A Scenario-Driven Approach to Model-Based Testing

Dave Arnold and Jean-Pierre Corriveau
School of Computer Science
Carleton University
Ottawa, CANADA
(darnold, jeanpier)@ecs.carleton.ca

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

Abstract—Tools for Model-Based Testing (MBT) use various algorithms and strategies to generate tests from a behavioral model of the IUT. Existing MBT tools are based on formal, typically state-based, specification languages. In state-of-the-art MBT tools, which we briefly review, executable tests are generated. But these tests are not directly traceable to an actual implementation-under-test (IUT). Instead, they are generally executable in a global space of states. As an alternative approach, we explore here a scenario-driven approach to MBT. Its key characteristic consists in allowing IUT-independent tests generated from scenarios to be transformed into test cases that can run on an actual IUT. A fully implemented framework that supports the automatic instrumentation of such test cases into an IUT, as well as their run-time monitoring is discussed.

Keywords—model-based testing, state-based models, scenario-based models, test case generation, executability

I. INTRODUCTION

Testing is one of the most expensive aspects of software development [1, 2]: Due to the reduction of time and skilled personnel, software is often not tested as thoroughly as it should be. According to Grieskamp [1] from Microsoft Research, current testing practices are not only laborious and expensive but often unsystematic, lacking an engineering methodology and discipline and adequate tool support.

Code-based testing constitutes one approach to software testing. Code-based testing tools (such as JUnit and its several adaptations to different languages, and very recently AutoTest [2]) allow for unit tests (i.e., tests pertaining to a procedure) to be specified in, or automatically generated from, code. Thus, such tests are implementation-specific.

In contrast, models independent of the implementation under test (hereafter IUT) and typically at a higher level of abstraction than code can be used to obtain tests. But, generally, such tests are semantically ‘disconnected’ from the IUT. In her seminal review of the state-of-the-art in software testing, Bertolino [3] elaborates: “A great deal of research focuses nowadays on model-based testing. The leading idea is to use models defined in software construction to drive the testing process, in particular to automatically generate the test cases. The pragmatic approach that testing research takes is that of following what is the current trend in modeling: whichever be the notation used, say e.g. UML or Z, we try to adapt to it a testing technique as effectively as possible [...]”

For Grieskamp [1], model-based testing (hereafter MBT) is a promising approach to testing, even if within Microsoft only about 5-10% of the product teams are using or have tried using MBT. Indeed, he indicates this low percentage can still be considered a success compared to the use of more formal quality assurance approaches, such as verification. Grieskamp argues that the slow adoption of MBT is, in part, the result of the extensive time required to learn the modeling notations, a lack of state-of-the-art authoring environments, missing scenario-based semantics, and the absence of integration with test management tools.

In this paper, we introduce an approach to MBT rooted in the semantics of scenarios. This approach is not limited to unit testing but also addresses the conformance of an execution of an IUT to one of the scenarios captured in our specification model. In other words, we present an MBT approach that uses an IUT-independent scenario-based model relevant to stakeholders in order to generate and execute unit and scenario test cases on a candidate IUT. To do so, we will discuss the fully implemented validation framework (hereafter VF) we developed for unit and scenario testing.

We first summarize in section II the key characteristics of current MBT tools. Then, in section III, we explain what distinguishes our work from the state of the art, we overview how our tool deals with some aspects of testing, and finally, we discuss the semantics we use and how they support MBT. Section IV provides an example used in the section V to explain scenario testing.

II. STATE OF THE ART IN MBT

Model-Based Testing involves the derivation of test cases, in whole or in part, from a model that describes at least some of the aspects of the IUT. Typically, MBT is the automation of a black-box test design. 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. Test cases derived from such a model are functional tests on the same level of abstraction as the model. The test cases are grouped together to form an abstract test suite. Such an abstract test suite cannot be directly executed against the IUT because the test cases are not at the same level of abstraction as the code. Therefore, the abstract test suite must be ‘transformed’ into an executable test suite that can operate within the IUT. Such a transformation is performed (typically manually) by mapping each abstract (model-generated) test case to a concrete test case suitable for execution on a particular IUT.

Using the definition for a model-based test generator put forth by Hartman [4], we find that model-based tools must have the following two characteristics:

- The model must define a specification of the IUT (i.e., a testable model), and
- The tool must supply a set of test generation directives that guide test case generation.

The output of an MBT tool is usually a set of test cases that include a sequence of stimuli to the IUT and the expected responses as predicted by the model. Here, we are going to focus on MBT tools that fulfill these two characteristics. Before doing so, we wish to distinguish such tools from other classes of tools such as test automation frameworks and dedicated modeling tools:

A test automation framework accepts tests that have been manually created, automatically generated, or pre-recorded. The automation framework then executes the test sequences without human interaction. Such frameworks lie beyond the scope of MBT as they require as inputs, test cases readily executable on an IUT. Consequently, we will not discuss further such frameworks.

Modeling tools are used to create models of (some aspects of) an IUT. Such tools provide support for model specification, analysis, and maintenance. It is an IUT that is modeled (as opposed to IUT-independent requirements). Furthermore, such tools do not generally have the ability to generate test cases. Consequently, we will not discuss further these tools.

Let us now go back to MBT tools. In general such tools accept a specification, an IUT, and a test purpose. The tools use the specification to create an internal model, usually based on a Finite State Machine (FSM) or FSM-like representation. The model is then traversed according to the test purpose leading to the generation of test cases. The test cases can be specified using different notations, such as TTCN [5] or an XML schema. For example, Lutess [6] accepts three elements as input for the automatic generation of a testing harness: a test sequence generator, the IUT, and an oracle. The test sequence generator is derived from an environment description written in the synchronous dataflow language Lustre [7]. The description is composed of a set of constraints that describe the set of interesting test sequences, operational profiles, properties to be tested, and behavioral patterns. The environment description can be viewed as a program that observes the input/output stream of the IUT. The environment then determines if a given test sequence is realistic with respect to the IUT, and then the oracle determines correctness. The oracle and the IUT can be provided in one of two ways. The first is as a synchronous program, and Lutess will handle the IUT as a black box. The second way is to supply the IUT as a program written in Lustre. In this case, Lutess automatically compiles and integrates the IUT into the test harness.

In order to determine the key features of existing MBT tools, let us consider some others: Lurette [8] is similar to Lutess. In contrast, whereas Lutess and Lurette start all test sequence generation from the initial state, GATEl [9] starts with a set of constraints on the last state of the sequence.

An AutoFocus [10] model is a hierarchical set of time-synchronous communicating Extended Finite State Machines (EFSMs) that use functional programs to represent guards and assignments. Models created in AutoFocus can be used for both code generation and testing. MBT in AutoFocus requires a model of the IUT, a test case specification, and the actual IUT. The test case specification can take one of three forms: functional, structural, or stochastic. Like GATEl, AutoFocus generates tests based on constraint logic programming.

The Conformance Kit was developed in the early 1990s by KPN research to support the automatic testing of protocol implementations [11]. The kit uses EFSMs to represent system specifications. In 1995, Philips extended the Conformance Kit creating a set of tools called the Philips Automated Conformance Tester (PHACT) [12]. PHACT extends the Conformance Kit with the ability to execute the computed TTCN-2 test cases against an IUT. PHACT uses three elements for test case execution: the supervisor, the stimulator, and the observer. The stimulator and the observer interact directly with the IUT. As such, for each candidate IUT, a dedicated simulator and observer must be created.

TorX is both an architecture for a flexible open testing tool for test case derivation and execution, and an implementation of such a tool [13]. TorX can be used with any modeling language that can be expressed as a Labeled Transition System (LTS).

Automated Generation and Execution of test suites for Distributed component-based Software (AGEDIS) [14] was a project running from October 2000 until the end of 2003. The goal of the AGEDIS project was to develop a methodology and tools for the automation of software testing, with emphasis on distributed component-based software systems. The AGEDIS Modeling Language (AML) is a UML 1.4 profile that serves as the behavioral modeling language. The behavior of each class is defined with a state machine using an action language.

Similarly, an executable specification language, called Abstract State Machine Language (AsmL), was developed by the Foundations of Software Engineering (FSE) group at Microsoft Research [15]. AsmL has a built-in conformance test facility that is based on two steps. First, the AsmL specification is transformed into a FSM, and then well-known FSM-based algorithms [16] are used to generate a test suite. The process of generating an FSM out of an AsmL is a difficult task that requires considerable expertise from the tester for defining both a suitable equivalence relation and a
relevance condition that prunes the state space into something manageable. It is also problematic that the AsmL to FSM transformation may yield a FSM that is non-deterministic. The AsmL test generator cannot handle non-deterministic FSMs. Microsoft Research is working to address the non-deterministic FSM generation issue and state space explosion problems in their current state-of-the-art tool: Spec Explorer [17].

Spec Explorer is a tool for testing reactive, object-oriented systems (which are the systems we consider). The inputs and outputs can be viewed as parameterized action labels that represent the invocations of methods with dynamically created object instances and other complex data structures.

A model in Spec Explorer is specified using Spec# [18]. 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, high-level data types, logical quantifiers (e.g., forAll and Exists), and a simple form of scenarios. A Spec# model can be viewed as a program. It can be compiled and executed just like a C# program. A Spec# model can call .NET framework code, and the model can be explored using Spec Explorer. More precisely, using a model specified in Spec#, Spec Explorer extracts a representative behavior of the system according to user-defined parameters for scenario control. Spec Explorer accomplishes this using a state exploration algorithm that works as follows: In a given model the current state determines the next possible invocations. An invocation is denoted by an action/parameter combination that is enabled by the preconditions in the current state. Spec Explorer then computes the successor states for each invocation. The process is repeated until there are no more states or invocations to explore. Parameters used for these invocations are provided by state-dependent parameter generators. Enabledness is determined by the precondition of an action. Spec Explorer also must use heuristics to prune the generated state space.

Spec Explorer indicates that an IUT conforms to the model if the following conditions are met:
- The IUT must be able to perform all transitions outgoing from the active state.
- The IUT must produce no transitions other than those outgoing from a passive state.
- Every test must terminate in an expected accepting state.

Offline testing reduces the model to a test suite that can be compiled to create a stand-alone test driver and oracle. In online, or on-the-fly, the tasks of testing model exploration and conformance testing are merged into a single algorithm. If the IUT is a non-distributed .NET program, then Spec Explorer will automatically generate the testing harness. In other cases the tester is required to write a wrapper around a distributed IUT.

Regardless of the specific model-based tool, all of the tools discussed above strive to express a model using an executable specification language. Such specification languages use guarded-update rules on a global data state representing the IUT. The rules describe transitions between data states and are labeled with actions. Each action corresponds to a method invocation in a test harness, or directly in an IUT. The rules can be parameterized by the tester using techniques such as equivalence partitioning.

The point to be grasped is that existing MBT tools are based on formal (typically state-based) languages. However such formal notations are often perceived as having a steep learning curve. This explains why Grieskamp [1] mentions the need for scenario-based MBT tools for conformance. In particular, we require a testable specification language that is executable, yet abstract enough to be understood by stakeholders (who are reported to more readily understand scenario-based semantics). For example, Spec# aims to reduce the gap between the specification language and the implementation language. But while Spec# can be easily picked up by someone with C# programming experience, in the context of conformance/validation, it is difficult for a stakeholder to understand it, as it resides at a low-level of abstraction. This raises a first question with respect to scenario-based MBT tools: what semantics and what level of abstraction would be appropriate for them?

The dominance of state-based semantics in MBT presents another difficulty. MBT is known to generate a large amount of tests even for small models. The large amount of tests turns out to be quite a problem in practice, rather than an advantage [1] and stems from the state explosion problem. A large number of states increases the time required to run the test suite, making it a significant cost factor. Grieskamp remarks that the solution to the state explosion problem is not stochastic on-the-fly (online) testing. To reduce the number of states, and thus the number of test cases, the notion of test selection is used to select a set of ‘representative’ tests from the model. Test selection is usually implemented through graph traversal techniques that are applied to an FSM corresponding to the model. Other techniques include parameter generation, pair-wise combination, equivalence partitioning, etc. However, users of MBT tools within Microsoft complain that they do not have enough fine-grained control over the test selection process [1].

With respect to this issue of controllability, we note that some of the tools we have examined in this section define the notion of a test purpose, which is used to select some desired behavior from the model. Grieskamp believes that instead of having additional notation for the description of test purposes, models themselves should be used to express such test purposes. In the end, he remarks that, in a scenario-based approach, test purposes would be inherent to the scenario. More precisely, if scenarios are conceptualized as generators (or equivalently, grammars) of paths, then test purposes correspond to sets of paths through scenarios. The question of test selection then becomes: across a set of scenarios, which specific paths must be tested?

The two questions raised above (namely, level of abstraction and test selection) with respect to scenario-based MBT lead us to discuss, in the next section, the characteristics of the specific scenario-based MBT tool we have developed.
III. AN ALTERNATIVE APPROACH TO MBT

A. On Rooting Executability in an Actual IUT

Model-Based Testing requires a model from which tests can be generated. The immediate question is to decide what should be the semantics of the language used to express such a model. Following Grieskamp’s [1] observation that industrial users of MBT approaches prefer scenario-based semantics, we have rooted our semantics in the notions of responsibilities and scenarios. Scenarios are conceptualized as grammars of responsibilities [19]. Each responsibility represents a simple action or task, such as the saving of a file, or the firing of an event. Intuitively, a responsibility is either 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 [2, 20] elements (which have been shown [2] to be very useful for unit testing). Such elements are typically used to express constraints on the state of the IUT before and after the execution of a responsibility or of a scenario: Preconditions specify constraints on the state of the IUT before the responsibility or scenario can be executed. Post-conditions specify constraints on the IUT’s state following a successful responsibility or scenario execution. We also provide the means to express invariants [2] (as illustrated in the example of the next section).

Most interestingly, the language (called Another Contract Language, hereafter ACL) we use to model IUT-independent requirements is not limited to functional dynamic testing (in contrast to AutoTest [2] and similar tools): it also supports static testing, as well as the evaluation of metrics (as explained shortly). This observation is crucial as it leads us to acknowledge that the version of MBT we adopt is fundamentally different from the one on which the approach of the previous section rests. Let us elaborate.

Existing MBT approaches use IUT-independent models that generate tests that are also independent of the details of an IUT. This is the case even for Spec# [18], despite its low level of abstraction. At best, in a few of these approaches, tests depend on the input/output behavior of a system (which is viewed as a black box). More importantly, the executability of these tests occurs in the semantic space of the specification language and is most often rooted in the concept of state explosion. For example, Spec# refers to queues of unreceived messages (a notion remote from any actual implementation) and relies on a global state explorer. In contrast, we root executability in the execution of the actual IUT developed by designers. This decision proceeds from the following observation: Whereas the development of an IUT by designers is unavoidable (because it constitutes the artifact to test), the creation of an executable model rooted in the semantics of a specification language is likely to represent a significant investment of time in the software development lifecycle, which does not eliminate the necessity of also testing the IUT. This is particularly problematic in light of the learning curve associated with current specification languages used in MBT (which may also make this task error-prone). Furthermore, even expertise with the semantics of executability in current MBT tools does not entail the absence of technical challenges such as state-based explosion (mentioned in section II).

But choosing to root executability of tests into the execution of the actual IUT presents an immediate problem: how can IUT-independent tests be executed by an IUT?

First it is important to acknowledge that in MBT, tests generated from an IUT-independent model must be themselves IUT-independent. Otherwise, we are essentially back to a code-centric approach and its disadvantages. In particular, the crucial distinction between the specification model (from which tests are generated) and the IUT is dangerously blurred.

Second, we suggest that enabling the execution of IUT-independent tests on actual IUT 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. A manual transformation (through the creation of so-called ‘glue code’) 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 model. Thus, we advocate an automated transformation, as well as automatic instrumentation. This has a crucial consequence for the semantics of ACL: our position is that only semantics that enable 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.

B. Using the Validation Framework

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 [19, 21], and design-by-contract constructs such as pre and post-conditions, and invariants [2, 20], as previously mentioned. Additional domain-specific constructs can also be added to the ACL, via modules known as plug-ins (as explained elsewhere [22]).

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 [22]. 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. 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.

In our VF, a set of bindings is represented by an XML file that contains tags linking contract elements to their IUT counterparts. Each IUT that is to be executed against a TRM must have a corresponding binding file.

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. Let us elaborate.

Since bindings provide a mapping from the TRM to a candidate IUT, details regarding the structure of the IUT are required. Obviously, these details are implementation-specific, and as such different binding algorithms may be required for different programming languages.

ABE supports an open approach to the automation of binding creation: different algorithms for finding bindings are separately implemented in different binding modules. Each binding module is implemented as a DLL (i.e., Dynamic Link Library) and is placed in a specific location relative to the folder in which our VF resides. Each such DLL must implement a specific interface we have defined, in order to be used as a binding module. Put simply, this interface allows the creator of a binding module to gain access to the internal structure of an IUT without having to get familiar with the (highly technical) internal representation of this structure.

Our VF uses only one binding module at a time. However, multiple modules can be used successively for the same TRM/IUT pair. That is, one module could be selected to infer as many bindings as possible, then a second module could be selected to infer any bindings not recognized by the first module.

We have implemented two such binding modules (for C# and C++) as part of the current release of our VF. 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) [22]. Missing bindings are specified manually.

We discuss further the importance of bindings for our approach to scenario testing in section V. For now, we must first overview the semantics of ACL.

C. Scenario-Based Semantics for MBT

ACL provides the user of our VF with built-in static checks, dynamic checks, and metric evaluators. (In addition, via a software development kit we wrote, plug-ins allow for the inclusion of user-specified static checks, dynamic checks, and metric evaluators into a TRM, see details in [22].)

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.

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 [2]. Indeed, for unit testing, our VF supports an approach similar to the one of AutoTest [2], as discussed in details elsewhere [22]. 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.

Due to space restrictions, only pre/post-conditions and invariants will be illustrated in the next section. The Container ACL model found in [22] provides a simple example that includes static and dynamic checks, as well as metric evaluators. It also illustrates how this ACL model is bound to different implementations and how the VF reports on the monitoring of an execution of an IUT.

D. Overview of Testing Process

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 IUT is executed by the VF. 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 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.

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 checks 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, as explained in Section V). 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.

### IV. An Example

We now present excerpts of a 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 just enough information to a) illustrate the semantics of ACL, especially those pertaining to scenarios and b) discuss the basics of unit and scenario testing in the VF. We use the // and /* */ to provide comments directly in the example, and otherwise rely on the reader’s ability to get the gist of the example.

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.

```csharp
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
{  Scalar Boolean EnforcePreRequisites =
   { true, default false }; // a course may have 0, 1 or 2 midterms
   [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 (1 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(); }
```

Parameters are values supplied to a contract. These values can be provided either via other contracts (i.e., statically), or at binding time (if 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 in Section V.
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);
}

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 purpose of this element is to ensure that the correct mark for the given student is recorded.

The ProjectCourse contract represents a refinement of the Course contract representing a single university course. The ProjectCourse is used for courses that also include a course project in addition to any number of assignments, mid-terms, and a final exam.

The binding point begins with the 'stub' symbol and that the parameter is set to not be enforced (via the above 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);
}

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 purpose of this element is to ensure that the correct mark for the given student is recorded.

The ProjectCourse contract represents a refinement of the Course contract representing a single university course. The ProjectCourse is used for courses that also include a course project in addition to any number of assignments, mid-terms, and a final exam.

The ProjectCourse contract represents a refinement of the Course contract representing a single university course. The ProjectCourse is used for courses that also include a course project in addition to any number of assignments, mid-terms, and a final exam.

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 purpose of this element is to ensure that the correct mark for the given student is recorded.

The ProjectCourse contract represents a refinement of the Course contract representing a single university course. The ProjectCourse is used for courses that also include a course project in addition to any number of assignments, mid-terms, and a final exam.

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 purpose of this element is to ensure that the correct mark for the given student is recorded.

The ProjectCourse contract represents a refinement of the Course contract representing a single university course. The ProjectCourse is used for courses that also include a course project in addition to any number of assignments, mid-terms, and a final exam.

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 purpose of this element is to ensure that the correct mark for the given student is recorded.

The ProjectCourse contract represents a refinement of the Course contract representing a single university course. The ProjectCourse is used for courses that also include a course project in addition to any number of assignments, mid-terms, and a final exam.

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 purpose of this element is to ensure that the correct mark for the given student is recorded.
no ambiguity between the stub and the responsibility

**DoAssignment**, despite using the same name.*]

```
stub Responsibility DoAssignment(tCourse c)
{ Contract Course course = c; boundpoint;
  [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 responsibilities and events. This illustrates another use of events, beyond triggers and terminations 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.*]

```
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 assignments, 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();
  TakeCourses(); } //end of scenario TakeCourses
```

/* Due to space restrictions we omit the University contract. We will merely mention that its main scenario, modeling a term, uses the keyword fire in order to fire events such as TermStarted, LastDayToDrop, and TermEnded.

We conclude with an example of inter-scenario relationships. 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
  { Contract University u;
    // creation of students and courses can be in any order
    (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 automatically c is of type tCourse
c = u.CreateCourse(dontcare, dontcare),
  (u.CancelCourse(c))?,
  observe(TermStarted)); }
} //other relations are omitted
}
```
V. UNIT AND SCENARIO TESTING

Having hinted above at how contract and scenario instance creation works, and how grammars of responsibilities are modeled, we now discuss how such grammars are tested. For simplicity, in this section, the term ‘scenario’ will refer to all forms of grammars of responsibilities in ACL (namely, responsibilities such as DoProject, scenarios and relations).

First, as explained earlier, the testable requirements model 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 as execute any dynamic checks, and gather metrics indicated.

The simplest strategy consists in not generating any tests from the TRM and instead leaving the provider of the IUT to supply a test suite: 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 [23, 24, 25] of the TRM is completely hidden from the creators of the TRM, which is highly problematic! In our opinion, stakeholders must have a say in ‘how much’ a TRM is to be tested.

Coverage at the level of unit testing\(^1\) is considerably simpler than for scenarios. It rests on the use of well-known combinatorial techniques (chapter 6 in [23]) to address, for example, the coverage of the Boolean clauses of pre- and post-conditions [2]. Also, as with AutoTest [\textit{Ibid.}], 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 \texttt{InstanceBind} keyword).

For scenario testing, the key idea is that scenarios are grammars and that, as with state machines [16, 23], there are well-known algorithms to obtain a selection of paths through a grammar according to some coverage criterion. Indeed, this idea has already been used in several distinct approaches to test case generation from scenarios [24, 25, 26]. Due to space restrictions, a survey of this research is omitted. 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 [23]) that enables this path to be taken. Consider, for example, scenario \texttt{RegisterForCourses} in our example. A test case \textbf{TCl} 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 (CFG) [23]. 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 [23]. Each coverage criterion is associated with a traversal algorithm that generates a test suite sufficient to cover the graph at hand. Binder [23] 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., [24, 25]). From this viewpoint, a test case can be thought of as a particular path of execution through a scenario. For example, for \textbf{TCl}:

```java
{   (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 capsize 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 \textbf{TCl}, the following:

```java
{   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) }
```

\(^1\) For the VF, a unit of testing is an IUT procedure corresponding to a responsibility of an instance of a contract.
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 seems very difficult to achieve even using parameters. For example, the number of courses varies from one test case to another for scenario RegisterForCourses...

VI. CONCLUSION

State-based approaches to model-based (MBT) testing offer the advantage of formal semantics, well-established techniques for test case generation, and executability rooted in the semantics of the specification model (in particular, in the notion of state exploration). Yet their adoption remains relatively low in industry, partly because of the preference of users for (often informal) scenario-driven semantics [1]. In this paper, we have introduced a framework that supports the automatic instrumentation and run-time monitoring of static and dynamic checks, as well as metric evaluation, and most importantly, scenarios. In our work, it is the execution of an actual implementation that is monitored against a testable model. But whereas test case generation is fully automated, test case execution requires a user to set relevant states of instances involved in the execution of a test case.

ACKNOWLEDGMENT

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

REFERENCES
