Automated Instrumentation of Contracts and Scenarios for Requirements Validation in .NET

Dave Arnold
School of Computer Science
Carleton University
Ottawa, Ontario, Canada
+1 613 520 4333
darnold@scs.carleton.ca

Jean-Pierre Corriveau
School of Computer Science
Carleton University
Ottawa, Ontario, Canada
+1 613 520 8750
jeanpier@scs.carleton.ca

ABSTRACT
During the development of an object-oriented reactive system, scenarios (such as UML’s use cases) may be used for the elicitation of functional and non-functional requirements. The contribution of this paper is the overview of a framework for the specification of a testable requirements model and the automated instrumentation of this model into an implementation in order to validate the model’s requirements against this implementation. Our testable model takes the form of contracts and is grounded in the notions of scenarios and responsibilities. More precisely, the validation of the requirements of this model depends on a user binding elements of contracts to actual procedures within a candidate implementation, (that also supplies test data). Once this is done, these requirements are validated against an execution. This validation consists in the invocation of both static and dynamic checks, the matching of scenarios, and the capture and evaluation of metrics for an execution. Metric evaluation allows our framework and testable model to also consider non-functional requirements.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification – model checking, programming by contract, and validation.

General Terms
Reliability, Languages, Validation.

Keywords
Validation, Contracts, Requirements, Scenarios.

1. INTRODUCTION
The use of scenarios [10] (e.g., UML use cases) as a method for requirements capture is not new. Indeed, several software design paradigms are grounded in the notions of responsibilities and scenarios [5]. This is the case, in particular, for object-oriented reactive systems, which are the ones we consider here. Conversely, current methods for automated model-driven testing [6] generally rest on state-based formalisms instead of scenarios [4], [12]. But Grieskamp [4] observes that state-based approaches to model-driven testing, while being far more popular than other formal (e.g., logic-based) approaches, are still poorly adopted in industry. We believe this is due, in part, to a lack of traceability [2], [5] between requirements capture and actual tests in code: Whereas the notions of responsibilities and scenarios readily map onto procedures and interactions of objects (ultimately leading to unit and interaction tests) in an implementation under test (IUT), such correspondence may be far more tenuous in the case of states (especially global system states, as opposed to states within objects [2]). Consequently, in this paper, we explore a scenario-based approach to model-driven testing.

In our work, during requirements capture, scenarios and responsibilities are augmented with static checks, dynamic checks, and metrics in order to define a system’s contracts1. Such contracts constitute what Binder calls a testable model [2]: a model from which tests (here for responsibilities and scenarios) can be extracted in a systematic way. These tests enable validating an IUT against the contracts of a system. In order to enforce strong traceability between a requirements model and actual tests associated with scenarios and responsibilities, we automate the instrumentation of such tests. We emphasize that, in contrast to many model-driven testing toolsets [6], we do not require the creation of glue code by a developer to connect tests (generated from a system’s requirement model) with an IUT. Furthermore, we wish to have a testable model that is decoupled from any particular implementation so that a single set of contracts can be tested against several candidate implementations. (Clearly, an implementation-specific approach to validation lacks abstraction and generality and thus decreases the possibility of test reuse across implementations.) The question then is how can our implementation-independent requirements model be used to generate tests that are automatically instrumented into an IUT? The answer lies in the idea of binding elements of a system’s contracts to actual procedures in the IUT. More specifically, some elements of a contract must be explicitly linked by the provider of an IUT to corresponding procedures within this IUT. For example, a responsibility of a scenario may have to be explicitly bound to a procedure in the IUT. If such mapping does not exist, then, alternatively, the IUT provider may define two unique events (observable_start_event and observable_end_event) associated with this responsibility and then bind these events to...

---

1 Static checks correspond to structural queries on an implementation, whereas dynamic checks and metric evaluators query an execution of an implementation. Such queries can be associated with scenarios and responsibilities (see [1]).
procedures of the IUT. Unlike glue code, no programming is involved in such bindings, which merely build explicit traceability links that enable the automated instrumentation of the tests generated from a system’s contracts in the IUT, as explained later.

The automated generation of fully instrumented and executable tests (missing only the test data provided by the IUT) entails the creation of a Validation Framework (VF) that is capable of inputting a requirements model, generating tests from it, and then executing such tests on one or more IUTs. The postulate of our work in developing such a VF is that responsibilities and scenarios constitute a sound semantic foundation for modeling the requirements of an object-oriented reactive system [5], [10].

Each scenario is specified as a grammar of responsibilities. Each responsibility represents a simple functional unit. In our work, a responsibility is either:

1. Bound to a procedure within an IUT.
2. Bound (via observable events, see above) to some procedures that through their interactions implement the responsibility.
3. Decomposed into a sub-grammar of responsibilities. In turn, each responsibility within the sub-grammar is then bound or further decomposed.

Static checks are restricted to a contract addressing a responsibility or a set of responsibilities (respectively bound to procedure(s) and class(es) of the IUT). Conversely, both scenarios and responsibilities can refer to dynamic checks (inspired from Design-by-Contract (DbC) [7], [9]). Such checks (e.g., pre- and post-conditions) are typically used to express constraints on the state of one or more objects (existing during a particular execution) before and after the execution of a scenario or responsibility. Most importantly, requirements captured as contracts do enable automated system testing [9] and do not rely on (a possible combinatorial explosion of) system states.

Furthermore, even simple software systems are composed of multiple scenarios. As such, validation includes not only the execution of individual scenarios, but also the execution of multiple, possibly interleaving, scenarios. Such scenario integration requires inter-scenario operators to support temporal ordering, concurrency, and distribution. Theoretical work on such operators is absent from many scenario-based testing work but is addressed at length in SCENT by Ryser and Glinz [10], [11]. Our VF makes operational portions of their work.

Beyond verifying the static and dynamic checks of scenarios and responsibilities, the grammar of responsibilities of each scenario, and inter-scenario relationships, our testable model must address non-functional requirements (such as performance, security, and usability). In our work, such non-functional requirements are test-enabled by gathering metrics during the execution of scenarios. Once a metric has been gathered, it needs to be analyzed, and ultimately, trade-off analysis must take place. Metric and trade-off analysis is an extremely domain specific activity. To support multiple domains within our VF, our framework supports pluggability (i.e., user-defined functional extensions). In other words, it is possible to define in our VF domain-specific extensions to our testable model, such extensions (called plug-ins) still enabling their automated instrumentation in an IUT. (Due to space limitations, the extension mechanism of our VF is discussed at length elsewhere [1].)

We offer an overview of our framework in the next section. (Due to space limitations, an example is presented elsewhere [1].)

2. THE FRAMEWORK

Our VF provides an open architecture for the specification and execution of a testable model. This testable model specifies static checks, dynamic checks, responsibilities, scenarios, metric capturers, and metric evaluators. A simplified graphical representation of the framework is shown in Figure 1. The VF is being integrated into Microsoft’s Visual Studio for ease of use and integration with existing development processes (unlike most automated model-driven testing approaches, as Grieskamp observes [4] in commenting on the lack of widespread use of such approaches in industry).

![Figure 1. The Conformance Testing Framework](image-url)
2.1 Input

The VF accepts three elements as input. The first element is the contracts specification that corresponds to the testable model. We have defined a high-level general purpose contract language known as Another Contract Language (ACL) [1]. The ACL is closely tied to requirements, by defining constructs for the representation of scenarios, responsibilities, and several general-purpose static and dynamic checks. (Details regarding the ACL syntax and semantics, as well as a complete case study illustrating its use can be found in [1].) Additional domain-specific constructs can also be added to the ACL, via VF plug-ins. Plug-ins are briefly introduced in Section 2.2.

Use of the ACL is not the only method for specifying a testable model. We are developing a contract language development kit that will allow for the creation of domain-specific contract specification languages. The use of multiple domain specific contract specification languages allows for the definition of a contract that is syntactically similar to the problem domain. For example, a contract language modeling an accounting system would require different syntactic/semantic constructs from one modeling a game. Furthermore, the contract language development kit can also be used for the creation of graphical contract notations, such as a UML profile.3

The second input element is one or more candidate IUTs against which the ACL contract system will be executed to determine conformance. Our VF accepts implementations in binary form. That is, source code for the IUT is not required. Through the use of Microsoft’s Phoenix Research Development Kit [8], the VF is able to open .NET managed executables, as well as C++ binaries that contain debugging information.4

Bindings represent the third and final input element to our framework. Each IUT has a unique set of bindings. Before a contract can be verified by the VF, some of its elements must be bound to implementation artifacts located within the IUT. Such bindings are essential for the automated instrumentation and execution of tests. The binding process begins with Phoenix being used to obtain a structural representation of the IUT. The VF’s binding tool then allows for the specification of a mapping between each responsibility defined within a contract to an actual procedure (or group of procedures) within the IUT (as previously mentioned). Most importantly, bindings allow contracts to be independent of implementation details, as specific procedure names, and parameter types/orders used within the IUT do not have to correspond to a similarly-named contract artifact. In addition, we repeat, such a binding step allows several candidate IUTs to be verified against a single testable model.

2.2 Openness through Plug-ins

In addition to the openness provided by the contract language development kit, we have created a plug-in development kit [1] to allow for the inclusion of user-specified static checks, dynamic checks, and metric evaluators. The creation of plug-ins is targeted towards test developers, rather than the contract writer. Plug-ins can be invoked within any contract specification language, including the ACL, to provide additional (domain-specific) framework functionality.

Static Checks

Static checks perform a check on the IUT that can be accomplished without execution. Examples of static checks include: analysis of field access modifiers, tests on inheritance depth, and the correct structural use of design patterns. A static check can be viewed as a method call: each check has a return type and may accept a fixed number of parameters. All static checks are guaranteed to be side-effect free.

Dynamic Checks

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. Like static checks, dynamic checks can be viewed as an operation with a return type and parameter set. Dynamic checks are also guaranteed to be side-effect free.

Metric Evaluators

Metric evaluators are used to analyze and report on the metrics gathered while the IUT was executing. Metric gathering is performed by the VF’s profiler in conjunction with any metrics gathered by the contract itself. Once the IUT has concluded execution, metric evaluators are invoked. Examples of metric evaluators include: performance, space, and network use analysis. Metric evaluators are side-effect free.

2.3 Compilation

Once a system’s contract has been specified, and at least one IUT has been provided, and the binding step has been completed, this contract is compiled into an intermediate language. The intermediate language is low-level and is accepted by the runtime portion of our VF. The purpose of such an intermediate language is to allow for multiple domain-specific contract languages to be executed by our common VF runtime. The compilation operation is represented by the triangle in Figure 1. Upon a successful compilation, all elements of the contract specification have been bound to any required IUT artifacts and any referenced plug-ins have been located and initialized (thus enabling their use). The result of such a compilation is a single intermediate language representation for each IUT containing all information required to automatically evaluate and execute the contracts of a system.

2.4 Evaluation and Execution

The evaluation of a contract begins with the analysis of the structural composition of the IUT, and with the execution of any static checks. Recall that the structural composition of the binary IUT is provided by Phoenix. Each static check then operates against the structure provided by Phoenix to determine a pass or fail result (reported by the VF). Once the execution of static checks is completed, the VF instruments the IUT with instructions required to monitor scenario execution, execute dynamic plug-ins, and capture metrics. Once instrumentation is completed, Phoenix is used to re-assemble the IUT so that it can be executed.

Next, the IUT is executed against a specialized profiler we provide as part of our VF. Our profiler tracks and records the execution tree/trace, the result of executing each dynamic check, and any requested metric data. The execution tree is also

---

3 The use of graphical contract notations is left for future work.
4 The debugging information is required to resolve symbol names during the binding process.
examined to determine if the execution of a scenario conforms to its grammar of responsibilities as specified within a contract. More precisely, in order to be matched, a scenario must first be triggered (via the occurrence of events and/or the satisfaction of dynamic checks such as the reception by an object of a value within a certain range). Once a scenario is triggered, it may fail to match its grammar. In this case, for dynamic checks, the failed match is recorded by the VF but IUT execution continues.

This scenario validation process is performed using a pattern matching approach (as opposed to a state-based approach used more frequently in model-driven testing (e.g., [3])). Such a pattern matching approach was selected to avoid the state explosion/scalability problem that typically afflicts state machine-based approaches [4].

During the execution of an IUT, the invocation of static and dynamic checks in scenarios and responsibilities addresses the validation of functional requirements. Next, the metric evaluators are invoked by the VF to analyze and report on the metric data that was gathered while the IUT was executing. Such analysis addresses the validation of non-functional requirements. All results from the evaluation and execution of contracts by the VF are displayed in a contract evaluation report.

3. CONCLUSIONS AND FUTURE WORK

We believe that validation requires a testable model that is capable of automatically generating instrumented checks. This model must capture functional and non-functional requirements in an implementation-independent way in order to reuse this model against several implementations. We avoid debatable separations (e.g., functional from non-functional requirements, structural from behavioral views, etc.) that, in our experience, increase the likelihood of a lack of traceability between distinct models and with an IUT. We also avoid formal, and in particular state-based, approaches whose traceability and scalability appears to be problematic in practice. Instead, our testable model is semantically rooted in the commonly-used notions of responsibilities and scenarios. Our work is also integrated into a widely-used development environment and, through plug-ins, enables domain specializations.

Additional work within the VF is required in several areas. One such area is the addition of language constructs and functionality to handle the testing of distributed applications. Such support will include the addition of inter-scenario operators supporting distribution and the coordination of multiple VF profilers operating on different machines. Also, the ability to generate a testable model from an augmented set of UML diagrams would be desirable. But it is important to acknowledge that UML models are not currently testable [2]. We will examine graphical modeling notations to determine what additional annotations would be required for the generation of a testable model that could be executed by our VF. Such generation would either result in the creation of an ACL contract or the direct generation of intermediate language. The graphical model(s) would also be required to have the ability to produce a binary candidate IUT and the corresponding bindings between the testable model and the IUT. At this time, this appears to be highly challenging.

4. ACKNOWLEDGMENTS

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

5. REFERENCES


