In the Requirements Lies the Power

Rand Waltzman
Department of Computer Science
Royal Institute of Technology
rand@nada.kth.se

Kristina Winbladh, Thomas A. Alspaugh
Debra J. Richardson
Department of Informatics
Bren School of Information and Computer Sciences
University of California, Irvine
{awinblad,alspaugh,djr}@ics.uci.edu

Abstract

System requirements expressed as scenarios represent a rich source of knowledge about a system and the context in which it is used. This is because the scenarios are the result of extensive collaborative efforts of a wide variety of stakeholders and are in a form to which all can relate. Ideally, they serve to represent the interests of all stakeholders at each stage of the development life cycle. Our focus in this paper is system testing against requirements. In particular, we show (1) how the knowledge represented in scenarios (using ScenarioML) can be directly transformed into an operational knowledge base in a rule-based programming language (JESS), (2) how this knowledge base can be used in system testing to compute, manage, and compare expectations of system behavior to actual system behavior relative to the requirements, and (3) how this can be achieved in a manner that is transparent to all stakeholders. The power of this approach derives from the peculiarly reflective character of knowledge based systems and their explicit use of meta-information and meta-information processing. We demonstrate the viability of our approach by its application to the AquaLush system in which we detected several violations of the system's stated requirements.

1 Introduction

One of the most important goals of software testing is to determine whether a system behaves the way the stakeholders want it to behave. From the stakeholders' point of view, this behavior is ideally described by a set of high-level requirements expressed so that they can be understood with a modicum of effort and without getting bogged down in formalisms that are obscure to all but a small group of specialists. Studies of industry practice [15] have shown that scenarios successfully satisfy these demands on high-level requirements. It is equally important to be able to test these requirements in a way that is transparent to the stakeholders. The stakeholders should be able to look at the test results and see that the system behaves in the way they expect based on the requirements. This demand, however, is not widely being met in practice in industry today. In this paper, we show how to meet this demand by viewing scenario-based requirements as a rich source of knowledge about a system and the context in which it is being used. We use that knowledge source to create a knowledge-based system that tests system behavior directly against the requirements.

The structures that make up a knowledge-based system (KBS) are qualitative models that describe the system's domain [6]. These models describe the domain in terms of causal, compositional, or sub-typical relationships among objects and events. They represent a partition of the world that provides a coherent picture of the domain and permits selective views of the domain for particular purposes. The KBS uses the models as a basis for computing actions in the domain. When we talk of inference in a KBS, we are really referring to a strategy for model manipulation.

The Artificial Intelligence (AI) community recognized early on that the acquisition of qualitative models, the knowledge of a KBS, is extremely difficult and a major bottleneck to construction of such systems [9]. A key observation of this paper is that a set of scenario-based requirements for a system is very close to being the kind of qualitative model required for a KBS that could manage system testing against requirements. A set of requirements scenarios represents the collective and agreed upon understanding and expertise of a wide variety of stakeholders. In the process of formulating scenarios, stakeholders have to hammer out definitions and specifications of domain concepts, terms, events, and processes. In particular, these include causal, compositional, and sub-typical relationships among objects and events of the domain — the same basic elements in the description of the qualitative models that make up a KBS. Since the scenarios explicitly represent the system's behavior, it is reasonably straightforward to translate them

into a KBS that models the systems behavior at the same qualitative level. The central type of inference, or model manipulation, done by the KBS in our approach is management and generation of expected system behavior and its comparison with actual system behavior during testing.

The explicit use of meta-information and meta-information processing distinguish knowledge-based systems from most other types of information processing systems and results in the ability of a KBS to reflect upon itself in the sense that the KBS knows what it knows, how it applies that knowledge to a particular problem, and how to explain that application. In our case, that means, for example, that if the KBS detects a mismatch between expected and observed system behavior during system test, it can use its requirements-based qualitative model of system behavior to explain the nature of the mismatch directly in terms of the requirements. This provides the transparency of the testing process previously mentioned. Such an explanatory facility could also be the basis of a tool that allows stakeholders to query the model and perform what-if type experiments in order to interactively explore the requirements.

Because we generate the KBS directly from requirements scenarios, we can do so at an early stage in system development. We can then use the KBS to assist in the design of requirements-based test scenarios. There are several advantages to designing such test scenarios at an early stage of system development. In general, the requirements receive additional validation before requirements problems can cause costly misunderstandings later in the development process [8]. In particular, we could discover (1) requirements that are not testable, (2) behaviors that are not directly observable and that might require the construction of special equipment for observing those behaviors, and (3) conflicting, incomplete, or ambiguous requirements.

In the remainder of this paper, we describe the transformation of the rich knowledge model represented by scenario-based requirements into a KBS that manages the process of testing the system against requirements in a way that is transparent to stakeholders. We show how this technique facilitates stakeholder involvement in two important phases of system development: (1) requirements formulation and evaluation and (2) system testing. We applied our approach to a sample system, AquaLush [7], a project developed elsewhere with a full range of artifacts. AquaLush is an automatic irrigation system that controls irrigation based on soil moisture levels rather than timing. We will use AquaLush artifacts and concepts to demonstrate our approach throughout the paper and validate our approach. In section 2, we summarize related work, and in section 3 we describe the requirements specification format that our approach uses. We present the details of our approach in section 4. In section 5 we briefly describe the results of our validation study and in section 6 we present our conclusions. Finally, we present some thoughts about future work in section 7.

2 Related Work

Testing research has focused mainly on *code-based testing*, in which tests are developed and chosen in order to achieve coverage of the implementation code. Although code-based testing can successfully detect faults in the code, it might not detect faults that produce plausible behavior but fails to meet the system's requirements. Furthermore, code-based testing implicitly excludes stakeholder participation.

Specification-based testing, on the other hand, is a testing technique whose purpose is to confirm the extent to which a system under development meets its specifications. Most specification-based testing approaches have focused on the component level and are typically expressed in the form of Labeled Transition Systems, Finite State Machines, state charts, or message sequence charts [10]. However, as in the case of code-based testing, stakeholders are implicitly excluded from participation as a result of the formal complexity of these representations. Since high-level requirements typically are less formal and more abstract than component specifications, requirements-based testing has been applied with only limited success [5, 11].

The KBS that we describe in this paper can be thought of, at least at some level, as what is known as a *test oracle*. A test oracle consists of two main parts: (1) expected output from the system under test, and (2) a procedure that compares the expected output with the actual output [12]. Oracles can either be human (i.e., manual checking of output) or automated (e.g., software), and although they seem essential to testing, they are often not easy to come by. Software oracles are not used extensively in common industrial practice. However, to the extent that software oracles are discussed, they completely lack the fundamental reflective characteristics of a KBS that are essential for transparent operation. The quality of human oracles and their time and effort is almost never taken into account in the evaluation of testing methods even though human testers are frequently unsure of the correctness of test output and must repeat their work every time the tests are run. This indicates, as recent work also shows, that test oracles can have a significant impact on test effectiveness and efficiency [16]. This finding highlights the importance of our KBS approach.

Our view that a set of scenario-based requirements is a rich source of system knowledge and a precursor to a KBS suggests that knowledge acquisition techniques developed over the years in the KBS community should be extremely relevant to the requirements engineering community. Reubenstein and Waters [13], among others, made this same observation in their work on requirements acquisition.

Little work has focused on using a KBS for

requirements-based testing. A notable exception is Samson [14]. She suggests that the quality of a test plan can be greatly improved by using a KBS to help match requirement types with test types. Her KBS is not based on the requirements themselves, but rather on a general knowledge of how to map classes of requirements onto classes of tests.

3 Scenarios

A *scenario* is a semi-formal description of uses of a system in terms of situations, interactions between agents, and events unfolding over time. ScenarioML is an XML language for scenarios [3, 4]. ScenarioML expresses scenarios with a combination of events, ontologies, references, and scenario parameters. The events are recursively structured from simple text events as a basis and include compound events grouping several events in a particular order (total or partial), event schemas such as iterated events and sets of alternative events, and episodes that specify another scenario as an event. Allen's interval algebra relations [2] express the temporal relationships among the parts of a compound event. This approach to events supports automated recognition of scenarios happening in a domain, derivation of one scenario from another (such as one or more test scenarios, or paths through a requirements scenario), and other automated processing. Ontologies give a way to describe the kinds of entities that can exist in a domain, define specific entities, and express the relationships among them. We use ontologies to help give the context of scenarios and (through scenario parameters) specify the range of entities that can appear in a specific scenario. We have seen that without ontologies and scenario parameters, it is difficult to derive adequate tests from requirements scenarios because there is no information with which to make them concrete, and there is little opportunity for automation of the process.

Fig. 1 shows a snippet of the "IrrigateScenario" from AquaLush. The scenario shows that the system should try to read each sensor three times, if it fails after three times the sensor is marked as failed. The next sequence in the scenario (Sequence 2) has a pre-condition based on the result of the sensor reads in the top of the scenario.

4 Requirements-Based Test Harness

Our KBS together with its interface to the system under test are components of what we call the test harness. In this section we describe the overall architecture and operation of the test harness.

4.1 Architecture

Fig. 2 shows the architecture of the test harness and its relation to a requirements scenario and its test scenarios. Its

SEQUENCE
1. ITERATION *for each* **sensor** *in* { the set of all sensors. }
   *. ALTERNATIVES
   A. SEQUENCE
      1. ITERATION • 0..2
         *. AquaLush failed to read the sensor.
      2. AquaLush read the sensor.
   B. SEQUENCE
      1. ITERATION • 3..3
         *. AquaLush failed to read the sensor.
      2. ALTERNATIVES
         A. AquaLush records that this particular sensor failed in persistent store.
         B. AquaLush alerts the operator that it cannot write to its persistent store.
2. SEQUENCE
   1. ITERATION *for each* **zone** *in* { in the set of all zones. }
      PRECONDITION moisture level LESS THAN critical moisture level & zone's sensor is working.

Figure 1. Scenario snippet

basic operation is to drive the system through a variety of paths through the scenarios, compare events output from the system to events in the requirements scenario, and evaluate whether or not they match. If there are mismatches between the expected and received events, the test harness alerts the tester and provides an explanation produced by the KBS. This process is illustrated in Fig. 3.

From the requirements scenario we map the scenario events to input and output functions that will connect the system under test and the test harness. We then use the requirements scenario and the mapping to create several test scenarios, with each test scenario tracing a particular path through the requirements scenario.

The events that we monitor typically involve both the system under test and the testing environment. The events are divided into a system output part, which the test harness should be monitoring occurrences of, and a system input part, which is the response to the output generated by the test harness and that drives the system. The test harness functions that monitor system output and generate system input are collectively referred to as the test harness API. The API is the primary interface between the system under test and the KBS. As an example, consider the scenario event "Failed to read sensor". The system output is "Read sensor", and maps to test harness function `SimSensorDevice.read()`. The KBS decides that the resulting system input is "fail", which maps to the return statement of `SimSensorDevice.read()`.

4.2 Test Scenarios and Test Cases

A *test scenario* corresponds to a particular path through a requirements scenario, and provides information about event responses along that path. A *test case* is a specialization of a test scenario created by adding concrete input data for the system under test. Each test case is an ordered list of event-responses. Each *event-response* contains information about the type of event, particular event parameters, and particular input that the KBS uses to drive the system.

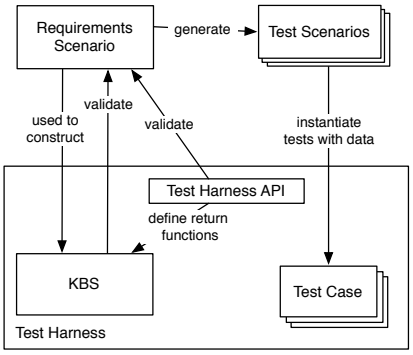

Figure 2. Test Harness Architecture

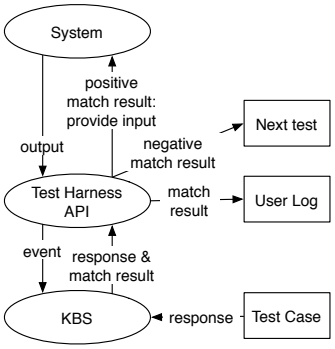

Figure 3. Test execution flow

For example, in the requirements scenario in Fig. 1, a path could be chosen corresponding to the case where the moisture level is below the critical value (see precondition for Sequence 2). The test scenario corresponding to that path contains that information without deciding concretely what the moisture level is. A test case for this test scenario is a further specialization and has concrete input data, such as a particular instance of a sensor and a moisture level of 10% when the critical moisture level is 50%. An event-response element in a test case for this test scenario would therefore look like: (event-response (event-type "Read sensor") (parameter S1) (outcome 10)), where event-type is the output event from the system to the test harness, and outcome is the input data from the harness to the system. Particular objects such as sensor S1 are instantiated instances of the ScenarioML ontology InstanceType element. The order of event-responses is derived from the sequence of events in the test scenario. There could for example, be two "Failed to read sensor" events followed by one "Success to read sensor" event. The KBS keeps track of appropriate input for each event. Fig. 4 shows a test case for the scenario snippet in Fig. 1. Although these test cases are executable code, our experience indicates that they are understandable with a modest effort by non-technical stakeholders.

```
(event-response (event-type "Read sensor") (parameter S1) (outcome fail))
(event-response (event-type "Read sensor") (parameter S1) (outcome fail))
(event-response (event-type "Read sensor") (parameter S1) (outcome 20))
(event-response (event-type "Read sensor") (parameter S2) (outcome fail))
(event-response (event-type "Read sensor") (parameter S2) (outcome fail))
(event-response (event-type "Read sensor") (parameter S2) (outcome fail))
(event-response (event-type "Read sensor") (parameter S3) (outcome 60))
(event-response (event-type "Read sensor") (parameter S4) (outcome 40))
(event-response (event-type "Read sensor") (parameter S4) (outcome fail))
...
```

Figure 4. Test case snippet

4.3 KBS

The KBS performs the following functions: (1) compute expected results based on test case and runtime information, (2) match events coming into the test harness from the system under test to expected events in the test case, and (3) use the information from the event-responses to drive the system.

The KBS is implemented in the rule-based programming language JESS (Java Expert System Shell) [1]. Fig. 5 shows a sample KBS rule. The qualitative model of the KBS is structured around different sets of tasks described by the requirements scenarios. For example, the rule in Fig. 5 partially implements the task of activating a zone for watering. We see this from the task condition pattern on the left hand side (LHS) of the rule. The remaining condition patterns on the LHS of the rule are satisfied if (1) the KBS has received a "Read sensor" event for the zone in question, (2) the number of times the system has attempted to read this sensor is below the maximum number allowed (three according to the requirements scenario), and (3) there exists an event-response in the test case that matches the event and the zone and that has not yet been processed, and whose outcome has the value of 'fail'. If the conditions in the LHS of the rule are satisfied, the KBS takes the actions specified in the right hand side of the rule. These actions are to update the model by (1) incrementing the number of times this sensor was read by 1, (2) noting that the requested sensor read failed, (3) noting that the event has been processed, and (4) noting that the event-response has been processed.

The KBS contains sufficient meta-information to be able to explain its model in a form understandable to stakeholders by (1) producing natural language versions of the rules, and (2) stating the fairly direct relation between the rule components of the model and the requirements in the scenario. As we illustrate in the next section, the KBS is also capable of generating explanations of expectation violations in terms that stakeholders can understand.

```
(defrule process-read-sensor-request-2
      ;System made read sensor request and sensor failed.
      (task (name activate-zone) (object ?zone))
      ?e <- (event (text "Read sensor") (parameter ?zone) (status matched))
      ?nsrwm <- (num-sensor-reads (value ?nsr&:(< ?nsr ?*max-sensor-reads*)))
      (zone (name ?zone))
      ?er <- (event-response (event-type "Read sensor") (parameter ?zone)
            (outcome fail) (status wait))
=>
      (modify ?nsrwm (value (+ ?nsr 1)))
      (bind ?*sensor-read-result* -1)
      (modify ?e (status processed))
      (modify ?er (status done))
)
```

Figure 5. JESS rule

5 Validation Study

In this section we briefly describe our experience constructing the test harness, generating tests, and running the tests on a sample software system, AquaLush [7].

We used the requirements and use cases to produce seven ScenarioML scenarios. We will use one of these scenarios, IrrigateScenario, to describe our work. We chose this scenario because it describes the main functionality of the system and relates it to several stakeholder goals. IrrigateScenario contains two major event sequences where the second one itself consists of three event sequences. The first major event sequence is illustrated in figure 1. From IrrigateScenario we derived 6 test scenarios and created 6 test cases out of the test scenarios. Finally, we ran our test cases against the released version of the system.

5.1 Testing Released Version

We detected a system event mismatch when we ran our first test case on the released version of the system (see test output below). For this test case, the test harness made the system think that sensor S1 failed to read three times in the first event sequence of the scenario. The system read the other sensors and then moved on to open the valves in the zone with sensor S3. Once the valves were open, the system should have started reading sensor S3 every minute until irrigation stopped. What happened instead was that after the valves opened in the zone with sensor S3, the system attempted to read sensor S1, even though this sensor was not in the zone that was being irrigated at the time and had actually been reported as not working.

```
...

An open valve event for V10 was received as
expected.

An unexpected request to read the sensor
from zone S1 was received. It was unexpected
since the system is currently irrigating zone S3.
This action is not in accord with Item 2.1
of IrrigateScenario.

A read sensor event for zone S3 was received as
expected while irrigating.

...
```

The KBS produced a moderately detailed explanation of the expectation violation including a reference to the item in IrrigateScenario that is the basis of the violation. This is a good example of the type of transparent behavior that facilitates stakeholder participation in the testing process.

Running the other test cases revealed additional problems, including some implementation faults, that we do not describe here due to lack of space.

6 Conclusion

We have shown how using the qualitative models of a KBS enables stakeholder participation in requirements-based testing. We have demonstrated the effectiveness of our approach by identifying a number of ways in which the published version of the AquaLush system does not satisfy its requirements. The explanations produced by the KBS comparing the actual system behavior with expected behavior clearly indicate the nature of the requirements violations in each of our six test cases.

The strength of our approach is derived from the synergy of two complementary technologies. On the one hand, we have ScenarioML. It provides a semi-formal language for describing requirements that is particularly well suited to the mechanization of their manipulation. These requirements are a rich source of stakeholder knowledge. On the other hand, we have the qualitative models of KBS technology implemented with the rule-based programming language JESS. These two technologies are deeply connected in that the KBS is directly derived from the ScenarioML requirements. JESS rules are highly modularized components of KBS models. A powerful feature of these rules is their direct relation to requirements scenarios. That is partly where their power lies in terms of directly exposing requirements violations. The data structures used to drive our rule-based programs are directly derived from the ScenarioML representations of the requirements. At this early stage of the research, we derive them by hand. But our experience so far has convinced us that the process of deriving JESS rules from ScenarioML scenarios can be completely automated.

7 Future Work

Our evaluation study has provided us with evidence that our approach is effective in revealing both potential specification problems and implementation faults. However, a testing approach also needs to be efficient. In order to make our approach both effective and efficient, we are currently working on different ways to automate the process.

One substantial task in our process has been to manually write the ScenarioML scenarios. Our group is currently implementing an Eclipse plug-in that will ease this task by providing a graphical interface for editing scenarios. We are also investigating ways to automatically generate test scenarios that cover all branches of a ScenarioML scenario. We will use instantiations of InstanceType elements in the scenario ontology and pre- and post-conditions as constraints and search through the scenario paths with those constraints until we have enough test scenarios to cover all branches and conditions. Since we view scenarios as a knowledge source, we also plan to investigate the possibility of adapting knowledge acquisition techniques from the KBS community. We also plan to auto-generate test cases in the form of JESS event-responses from the test scenarios by either random selection of input data that satisfies the path, manual selection of data, or heuristic selection based on boundary value analysis of the ontology.

As we indicated in the last section, the KBS is currently created manually. In the future we expect to be able to generate it automatically from a set of requirements expressed in ScenarioML. We are aware of the possible need to extend the ScenarioML language to be able to express information needed for this task that is not necessary for the requirements themselves. We also need tools that will allow stakeholders to statically (i.e., not in the context of a running test) inspect and understand the qualitative models of the KBS - especially in terms of the requirements.

Once automation is in place, we will perform a more substantial validation study to evaluate the effectiveness of test cases that are auto-generated from ScenarioML. We also want to evaluate the efficiency of using our approach with automation in place compared to other testing approaches. In order for such an evaluation to be convincing we need to develop a framework for evaluating different testing approaches with regard to effectiveness and efficiency. It is generally known that faults that stem from requirements misunderstandings are expensive to fix late in the development cycle. Time spent on specifying and validating the requirements for the system could therefore be gained by a lower number of severe faults later in the cycle. We intend to develop a framework that classifies the different types of faults an approach can find and measures time spent on specification development and testing all activities.

References

[1] JESS (Java Expert System Shell). `http://herzberg.ca.sandia.gov/jess/`.

[2] J. F. Allen. Maintaining knowledge about temporal intervals. *CACM*, 26(11):832–843, 1983.

[3] T. A. Alspaugh, S. E. Sim, K. Winbladh, M. Diallo, H. Ziv, and D. J. Richardson. The importance of clarity in usable requirements specification formats. Technical Report UCI-ISR-06-14, Inst. for Softw. Res., Univ. of Cal., Irvine, 2006.

[4] T. A. Alspaugh, B. Tomlinson, and E. Baumer. Using social agents to visualize software scenarios. In *SoftVis'06*, pages 87–94, 2006.

[5] L. C. Briand and Y. Labiche. A UML-based approach to system testing. *Softw. and Syst. Mod.*, 1(1):10–42, 2002.

[6] W. J. Clancey. Viewing knowledge bases as qualitative models. Technical Report STAN-CS-8, Stanford University, Computer Science Department, May 1986.

[7] C. Fox. *Introduction to Software Engineering Design: Processes, Principles and Patterns with UML2*. Addison Wesley, 2006.

[8] D. Graham. Requirements and testing: Seven missing-link myths. *IEEE Softw.*, 19(5):15–17, 2002.

[9] F. Hayes-Roth, Waterman, and D. Lenat. *Building Expert Systems*. Addison-Wesley, 1983.

[10] H. Muccini, M. S. Dias, and D. J. Richardson. Systematic testing of software architectures in the C2 style. In *FASE'04*, volume 2984 of *Lect. Notes in Comp. Sci.*, pages 295–309, 2004.

[11] C. Nebut, F. Fleurey, Y. L. Traon, and J. M. Jezequel. Automatic test generation: A use case driven approach. 2006.

[12] T. O. O'Malley, D. J. Richardson, and L. K. Dillon. Efficient specification-based oracles for critical systems. In *CSS'96*, 1996.

[13] H. B. Reubenstein and R. C. Waters. The Requirements Apprentice: Automated assistance for requirements acquisition. *IEEE Trans. on Softw. Eng.*, 17(3):226–240, 1991.

[14] D. Samson. Knowledge-based test planning: Framework for a knowledge-based system to prepare a system test plan from system requirements. *The Journal of Syst. and Softw.*, 20(2):115–124, 1993.

[15] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer. Scenarios in system development: Current practice. *IEEE Softw.*, 15(2):34–45, 1998.

[16] Q. Xie and A. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Trans. on Softw. Eng. and Method.*, 2006.