2010 Eighth ACIS International Conference on Software Engineering Research, Management and Applications

MODELING AND VALIDATING REQUIREMENTS USING EXECUTABLE CONTRACTS AND SCENARIOS

Dave Arnold and Jean-Pierre Corriveau
School of Computer Science
Carleton University
1125 Colonel By Drive
Ottawa, CANADA, K1S 5B5
(darnold, jeanpier}@scs.carleton.ca

Wei Shi
Faculty of Business and Information Technology
UOIT
2000 Simcoe North
Oshawa, CANADA, L1H 7K4
Wei.Shi@uoit.ca

***Abstract*—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). In Model-Based Testing, much work has been done on the generation of functional test cases. But few approaches tackle the executability of such test cases. And those that do, offer a solution in which tests and test cases are not directly traceable back to the *actual* behavior of an IUT. Furthermore, very few approaches tackle non-functional requirements. Consequently, we have implemented a validation framework that *does* support the modeling and automated validation of a set of functional and non-functional requirements against several candidates IUTs. We report here on the key characteristics of this prototype and briefly discuss lessons learnt from its use in the context of a graduate course.**

Keywords- validation; model-based testing; executability

I. ON MODEL-BASED TESTING

It is widely accepted that Requirements Engineering aims at providing a bridge between the stakeholders and the developers of a computer-based system, each having their own specific viewpoints and concerns [1]. From a practical standpoint, this bridge must be an operational one, rooted in the key notion of quality [2]. That is, ultimately, the needs of stakeholders must be *validated* [3] against the actual behavior of an *implementation under test* (IUT). The act of validating stakeholders' needs consists in determining if an IUT satisfies the (functional *and* non-functional) requirements of the stakeholders [2, 3].

Current approaches to validation fall into two categories: code-centric and model-centric. A code-centric approach to validation, such as Test-Driven Development (TDD) [4], uses *test cases* (as opposed to more abstract *tests* [3][1]) 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 the system. The test case is executed against the software system, usually resulting in a failure. Code is then added to the implementation until the test case succeeds. The process repeats until all of the requirements (known somehow by a developer) have been satisfied. Such test cases do not explicitly model the needs of stakeholders, and they are implementation-specific. Indeed, implementation details pervade code-centric approaches, making them difficult for stakeholders to understand and use.

In contrast, in a model-centric approach, such as Model-Based Testing (MBT) [5], the needs of stakeholders are specified in models from which tests are extracted to 'drive' validation. These models (whether in a graphical or text-based format), usually capture a *partial* representation of the IUT's behavior, 'partial' especially because the model abstracts away some of the implementation details. Tests derived from such a model are functional tests at the same level of abstraction as the model. Such tests are grouped together to form an abstract test suite. Such a test suite *cannot* be directly executed against an IUT because the tests are not at the same level of detail as the code. The question then is: how are requirements validated against the behavior of an actual IUT? We identify three approaches:

The first approach consists in augmenting abstract requirement models with implementation-specific details capable of generating tests (and possibly test cases). For example, Briand and Labiche [6] advocate the use of UML activity diagrams for the modeling and validation of system-level functional requirements. Their method requires the use of the Object Constraint Language (OCL) [7] to capture implementation-specific constraints. In particular, the OCL requires making explicit references to variables within an IUT. Thus, requirements models involving the OCL are tightly coupled to specific implementation details. This is highly problematic: any change to the implementation must 'percolate up' to the requirements model, a maintenance nightmare. Furthermore:

- The presence of implementation details considerably reduces the understandability of such models by stakeholders (and appears to contradict the very nature of *model-driven* testing).
- The possibility of having a requirements model abstract enough to be applicable to several IUTs is lost.
- The OCL does not address non-functional requirements.

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

978-0-7695-4075-7/10 $26.00 © 2010 IEEE
DOI 10.1109/SERA.2010.46

- The generation of tests and test cases from the OCL requires the use of stand-alone tools (such as [8, 9]).

Finally, the OCL does not address interactions between components. Thus, for that crucial part of their method, Briand and Labiche start from activity diagrams in order to obtain (through multiple transformations) abstract test sequences, which are disconnected from the actual implementation. How such test sequences are to become executable and how their validation (which involves monitoring actual sequences of procedure calls during the execution of an IUT) is not addressed. This leads us directly to the second approach to such problems.

In order to bridge the gap between the abstract tests and an actual implementation, the widespread solution consists in using 'glue code.'. That is, tests obtained from the models are subsequently coded or, more precisely, have some of their corresponding test cases coded (typically manually). In other words, glue code requires a programmer to make abstract tests become operational (i.e., usable for validation) via programming. The resulting code (which includes not only executable test cases but also test drivers and oracles [3]) must generally be handcrafted and may end up not corresponding to the abstract tests. Moreover, this glue code is implementation-specific: both its reusability across IUTs and its maintainability are highly problematic. Ultimately, the creation of glue code is a non-automated endeavor that is time-consuming and just as error-prone as development of the original IUT.

In light of the pitfalls of the previous approaches, researchers at Microsoft have developed, over the last several years, what we consider to be the state-of-the-art tool for model-based testing, namely Spec Explorer [10, 11]: "Spec Explorer 2010 is a tool that extends Visual Studio for modeling software behavior, analyzing that behavior by graphical visualization, model checking; and generating standalone test code from models. Behavior is modeled in two ways: by writing rule machines in C# (with dynamic data-defined state spaces) and by defining scenarios as action patterns in a regular-expression style." [12].

Like many other MBT methods, Spec Explorer generates test cases from a graph corresponding to a state machine defined in terms of global data spaces. More specifically, the modeling of requirements consists in defining an infinite transition system T_P through a Spec# program P (where Spec# is in fact a superset of C#). Exploration (which *attempts* to avoid state explosion through the use of complex heuristics and of action scenarios) reduces T_P to a finite test graph *G*. Test cases are subsequently generated by traversing *G* (using coverage techniques [3] and/or user defined sequences of actions). In a given state, the testing component of Spec Explorer determines which actions are enabled and thus which states of *G* are reachable. Spec Explorer then uses the current test case to select one of these reachable states and executes the corresponding *controllable* action (i.e., an action that defines one or more method invocations to be observed in P). Spec Explorer does provide the test harness that allows for such actions to be executed in P and for the resulting method invocations in P to be observed (in order to determine the resulting state in *G*). The key point is that, in this approach, it is P that is tested. But P *is not* an actual IUT; it is a model in Spec# of an actual IUT.

Thus, the question remains: how can requirements be modeled and validated against the behavior of an actual IUT? It is this question that we address in this paper. Our objective here is to introduce our proposed solution, which we have implemented in a tool called the Validation Framework (VF). Due to space restrictions, we purposely do not go into the details of the realization of our VF[2]: through the overview of our VF the specific contribution in this paper consists in suggesting that, indeed, it is possible to have an implementation-independent requirements model be 'semi-automatically' (as explained later) validated against an actual IUT.

First, in subsection 2.1, we argue for a requirements model that is testable and independent of implementation details (because it may have to be validated against several IUTs). Then, in subsection 2.2, we summarize the *modus operandi* of the validation framework we are proposing. In section 3, through the walkthrough of a very simple example, we overview the semantics this VF offers for the expression of a Testable Requirements Model (TRM). As explained in 2.2, in order for this TRM to be validated against an IUT, a set of *bindings* must be created. This process is explained in section 4. We conclude with some remarks stemming from our tool being used by the students of a graduate course in object-oriented software engineering at Carleton University.

II. THE VALIDATION FRAMEWORK

A. One model, several IUTs

The solution we propose proceeds from the integration of two well-known ideas: a) expressing requirements in terms of *responsibilities* [16, 17] and *scenarios* [17, 18, 19, 20, 21] and b) organizing such responsibilities and scenarios into *contracts* [22, 23, 24] that can be *bound* to components of an actual system. In essence, it is this integration that we discuss in the rest of this paper.

Semantically, our work proceeds from noticing that in his explanation of the limited adoption of a previous version of Spec Explorer within Microsoft, Grieskamp [11] proposes several possible reasons. In particular, he remarks that developers and testers alike significantly prefer the intuitive nature of scenarios to the semantics underlying more formal (not only logic-based and set theoretic, but also state-based) approaches. Indeed, the use of scenarios (e.g., UML's use cases) as a method for requirements capture is commonly accepted. Consequently, our work seeks to capture functional requirements specified in the form of

[2] The tool, its code and documentation, as well as the extensive test suites and case studies used for its validation are all accessible from [13]. The VF consists of over 250,000 lines of mostly C# code and requires the use of Microsoft's Visual Studio 2008, and Microsoft's Phoenix SDK [14] (as briefly discussed later in this paper and explained at length in [15]).

scenarios[3]. Most importantly, this decision allows us to reuse the existing seminal work on generating tests from scenarios (e.g., [3, 18]), especially the algorithms of [6] and [24]. However, the generation of tests and the instantiation of such tests into executable test cases constitute two complex tasks that lie beyond the scope of the current paper. Consequently, those two topics are discussed at length elsewhere [25].

Another premise of our proposal is the adoption of a genuine model-based outlook: the requirements model that is to be validated must be implementation independent. More specifically, we require that our Testable Requirements Model (TRM) be decoupled from any particular implementation so that a single TRM may be validated against several candidate implementations. By candidate, we mean an IUT that is to be validated 'against' the requirements defined by the TRM. Thus, our work supports applying a single TRM to several candidate implementations (one at a time) without requiring any modification to this TRM. The 'results' (which will be discussed later) generated by validating the TRM against each candidate IUT can be used for the behavioral comparison of different IUTs. As an example of such a use of our VF, consider several vendors applying for certification against a standard. The standard would be represented by a single TRM that would be validated against each of the candidate implementations. Please note that the process and issues pertaining to the development of candidate IUTs lie outside the scope of our work: we merely postulate the eventual availability of one or more IUTs to validate against a TRM. Furthermore, we remark that our approach to validation does not assume that the implementation source code is available. This is highly desirable in several situations such as offshore outsourcing [26].

B. Using the Validation Framework

1) Inputs to the Validation Framework

Our VF operates on three input elements. The first element is the Testable Requirements Model (TRM) expressed using ACL (Another Contract Language) [15], a high-level general-purpose requirements language we have created. We use here the word 'contract' because a TRM is formed of a set of contracts, as will be illustrated in section 3. The ACL is closely tied to requirements by defining constructs for the representation of goals and beliefs [27], scenarios [17, 18, 19], and design-by-contract concepts such as pre and post-conditions [22]. Additional domain-specific constructs can also be added to the ACL, via modules known as *plug-ins* (which will be discussed in 2.2.2).

The second input element is the candidate IUT against which the TRM will be executed. Recall that the framework accepts IUTs in binary form. Using Microsoft's Phoenix Software Development Kit [14], the framework is able to open .NET managed executables and Active Server Page (ASP) .NET web applications.

[3] Please note that our semantics are *not* limited to functional requirements, as will be illustrated in 2.2.2 and Section 3.

Bindings represent the third and final input element required by the VF. Before a TRM can be executed the types, responsibilities, and observability requirements (see section 3) specified in a TRM must be bound to concrete implementation artifacts located within the IUT. Phoenix is used to open the binary IUT for the purposes of obtaining a structural representation. 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. (As will be discussed in section 4, our binding tool is in fact able to automatically infer most of the bindings required between a TRM and an IUT.) Bindings constitute the key facet of our approach:

- Like the well-known mediator pattern [28], bindings *decouple* a TRM from any IUT. Only the binding tool 'knows' to which component of an IUT each element of a TRM maps.
- Because each IUT has its set of bindings to a TRM, several candidate IUTs can indeed be validated against a same TRM.

In other words, bindings constitute the bridge between a TRM and an actual IUT, a bridge that is generally absent from other MBT methods.

2) Plugins

The ACL provides the user of our VF with built-in static checks, dynamic checks, and metric evaluators (see below and section 3). In addition, plug-ins allow for the inclusion of *user-specified* static checks, dynamic checks, and metric evaluators into a TRM. In other words, the language we use to capture requirements has 'open' semantics inasmuch as new constructs can be defined by a user, provided this user can specify how to test such constructs in an IUT. Let us elaborate.

The creation of plug-ins is targeted toward developers of 'checks' for our VF, rather than the author of a TRM. More precisely, we have created specialized plug-in software development kits (SDKs) to aid in the creation of static checks, dynamic checks, and metric evaluators:

A static check performs a check on the IUT that can be accomplished without execution. Examples of static checks include checks involving inheritance (e.g., type A must be a descendant of type B), checks on types (e.g., type A must contain a variable of type B) and even the correct use of structural design patterns [28]. 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. Examples of dynamic checks include: testing the value of a variable at a given point, ensuring a given state exists within an object, and validating data sent between two different objects. As with static checks, a dynamic check 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 performed by the validation framework. 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.

Through the use of plug-ins, researchers in static testing, dynamic testing, and metric evaluation can contribute to our VF without having to create an *ad hoc* approach to the integration of their work within the VF. Instead, using the appropriate SDK, they can create plug-ins corresponding to their work. Over time this approach should lead to a rich set of plug-ins, which, *de facto*, will improve the applicability of ACL to different domains.

3) Validation 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.

Execution of a Testable Requirements Model 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 required by the TRM. The execution paths are used to determine if each scenario execution matches the grammar of responsibilities (see section 3) corresponding to it within the TRM. 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 were observed. 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.

III. A SIMPLE EXAMPLE

As in Use Case Maps [17, 18], in ACL scenarios are conceptualized as grammars of responsibilities. 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. In addition to responsibilities and scenarios, the ACL offers a set of Design-by-Contract [22] elements. The latter are typically used to express constraints on the state of the IUT before and after execution of a scenario or responsibility: Preconditions specify constraints on the state of the IUT before a responsibility or scenario can be executed. Post-conditions specify constraints on the IUT's state following a successful responsibility or scenario execution. The ACL also provides the means to express invariants (as illustrated below).

When a pre- or post-condition fails, execution proceeds but the failure is logged in the Contract Evaluation Report (CER). Also, when a scenario is executed by an IUT, the specified grammar of responsibilities must hold. That is, the responsibilities that compose the scenario must be executed in such 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 IUT does not match the TRM. This mismatch is also reported in the CER.

The following annotated example summarizes several of the semantics currently supported by our contract language and VF. (Note however that inter-scenario relationships [19] are missing here but are offered by the ACL.) The // and /* */ delimit comments aimed at explaining the key points of the example, which models a simple generic container (storing elements of type tItem). We have included example usage of inheritance in order to demonstrate how our work supports the compositionality of contracts through this mechanism. Contract elements without comments are assumed to be self-explanatory.

```
Import Core;
Namespace My.Examples
{
/*An abstract contract is NOT bound to a type within the IUT. Also, T will be bound upon ContainerBased being refined.
A contract may define variables, which will be kept by the VF.*/
abstract Contract ContainerBase<Type T>
{
 Scalar Integer size;
          // Number of elements in the container.

/* An observability is a query-method that is used to provide state information about the IUT. That is, they are read-only methods that acquire and return a value stored by the IUT. */
 Observability Boolean    IsFull();
 Observability Boolean    IsEmpty();
 Observability T ItemAt(Integer index);
 Observability Integer    Size();

/*An abstract observability MUST be refined in a derived contract. */
abstract Observability Boolean  HasItem(T aItem);

/* The body of the "new" responsibility is executed immediately following the creation of a new contract instance. */

 Responsibility new()
{ size = 0;   Post(IsEmpty() == true); }
```

```
/* The body of the "finalize" responsibility is executed immediately before the destruction of the current contract instance. */
Responsibility finalize()
{ Pre(IsEmpty() == true); }

/* 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 SizeCheck
{ Check(context.size >= 0);
  Check(context.size == Size()); }

/* This responsibility defines pre- and post- conditions for any addition. It is not to be bound but rather to be extended by actual responsibilities.
The keyword 'Execute' indicates where execution occurs. */
Responsibility GenericAddition(T aItem)
{ Pre(aItem not= null);   Pre(IsFull() == false);  Execute();
  size = size + 1;
  Post(HasItem(aItem));  }

/* This responsibility extends GenericAddition. It therefore reuses the pre- and post-conditions of GenericAddition.
It does not add any other checks to those of GenericAddition.
But Add can (and will) be refined in the contract that extends the current abstract one. */
Responsibility Add(T aItem) extends GenericAddition(aItem)
{ Execute(); }

/* Insert also extends GenericAddition and thus reuses its pre- and post-conditions.
But it also adds pre- and post-conditions of its own due to the fact that its interface involves the use of an index. */

Responsibility Insert(Integer index, T aItem)
  extends GenericAddition(aItem)
{ Pre(index >= 0);          Execute();
  Post(ItemAt(index) == aItem); }

/* Responsibility Remove returns the element removed.
The keyword 'value' denotes the return value.  */
Responsibility T Remove()
{ Pre(IsEmpty() == false); Execute();
  size = size - 1;
  Post(value not= null);
  Post(HasItem(value) == false); }

Responsibility RemoveElement(T aItem)
{ Pre(IsEmpty() == false); Pre(HasItem(aItem) == true);
  Execute();
  size = size - 1;
  Post(HasItem(aItem) == false); }

/* The following scenario merely consists of a trigger statement and a terminate statement. There is no grammar of responsibilities between these two statements (in contrast to most scenarios.)
This scenario captures the fact that the addition of an element x must eventually be followed by removal of x.
Here Add or Insert trigger the scenario, and Remove or RemoveElement terminate it.
Notice the use of the 'dontcare' keyword for the first parameter of Insert.  */
Scenario AddAndRemove
{ once Scalar T x;
  Trigger(Add(x) | Insert(dontcare, x)),
  Terminate((x == Remove()) | (RemoveElement(x)));  }

}  // End of contract ContainerBase

/* A TRM must include a main contract. It typically includes several other contracts.
The main contract of a TRM must be bound to a type of the IUT. Here Container inherits from ContainerBase.
Single and multiple inheritance are supported for composing contracts together.
Also, note that T in ContainerBase is explicitly  bound here to the type tItem (using syntax similar to templates in C++).
*/
MainContract Container extends ContainerBase<tItem>
{ List Integer container_times;
// Amount of time that each item spends in the container.
  Scalar Timer item_timer;
/* Timer is a built-in type of our VF
A single timer can be used to time multiple items concurrently. */
  Scalar Integer number_of_items;
/* Used to store the total number of items that are stored by the container during execution.  */

/* The abstract responsibility of ContainerBase is now refined.  */
refine Observability Boolean HasItem(tItem item)
{ tItem x;  Boolean result = false;
  loop(0 to Size())
          { x = ItemAt(counter);
            result = result || x == item;          }
  value = result; // Value is the keyword for return value.
          }
/* A parameter can be set explicitly, or using the binding tool of section 4, or set at run-time. Here, it controls whether the static check below is to be performed or not. */
Parameters
{ Scalar Boolean CheckMembers; }
```

```
/* What follows is a static check that uses the plug-in static check HasMemberOfType to verify if the container holds instances of type tItem. This check is performed only if parameter CheckMembers is true. A belief is merely a message logged in the report (CER) produced by the VF. */
Structure
{ choice(Parameters.CheckMembers) == true
    { Belief CheckMember("There should be a member in our container to hold elements of type tItem")
            { HasMemberOfType(tItem); }    }    }

/* We refine new: The post-conditions of the parent contract are checked before the body below is executed. */
refine Responsibility new()
{  number_of_items = 0;
   container_times.Init(); }

/* The 'fire' keyword is used to create an instance of an event that can, in turn, trigger or be observed in scenarios. */
refine Responsibility finalize()
{  fire(ContainerDone); }

/* Next, Add, Insert, Remove and RemoveElement from the ContainerBase contract are further refined to use timers. More specifically, the scenario AddAndRemove (in the parent contract) creates an instance of itself for each element that is added to the container. This allows us to start a timer in Add or Insert upon insertion of an element and to stop that timer when that element is removed. In turn, this allows us to store the time spent by an element in the container.   */
refine Responsibility Add(tItem item)
{ Pre(HasItem(item) == false);       Execute();
  item_timer.Start(item);  // Built-in way to start a timer.
  number_of_items = number_of_items + 1; }

refine Responsibility Insert(Integer index, tItem item)
{ Pre(HasItem(item) == false);       Execute();
  item_timer.Start(item);
  number_of_items = number_of_items + 1;   }

refine Responsibility tItem Remove()
{ Execute();
  item_timer.Stop(value);
  container_times.Add(item_timer.Value(value));  }

refine Responsibility RemoveElement(tItem item)
{ Execute();
  item_timer.Stop(item);
  container_times.Add(item_timer.Value(item));  }

/* This responsibility is to be used in the scenario ContainerLifetime below. RemoveScn abstracts away which of the two Remove responsibilities is used.
Notice again the use of keyword 'dontcare'.   */
Responsibility RemoveScn()
{ Remove() | RemoveElement(dontcare);  }

/* A stub responsibility is a place holder for one or more responsibilities. Here, we have only one choice, the default one, the responsibility Add.  Parameters and other mechanisms could be used to select between different kinds of addition, as illustrated elsewhere [15]. */
stub Responsibility AddElement(tItem item)
{ Pre(item not= null);
  [Default] Add(item); }

/* This scenario illustrates a Trigger being followed by a grammar of responsibilities and then a Terminate statement. In this case, the Terminate MUST be preceded by an 'observe' statement specifying the event that enables this termination.

In the following scenario, a new scenario instance is created each time a new container is constructed (via the new responsibility). The responsibility new acts as the trigger.
The ',' denotes the 'follow' operator.
An atomic block defines a grammar of responsibilities so that no other responsibilities of this contract instance are allowed to execute except the ones specified within the grammar.
The scenario must observe the event ContainerDone before concluding by proceeding with the execution of  finalize (which fires the event ContainerDone before its checks. This semantic 'contortion' is due to the way scenario instances are monitored. */

Scenario ContainerLifetime
{ Trigger(new()),
  atomic
    { (Add(dontcare) | Insert(dontcare, dontcare))*,
            (RemoveScn())*; },
  observe(ContainerDone),
  Terminate(finalize()); }

/* A list of integers representing the amount of time that each element spent in our container. */
Metric List Integer ContainerTimes()
{ context.container_times; }

// Total number of items that were stored in the container.
Metric Scalar Integer NumberOfItems()
{ context.number_of_items; }

// This section builds the evaluation report.
Reports
// {0} is where the reported result goes in the output string.
{ Report(          "The average time in the container is {0} milliseconds",     AvgMetric(ContainerTimes())); // Plug-in AvgMetric
```

```
/* A report all statement performs the exact same way as
the report statement, except that it generates a single result
for all contract instances.  */
  ReportAll("The average time in all containers is {0}
milliseconds",     AvgMetric(ContainerTimes()));
  Report("The number of items added to the container is
{0}",     NumberOfItems());
  ReportAll("The number of items added to all containers is
{0}",     NumberOfItems());                              }

/* The type tItem used for the elements of the container
cannot be type bound to the container nor any of its
descendants. So, here, we do not allow lists of lists. */
Exports
{ Type tItem conforms Item
            { not context;  not derived context; }          }

} }
```

To conclude, we remark that this single TRM has been applied to several simple data structures (e.g., different kinds of arrays and linked lists) implemented in C# and C++/CLI, with and without coding errors (in order to verify responsibility and scenario failure).

IV. TRACEBILITY THROUGH BINDINGS

The crucial point to grasp with respect to the semantics supported by the ACL and overviewed in the previous section is that they do support automated validation. That is, once a TRM is linked 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 execution). Thus, our whole approach to validation hinges on the ability to link a TRM to an IUT. To do so involves the creation of *bindings*. More precisely, our framework is able to capture (and even partially infer) a set of mappings, called bindings, between elements of the implementation-independent TRM and procedures and types of the IUT. It is the creation of such bindings that eliminates the need for the development of glue code. Let us elaborate.

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. However, rather than having to directly edit the XML binding code, we have integrated a binding tool into the VF. The binding tool provides a graphical way to view and specify the binding information. Figure 1 provides a snapshot of the main window of our binding tool (opened on the contract given in section 3).

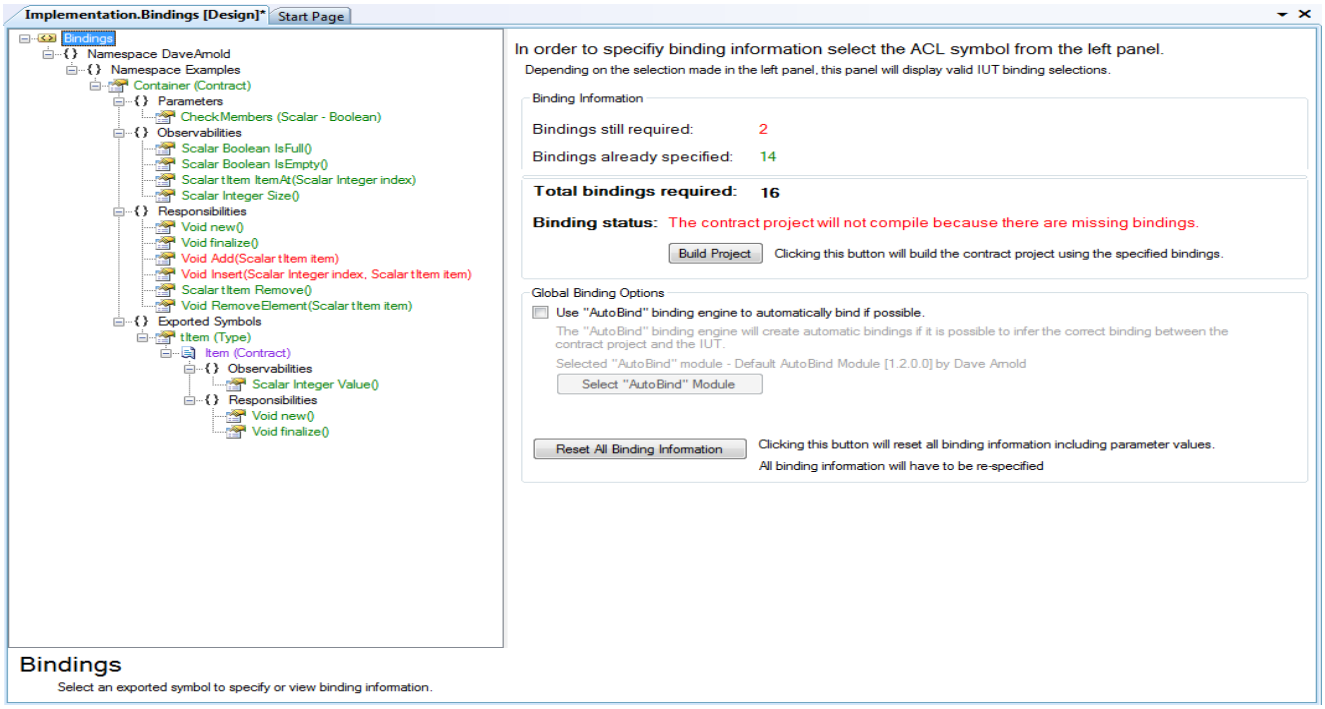

Figure 1. The Binding Tool

On the left, the binding tree displays each contract element that must be bound to an IUT counterpart. These elements include: contracts, parameters, observability methods, responsibilities, and exported types. (As previously mentioned scenarios are grammars of responsibilities and, as such, have no corresponding elements in an IUT.) The contents of the binding tree are generated by the ACL compiler as it compiles a TRM.

The binding status of the elements of contracts to be bound is summarized in the color of each item in the binding tree: green for those successfully bound, red otherwise. If a bound contract element is selected in the

binding tree, then information about its corresponding IUT structural element will be displayed under the selection.

On the right, in Figure 1, information about the overall binding status of a TRM (i.e., a set of contracts) is given. Here, two bindings are missing and thus the VF cannot validate the IUT against the TRM. Users of our VF can bind contract elements to procedures and types within 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.

The 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 where the 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 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). Missing bindings were specified manually. That is, from the binding tree of Figure 1, it is always possible to select a TRM element to bind and then manually select the IUT entity it corresponds to (even overriding bindings obtained through binding modules). Thus, we view our overall approach to validation as being 'semi-automatic'.

The ABE uses a binding module (which, in essence, defines what constitutes a successful match) as follows: First, each contract within the TRM is bound to a type within the IUT. The ABE examines all types defined within the IUT. The types are compared by name and structure to determine the correct binding. Structural comparison entails looking for procedures within the type to determine if the observability methods and responsibilities defined within the contract could also be bound. Once all of the contracts have been bound, the ABE binds observability methods. As each such method represents an observation requirement imposed on the IUT, the corresponding IUT procedure must be side-effect free. That is, invocation of the IUT procedure bound to the observability method must not alter the state of the IUT. To enforce this, the ABE ensures any candidate IUT procedure is indeed side-effect free (a non-trivial technical detail explained elsewhere [15]). Once a set of candidate query-methods is selected, procedure name, return type, and parameters (number, types, and order) are all examined to select a corresponding IUT procedure for binding. Following the binding of observability methods, the ABE binds responsibilities. Each responsibility is bound to one or more IUT procedures. The ABE begins by looking for a single procedure that can be bound to the responsibility. The procedure name, return type, and parameters (number, types, and order) for each IUT procedure are examined to find a corresponding match. If an individual IUT procedure cannot be located, the ABE begins to analyze groups of procedures that could be used in combination to create the required responsibility binding. Once all contracts, observability methods, and responsibilities are bound, the ABE will bind any remaining exported symbols using the same methodology for binding contracts to IUT types.

If at any point the selected binding module is unable to determine a binding for an element within the TRM, the module will skip the binding and move on. Once this automatic binding process ends, further bindings can be specified manually, or another binding module may be applied to the TRM.

Finally, each time the binding data is updated, the ABE will run to see if the updated binding data allows for additional bindings to be inferred. That is, if a binding is completed manually, it is possible that the ABE will infer several additional bindings at the same time. Also, because the ABE runs each time the binding data is updated, it is possible that if a binding is removed, the binding will be reestablished instantly because the ABE found a match. But once a binding has been established, either manually or via the ABE, future executions of the ABE will not change the binding. That is, the ABE only operates on bindings that have yet to be specified (unless the user asks for complete regeneration).

In summary, bindings not only eliminate a frequent and problematic traceability gap between a requirements model and an actual implementation to validate; they also enable the semantics of this model to be operational. That is, through bindings, all the static and dynamic checks, metric evaluators, and scenarios of an implementation-independent requirements model captured in our VF can be automatically instrumented in an actual IUT (as opposed to a model of an actual IUT) and automatically validated. Such validation results in the production of a report, which is displayed by the VF.

V. DISCUSSION AND CONCLUSION

The Validation Framework (VF) we propose allows for the validation of an implementation-independent Testable Requirements Model (TRM) against one or more candidate *actual* Implementations Under Test (IUTs) without the need for glue code to bridge from requirements to IUT. Nor is it

necessary to augment this requirement model with implementation-specific information (such as OCL constraints)[4]. From our viewpoint, the approach we propose is truly model-based: the requirements model is independent of any implementation but imposes (structural and/or behavioral) constraints (via pre- and post-conditions, invariants and grammars of responsibilities) on all candidate implementations that could be validated against this model.

As previously mentioned, we carried out significant initial testing on the VF. First, a comprehensive suite was used for the ACL compiler. Then, five extensive case studies were developed to verify the handling of static and dynamic checks, as well as scenario monitoring and the evaluation of metrics. All this work ultimately demonstrated the feasibility of the approach we are proposing. But we needed to have the VF evaluated independently. To do so, we asked the students of a graduate course one of the authors was teaching this term to learn the ACL and the VF in order to assess their usefulness. Each student had to choose a domain, model the requirements of this domain via ACL, and finally bind their TRM to different implementations in order to obtain several evaluation reports. The domains they selected ranged from simple data types (e.g., developing a hierarchy of contracts for a binary search tree and some its well-known specializations such as AVL trees) to simple games (e.g., BlackJack) to a subset of an actual military system! The resulting assessments from the students were surprisingly convergent:

- The semantics of ACL were seen has generally necessary and sufficient across the different selected domains. Time-constraints did not make it possible however to explore the use of plug-ins.

- The instrumentation of an actual IUT by the VF takes place at a low level where highly technical details (such as the ordering of instantiations) can complicate drastically reliable scenario monitoring. Bugs were discovered and fixed, but they showed that the VF is tightly coupled to the inner workings of .NET and that maintenance of the VF requires great technical expertise in this domain. This is especially the case for the creation and use of the test harness capable of monitoring complex scenarios (which most MBT approaches simply do not tackle).

- While automatic binding generally works, subsequent user-specified modifications to generated bindings were arduous at times. Users appreciated the fact that the VF does allow binding a responsibility to a procedure of a totally different name. But the management of bindings over several iterations of TRM refinement was deemed rather 'primitive'. More work is required on both binding generation and binding management.

- Integration within Visual Studio was greatly appreciated but it was noted by some that some features of the latter (such as Intellisense) were not available in the VF. Such additions are feasible and will be included in a subsequent release of our tool.

- There was some 'trial and error', unless bindings were specified entirely manually. Furthermore, some users complained of having to 'retrofit' the implementation at hand with code to handle the observability requirements spelled out in a TRM. In fact, this is expected and likely unavoidable: because the TRM is decoupled from any implementation, it must define a 'validation interface' (especially via observability methods) that IUTs will have to support. Such a strategy is also at the root of several design patterns [28] (which 'impose' an interface on a set of classes by defining an abstract parent class for all of them).

- All users complained that if their contracts compiled but did not run correctly, a) no debugging was available and b) the generated reports were not useful in locating mistakes, but merely in reporting their presence. However, the fact is that the VF is a program that stops and monitors the execution of another program (the IUT). As such, it would be extremely (nay impossible) difficult to support debugging the execution of the VF (which, itself, can be conceptualized as a debugger).

- Some users felt they had to modify their IUTs in order for the latter to "bind correctly" to their TRM. Clearly this is wrong: it is the TRM that has to be abstract enough to accommodate one or more IUTs. But this misconception emphasizes the reality of using our current VF: the validation of an IUT largely depends on the ability to bind that IUT to a TRM. But this is easier said than done. For example, the TRM does not offer 'pointer semantics', which eliminates any hope of binding to non-managed C++. In turn, this limits the applicability of the VF (for example, to the validation of C++ data type libraries such as STL and BOOST, and other non-managed sources).

In the end, we must conclude that, while the proposed approach and its corresponding tool do offer a novel solution for the semi-automated validation of a requirements model, in practice our work turns out to be very specific to the .NET platform and to managed (i.e., garbage-collected) languages such as C#.

VI. ACKNOWLEDGMENTS

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

[4] As previously mentioned, it is quite problematic to include implementation-specific details in a requirements model.

VII. REFERENCES

[1] Nuseibeh, B.A, Easterbrook, S.M.: *Requirements Engineering: A Roadmap*. In Proceedings of the International Conference on Software Engineering (ICSE-2000), 4-11 June 2000, Limerick, Ireland, ACM Press.

[2] Meyer, B.: The Unspoken Revolution in Software Engineering. In *IEEE Computer*, vol. 39, no. 1, pp. 121-123.

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

[4] Beck, K.: *Test-Driven Development: By Example*. Addison-Wesley, November 2002.

[5] Bertolino, A.: *Software Testing Research: Achievements, Challenges and Dreams*. In proceedings of the Future of Software Engineering, FOSE'07, IEEE Computer Society Press, Minneapolis, MN, pp. 85-103, May 2007.

[6] Briand, L., Labiche, Y.: A UML-Based Approach to System Testing. In *Journal of Software and Systems Modeling*, vol. 1, no. 1, pp. 10-42, Springer, January 2002.

[7] The Object Management Group (OMG): *The Object Constraint Language (OCL) Specification*. Version 2.0, OMG Document #06-05-01, May 2006.

[8] Dresden University of Technology: The Dresden OCL Toolkit. DOI= http://dresden-ocl.sourceforge.net/index.php

[9] Klasse Objecten: The Octopus Tools, DOI= http://www.klasse.nl/octopus/index.html

[10] 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.

[11] Grieskamp, W.: *Multi-Paradigmatic Model-Based Testing*, Technical Report #MSR-TR-2006-111, Microsoft Research, August 2006.

[12] Microsoft Spec Explorer 2010, http://msdn.microsoft.com/en-us/devlabs/ee692301.aspx, accessed December 2009.

[13] The Validation Framework, http://vf.davearnold.ca accessed December 2009.

[14] Microsoft Research: *Microsoft Phoenix Research Development Kit* https://connect.microsoft.com/Phoenix accessed December 2009

[15] Arnold, D.: *An Open Framework for the Specification and Execution of a Testable Requirements Model*, Doctoral dissertation, School of Computer Science, Carleton University, April 2009.

[16] Wirfs-Brock, R. and McKean, A., *Object Design: Roles, Responsibilities and Collaborations*, Addison-Wesley, 2002.

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

[18] Amyot, D., Weiss, M., and Logrippo, L.: Generation of test purposes from Use Case Maps. In *Journal of Computer Networks*, vol. 49, no. 5, pp. 643-660, Elsevier, 2005.

[19] Ryser, J., Glinz, M.: SCENT: *A Method Employing Scenarios to Systematically Derive Test Cases for System Test*. Technical Report. University of Zurich, 2003. DOI= http://www.ifi.uzh.ch/rerg/research/scent/

[20] Weidenkaupt, K., Pohl, K., Jarke, M., and Haumer, P.: *Scenario Usage in System Development: A Report on Current Practice*. In proceedings of the 3rd International Conference on Requirements Engineering: Putting Requirements Engineering to Practice (ICRE'98), pp. 222-241, April 1998.

[21] Jacobson, I.: *Object-Oriented Software Engineering*. ACM Press, New York, 1992.

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

[23] Helm, R., Holland, I., Gangopadhyay, D.: Contracts: *Specifying Behavioral Compositions in Object-Oriented Systems*. In proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA'90), pp. 169-180, October 1990.

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

[25] Corriveau, J.-P. and Shi, W., *A Scenario-Driven Approach to Model-Based Testing*, submitted to the Third International Conference on Software Testing, Verification and Validation, Paris, April 2010.

[26] Corriveau, J.P.: *Testable Requirements for Offshore Outsourcing*. In proceedings of SEAFOOD'07, Zurich, February 2007.

[27] International Telecommunications Union (ITU): *User Requirements Notation* (URN). ITU-TS Recommendation Z.151, 2008.

[28] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.