Scenario-Based Validation

Beyond the User Requirements Notation

Dave ARNOLD and Jean-Pierre CORRIVEAU
School of Computer Science
Carleton University
Ottawa, CANADA
{darnold, jeanpier}@scs.carleton.ca

Wei SHI
Faculty of Business and IT
UOIT
Oshawa, CANADA
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 test cases are not directly traceable back to the actual behavior and components of an IUT. Furthermore, extremely few approaches tackle non-functional requirements. Indeed, the User Requirements Notation (URN) is one of few proposals that address the modeling and validation of both functional and non-functional requirements. But if the URN is to support traceability and executability of tests cases with respect to an actual IUT, then the "URN puzzle" must be modified: it must be augmented with a testable model for functional and non-functional requirements, an IUT, and explicit bindings between the two. We explain how these three additions are used in our implemented framework in order to support scenario-based validation.**

Keywords-validation; scenarios; contracts; user requirements notation; model-based testing

I. INTRODUCTION

A quality-driven [1] approach to software development and testing demands that, ultimately, the requirements of stakeholders be validated [2] against the actual behavior of an implementation under test (IUT). Thus, the general problem at hand is test case generation and execution.

From this viewpoint, we remark that modeling tools are used to create models of (some aspects of) an IUT. Examples of modeling tools include IBM Rational Rose [3], Borland Together [4], and Telelogic's (now IBM) Tau [5], among many others. 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 generally do not have the ability to generate test cases, let alone run and monitor them. Consequently, they are not relevant here.

Similarly, a test automation framework (e.g., Rational Robot by IBM [6]) accepts tests that already have been manually created, automatically generated, or pre-recorded. Most importantly, such test cases must be readily executable on an IUT. The automation framework then executes the test sequences without human interaction. Such an approach bypasses a key problem, namely the generation of executable test cases. Consequently, we will not discuss further such frameworks.

Code-based testing constitutes one approach to software testing. But a code-centric testing method, such as in test-driven design [7], does not explicitly model the requirements of stakeholders. Furthermore, code-based testing tools (such as JUnit [8] and its several adaptations to different languages, and very recently AutoTest [9]) allow only for unit tests [2] (i.e., tests pertaining to a procedure, not to a scenario/cluster [2]) to be specified in, or automatically generated from, code. And such tests are implementation-specific.

In model-based testing [10] (MBT), the requirements of stakeholders are to be captured in implementation-independent models from which tests are to be extracted to drive the task of validation [2]. Such requirements can be categorized into functional and non-functional[1] ones (such as performance, usability, etc.), the former receiving almost all of the attention in modeling and validation literature.

An MBT tool uses various algorithms and strategies to generate tests from a behavioral model of the IUT. Test cases derived from such a model are functional tests at the same level of abstraction as the model. From this viewpoint, MBT approaches can be separated into those that support executable test cases (e.g., [11, 12]), and those that do not (e.g., [2, 13, 14]). It is important however to clarify what is exactly meant by executability in the former category. Existing MBT tools that support executability are systematically grounded in state-based semantics. That is, the execution of the generated tests occurs in the semantic space of the specification language and is most often rooted in the concept of state exploration [11]. For example, in Spec# [11], executability relies on queues of *unreceived* messages (a notion incompatible with any *actual* execution of code) as well as on a global state explorer (which must deal with the difficult problem of state explosion [11]). The point to be grasped is that there is no traceability between test cases generated and executed (with respect to a space of states), and executions of an actual implementation-under-test (IUT). But traceability between a model, the test cases

[1] Also called 'quality of service' requirements.

generated from it, and an IUT, is necessary if stakeholders are to partake in the validation of this IUT, and in particular, in the selection of test cases.

The question then is two-fold: is it possible to have an approach to MBT that a) addresses both functional and non-functional requirements and b) relies on test cases executable against an actual IUT? Our claim is that it is. In order to support this contention, we will introduce here a requirements specification language and explain how it is used for validation. Our proposal proceeds from the User Requirements Notation [15] (URN), one of few MBT approaches that address the modeling and validation of both functional and non-functional requirements. An overview of this conceptual and methodological framework is provided in Fig. 1 (from [15]).

Currently, the URN offers limited test case generation [16], and the executability of such test cases is not addressed. Our contention in this paper is that if the URN is to indeed support the validation of the requirements of stakeholders against the actual behavior of an IUT, then the "URN puzzle" of Figure 1 must be modified. In the next section, we explore the nature of these modifications, which we will motivate from the need to support the executability of test cases against an actual implementation. In particular, we will justify why we add both a testable requirements model (TRM) and an IUT to the puzzle, as well as a) our validation framework (VF) [17] and in particular b) explicit bindings between a TRM and an IUT. Then, in section 3, we present an example of our URN-based approach to validation using our VF.

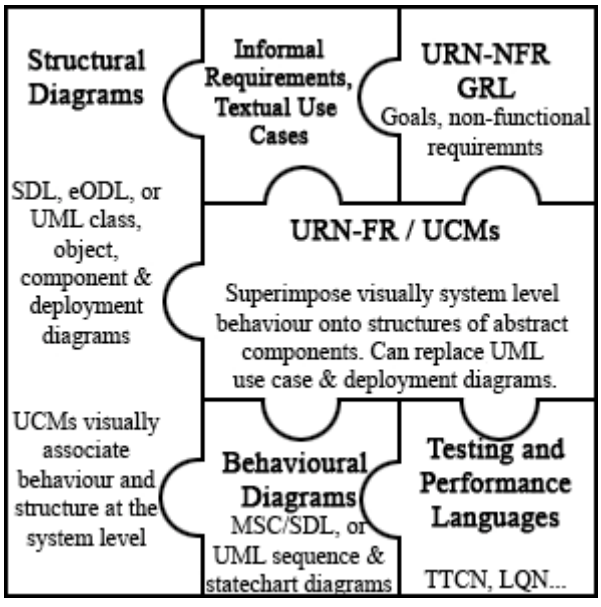

Figure 1. The URN modeling puzzle

II. AUGMENTING THE URN

A. Why Start with the URN?

The URN is a two-headed proposal. URN-NFR addresses non-functional requirements (NFRs), capturing them using the Goal-Requirements Language (GRL) [18]. Such a model aims at highlighting how some facets of a system (*e.g.*, tasks, procedures) contribute (positively or negatively) to the satisfaction of NFRs. Little exists in terms of proposals for the systematic production of tests from GRL models.

Functional goals (FRs) are captured using Use Case Maps (UCMs) [19], which consist of *scenarios* forming temporal flows of *responsibilities*.

Both URN-NFR (*i.e.*, GRL) and URN-FR (*i.e.*, UCM) models are taken to proceed from informal requirements organized into textual use cases [2]. Thus, URN is a *scenario-driven* approach to MBT, in contrast to state-based approaches to MBT. This is an important difference, especially with respect to the validation of the requirements of stakeholders. In particular, Grieskamp [20], who developed SpecExplorer [11] at Microsoft, remarks that the low adoption rate of state-based MBT tools in industry depends, amongst other factors, on the learning curve associated with such semantics and the lack of support for more stakeholder-friendly scenario-based semantics.

Thus, URN constitutes an interesting starting point for an MBT tool for several reasons: i) it is scenario-based, requirements oriented (i.e., high-level) and implementation independent, ii) it does generate functional test cases [16], iii) it does address the modeling of non-functional requirements and iv) as suggested by Fig. 1, it integrates well [15] in a development process that aims to address the whole modeling/testing enterprise (from use cases 'down to' detailed Message Sequence Charts [21] and test languages such as TTCN [22] and LQN [23][2]).

Furthermore, since a use case map can be conceptualized as a *grammar* of responsibilities [19], the technique of *path sensitization* [2] can be used to generate a test suite with respect to a specific *coverage criterion* [2], as explained in [16]. The point is that there exist algorithms to 'cover' use case maps via a suite of generated test cases (as there are for Binder's extended use cases [2] and use cases augmented with contracts [13]). Similarly, test case generation algorithms for state machine coverage have been extensively documented (e.g., chapter 7 of [2]). The point then is that test case generation is *not* a significant hurdle for validation (and thus, due to space restrictions, will not be discussed further here). (Details of our approach to test case generation are available in [24].)

It is the executability of such test cases against executions of an implementation under test (IUT) that is challenging. In particular, such executability requires the automatic instrumentation of test cases in an IUT. For

[2] More precisely, as Buhr [19] explains, use cases are first expressed as temporal flows of responsibilities (e.g., one flow per scenario of a use case) and then, and only then, are these responsibilities packaged into objects and ultimately classes, and their detailed interactions worked out.

example, it's one thing to use path sensitization to generate test cases from UCMs. It's another to develop a profiler that can check whether or not an actual execution of the IUT conforms or not with the scenario models of such UCMs. In fact, unless the semantic elements of UCMs are associated with actual method calls and execution paths, such profiling is not possible.[3] Furthermore, the few tools that do support such profiling are, in practice, quite limited. Consider, for example, Rational's TAU and its Validator [5]. TAU 'supports' Message Sequence Charts (MSCs) [21], a semantically rich scenario-modeling notation (which is too detailed to serve for requirements modeling). However, scenario profiling is *actually* restricted to the simplest semantic elements of MSCs. And profiling is *not* scenario-based but state-based! In other words, what is monitored are sequences of states, not a path of execution (consisting of a sequence of events/method calls).

We also remark that TAU 'supports' the automatic generation of tests captured in TTCN [22], a standard test notation. But the semantics of TTCN *cannot* handle the complexities of scenario notations such as UCMs and MSCs: TTCN was not meant to and cannot tackle the validation of complex scenarios (whereas our proposal does [24]).

In the rest of section 2, our task thus consists in explaining how, starting with the URN, we can get to test cases executable against an IUT.

B. On A Traceable Requirements Model

The URN puzzle is our starting point as it is, we repeat, one of few proposals that address both functional and non-functional requirements.

As a first modification, we propose to reorganize the URN puzzle so that it separates two distinct viewpoints, namely the one of the stakeholders and the one of the developers. This initial modification is given in Fig. 2.

Fig. 2 emphasizes that models relevant to stakeholders must be independent of any particular implementation. It also conveys the fact that such models must address both functional and non-functional requirements (which proceed from informal requirements). It does not address the generation of test cases executable against an IUT. For that, we require another model, which we will call a *Traceable Requirements Model* (TRM). Let us elaborate on the properties of such a model.

First and most importantly, this model must be *traceable*: it must be possible to associate its semantic elements to elements of an IUT. (Ideally, this process should be fully automated. More on this shortly.) In our opinion, it is this traceability that enables bridging between the stakeholders' viewpoint and the one of the IUT developers, as will be explained shortly. The need for traceability between a TRM and an IUT immediately suggests adding *both* a Traceable Requirements Model and an IUT to the elements already in Fig. 2.

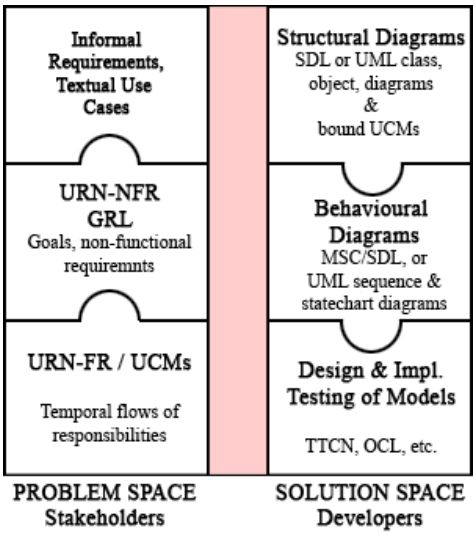

Figure 2. The Two Viewpoints of the URN puzzle[4]

Second, the TRM must also be a *unified* requirements model inasmuch as its semantics must address both FRs *and* NFRs. This is an important decision: having a single model for FRs and NFRs allows us to associate *metric evaluators* to responsibilities and scenarios, as will be illustrated in section 3.

Third, the TRM must be semantically *comprehensive*: To enable the evaluation of NFRs, the TRM supports metric evaluators, as just mentioned above. In addition, it is rooted in the semantic notions of *responsibilities* and *scenarios* found in UCMs (from which it is to be derived). Following [13], we further enhance the scenarios of a TRM with elements of design-by-contract [27] (i.e., pre- and post-conditions, invariants). (This allows us to reuse the test case generation algorithms proposed in [13].) Following Ryser and Glinz [28], we also include semantics for the all-important (and generally forgotten) *inter-scenario relationships*. Static checks (see section 3) complete the semantics of our TRM in order to support a simple form of static analysis [17]. In our current implementation, all these semantic elements are expressed in our own requirements specification language called *Another Contract Language* (ACL) [17].

Our TRM is textual (*i.e.*, non-diagrammatic). Thus, stakeholders are to initially develop use cases, then GRL and UCM diagrams, and then, and only then, refine the responsibilities, scenarios and metrics identified in such diagrams into a TRM. This step is *not* automated and requires the participation of someone familiar with a) the semantics of ACL, b) the test case generation process we support [24] (in order to help stakeholders in selecting test

[3] It must be emphasized that by grounding executability in the semantics their specification languages (as opposed to *executions* of an IUT), state-based MBT tools essentially eliminate the whole issue of instrumentation and profiling, at the expense of traceability between test cases and actual executions of an IUT.

[4] The OCL is included as a testing language. Its usage is for unit testing and OCL is IUT specific. It does not tackle scenario conformance [25]. SDL [26] is an older state-based modeling language.

cases) and c) the binding process used to link elements of a TRM to elements of an IUT (as explained shortly). Details of how a TRM is to be 'derived' from URN models are not relevant here. It only matters that the semantic closeness of our TRM to URN models makes this quite straightforward.

Static checks correspond to structural queries on an implementation, whereas design-by-contract elements [18] (i.e., pre- and post-conditions, invariants) and the evaluation of metrics constitute types of dynamic checks that query an execution of an IUT. Most importantly, from an operational standpoint, we view scenarios as complex dynamic checks involving run-time monitoring (i.e., profiling).

Validating static and dynamic checks against an IUT produces a 'validation report.' This report must provide the outcome of each static and dynamic check, and metric evaluation(s) for a specific execution of an IUT. Static and dynamic checks either pass or fail. That is, a check must be directly executable and have a well-defined outcome. If a check fails, information regarding the failure (e.g., values used during the evaluation of a pre- or post-condition) should be reported, but the execution proceeds if possible. Conversely, the result of a metric evaluator is a value (such as a performance counter) that is directly included in the validation report.

Because a TRM includes scenarios, a validation report must also provide information pertaining to the monitoring of the execution of each scenario's *instances* and of their corresponding responsibilities. Each scenario may be triggered several times during an execution, leading to several distinct instances of that scenario. Run-time monitoring involves tracking scenario instances and knowing how to match them against parts of the execution of an IUT. When the execution path of an IUT differs from a relevant scenario, the exact responsibility where execution deviated is indicated in the report. Further scenario execution information, such as the number of times a particular scenario has executed, and what data was used/changed during its execution is also included in the report.

Our Validation Framework (VF) supports all the reporting tasks outlined above. The resulting report is known as a Contract Evaluation Report (CER) [17].

In order for the validation of a TRM against an IUT to be possible, we require a) a mechanism to link a TRM to an IUT and b) a VF that provides full instrumentation for a specific TRM/IUT pair. We now explore these two issues.

C. *Bridging the Traceability Gap*

We require that the semantics of the TRM be *decoupled* from any particular implementation (and programming language) so that a single TRM may be tested against several candidate implementations. The immediate question then is how to bridge between an implementation-independent TRM (i.e., a stakeholder's viewpoint) and an IUT (i.e., the developer's viewpoint)? The answer lies in the idea of *binding* elements of the TRM to actual methods within the IUT. More specifically, some elements of the TRM must be explicitly linked (by the provider of an IUT) to corresponding methods within this IUT, and some methods may have to be added to the IUT to enable the observability [2] that is required to determine if a given requirement is satisfied.

Bindings act as a mapping between the TRM and the IUT. Relevant elements of the TRM must be bound to concrete IUT counterparts. Such a binding process is ideally automated, thus providing an automatic connection between the TRM and IUT. Our VF includes such binding support. Let us elaborate.

Our binding tool allows for the automatic specification and display of binding information. Such binding information links elements of a TRM to concrete elements located within an IUT. Bindings can be specified manually or using the Automated Binding Engine (ABE). The ABE is not limited to a specific binding algorithm. Rather, it is open in nature and uses extension modules that are developed using a software development kit [17]. We have implemented two ABE extension modules as part of the current VF. Details regarding how bindings are inferred can be found elsewhere [17]. Most importantly, bindings allow the TRM to be independent of implementation details, as specific IUT method names, and parameter types/orders used within the IUT *do not* have to correspond to a similarly named TRM artifact. (In fact, one responsibility can be bound to several distinct procedures and even to a sequence of procedures [17].) In addition, such a binding process allows several candidate IUTs to be verified against a single testable model.

Once binding information is specified, our VF is able to execute an IUT against a TRM. The process begins with the execution of static checks. We first use Microsoft's Phoenix Research Development Kit [29] to gather structural and behavioral elements of the IUT (which is input as an executable, not source code). Once such information is gathered, the actual static checks are executed. After all of the static checks have been executed, the VF then executes the IUT, by launching it as a new process. Our VF acts as a specialized debugger and profiler, keeping track of instance creation, method invocation, and instance destruction. In addition, the VF will pause the execution of the IUT as needed in order to execute various checks. At the same time the IUT is launched, our VF also maintains a separate execution environment for the TRM. This environment contains any instances of TRM artifacts (e.g., scenarios) that have been created as a result of IUT execution. Once execution of the IUT has completed, the VF gathers metric information obtained during execution of the IUT and performs metric evaluation. Such evaluation consists of the VF invoking metric evaluators to analyze and report on the gathered metric information. Finally, a CER is generated displaying test outcomes covering all aspects of a candidate IUT executed against the TRM. (Details regarding the specific CER contents can be found in [17]).

Fig. 3 summarizes how using a TRM, an IUT, bindings between the two, our VF and the report it generates, fits into the URN-based approach we propose.

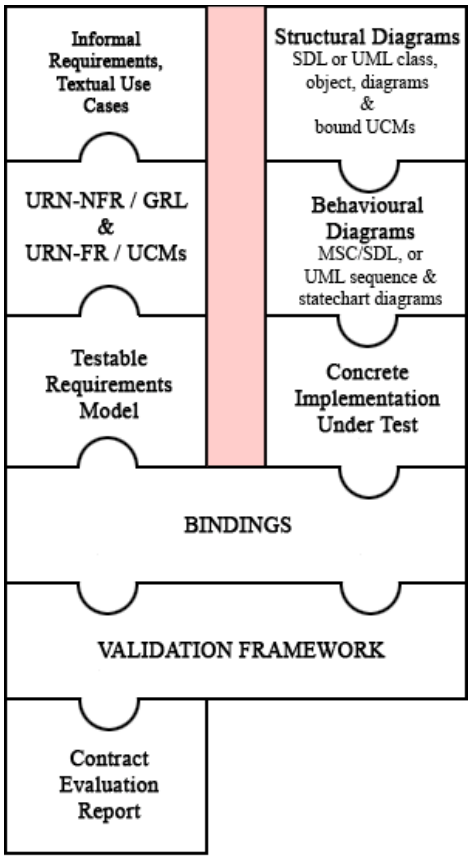

Figure 3. Completed URN puzzle

III. An Implemented Example

A. URN Models for a Simple Container

Our example models a *very* simple container able to add, remove, and search for items (without multiple occurrences of an item, for simplicity). (See [17] for more extensive case studies.) Using the URN we begin with a UCM representing how an external entity, the 'user,' interacts with our container. As Fig. 4 illustrates, the user is able to add, remove, and search for items in any order. Also, the user may choose to not interact with the container (which corresponds to the empty 'center' path in Fig. 4).

This UCM represents only some of the functional aspects of the container (omitting, for example, creation and destruction).

To model non-functional aspects, we have created the minimalist GRL diagram shown in Fig. 5. This figure defines a goal named 'Item Search' representing the identical functional aspect found within our UCM. The GRL diagram also includes a softgoal (i.e., NFR) named 'Performance.' In our simple example, we wish to have high performance (that is, achieve a constant search time) when we are searching for an item. The GRL diagram illustrates how we believe three potential implementation strategies will affect the search performance softgoal.

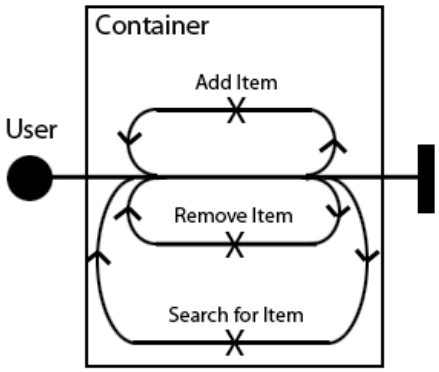

Figure 4. Container example UCM diagram

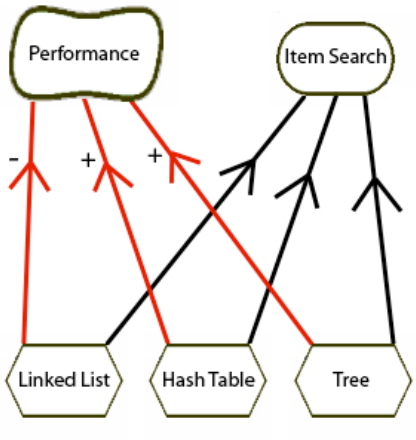

Figure 5. Container example GRL diagram

The diagrams in Figs. 4 and 5 capture at a high level of abstraction the requirements of our very simple example. Most importantly, we observe that neither diagram is testable per se. Furthermore, should test cases be generated outside of our VF, they would necessarily be *disconnected* from actual implementations of such a container. Not only would a developer have to associate the responsibilities of the UCMs to actual methods in an IUT, but also instrumentation code would be required to observe that an item is properly added, looked up and removed. Code would also be required to verify the validity of an actual sequence of method calls against the grammar of responsibilities captured in the UCM. For GRL, code would be required to measure search performance (*e.g.*, in terms of look-ups in the container). All of this so-called 'glue code' would most likely be very implementation specific, having to be considerably modified across candidate IUTs for the container.

In our opinion, developing a TRM using our framework constitutes a better alternative to IUT-specific 'glue code' because the former enables the validation of a single TRM

across different IUTs[5]. This is made possible by having our VF support the *automatic instrumentation* of a TRM in an IUT, once bindings have been supplied. From a validation viewpoint, a stakeholder is 'shielded' from IUT-specific considerations and only focuses on refining URN models into a TRM: it is the developer of an IUT who must supply the IUT-specific bindings that enable automatic instrumentation.

B. *Creating a TRM for the Container Example*

We begin our example TRM with a first rather abstract version that proceeds from the Use Case Map. This TRM is shown in Fig. 6 below:

```
Contract Container {
 Scalar Integer size;
 Observability Boolean
   HasItem(tElement item);
 Responsibility Add(tElement item) {
    Pre(HasItem(item) == false);
    Execute();
    size = size + 1;
    Post(HasItem(item) == true);
 }
 Responsibility Boolean
   Search(tElement item){
    Pre(item not= null);
    Execute();
    Post(HasItem(item) == value);
 }
 Responsibility
   Remove(tElement item) {
    Pre(HasItem(item) == true);
    Execute();
    size = size - 1;
    Post(HasItem(item) == false);
 }
 Exports {
    Type tElement { not context; }
 }
}
```

Figure 6. A TRM proceeding from use cases

In ACL, responsibilities and scenarios are grouped into contracts. Our TRM starts with the definition of a single contract named *Container*. The TRM can contain any number of contracts; however we will only use one in our example. The body of our contract begins with the definition of a contract instance variable. Contract instance variables provide a way to store information within the TRM. It should be noted that contract variables are separate from variables found within the IUT (and are maintained by our VF). Our contract instance variable is named 'size' and will be used to store the number of items that should be in the container at a given time (according to the TRM).

[5] As previously mentioned, a TRM must also support automatic test case generation which, without going in any further details, takes the form of test procedures automatically inserted in, and called from, the main procedure of the IUT.

Next, the contract defines an observability procedure (hereafter simply 'an observability'). Such a procedure represents an observation requirement imposed on the bound IUT. The 'HasItem' observability is used to check if a given item is stored within the container. Note that the item type is 'tElement.' The 'tElement' identifier represents the type of element that is being stored in the container. Details regarding how the 'tElement' type is bound to the IUT will follow shortly.

The 'Add' responsibility represents the task of adding a new item to the container. The item to add is represented by the single parameter passed to the responsibility. The body of the responsibility begins with a single precondition to ensure that the item that is being added to the container is not already stored in the container. The *Execute* statement, indicates the point where the bound IUT method(s) will execute. That is, the IUT will be instructed to carry out at that point the actual task of adding the requested item to the container. Next the 'size' contract variable is incremented to reflect the addition of the new item. Finally a post-condition is specified to ensure that the given item has indeed been added to the container.

The 'Search' responsibility returns a Boolean value indicating if the given item is stored within the container. The body of the responsibility begins with a precondition to ensure that the given item is valid. The *Execute* statement follows and instructs the IUT to perform the actual search. Finally a post-condition is specified to ensure that the value returned by the IUT method (a Boolean in this case), denoted by the 'value' keyword, is the same as the one obtained from the 'HasItem' observability.

The final responsibility, 'Remove,' removes the given item from the container. The body of the responsibility contains a single precondition to ensure that the item requested for removal is actually in the container. The *Execute* statement removes the actual item from the IUT. Finally, the 'size' contract variable is decremented to reflect the removal, and a post-condition is specified to ensure that the item was actually removed from the container. The specification of these three responsibilities proceeds directly from the UCM of Fig. 4.

The final semantic element within our contract is an Exports section. Each contract may contain at most one Exports section, which is used to define symbols that are external to the contract, that is, types that must be bound to concrete IUT elements for contract execution. Our contract contains a single export entry for the 'tElement' symbol. The export indicates that the 'tElement' symbol must be bound to a type within the IUT, and that this type cannot be the same type as the one representing the container. That is, this simple TRM does not support a container of containers.

The key point for now is that this TRM, despite its simplicity, proceeds from a UCM and can be bound to an IUT, thus enabling (albeit *very* high-level) validation.

Our initial TRM does not take into account scenarios or NFRs. We will now refine it to include the additional requirements captured by the UCM of Fig. 4, and the GRL diagram of Fig. 5. The resulting TRM is shown in Fig. 7 (refinements in bold).

```
Contract Container {
 Scalar Integer size;
 Scalar Timer search_timer;
 Observability Boolean
    HasItem(tElement item);
 Responsibility Add(tElement item) {
    Pre(HasItem(item) == false);
    Execute();
    size = size + 1;
    Post(HasItem(item) == true); }
 Responsibility Boolean
  Search(tElement item){
    Pre(item not= null);
    search_timer.Start(item);
    Execute();
    search_timer.Stop(item);
    Post(HasItem(item) == value); }
 Responsibility Remove(tElement item) {
    Pre(HasItem(item) == true);
    Execute();
    size = size - 1;
    Post(HasItem(item) == false); }
 Scenario AddSearchRemove {
    once Scalar tElement x;
    Trigger(Add(x)),
    Search(x)*,
    Terminate(Remove(x));
 }
 Scenario Lifetime {
    Trigger(new()),
    (
      Add(dontcare) | Search(dontcare) |
      Remove(dontcare)
    )*,
    Terminate(finalize()); }
 Metric List Integer TimesToSearch() {
    search_timer.Values(); }
 Reports {
    ReportAll(
       "The average search time is: {0}",
       AvgMetric(TimesToSearch()));}
 Exports {
    Type tElement { not context; }}}
```

Figure 7. Refined TRM

The refinements begin with the addition of the 'search_timer' contract instance variable. The variable will be used to store the amount of time required for an item to be found within the container. That is, the timer will be used in determining the satisfaction of the performance softgoal (as explained shortly).

The 'Search' responsibility is refined to include the starting of a timer keyed on the item being searched for. The same timer is stopped following the *Execute* statement. That is, the timer will record the amount of time that it takes the IUT to search for each item.

Our refined contract contains two scenarios: the first, named 'AddSearchRemove,' specifies the behavior of a single item within the container, whereas the second, named 'Lifetime,' specifies how the container can be used to store any number of items as per the UCM in Fig. 4. The body of the 'AddSearchRemove' scenario begins with the declaration of a scenario variable, named 'x.' The purpose of this scenario variable is to represent an individual element of the container. Note the use of the 'once' keyword. This keyword states that the scenario variable can only be assigned a single time. If additional assignments occur the contract will fail[6]. A scenario begins when its triggering event occurs. The triggering event for the 'AddSearchRemove' scenario is the successful execution of the 'Add' responsibility. The 'x' scenario variable will be assigned the item added to the container. Each time a unique item is added, a new scenario instance is created. Next, the 'Search' responsibility can be executed zero or more times (denoted by the '*' operator). The 'x' scenario variable is used as a parameter for this responsibility to ensure the search is performed on the same element that was added to the container earlier in that specific scenario. Finally, the scenario terminates following execution of the 'Remove' responsibility for that specific item. Any scenarios that have not terminated, or do not follow the specified scenario grammar are taken to have failed. Such a failure is logged in the CER.

The 'Lifetime' scenario addresses the lifetime of the container. The scenario is triggered when the special 'new' responsibility is executed. That is, the scenario is triggered when a new container is instantiated. Once the scenario is triggered, any number of add, search, and remove operations may take place. The 'or' ('|') operator indicates that there is no specified ordering for the execution of these responsibilities. The whole group of responsibilities can execute zero or more times (denoted by the '*' operator). That is, it is possible that no operations are performed on the container (as per the UCM). Also, note the use of the 'dontcare' keyword: it indicates that the values of the parameters (*i.e.*, the items) for these responsibilities are not important from the viewpoint of this scenario. Finally, the scenario terminates after the 'finalize' responsibility has executed, that is, when the container is destroyed.

The previous constructs are used to specify functional aspects of the TRM. Our contract continues with non-functional constructs. The first of these is a metric named 'TimesToSearch.' This metric, when invoked, will return a list of integer values representing the times gathered when the 'Search' responsibility executed. These values are obtained by using the 'Values' method (predefined by our VF) on the 'search_timer' contract variable. (For more information on timers see [17].) The second non-functional construct is in the 'Reports' section. The 'Reports' section is evaluated after the IUT has finished executing so that any NFR information gathered during execution can be evaluated. In our example, the body of the 'Reports' section uses the built-in 'AvgMetric' metric evaluator to compute the average search time, obtained from the list of times found in the 'TimesToSearch' metric. The result of such metric evaluation is then displayed on the CER (for the user

[6] The rationale for such a keyword is that our implemented large case studies [17] suggest it is frequent that a contract or scenario variable is only assigned to once, and that additional assignments are in fact specification errors within a TRM.

to determine whether the softgoal is or is not satisfied). It should be noted that our VF uses an open semantic model so that user-defined and domain-specific metric evaluators can be defined via the VF and invoked from within a 'Reports' section (for details, see [17])

Our TRM could have been refined further with the addition of invariants, pre and post-conditions creating stricter responsibility definitions, *etc*. Such refinements were omitted due to space constraints. For the complete example, see [17].

To conclude, let us add that this TRM can also include inter-scenario relationships. For example, should we want to express that a test case is to involve at most 8 instances of the AddAndRemove scenario, and at most 4 of the ContainerLifeTime one, all executing independently, we would write: (where || denotes concurrent execution)

```
Interaction ContainerInter
  { Relation MultipleContainers
      { Contract Container c;
        c.AddAndRemove[8]||
         c.ContainerLifetime[4];} }
```

C. Binding the TRM to an IUT

Our VF uses the previously mentioned ABE modules to automatically infer bindings between a TRM and an IUT. For the purposes of our example, we will outline the bindings necessary to bind the contract of Fig. 7 to a candidate IUT. The complete binding table can be found in Table 1. The first element to be bound is the contract itself. Each contract must be bound to a type within the IUT. In our example, for simplicity, the contract is bound to a single IUT type representing a single specific container (*e.g.*, a linked list in C++). Once a contract is bound to an IUT type, elements of this contract are bound to IUT elements located within the IUT type bound to this contract. Such binding process begins with any export entries that may be present in the contract. Recall that our container example refers to a single export entry for the 'tElement' symbol. A binding will thus be required between the 'tElement' symbol and a type within the IUT representing the single element type stored within this particular container. Next, observability procedures are bound. Our contract specifies a single observability that will be, likely automatically, bound to a corresponding method within the IUT type bound to the contract. For an observability binding to be successful, the return type and parameter types (but not the name) of the observability and the IUT method must match. The IUT method must also be side effect free (which is enforced by the VF, see [17]).

Next bindings for the 'Add,' 'Search,' and 'Remove' responsibilities are handled. In our VF, a responsibility can be bound to any number of IUT methods. For the purpose of our simple example, we will assume a one-to-one binding. (Details for binding to several methods can be found in [17].) Such a binding will then require the IUT method to have the same return and parameter types as the responsibility in order to enable an automatic binding. This completes the binding process. The total number of bindings required in our example is six[7].

Once the binding process is complete, our VF automatically compiles the TRM, instruments it into the IUT, runs the IUT and evaluates this IUT against the compiled TRM producing a CER.

D. Validation of the Container Example

With the VF acting as a run-time monitoring system, we are able to capture events as the bound IUT executes. Each time a type is instantiated, the VF checks to see if any contract within the TRM is bound to that IUT type. If so, a new contract instance is created to represent the new IUT type instance. As the IUT continues to execute, any responsibilities that are bound to methods within the corresponding IUT type will be evaluated when their corresponding IUT methods execute. (Details regarding the evaluation of a bound responsibility and corresponding scenario grammar execution can be found in [17, 24].)

Once the IUT finishes its execution, the VF notifies any scenario instances that have yet to terminate that the IUT has completed, and that all such incomplete scenario instances fail. Next, the result of executing each scenario is written to the CER. Such information includes the execution grammar specified in the TRM, the actual execution trace, and any unexpected responsibilities or observable events.

The final step is the evaluation of the 'Reports' section possibly found in each contract (see Fig. 7). Such evaluation consists in the VF invoking the previously discussed metric evaluators to analyze and report on the metric information gathered during IUT execution. The gathering of all data used in the CER, including metrics specified in the TRM, is performed by the VF. The results of metric analysis are formatted (as per the string provided to the 'ReportAll' statement(s)) and written to the CER. Evaluation of each contact's 'Reports' section completes the process of validating a candidate IUT against a TRM. In other words, from our viewpoint, validation is incomplete unless its outcomes can be reported to stakeholders. Let us briefly elaborate.

E. The Contract Evaluation Report

The CER displays results covering all aspects of a candidate IUT validated against the TRM. The results of each contract, contract instance, observability, invariant, responsibility, scenario, and metric analysis are shown. Each distinct candidate IUT executed by our VF will have a separate CER. Multiple CERs can be compared to find differences between candidate IUTs (in order, for example, to decide which IUT to select based on the satisfaction of a softgoal).

The CER itself is presented using a tree that displays each element of the TRM in either green or red. Elements in green represent areas where execution of the candidate IUT followed the specified TRM. Elements in red represent areas

[7] 1 contract + 1 observability + 3 responsibilities + 1 export.

where execution of the candidate IUT deviated from the TRM. Upon selection, each node within the CER tree displays a report view showcasing precise execution details based on the currently selected tree item. Figure 8 shows both the tree and report views generated by the execution of our container TRM against an IUT using the bindings provided in Table 1. Fig. 8 displays the scenario instance view. The scenario instance view provides the actual triggering and termination events for a given scenario, as well as the exact execution trace that was observed during execution of the selected scenario instance. In addition, details regarding the execution of any pre- and post-conditions (and other dynamic checks, such as invariants, executed as part of the scenario instance) are also provided. The CER has a total of 22 distinct report views that can be displayed depending on the selection made in the report tree. (For details about information written to the CER and corresponding views, see [17].)

TABLE I. BINDING TABLE FOR THE CONTAINER TESTABLE REQUIREMENTS MODEL

TRM Element Name	TRM Type	IUT Bindpoint	IUT Type
Container	Contract	Examples::Container	Class
tElement	Exported Type	int Examples::ContainerItem	Class
Boolean Container.HasItem(tElement item)	Observability	bool Examples::Container.HasItem(Examples::ContainerItem)	Method
Void Add(tElement item)	Responsibility	void Examples::Container.Add(Examples::ContainerItem)	Method
Boolean Search(tElement item)	Responsibility	bool Examples::Container.Search(Examples::ContainerItem)	Method
Void Remove(tElementItem item)	Responsibility	void Examples::Container.Remove(Examples::ContainerItem)	Method

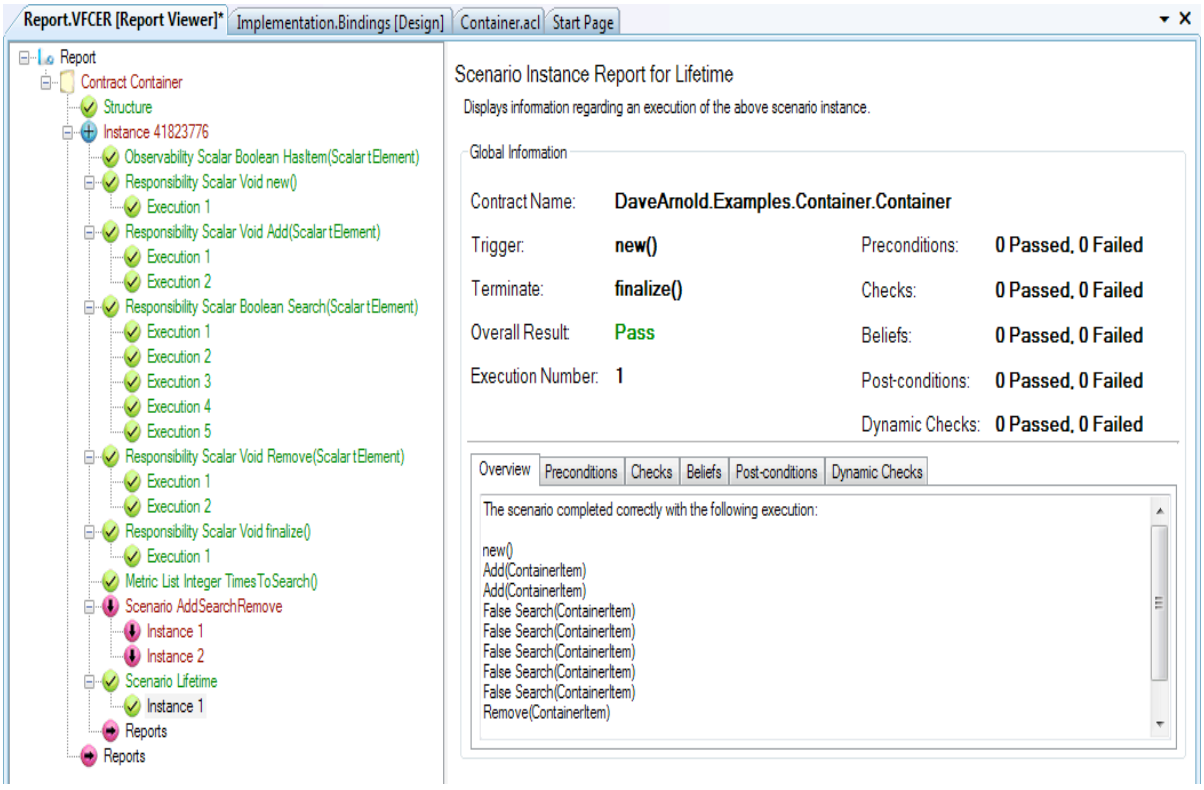

Figure 8. Validation Framework Report Viewer - Scenario Instance View

IV. CONCLUSION

Most current tools for model-based testing (MBT) are state-based. Such tools offer the advantage of formal semantics and well-established techniques for test case generation. But their executability is rooted in the semantics of their specification model (in particular, in the notion of state exploration). Furthermore, their adoption remains relatively low in industry, partly because of the preference of users for (often informal) scenario-driven semantics [1]. Notwithstanding the semantic features of our requirements language and the functionality of our tool that we cannot elaborate upon (due to space constraints), our claim here is that it is feasible to have a URN-based MBT approach that a)

allows its user to express the requirements of stakeholders first using the URN, then in a Testable Requirement Model (TRM), b) supports the automatic test case generation from this TRM (through path sensitization algorithms that we discuss at length elsewhere), c) bridges between the TRM and an Implementation under test (IUT) via the use of our binding tool, and d) enables the validation of the TRM against this IUT (through the automatic instrumentation of the static and dynamic checks, as well as scenarios of the TRM in this IUT). We have implemented a validation framework to support this MBT approach. Its most distinguishing feature, in our opinion, are the semantics of the requirements language it supports for the specification of a TRM: whereas state-based MBT approaches have what can be tested proceed from the semantics of state exploration, the semantics of our requirements language proceed from what can be automatically instrumented and monitored at run-time.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Meyer, B. "The Unspoken Revolution in Software Engineering", *IEEE Computer*, January 2006, vol. 39, no. 1, pp. 121-123.

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

[3] IBM: *Rational Rose*. http://www-306.ibm.com/software/awdtools/developer/rose/index.html, accessed October 2009.

[4] Borland: *Borland Together*. http://www.borland.com/us/products/together/index.html, accessed October 2009.

[5] IBM: Rational TAU, http://www-01.ibm.com/software/awdtools/tau/ accessed October 2009.

[6] IBM: Rational Robot, http://www-01.ibm.com/software/awdtools/tester/robot/ accessed October 2009.

[7] Beck, K., Test Driven Development By Example, Addison-Wesley Professional, Reading, MA, 2002.

[8] JUnit: http://www.junit.org/ accessed October 2009.

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

[10] Bertolino, A., "Software Testing Research: Achievements, Challenges and Dreams", *Future of Software Engineering (FOSE'07)*, IEEE Press, Minneapolis, May 2007, pp. 85-103.

[11] Campbell, C., *et al*., *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*. Microsoft Research Technical Report #MSR-TR-2005-59, May 2005.

[12] Foster, H., Uchitel, S., Magee, J. Jeff and Kramer, J., *A Tool for Model-Based Verification of Web Service Compositions and Choreography*, International Conference on Software Engineering (ICSE) 2006.

[13] Nebut, C., Fleurey, F., Traon, Y.L. and Jezequel, J., "Requirements by Contracts allow Automated System Testing", *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE)*, Washington, DC, November 2003, pp. 85-105.

[14] Somé, S., *Use Cases based Requirements Validation with Scenarios* 13th IEEE International Conference on Requirements Engineering (RE 2005), September 2005.

[15] Amyot, D., "Introduction to the User Requirements Notation: learning by example", *Computer Networks: The International Journal of Computer and Telecommunications Networking*, Elsevier Inc., New York, June 2003, vol. 42, no. 3, pp. 285-301.

[16] Miga, A., "Application of Use Case Maps to System Design with Tool Support", *Masters Thesis*, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, October 1998.

[17] Arnold, D., The Validation Framework and its examples, http://vf.davearnold.ca/, accessed January 2010.

[18] Yu, E., and Mylopoulos, J., "Why Goal-Oriented Requirements Engineering", *Proceedings of the 4th International Workshop on Requirements Engineering: Foundations of Software Quality*, Pisa, Italy, June 1998, pp. 15-22.

[19] Buhr, R.J.A., and Casselman, R.S., *Use Case Maps for Object Oriented Systems*, Prentice-Hall, USA, 1995.

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

[21] International Telecommunication Union (ITU), "Message Sequence Chart (MSC)", *ITU-TS Recommendation Z.120*, 1996.

[22] International Standards Organization (ISO), "TTCN Standard", *ISO/IEC Standard #9646-3*, 1992.

[23] Woodside, M., Franks, G. and Petriu, D., "The Future of Software Performance Engineering", *Future of Software Engineering (FOSE'07)*, IEEE Press, Minneapolis, May 2007, pp. 171-187.

[24] Arnold, D., Corriveau, J.-P. and Shi, W., "A Scenario-Driven Approach to Model-Based Testing", http://people.scs.carleton.ca/~jeanpier/VF_test_generation.pdf accessed January 2010.

[25] OMG's Object Constraint Language http://www.omg.org/technology/documents/formal/ocl.htm accessed October 2009.

[26] ITU's Specification and Description Language (SDL) http://www.itu.int/ITU-T/studygroups/com10/languages/Z.100_1199.pdf accessed October 2009.

[27] Meyer, B., "Design by Contract", *IEEE Computer*, IEEE Press, New York, November 1992, pp. 40-51.

[28] Ryser, J. and Glinz, M., "SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test", *Technical Report*, University of Zurich, February 2000, pp. 1-116.

[29] Microsoft Research, *Phoenix Research Development Kit*, June 2007, http://research.microsoft.com/phoenix accessed October 2009.