Open Framework for Conformance Testing via Scenarios

Dave Arnold
School Of Computer Science
Carleton University
Ottawa, Ontario, Canada
darnold@scs.carleton.ca

Jean-Pierre Corriveau
School Of Computer Science
Carleton University
Ottawa, Ontario, Canada
jeanpier@scs.carleton.ca

Vojislav Radonjic
School Of Computer Science
Carleton University
Ottawa, Ontario, Canada
radonjic@scs.carleton.ca

Abstract

Scenarios are vital for the specification of software systems. We are developing an open framework for the specification, execution, and conformance evaluation of scenarios. The scenarios define a contract which is bound to an implementation under test. The scenarios are executed by our framework to ensure conformance against the contract.

Categories and Subject Descriptors D.2.4 [**Software Engineering**]: Software/Program Verification – *assertion checkers, class invariants, model checking, programming by contract*.

General Terms Reliability, Verification

Keywords Conformance Testing; Contracts; Metrics; Non-Functional Requirements; Scenarios

1. Introduction

The use of scenarios as a method for requirements capture is not a new one. Several software design paradigms focus on the use of scenarios, such as UML use cases [3]. While scenarios are being used at the software design level, they are not being used in practice for software testing activities [2]. One aspect of software testing is conformance testing: to determine whether a system meets its requirements. If the system requirements are specified using scenarios, intuitively it would make sense to use the same scenarios to perform conformance testing. To this end, we propose an open framework which supports the specification, execution, and conformance evaluation of scenarios. Our poster presents such a framework, and is organized as follows: Section 2 provides a background about scenarios. Section 3 presents our framework, and Section 4 contains concluding remarks.

2. Background

Scenarios are a method for requirements capture that focuses on system flow and business rules [3]. Scenarios can be decomposed into a grammar of responsibilities. Each responsibility can be viewed as a simple action, such as the saving of a file, or the firing of an event. A responsibility can be mapped to a method found within an implementation. In addition to the grammar of responsibilities, a scenario may also include a set of Design-by-Contract (DbC) [4] elements. DbC elements are used to express constraints on the state of the system before and after scenario execution.

When a scenario is executed by the system, 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 actions which are not defined by the scenario are executed, that system does not conform to the specification.

Even a simple software system is composed of multiple scenarios. As such, conformance testing 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 has already been done by Ryser and Glinz [7] in SCENT. Our proposed framework seeks to operationalize portions of their work.

Scenarios provide a method for testing functional conformance. Scenarios do not contain non-functional elements, such as performance, security, and usability constraints. Along with the scenario specifications, our framework also allows for non-functional constructs to be used for the capture and analysis of non-functional information gathered during scenario execution.

3. The Framework

Our proposed framework provides an open architecture for static checks, scenarios, and metric evaluators. A simplified graphical representation of the framework is shown in Figure 1. The framework has been integrated into Microsoft's Visual Studio for ease of use.

High-level Contract Specification

The framework operates using a contract specification, an Implementation Under Test (IUT), and bindings between the two. We have created a contract language development kit, which allows for the creation of domain specific contract specification languages. For our purposes we have defined a high-level contract language known as Another

OOPSLA'07 October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-865-7/07/0010.

Contract Language (ACL). ACL is closely tied to requirements, and has constructs for the specification of goals, beliefs [6], scenarios, and several lower-level constructs, such as pre and post-conditions [4]. To allow for domain-specific constructs, ACL and other framework modules support extensions. Extensions will be discussed shortly.

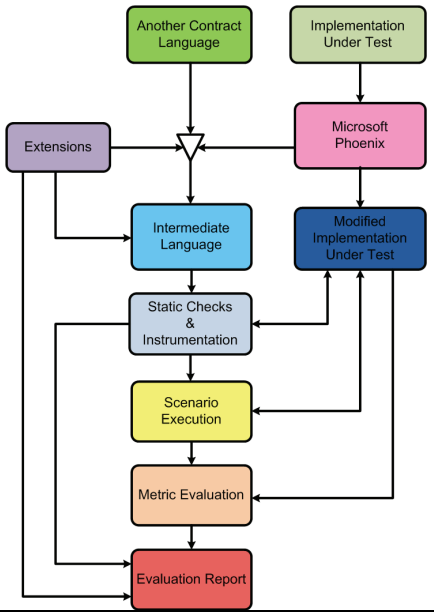

Figure 1. Framework overview.

Bindings

Once a contract has been created, the next step is to bind contract elements to implementation artifacts located within the IUT. The binding operation is performed using Microsoft's Phoenix Framework [5] to open the compiled IUT and to create a corresponding structural representation. Using the structural representation, our binding tool maps the goals, beliefs, scenarios and their containing responsibilities to types and methods found within the IUT. The binding allows the contract to be independent of implementation details, and allows for several candidate IUTs to be tested using a single contract.

Extensions

Extensions are analogous to a library found within a programming language. Extensions provide functionality not provided by the framework itself. Extensions can be one of two types: static or dynamic. Static extensions operate on the structure of the IUT. An example of a static extension would be the examination of the IUT's structure to check for the proper application of a design pattern [1]. Dynamic extensions are used to perform checks as the scenarios are being executed. A special type of dynamic check is known as a metric evaluator. Metric evaluators are used to analyze metrics gathered while executing the scenarios. An example of a metric evaluator would be a comparison of performance vs. storage metrics.

Contract Compilation

Once the contract has been defined and bound to an IUT, it is compiled into an intermediate language. The purpose of the intermediate language is to allow for multiple high-level contract languages and representations to be used within our common evaluation run-time. 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 IUT artifacts and any required extensions have been located and initialized.

Contract Evaluation

The first step in evaluating a contract is to examine the structural composition of the IUT and to execute the static checks. At this point, the IUT is also instrumented with run-time checks required for scenario execution, dynamic extensions, and metric capture.

Next, the IUT is executed against a profiler that tracks and records the execution tree, the results of the instrumented run-time checks, and the metric data. The execution tree is then examined to determine if each scenario execution conforms to the scenario grammar prescribed in the contract. The metric data is analyzed by the extensions to yield non-functional information about the IUT. All results are displayed in a contract evaluation report.

4. Conclusion

Our framework provides an open architecture for the specification of contracts, which are grounded at the system requirements level. Elements of a contract are bound to structural elements found within the IUT. The framework executes static checks and instruments the IUT for scenario execution and metric gathering. Once execution is complete, the framework examines the execution tree to determine if the scenarios were executed according to their inter-scenario relationships. Metrics are analyzed using the metric evaluators. The result of evaluating a contract is a report indicating the outcome of the static checks, scenario execution, and metric evaluation.

References

[1] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[2] Grieskamp, W. Multi-Paradigmatic Model-Based Testing. Technical Report #MSR-TR-2006-1111. Microsoft Research, August 2006.

[3] Jacobson, I. Object-Oriented Software Engineering. Addison-Wesley Professional, 1992.

[4] Meyer, B. Design by Contract. IEEE Computer, vol. 25, no. 10, pp. 40-51, October 1992.

[5] Microsoft Corporation. Phoenix Research Development Kit. March 2007. http://research.microsoft.com/phoenix/

[6] Mylopoulos, J., Chung, L., and Yu, E. From object-oriented to goal-oriented requirements analysis. Communications of the ACM. vol. 42, no. 1, pp. 31-37, January 1999.

[7] Ryser, J., and Glinz, M. SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test. Technical Report #2000-03. University of Zurich. February 2000.