Generating Verifiable Test Scenarios

Conference Paper · January 2011

CITATION
1

READS
42

2 authors:

Jean-Pierre Corriveau
Carleton University
94 PUBLICATIONS **467** CITATIONS
SEE PROFILE

Wei Shi
Carleton University
97 PUBLICATIONS **577** CITATIONS
SEE PROFILE

Generating Verifiable Test Scenarios

J.-P. Corriveau[1] **and W. Shi**[2]
[1]Computer Science, Carleton University, Ottawa, Ontario, Canada
[2]Business and IT, UOIT, Oshawa, Ontario, Canada

Abstract - *Many approaches that address scenario testing do so using models semantically distant from an implementation under test (IUT). While test paths can be generated from such models, these paths typically do not include path sensitization data and are not testable against an actual execution of an IUT. In this paper, we explain how the Validation Framework (VF) we have developed models scenarios and parse these into test scenarios. The latter do include path sensitization data and are verified by the VF against an actual execution of an IUT.*

Keywords: validation, model-based testing, scenarios

1 Introduction

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). That is, there needs to be a systematic (ideally objective and automated) approach to the validation of the requirements of the stakeholder against the *actual behavior* of an IUT [1, 2]. Unfortunately, most often, there is no such systematic approach to validation. Quite on the contrary, in practice, testers mostly carry out only extensive *unit testing* [3]. In this paper, we are instead concerned with scenario testing.

Model-Based Testing (MBT) involves the derivation of test cases from a model (or set of models) that describes some of the aspects of an IUT. More precisely, an MBT method/tool uses various algorithms and strategies to generate tests from a behavioral model of an IUT. Scenarios [3, 4, 5, 6, 7] constitute one type of behavioral model (another being state machines [3]). 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 cases derived from such a model are functional ones expressed at the same level of abstraction as the model. Such test cases are grouped together to form an abstract test suite. Most importantly, such an abstract test suite *cannot* be directly validated against the execution of an IUT because the test cases are not at the same level of abstraction as the code. Indeed, several MBT tools claim to offer test generation and test execution. But it is important to understand that in the context of MBT, 'test execution' generally consists in *symbolic execution*, that is, is carried out using a (typically state-based) model of the IUT, *not* the actual IUT. For example, Spec Explorer [8, 9, 10] stands out as an industrial-strength MBT tool with a specification language (Spec #) rich enough to capture a detailed state-based model of an IUT's behavior, as well as user-defined scenarios. But, the latter are not taken as defining a grammar of valid and invalid sequences of procedures of an IUT. Instead, these scenarios are interpreted in terms of states of the model and are validated via their symbolic execution. In contrast, here we propose an approach to scenario testing that generates test scenarios that can be validated against the execution of an IUT. To do so, we first briefly summarize in the next section the Validation Framework (VF) we have introduced elsewhere [2, 11]. The key characteristic of that tool is that implementation-independent specifications are bound to actual types and procedures of an IUT in order to enable the validation of the specification model against the execution of an IUT. Examples of scenarios in the VF are presented in section 3 and their testing is discussed in section 4.

Before concluding this introduction we remark that tools meant to only capture models (e.g., IBM's Rational Rose[12]) are not relevant to this paper, as they are not meant to address scenario testing. The same observation holds for *test automation frameworks* (e.g., IBM's Rational Robot [13]) in which there is no test generation from models (for the simple reason that such frameworks are not model-based). In the same vein, code-based testing tools (such JAVA's JUnit [14] and AutoTest [15]) mostly allow for *unit* tests (i.e., tests pertaining to a procedure, as opposed to tests addressing scenarios) to be specified in, or automatically generated from, code. Unit tests are semantically much simpler than scenario tests. Furthermore, such unit tests are implementation-specific and difficult (nay impossible) to trace back to the requirements of stakeholders (which are typically implementation-independent). For these reasons, code-based testing tools will not be discussed further here.

2 VF: An Alternative for MBT

In order to validate the requirements of a stakeholder against the actual behavior of an IUT, it is necessary to have a specification language from which tests can be generated and executed 'against' an actual IUT (as opposed to a model of the latter). We have described such an approach and its corresponding tool, the VF, at length elsewhere [2, 11, 16]. Our VF 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 [17] such as pre and post-conditions, and invariants. The second input element is the candidate IUT against which the TRM will be executed. 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* requirements of the TRM (see next section) 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. In particular, our binding tool is able to automatically infer most of the bindings required between a TRM and an IUT [11, 16]. 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.

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. 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 and are available at [11].) The validation of a TRM begins with a structural analysis of the candidate IUT, and with the execution of any *static checks* (e.g., a type inherits from another). Following execution of the static checks, the VF starts and monitors the execution of 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 user-specified metrics [11] 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 example). 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.

The key point of this overview is that once a TRM is bound to an IUT, all checks are automatically instrumented in the IUT whose execution is also controlled by the VF. As explained above, this enables verifying that actual sequences of procedures occurring during an execution of an IUT 'obey' the grammar of valid sequences defined in ACL scenarios. As we will illustrate in section IV, scenario testing proper goes beyond this sort of validation.

3 An Example

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 scenario testing. Consequently, 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 scenario testing in the VF. (The complete example is available at [11] and is 49 pages long. Most importantly it does compile and run, as is the case with all other examples in [11]. The reader may verify this claim by downloading the tool and trying it out!) This example is at a level of abstraction similar to models specified in Spec# [8, 9], and appears to most readers to be surprisingly low (in fact akin to programming). We remark that the level of abstraction of specification languages for MBT varies considerably [10]. In particular, we must emphasize that Spec# [9] (which supports Spec Explorer [8], the only industrial-strength MBT tool we know) works at a low level of abstraction (similar to that of ACL) because it deals with real systems, not toy examples. For such systems, semantic expressiveness and scalability are essential.

In the following example, we use the // and /* */ to provide comments in context, as this facilitates the presentation of ACL's features. Our example starts with the Course contract, which 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 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. */

Parameters
{ Scalar Boolean InstanceBind HasFinal =
   { default true, false };
// other parameters have been omitted
}
/* 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 Integer
MarkForStudent(tStudent student);
Invariant IsFullCheck
{ Students().Length() <= CapSize(); }
/* Here is a simple responsibility, which will be used if
prerequisites are set to not be enforced. 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,
RemoveStudent, new, finalize...  see [2, 11] */
// a scenario defines a grammar of responsibilities
Scenario ReportMarks
{  Trigger(observe(TermEnded)),
once Scalar Contract University u = instance;
each(Students())
//built-in variable iterator accesses students one at a time
{u.ReportMark(context, iterator, MarkForStudent(iterator))},
Terminate(fire(MarksRecorded));     }
Exports
{ Type tStudent conforms Student { University::tStudent; }
}}} //end of Course contract

/* The Student contract represents an individual student
enrolled at a university who is able to take courses.  */
Namespace Examples.School {
Contract Student  {
//lots of observabilities and responsibilities are omitted.
/* 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. We leave it to the reader to either figure out the
details or read them elsewhere [11]. */
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 up to the max # of courses for a full time
  }
   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. */
MainContract University
 { Parameters {
[1-100] Scalar Integer InstanceBind UniversityCourses; }
```

```
/* several parameters, observabilities, responsibilities and
some scenarios were omitted. */
Observability List tCourse Courses();
Observability List tStudent Students();
Responsibility ReportMark
/* 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. */
   (tCourse course, tStudent student, Integer mark)
// the number of failures is recorded
 {  choice(mark) < Parameters.PassRate
       {  student.bindpoint.failures =
            student.bindpoint.failures + 1;  }  }
Responsibility CalculatePassFail()  {
 each(Students())
    choice(iterator.bindpoint.failures) >= 2
   FailStudent(iterator);
   alternative
   PassStudent(iterator);
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());
 }
```

4 Scenario Testing with the VF

In ACL, scenarios are expressed as *grammars of responsibilities*, much like in Use Cases [5] and Use Case Maps [7]. As with any sort of grammar, there are well-known algorithms (e.g., [18, 19, 20]) to obtain a selection of paths through a grammar of responsibilities according to some *coverage criterion* [3]. In essence, a scenario (or responsibility) is parsed and transformed into a form of control flow graph [*Ibid.*] from which paths are easily extractable. However, such paths are not executable (and thus are often referred to as *test purposes*). As in several other methods, the VF currently supports the 'all branches' coverage criterion. Consider, for example:

```
Scenario X
{   Trigger (observe(eventA)) ,
    choice(failures) >= 2     responsibilityA();
      alternative             responsibilityB();
 Terminate (fire(eventB));    }
```

Here, two branches need to be covered: one for 2 or more failures, one for less than 2 failures. In order to control this branching, a *path sensitization variable* (PSV) is required leading to two test cases: one for which the value of the PSV is set to 2 or more, another with a value less than 2. Currently, this form of *equivalence partitioning* [3] requires the user to set the actual PSV value used for each test case (for it cannot be inferred by the tool whether, for example, -1 or 0 are valid or not, and what is the maximum valid value for this PSV). It must also be pointed out that only valid PSV values are relevant for testing a scenario, as will be explained shortly.

As Binder explains at length [3], coverage of loops requires that they be flattened (i.e., 'unrolled'). So scenario **ReportMarks** in the *Course* contract will require a PSV to control the generation of the different paths associated with this loop. This PSV corresponds to the number of students in the course at hand. Currently, using the VF, minimally two test cases are generated: one for the minimum number of iterations and one for the maximum. If possible, a third test case for a number of iterations between this minimum and maximum is also generated. Thus, for the loop in **ReportMarks**, three test cases will be generated: one for a course with a minimum number of students, one for a course with the maximum number of students this course allows (i.e., its capsize), and one for a course with a number of students between this minimum and maximum.

Transforming a scenario into a graph from which paths can be extracted is not necessarily trivial as studying scenarios **RegisterForCourses** and **TakeCourses** in contract *Student* should make clear: combining branching statements (e.g., choice/alternative) with one another and with loop statements (e.g., redo and [] blocks) leads to complex control structures. The introduction of possible concurrent paths (through the *parallel* statement) further complicates this task. But, as previously mentioned, path generation is a well-understood process [18, 19, 20] for which we merely reuse existing solutions (by adapting them to the syntax and semantics of ACL). Conversely, the identification of PSVs requires an ACL-specific solution, which can be discussed only after we first understand what the VF offers in terms of support for scenario testing.

As hinted in section 2, the primary role of the VF is to monitor the execution of an IUT and report on violations of static and dynamic checks (such as violations of pre- and post-conditions of responsibilities, of invariants of contracts, and of grammars of scenarios). Focusing specifically in this paper on scenarios, we remark that an instance of a contract is created each time an instance of the type to which the contract has been bound is made. So, for example, each instance of a course created during the execution of the IUT is monitored by a corresponding instance of the *Course* contract. And each contract instance creates an instance of one of its scenarios once this scenario is triggered. So, for example, once the term ends, each course contract instance will create its scenario instance for scenario **ReportMarks** in order to monitor the grammar of that scenario for that specific course. A scenario violation will occur, for example, in any course for which its scenario instance for scenario **ReportMarks** fails to observe the responsibility *ReportMark*() of the university being called for the exact number of students in the course at hand. A scenario violation will also be recorded if the execution of the IUT terminates without a scenario having its *Terminate* condition satisfied.

The more complex the semantics of a scenario, the more complex its grammar and the more numerous its sources of violations. Consider, for example, scenario **TakeCourses** in contract *Student*. The key observation is that there is a single instance of this scenario for each student. Thus, this single scenario instance addresses the completion of *all* the courses taken by this student in that term. To do so, it verifies (amongst other checks) that the exact number of assignments for course *c* is performed (by tracking how many times the *DoAssignment*() responsibility is invoked with *c* as parameter). Should the student not complete the required number of assignments (or midterms, etc.) in any of her courses during the term at hand, then the VF will report a violation for scenario **TakeCourses**.

Most importantly, it is crucial to understand that, because responsibilities found in ACL contracts are bound to actual procedures of an IUT, scenario validation using the VF ensures that actual sequences of procedure calls (monitored during the execution of an IUT) 'obey' the grammar of the scenario(s) relevant to these procedures. With respect to scenario testing, this modus operandi of the VF defines what is *observable* [3]. But scenario testing requires that we address not only observability but also controllability [*Ibid.*]. To do so, let us return to scenario **ReportMarks**. As previously mentioned, testing this scenario involves one PSV for the flattening of the *each* statement. **ReportMarks also** invokes responsibility *ReportMark*() in the *University* contract. This further complicates PSV identification, as discussed at the end of this section.

For now, the immediate question is how test cases generated for a scenario are to be executed. ACL is an implementation-independent specification language and thus executable code cannot be generated from it. However, because ACL contracts are bound to classes and ACL observabilities and responsibilities to procedures, ACL scenarios can be used to monitor the correct execution of specific test cases. Let us elaborate on this by continuing our discussion of the testing of scenario **ReportMarks**.

In the context of testing the example university system, the execution of a corresponding IUT will involve the execution of a test suite, that is, of a set of test cases. The task of creating this test suite lies with the developer/tester: the role of the VF is to generate what we call *test scenarios* that address the coverage of the ACL contracts (not IUT code!) modeling the university system. Specifically, for scenario **ReportMarks**, this involves the following steps:
1) this scenario is selected in the testing window of the VF. (The user chooses which scenarios to test in a particular execution of the IUT.)
2) the scenario is parsed and the user is asked to input a name (e.g., *numberOfStudents*) for the PSV required to flatten the loop of the scenario (and for other identified PSVs, if any).
3) the user is prompted to input a minimum and a maximum default value for *numberOfStudents*.
4) the user may flag the observability MarkForStudent as a 'IUTProvided', in which case it is understood marks for students will be supplied via the execution of the IUT. Alternatively, the user may flag this observability as 'toInput', in which case each mark will have to be input by the tester at run-time. (This alternative is useful when wanting to avoid writing a multitude of similar test cases differing only with respect to these marks.)
5) From the information above, the VF generates a test scenario for the minimum value of *numberOfStudents* and another for its maximum value. If an in-between value is possible, the VF generates a third tes scenario for this value. (Examples of such test scenarios are discussed shortly.)
6) As the IUT executes, beyond the monitoring offered by scenario **ReportMarks**, the VF also monitors the generated test scenarios. A test scenario is 'covered' if it is triggered and terminates correctly. That is, whereas scenarios are monitored for violations, in the case of test scenarios, it is their occurrence at least once during the execution of an IUT that is reported to the user. (Thus, once the occurrence of a test scenario has been detected, the VF prevents this test scenario from being triggered again.)
7) At the end of the execution of an IUT, the VF reports on the occurrence or absence of each test scenario. The user can assess how much coverage of each one the scenarios has been achieved and add more test cases to the IUT where need be (in order to have more test scenarios covered).

The nature of a test scenario is best understood through a simple example. Consider the case for which scenario **ReportMarks** is to be tested with a minimum number of students. The generated test scenario is:

```
TestScenario ReportMarks-1
{  Trigger(observe(TermEnded)),
once Scalar Contract University u = instance;
```

```
Check(Students.Length() == numberOfStudents.Min());
Terminate(fire(MarksRecorded));     }
```

This test scenario will have been covered once any empty course completes. Now consider the case for which scenario **ReportMarks** is to be tested with a maximum number of students. The self-explanatory generated test scenario is:

```
TestScenario ReportMarks-2
{  Trigger(observe(TermEnded)),
once Scalar Contract University u = instance;
Check(Students.Length() == numberOfStudents.Max());
Terminate(fire(MarksRecorded));
```

Should the tester want to use a maximum number of students specific to each course, then the appropriate Check statement would be:

```
Check(Students.Length() == context.CapSize());
```

While these simple examples summarize the role of generated test scenarios with respect to scenario coverage, they do not convey the complexity of i) PSV identification and ii) generation in more complex scenarios. We discuss these issues next.

Consider scenario **RegisterForCourses** in contract *Student*. Whether a student is full-time or part-time leads to two different *sets* of generated test scenarios. For each of these two sets, there are several other PSVs:

- the number of courses successfully registered in (required to flatten out the statement [0-u.Parameters.MaxCoursesForFTStudents] for full-time student (or its equivalent for part-timers).

Here the minimum and maximum values are explicitly captured and need not be asked from the user.

- the maximum number of iterations of the redo statement, that is how many courses that are full can be selected before one non-full one is found. Because of the semantics of the redo, the minimum is implicit: if the first course selected is available, the redo does not execute.

Given these three PSVs, the VF will generate test scenarios corresponding to different combinations of values for these PSVs. For example:

- a full-time (or part-time) student who does not attempt to register in any course
- a full-time (or part-time) student who registers in the maximum allowable number of courses for her status, all these courses being immediately available (i.e., non full and thus no redo is performed).
- a full-time (or part-time) student who registers in the maximum allowable number of courses for her status, each of these available courses being selected only after the maximum number of retries (i.e., full courses) has been attempted.

In summary, a scenario can have several PSVs associated with it and (via boundary analysis [3] on these PSVs) a test scenario will be generated for each possible valid combination of PSV values. (Representations and algorithms for such combinations are discussed at length in chapter 6 of [3].) Due to the semantic richness of ACL, such combinatorial testing can be quite complex. Consider, for example, scenario **TakeCourses** in contract *Student*. The *parallel* statement and or (|) operators used in the atomic block allow for the requirements of a course to be addressed in *any* order as previously mentioned. And, for assignments and midterms, loops are involved and must be flattened (thus requiring a min/max for the number of assignments and the number of midterms). In contrast, for the possible project and final, the choice is Boolean: a course has 0 or 1 project and 0 or 1 final. What further complicates the corresponding control flow graph is the *parallel* statement used at the start of the scenario to allow a single student to have the requirements of *several* courses taken the same term be addressed concurrently. Whereas a loop requires one PSV, a *parallel* statement requires two: i) the min/max number of instances (defined in this scenario by CurrentCourses().Length()) and ii) the min/max number of concurrent instances. That is, for this scenario to be thoroughly covered, we need to know not only the maximum number of courses a student can take in a term but also how many of these courses can have their requirements be concurrent. Only with these two PSVs can we test not only courses with simultaneous requirements, but also courses whose requirements do *not* overlap in time (i.e., minimum concurrency is set to 0). A last PSV (capturing the min/max number of courses dropped) is required to control whether or not courses are droppedduring a term.

Three observations proceed from this example:

1) Test scenarios are *not* meant to address issues such as trying to drop a course after the last day to drop: the scenario itself will catch such violations.

2) It should be clear that not all paths are covered: there is a combinatorial explosion of possible paths, especially in light of the interleaving [18] resulting from the use of *parallel* statements. This is why we rely on an existing algorithm for 'all-branches' coverage [3, 18].

3) The scope of a PSV is a test scenario. Thus there is no way in **TakeCourses** to associate a maximum number of assignments to a specific course. Semantically this could be desirable but would not only greatly complicate the generation of combinations of PSV values, but also dramatically decrease the usability of our testing approach (as a multitude of user inputs would be required).

The fact that a scenario invokes one or more responsibilities constitutes another source of complexity in identifying PSVs. Let us return to scenario **ReportMarks** of the *Course* contract. It invokes responsibility *ReportMark*() of the *University* contract. We postulate that, in the context of testing **ReportMarks**, both branches (i.e., a failing grade or, implicitly, a passing one) of *ReportMark*() must be covered (regardless of unit testing on the procedure bound to this responsibility). The PSV to control this is generated by the VF in the scope **ReportMarks**. Its name summarizes best its semantics: *numberOfFailingMarks* (over the scenario's execution). As usual, it is left to the user to specify the min and max for this PSV (from which the system will generate

an in-between value, if possible). Once, this is done, test scenario generation can proceed: the loop of the scenario leads to 2 or 3 paths (as explained earlier), each of which now branching into one of the 2 or 3 paths associated with *numberOfFailingMarks*. Thus, the VF generates a minimum of 4 test scenarios for **ReportMarks**: (min # of students, min # of failures), (min # of students, max # of failures), (max # of students, min # of failures) and (max # of students, max # of failures).

There two points to make here: i) and ii) the generation of test scenarios does not merely proceed from generating combination of PSV values; it also involves the flow analysis of the scenario (and the responsibilities it invokes). Thus, in summary, the generation and validation of test scenarios in ACL is a complex task relying on user input for setting up correctly boundary value testing.

5 Conclusion

We have presented elsewhere i) ACL, a semantically rich implementation independent specification language (that supports design by contract, responsibilities and scenarios) and ii) the tool (the VF) that enables its user to bind ACL specifications to the types and procedures of an IUT, thus enabling the validation of an ACL against an actual execution of an IUT. In this paper we have focused specifically on scenario testing and we have overviewed how ACL specifications can be used to generated test scenarios and their path sensitization data. Most importantly, using the VF, these test scenarios can be validated against an actual execution of an IUT. We believe only Spec Explorer [8] offers similar semantic richness and scenario testing capabilities, albeit using the symbolic execution of a complex finite state machine.

Acknowledgments

We thank Katie McClean for her implementation of the ACL parsing required to generate the PSVs. And thanks to NSERC!

6 References

[1] B. Meyer, The Unspoken Revolution in Software Engineering, *IEEE Computer*, January 2006.

[2] D. Arnold, J.-P. Corriveau and W. Shi, *Reconciling Offshore Outscouring with Model-Based Testing*, *SEAFOOD*, Saint Peterburg, Russia, June 2010.

[3] R., Binder, *Testing Object-Oriented Systems*, Addison-Wesley Professional, Reading, MA, 2000.

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

[5] D. Rosenberg and M. Stephens, *Use Case Driven Object Modeling with UMLTheory and Practice*, APress, 2007.

[6] Message Sequence Charts, http://www.itu.int/ITU-T/2005-2008/com17/languages/Z120.pdf

[7] Buhr, R.J.A., Casselman, R.: *Use Case Maps for Object Oriented Systems*. Prentice Hall, November 1995.

[8] C. Campbell, W., Grieskamp, L., Nachmanson, W., Schulte, N., Tillmann, and M.Veanes. *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*. Microsoft Research Technical Report #MSR-TR-2005-59, May 2005.

[9] Microsoft Research: *Spec# Tool*. http://research.microsoft.com/specsharp

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

[11] D. Arnold, The Validation Framework and its examples, http://vf.davearnold.ca/.

[12] IBM: *Rational Rose*. http://www-306.ibm.com/software/awdtools/developer/rose/index.html

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

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

[15] B. Meyer et al., Programs that test themselves, *IEEE Computer*, vol.42(9), September 2009, pp.46-55.

[16] B. Meyer, Design by Contract. In *IEEE Computer*, vol. 25, no. 10, pp. 40-51, IEEE Press, New York, October 1992.

[17] D. Arnold, J.-P. Corriveau and W. Shi,, Modeling and Validating Requirements using Executable Contracts and Scenarios, *SERA*, Montreal, May 2010.

[18] L. Briand and Y. Labiche, *A UML-Based Approach to System Testing*, Lecture Notes In Computer Science; Vol. 2185, Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 194 - 208

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

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