execution, the metric evaluators are invoked. Examples of a metric evaluator include: performance, space, and network use analysis. Metric evaluators are side effect free. Most importantly, they allow our proposed tool to tackle the validation of non-functional requirements, something that is essentially downplayed in most existing approaches and tools for MBT. In fact, to the best of our knowledge, very few, if any, MBT tool, except ours, addresses both (static and dynamic) functional *and* non-functional requirements (via metric evaluators) within the same specification language. As a matter of fact, ACL offers a semantic richness (static and dynamic testing, responsibilities and scenarios, metric evaluators) that enables the contractor to develop a requirements model (to be included in the outsourcing contract) that can address system-level behavior of an IUT, *as well as* scenario and unit testing, static analysis and non-functional requirements. We believe this is absolutely necessary if indeed ACL is to offer the semantic flexibility (i.e., high- *and* low-level semantics) that industrial outsourcing projects require [4].

Once the TRM has been specified and bound to a candidate IUT, the TRM is compiled. Upon a successful compilation, all elements of the TRM have been bound to IUT artifacts and any required plug-ins have been located and initialized. The result of such a compilation is a single file that contains all information required to execute the TRM against a candidate IUT. (Details lie beyond the scope of this paper.) The validation of

a Testable Requirements Model (TRM) begins with a structural analysis of the candidate IUT, and with execution of any static checks. Following execution of the static checks, the VF executes the IUT. The VF is able to track and record the execution paths generated by the IUT, as well as execute any dynamic checks, and gather metrics indicated by the TRM. The execution paths are used to determine if each scenario execution *matches* the grammar of responsibilities corresponding to it within the TRM (see next section). Next, metric evaluators are used to analyze and interpret any metric data that was gathered during execution of the IUT. All of the results generated from execution of the TRM against the candidate IUT are written to a Contract Evaluation Report (CER). The generation of the CER completes the process of executing a TRM against a candidate IUT. The CER indicates where the candidate IUT matches the TRM, and where any deviations from the TRM were observed. For example, when a pre- or post-condition fails, the execution proceeds but that failure is logged in the CER. Also, when a scenario is executed by an IUT, the specified grammar of responsibilities must hold in order for the scenario to be considered to have succeeded. That is, for success, the responsibilities that compose the scenario must be executed in an order that satisfies the grammar. If the scenario cannot be executed, or responsibilities/events that are not defined by the scenario are executed, then the scenario is deemed to have failed. This mismatch is also reported in the CER. Several quality control and analysis methods could then be used to analyze the generated CER and apply their findings to the software development process, or calculate information important to management and other stakeholders. Such methods currently lie beyond the scope of our work.

The key point of this overview is that once a TRM is bound to an IUT, all dynamic checks and metrics evaluators are automatically instrumented in the IUT whose execution is also controlled by the VF (e.g., in order to monitor scenario instance creation and matching). In order to discuss scenario testing, we must first illustrate the semantics of the language we use to specify a TRM, especially those features that pertain to unit and scenario testing.

3 A Partial Example

We now present excerpts of a medium-size case study that deals with how students register in their courses at a university and how they obtain grades for these courses. Our intent is not to motivate the contracts used, nor to explain at length how this example is processed. Instead, we aim at providing the reader with a substantial (i.e., non-trivial) example in order to a) illustrate the semantics of ACL, especially pertaining to scenarios and b) subsequently discuss the basics of unit and scenario testing in the VF. (The complete example is in [16] and is 49 pages long. Most importantly it does compile and run.)

We use the // and /* */ to provide comments directly in the example, as this facilitates presenting explanations in the ACL rather than after it.

The Course contract represents a single university course. A course is created by the university, and consists of a name, code, a list of prerequisites, a list of students currently enrolled in the course, and a size limit on the number of students that can take the course.

```
Namespace Examples.School{
Contract Course {
/* Once the contract Course is bound to a type of the IUT, each time an instance of
this type is created, a new instance of contract Course is associated with it. A contract
can be bound to several types.
    Parameters are values supplied to a contract. These values can be provided either
via other contracts (i.e., statically), or at binding time (in which case all instances of
the class bound to this contract will share the same parameter values). Finally,
parameter values can be left unspecified until run-time, using the keyword
InstanceBind, in which case, each time an instance of a class bound to this contract is
created, the VF will prompt the user for all required parameter values. These three
options are extremely important in dealing with test case generation, as explained
later. */

Parameters
{ Scalar Boolean EnforcePreRequisites = { true, default false };
  [0-2] Scalar Integer InstanceBind NumMidterms = 1;
  [0-5] Scalar Integer InstanceBind NumAssignments = 1;
 Scalar Boolean InstanceBind HasFinal = { default true, false };          }

/* An observability is a query-method that is used to provide state information about
the IUT to the TRM. That is, they are read-only methods that acquire and return a
value stored by the IUT. */
Observability String Name();
Observability Integer Code();
/* The VF supports scalars (one value) and Lists (set of values).
Other observabilities skipped here include a list of students, a cap size, a list of
prerequisites, the weight for assignments, etc. We now look at some more of these: */

//example of an observability with a parameter
Observability Integer  MarkForStudent(tStudent student);
// example of statically setting a parameter
Observability Boolean HasFinal()   { Parameters.HasFinal == true; }

/* The responsibilities new and finalize are special:
The body of the new responsibility is executed immediately following the creation of
a new contract instance.    It only can use post-conditions       */
Responsibility new()
{  Post(Name() not= null); Post(Code() not= 0);
   Post(Students().Length() == 0); Post(TotalMarks() == 100); }
/* The body of the finalize responsibility is executed immediately before the
destruction of the current contract instance.    */
Responsibility finalize()  { Pre(Students().Length() == 0);  }

/* Invariants provide a way to specify a set of checks that are to be executed before
and after the execution of all bound responsibilities. Invariants precede pre-conditions
and follow post-conditions.          */
Invariant IsFullCheck {  Students().Length() <= CapSize(); }
```

```
/* A stub is a choice point where one of several possible responsibilities is selected
based on some criterion. Here, we use the value of a parameter to choose.  */
stub Responsibility AddStudent(tStudent s)
{ [Default] AddStudentNoPreReqCheck(s);
  [Parameters.EnforcePreRequisites == true]  AddStudentPreReqCheck(s);    }

/* Here is a simple responsibility, which will be used if prerequisites are set to not be
enforced (via the parameter). It just checks that the student is not already in the
course.  The keyword Execute indicates where execution occurs. */
Responsibility AddStudentNoPreReqCheck(tStudent s)
{  Pre(Students().Contains(s) == false);  Execute();
   Post(Students().Contains(s) == true);   }
/* Other responsibilities include AddStudentPreReqCheck and RemoveStudent. */

// a scenario defines a grammar of responsibilities
Scenario ReportMarks
{  Trigger(observe(TermEnded)),     //it must be triggered by an event (a symbol)
/* Each time a scenario's trigger is satisfied,  it creates a new scenario instance (of
itself). Instances of contracts can be referenced by other contracts.
Semantically, this is a powerful mechanism of our VF.  Here, the example is simple
because there's only 1 instance of the contract for the university.  A complex
example, which involves the keyword bindpoint will be presented later in this section.
The keyword once states that the variable does not change once its value is set. */
once Scalar Contract University u = instance;

/* The body of the each statement contains a single grammar element.  It captures that
the fact that ReportMark responsibility (defined in the university contract instance u)
must be invoked using our course and the current student (denoted by the iterator
keyword) and that the mark reported by the MarkForStudent() observability method
is provided.  The purpose of this element is to ensure that the correct mark for the
given student is recorded.
each(Students()) {u.ReportMark(context, iterator, MarkForStudent(iterator))},
/* The ',' operator is the 'follows' operator above.

The scenario is completed with a termination condition.  A scenario's termination
condition is used to specify when the scenario is complete.    */
Terminate(fire(MarksRecorded));     }
/* The Exports section defines the IUT binding points required for the contract, and
also includes any binding constraints.  Our Course contract only requires a single
binding point: tStudent.  The binding point begins with the Type keyword to denote
that the binding of the tStudent symbol is to be made against a type within the IUT.
    The conforms keyword is used to indicate that the IUT type that the tStudent sym-
bol is bound to will automatically have the Student contract applied to it. The tStu-
dent binding point contains a single binding rule that states that the IUT type that is
bound to the tStudent symbol must be the same type that is bound to the tStudent
symbol found within the University contract.  */
```

```
Exports
{ Type tStudent conforms Student { University::tStudent; }
}}} //end of Course contract

/*The ProjectCourse contract represents a refinement of the Course contract. It is
used for courses that also include a course project.*/
Namespace Examples.School
{ Contract ProjectCourse extends Course {
/* contracts can be organized hierarchically using the extends keyword. Usual rules
for how to deal with pre- and post-conditions [12] apply. */
Parameters
{ Scalar Boolean InstanceBind HasProject = { default true, false };
//the rest of the contract is not relevant to this paper
} //end of ProjectCourse contract

/* The Student contract represents an individual student enrolled at a university. */
Namespace Examples.School {
Contract Student {
Observability List Integer CompletedCourses();
/* scalar (failures) and observabilities (CurrentCourses, StudentNumber, Name,
IsFullTime ) are then defined, as well as responsibilities: SelectCourse (which
returns a course), RegisterCourse , DropCourse, RegisterCourse, DoAssignment,
DoMidterm, DoProject and DoFinal.
We now look at stub DoAssignment. The rationale for using a stub is that is if a stu-
dent is not taking a course that has any assignments, then the student will not require a
corresponding responsibility (which is declared after the definition of the stub of the
same name). The body of the DoAssignment responsibility stub begins with the ex-
traction of the Course contract instance from the provided IUT instance representing
the given course that we are doing the assignment for. This is achieved using the
keyword bindpoint. Note that if the resultant contract type does not match the de-
clared type, a run-time error will be issued.
    It is important to note that stubs are not bound but instead reduced to a responsibil-
ity (which is then bound). So there is no ambiguity between the stub and the responsi-
bility DoAssignment, despite using the same name.*/
stub Responsibility DoAssignment(tCourse c)
{ Contract Course course = c.bindpoint;
  [course.Parameters.NumAssignments > 0] DoAssignment(c);   }
Responsibility DoAssignment(tCourse c);

/* The DoProject() responsibility is an example of a responsibility that is not bound to
a corresponding IUT procedure, but rather is specified as a grammar of other respon-
sibilities and events. This illustrates another use of events, beyond triggers and termi-
nations of scenarios. Events exist at run-time in the space of execution of the TRM, as
opposed to the one of the IUT. The run-time space of the TRM maintains all relevant
contract information, including contract and scenario instances.*/
Responsibility DoProject(tCourse c)
{       FormATeam(c), observe(TeamFinalized), WorkOnProject(c);     }
```

```
/* We include the two scenarios of this contract to illustrate the semantic complexity
our VF can currently handle. The keyword atomic defines a grammar of
responsibilities such that no other responsibilities of this contract instance are allowed
to execute except the ones specified within the grammar. Due to space limitations, we
leave it to the reader to either figure out the details or read them elsewhere [16]. */
Scenario RegisterForCourses {
Scalar tCourse course;      Contract University u = instance;
Trigger(observe(CoursesCreated), IsCreated()),
/*triggered IF courses have been created and the student is also created (isCreated is a
responsibility in this contract) */
 choice(IsFullTime()) true
  {  ( atomic
     { course = SelectCourse(u.Courses()),
       //via bindpoint get the instance for the selected course
       choice(course.bindpoint.IsFull())  true
         {  course = SelectCourse(u.Courses()), redo}  }, //until a course is not full
   // keyword context refers to the current contract instance
     u.RegisterStudentForCourse(context, course),
     RegisterCourse(course)
    )  [0-u.Parameters.MaxCoursesForFTStudents] //repeat atomic
  }
   alternative(false)  //student is part-time (PT)
   { ( atomic
       {  course = SelectCourse(u.Courses()),
          choice(course.bindpoint.IsFull())  true
           { course = SelectCourse(u.Courses()), redo   }},
       u.RegisterStudentForCourse(context, course),
       RegisterCourse(course)
       )[0-u.Parameters.MaxCoursesForPTStudents] },
Terminate();  } //end of scenario RegisterForCourses

Scenario TakeCourses {
failures = 0; //number of failures in the current term
Trigger(observe(TermStarted)),
parallel
{ // for all courses of that term
 Contract Course course = instance;
  //if and only if that course is one taken by this student
  Check(CurrentCourses().Contains(course.bindpoint));
  atomic
   { (parallel//can do assignmnts, midterm, proj concurrently
       { (DoAssignment(course.bindpoint))
         [course.Parameters.NumAssignments]  }
      |  //use OR, not AND, to avoid ordering
         (DoMidterm(course.bindpoint))
          [course.Parameters.NumMidterms]
      |
```

```
            (DoProject(course.bindpoint))
            [course sameas ProjectCourse &&
                course.Parameters.HasProject]
        ),
        (DoFinal(course.bindpoint)
          [course.Parameters.HasFinal]
          }
      alternative( not observe(LastDayToDrop))
        { DropCourse(course.bindpoint) }
}[CurrentCourses().Length()];
Terminate();  } //end of scenario TakeCourses

/* We omit most of the University contract, which does not add to this presentation.
We mostly focus below on its main scenario, which pertains to modeling a term. */

MainContract University
 {Parameters
    { [1-100] Scalar Integer InstanceBind UniversityCourses;
      Scalar Integer MaxCoursesForFTStudents = 4;
      Scalar Integer MaxCoursesForPTStudents = 2;
      Scalar Integer PassRate = 70;   }
/* several observabilities, responsibilities and some scenarios are omitted.  */

Responsibility ReportMark     (tCourse course, tStudent student, Integer mark)
/* The course, the student and the mark to be recorded are provided as parameters.
Each parameter of the responsibility is bound to a parameter of the procedure bound
to this responsibility. */
   {  choice(mark) < Parameters.PassRate   // the number of failures is recorded
          {  student.bindpoint.failures = student.bindpoint.failures + 1;  }  }
Scenario CreateCourses
    { Trigger(new()),
/* all courses of the term, in Parameters.UniversityCourses, must be created */
      CreateCourse(dontcare, dontcare) [Parameters.UniversityCourses],
       Terminate(fire(CoursesCreated));   }
Scenario CreateStudents
    { Trigger(new()), CreateStudent(dontcare)+, Terminate(finalize());   }

Scenario Term
/* term management via doing responsibilities in a particular order and firing the
corresponding events  */
    { Trigger(new()),
     ( CreateCourse()[Parameters.UniversityCourses],
       TermStarted(), fire(TermStarted), LastDayToDrop(), fire(LastDayToDrop),
       TermEnded(), fire(TermEnded), observe(MarksRecorded)
                [Parameters.UniversityCourses],
        CalculatePassFail(), DestroyCourse()[Parameters.UniversityCourses],
```

```
        fire(TermComplete)
      )+,
     Terminate(finalize());
 }
/* We conclude with an example of inter-scenario relationships [6]. Whereas a scenario instance is tied to a contract instance (its 'owner'), relations are not. This complicates their monitoring: each time a scenario instance is created or terminates, all relations referring to this scenario must be verified for compliance. */
Interaction School
{ Relation Creation // creation of students and courses can be in any order
   { Contract University u; (u.CreateStudents || u.CreateCourses); }
  Relation Cancelling
/* for any course created by the university, cancelling it is optional until the term starts. The keyword dontcare is used to ignore the name and code of the courses.
The use of dontcare simplifies test case generation!! */
   { Contract University u; Instance c; //The VF figures out c is of type tCourse
     c = u.CreateCourse(dontcare, dontcare), (u.CancelCourse(c))?,
     observe(TermStarted); } }   //other relations are omitted
} of Interaction School
```

4 Scenario Testing

In this section, we mainly focus on how scenarios expressed in ACL can be validated against the actual behavior of an IUT. For simplicity, hereafter, the term 'scenario' will refer to all forms of grammars of responsibilities in ACL (namely, responsibilities such as ***DoProject***, as well as scenarios and relations).

First, as explained earlier, the TRM will be bound to an IUT and compiled. Its dynamic checks (and metric evaluators) will be automatically instrumented in the selected IUT. Then testing proper begins. Static checks will be verified and their outcomes logged. Then the VF runs the IUT. Recall that the VF is able to track and record the execution paths generated by the IUT, as well as execute any dynamic checks, and gather metrics indicated by the TRM. Such execution paths are automatically *matched* against the scenarios of the TRM. Conceptually this task is not complex. Its implementation, however, is not trivial: the VF must monitor/stop/restart the execution of procedures of the IUT, as well as keep track of all contract and scenario instance creation and termination, and all events. (Details are given in [16].)

Second, the VF must tackle test case generation. The simplest strategy consists in *not* generating any tests from the TRM and instead leaving the creator of the IUT to supply a test suite. In this case, the TRM captures what is valid and what is not, and the VF provides automatic instrumentation, as well as run-time monitoring and logging of scenario satisfaction or failure. With this strategy, it is entirely left to the IUT-provider to code a test suite that defines a space of executions of the IUT in which these contracts/scenarios are validated. The difficulty with this approach is that the issue of *coverage* [3] of the TRM is completely hidden from the creators of the TRM (i.e., stakeholders such as the contractor and/or the testers), which is problematic! In our opinion, stakeholders must have a say in 'how much' a TRM is to be tested [2].

Coverage at the level of unit testing is considerably simpler than for scenarios. It rests on the use of well-known combinatorial techniques (chapter 6 in [3]) to address, for example, the coverage of the Boolean clauses of pre- and post-conditions. Also, as with AutoTest [15], instances can be automatically created with random states (that must satisfy the relevant invariants). Alternatively, the variables needed to (possibly partially) set the state of an instance under test (e.g., whether or not a student is full-time) can be defined statically, or when binding the TRM to an IUT, or at run-time (e.g., via the use of the *InstanceBind* keyword).

For scenario testing, the key idea is that scenarios are *grammars* and that, as with state machines, there are well-known algorithms to obtain a selection of paths through a grammar according to some coverage criterion [2]. Indeed, this idea has already been used in several distinct approaches to test case generation from scenarios [e.g., 17, 18]. Here, we only need to know that a test case consists of a specific path through a scenario, as well as the data (called *path sensitization data* [2, 3]) that enables this path to be taken.

Consider, for example, scenario ***RegisterForCourses*** in our example. A test case **TC1** through this scenario could be that a part-time student selects a first course which is full, then selects two that are not (and thus, for which registration proceeds). The question then is: how can this test case be generated *and* executed by our VF?

With respect to test case generation, the process starts by having the VF produce an internal representation of each scenario to test as a control flow graph [3]. Then, for each such graph, the user of the VF selects a coverage criterion. At this point in time, the VF offers "all-statements" and "all-branches" coverage [3, 18]. Each coverage criterion is associated with a traversal algorithm that generates a test suite sufficient to cover the graph at hand. Binder [3] discusses at length such algorithms (e.g., how conditional statements and loops are covered), which are also at the basis of similar approaches (e.g., [17, 18]). From this viewpoint, a test case can be thought of as a particular path of execution through a scenario. For example, for **TC1**:

```
{   (observe (CoursesCreated) == true), (context.IsCreated() == true),
    (context.IsFullTime() == false) ,
    (assign (course)) , (course.bindpoint.IsFull() == true)
    (assign (course)), (course.bindpoint.IsFull() == false)
    u.RegisterStudentForCourse(context, course),
    RegisterCourse(course),
    (assign (course)), (course.bindpoint.IsFull() == false)
    u.RegisterStudentForCourse(context, course),
    RegisterCourse(course),          Terminate() }
```

This path corresponds to the following specific branching decisions: the event *CoursesCreated* has been fired, the contract instance at hand (i.e., the *context*, which is a student) is created (i.e., student number is not 0) and is not full time, the first course attempted is full, the second is not, the atomic block is to execute twice, the third course attempted is not full.

It is crucial to understand that such a path is *not* executable: we need actual instances (i.e., a student, a university, and three courses). And such instances must typically be *set* to states that enable the execution this specific test case (instead of

'ending up' in a specific state via long set ups). For example, it is far more efficient to set a course to be full (via the relevant instance variable) than to create a number of students corresponding to this course's maximum size and then to add each of these students to the course!

Consequently, we adopt a representation of a test case that requires the VF to invoke the binding tool when required. Consider, for example for **TC1**, the following:

{ Create(context), Create(u), fire(CoursesCreated), Set(context,IsCreated(), true), Set(context,IsFullTime(), false), Set(course,IsFull(), true), Set(course,IsFull(), false), Set(course,IsFull(), false) }

The *Create* operation requires that the VF instantiate the required sole argument and then open the binding tool on it in order for the user to set all relevant instance variables. Events can be directly fired (i.e., no need to go through university in our example to fire *CoursesCreated*) in order to force a particular path of execution. Finally, the *Set* operation opens a binding tool on its first argument which must be an existing instance. The user then sets one or more variables of that instance. Then the second argument is called and its return value is compared to the third argument. The binding tool does not close before these two values match, ensuring that the selected branching is indeed executed.

In summary, in our VF, test case execution is semi-automatic: it requires that, during execution the binding tool be used to set some of the variables of the relevant instances in order to execute the selected path(s) through a scenario. It is left to the user to know which variables to set (e.g., student number must not be zero for *IsCreated* to return true). More sophisticated, less cumbersome approaches to test case generation would require that the VF carry out a deeper analysis of a scenario and of the responsibilities it refers to in order to determine which specific variables of which specific instances determine this scenario's flow of control. Such an analysis is currently beyond the scope of our work.

Finally, we emphasize that a fully automated purely static approach to test case generation and execution appears to be very difficult to achieve even using parameters. For example, the number of courses does vary from one test case to another for scenario ***RegisterForCourses***…

5 Conclusion

In this paper, we presented a tool that supports the *automated* validation of the requirements of the contractor against the *actual behavior* of an IUT delivered by the contracted. This is achieved via the automatic generation of executable test cases from a testable IUT-independent model of the requirements. A fair amount of experimentation and testing has been carried out on the VF. First, a comprehensive suite was used for the ACL compiler. Then, five extensive case studies [16] were developed to verify the handling of static and dynamic checks, as well as scenario monitoring and metrics evaluation. Finally, the tool was used in a graduate course one of the authors. The feedback we received stated clearly that, as with Spec#, ACL 'feels like' a programming language and thus may be of limited interest to the contractor and the testers alike. While we believe ACL's semantic richness is required to deal with industrial

(i.e., non-toy) examples, the question is: what can be done? At this point in time, we believe the answer lies in understanding Meyer's seminal idea of programs that test themselves: the more errors will be automatically detectable within an IUT through the use of a programming language and tools such as AutoTest [15], the more the semantics of ACL and, in turn, requirements models captured using ACL will be simplified. Most importantly, such simplifications should also ultimately lead to a simpler if not more automated creation of bindings and of executable test cases.

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

References

1. Meyer, B.: The Unspoken Revolution in Software Engineering. IEEE Computer 39(1), 121–123 (2006)
2. Corriveau, J.-P.: Testable Requirements for Offshore Outsourcing. In: Meyer, B., Joseph, M. (eds.) SEAFOOD 2007. LNCS, vol. 4716, pp. 27–43. Springer, Heidelberg (2007)
3. Binder, R.: Testing Object-Oriented Systems. Addison-Wesley Professional, Reading (2000)
4. Grieskamp, W.: Multi-Paradigmatic Model-Based Testing. Technical Report #MSR-TR-2006-111, Microsoft Research (2006)
5. International Telecommunications Union: The Evolution of TTCN, `http://www.itu.int/ITU-T/studygroups/com17/ttcn.html`
6. Ryser, J., Glinz, M.: SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test. Technical Report. University of Zurich (2003)
7. Beck, K.: Test-Driven Development: By Example. Addison-Wesley, Reading (2002)
8. Arnold, D., Corriveau, J.-P., Shi, W.: Validation against Actual Behavior: Still a Challenge for Testing Tools. In: Software Engineering Research and Practice (SERP). CSREA Press (July 2010)
9. Cucumber, `http://cukes.info/`
10. Microsoft Research: Spec# Tool, `http://research.microsoft.com/specsharp`
11. Veanes, M., Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N.: Model-based testing of object-oriented reactive systems with Spec Explorer, Tech. Rep. MSR-TR-2005-59, Microsoft Research (2005)
12. Meyer, B.: Design by Contract. IEEE Computer 25(10), 40–51 (1992)
13. Arnold, D., Corriveau, J.-P., Shi, W.: Modeling and Validating Requirements using Executable Contracts and Scenarios. In: Software Engineering Research, Management & Applications (SERA 2010). Springer, Heidelberg (May 2010)
14. Buhr, R.J.A., Casselman, R.: Use Case Maps for Object Oriented Systems. Prentice Hall, New York (1995)
15. Meyer, B., Fiva, A., Ciupa, I., Leitner, A., Wei, Y., Stapf, E.: Programs that test themselves. IEEE Computer 42, 46–55 (2009)
16. Arnold, D., Corriveau, J.-P.: The Validation Framework and its examples, `http://vf.davearnold.ca/`
17. Briand, L., Labiche, Y.: A UML-Based Approach to System Testing. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 194–208. Springer, Heidelberg (2001)
18. Nebut, C., Fleury, F., Le Traon, Y., Jézéquel, J.M.: Automatic Test Generation: A Use Case Driven Approach. IEEE Transactions on Software Engineering 32, 140–155 (2006)