ISR Institute for Software Research
University of California, Irvine

Meeting the Requirements and Living Up to Expectations

Kristina Winbladh
University of California, Irvine
awinblad@ics.uci.edu

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

Thomas A. Alspaugh
University of California, Irvine
alspaugh@ics.uci.edu

Debra J. Richardson
University of California, Irvine
djr@ics.uci.edu

January 2007

ISR Technical Report # UCI-ISR-07-1

Institute for Software Research
ICS2 110
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

Meeting the Requirements and Living Up to Expectations

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

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

ISR Technical Report UCI-ISR-07-1
January 30, 2007

Abstract: To produce better quality software at reasonable cost, we propose *requirements-based testing*, in which testing is driven directly from the requirements and faults that prevent the product from meeting its requirements are detected. Our approach makes use of requirements in the form of goals and scenarios. From these we generate test scenarios that drive the system under test through particular paths of the scenarios, and a test harness that verifies the system follows the particular path and meets its conditions. Because our test scenarios are derived directly from the requirements, a major benefit of the process of writing test scenarios is the identification of poorly formulated requirements. We applied our approach to a sample software system and to mutants of it generated by MuJava. Our approach was effective at finding implementation faults that caused the system to diverge from the requirements.

Meeting the Requirements and Living Up to Expectations

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

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

ISR Technical Report UCI-ISR-07-1
January 30, 2007

Abstract

To produce better quality software at reasonable cost, we propose *requirements-based testing*, in which testing is driven directly from the requirements and faults that prevent the product from meeting its requirements are detected. Our approach makes use of requirements in the form of goals and scenarios. From these we generate test scenarios that drive the system under test through particular paths of the scenarios, and a test harness that verifies the system follows the particular path and meets its conditions. Because our test scenarios are derived directly from the requirements, a major benefit of the process of writing test scenarios is the identification of poorly formulated requirements. We applied our approach to a sample software system and to mutants of it generated by MuJava. Our approach was effective at finding implementation faults that caused the system to diverge from the requirements.

1 Introduction

The goal of software testing is to assess the quality of a software product by finding faults in it. The effectiveness and efficiency of a testing approach can be characterized in terms of tradeoffs between the effort of using the approach and its ability to find faults. Specification-based testing is effective and efficient

in the sense that testing resources are focused on the most important behavior of the system and in addition to code faults it can also reveal faults in the specification early. When specification faults are revealed early, we can prevent these from propagating into the final product. We believe that *requirements-based testing* is a particularly effective specification-based testing approach. In requirements-based testing, the requirements serve as the basis of (1) the validation of the system under test and (2) the scenarios that drive the system while it is tested. In addition to the benefits of specification-based testing, requirements-based testing tests against the requirements most important to stakeholders and assesses quality of the system under test in terms stakeholders understand.

The objective of our work is to utilize the natural symbiotic relation between requirements and testing to produce systems that better satisfy stakeholder needs. We do this by using the testing process to validate the requirements, and using test results to demonstrate the quality of the software in the extent that it meets the requirements. The relation is depicted in Fig. 1 as a three-way exchange of benefits. The first benefit is that results of tests generated from the requirements provide meaningful insights about the quality of the system under test, in terms of how well it meets its requirements. The second benefit is that early test design helps to build quality into the product by validating the requirements and inhibiting defect multiplication. Testing against a specification generates questions about that specification and challenges it. If test design is considered early and driven by requirements, the requirements receive additional validation before requirements problems can cause costly misunderstandings later in the development process [7]. The third benefit is that tests generated from requirements, as opposed to later specifications and models of code, offer substantial opportunities for testing efficiency and allow the generated test cases to be directly traced back to high-level requirements [17].

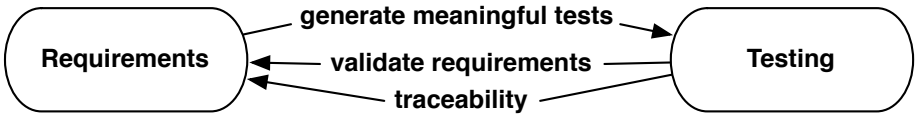

Figure 1: Symbiotic relation between requirements and testing

In this paper we will describe a requirements-based testing approach in which tests are directly driven by and checked against requirements in the form of scenarios. We applied our approach to a sample system, AquaLush [6], 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 describe the results of our validation study and we

conclude the paper with lessons learned and future work in section 6.

2 Related Work

All too often, testing is cut short when budgets or resources are short, does not correspond to the original requirements, and is generally inadequate, inefficient, and ineffective. Software engineering researchers have long warned of and pointed out the enormous risks associated with neglecting software testing [15]. Prudent cost management therefore requires that software testing also be efficient. 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 behavior that is plausible but fails to meet the system's requirements.

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. Whereas code-based testing strives to exercise as much of the implementation as possible in order to reveal faults in the implementation, specification-based testing strives to examine whether the implementation meets as much of its specification as possible, in order to reveal faults that prevent the implementation from doing so. Most specification-based testing approaches have focused on lower-level specifications that are typically expressed in the form of Labeled Transition Systems, Finite State Machines (FSM), state charts, or message sequence charts (depending on the approach). It is possible to automatically generate test suites from these that can determine how well the system conforms to its specification [11]. Since high-level requirements typically are less formal and more abstract than component specifications, specification-based testing has not been successfully applied to them before now.

There has been work on test case generation from requirements, particularly from UML use cases [4, 12]. These approaches however, seem to depend on design information. Our approach differs from these in that it is purely requirements-based, i.e. we do not make use of design or implementation information when creating test cases or test oracles. More than one set of design choices and implementations can therefore be valid, and be tested against one set of requirements. Although high-level goals and scenarios are not as formal as FSMs, we believe that testing against requirements expressed in these terms addresses many of the commonly recognized problems in creating quality software under acceptable budget and time constraints.

The mechanism to evaluate whether the output of a test scenario is correct is known as a *test oracle*, and the belief that the tester is able to determine whether the test output is correct is known as the *oracle assumption* [16]. 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 [13]. 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. For all intents and purposes, software oracles do not exist in common industrial practice, while 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 [20]. We propose a more automated approach in which we test the system against test oracles derived from requirements, so that results can be automatically and reliably checked. In prior work we manually constructed these oracles from the pre- and post-conditions of goals in the plans, so that the oracles could answer whether the system's state changed from a goal's pre-condition to its post-condition. Since pre- and post-conditions are expressed formally in predicate logic, there is good reason to believe that test oracles for them can be automatically generated.

Work on relationships between goals and refinement of goals into operationalizable requirements has been carried out for at least a decade. This work includes the reduction of goals to functional and non-functional requirements. Of particular interest is the work on goal refinement by Lamsweerde *et al.* [9], Mylopoulos, Chung, Yu, *et al.* [5], and Rolland *et al.* [14], specifically the work on mapping goals to requirements scenarios.

Preliminary results of our previous work indicated that testing against goals and plans can successfully distinguish false positive test results and domain knowledge errors [19]. Our previous work also indicated that we can infer the achievement of high-level stakeholder goals from the achievement of goals at the implementation level [18]. We found evidence that stakeholders prefer ScenarioML scenarios over use cases and message sequence charts [2]. Although our preliminary results are promising, we detected some drawbacks using goal-annotations in the source code and not providing support for test generation. We now feel encouraged to explore the development of a requirements-based testing framework that addresses these issues.

3 Goals and Scenarios

Goals describe stakeholders' intentions for the system and state objectives that the system should meet. High-level stakeholder goals are step-wise refined in an AND/OR goal-graph into lower-level functional goals. At some point in the refinement process, the order of sub-goals can become relevant. We therefore introduce the concept of plans. A *plan* is an abstract description of how to satisfy some goal by satisfying its sub-goals in some sequence. We use GoalML, an XML-based language, to express goals and plans [19, 18]. XML-based languages are particularly useful for supporting test automation and production of human readable versions of scenarios and goal models.

At an even lower level of abstraction, goals can be refined by scenarios. A *scenario* describes a use of a system in terms of situations, interactions between agents, and events unfolding over time. ScenarioML is an XML lan-

guage for scenarios [2, 3]. ScenarioML expresses scenarios with a combination of events, ontologies, references, and scenario parameters. The events are recursively structured, starting from simple text events as a basis, and including 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 [1] 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.

Figure 2 shows a partial goal graph. The figure illustrates the refinement of a top-level stakeholder goal "Create a high-quality maintainable product" through various lower level goals such as "AquaLush must irrigate only until the critical moisture level is achieved" and "AquaLush must allow operation in either manual or automatic mode" down to the "IrrigateScenario". The "IrrigateScenario" is a refinement of the goals, because if the system does what the scenario specifies, there is confidence that the corresponding stakeholder goals are met. Figure 3 shows a small snippet of the "IrrigateScenario". 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 Testing

Testing is typically be divided into four major activities: test design, test generation, test execution, and test evaluation. In the following sections we will describe our testing approach through these activities and illustrate it with a running example from AquaLush.

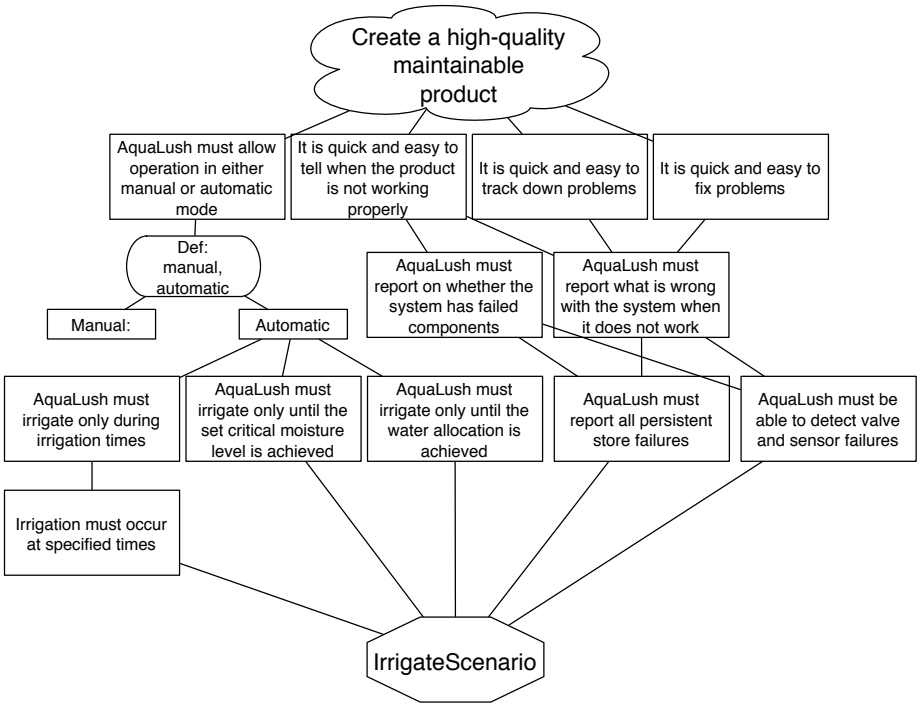

Figure 2: Partial goal-graph

4.1 Test Design

We believe the testing process should begin as early as possible. We therefore suggest that test design should begin as soon as there are requirements. Figure 4 shows a diagram of the test design tasks in our approach. The goal of our testing approach 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. The events can be divided into three kinds of events: internal, boundary, and external. *Internal events* are events inside the system under test, and are not checkable through specification-based testing. An example of such an event is "AquaLush places the zone on an active zone list". This event is internal to the AquaLush system, and cannot be detected by monitoring the system from the outside. Internal events are typically not requirements-level events because they deal with design level issues. *Boundary events* are events that involve both the system under test and the testing environment. An example of such an event is "AquaLush failed to read the sensor". This event is an output from the system under test to an external function, which means that we can monitor the occurrence of the event from outside the system under test. Most of the events in our requirements scenarios are of this type. *External events* are events that happen

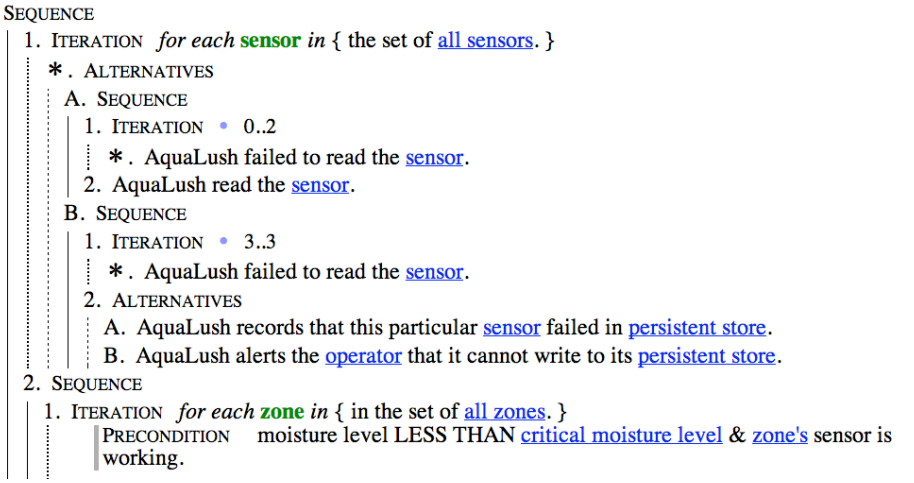

Figure 3: Scenario snippet

outside the system under test. An example of such an event is "It is raining". This is an event that does impact the moisture level of the soil that AquaLush irrigates. Although we do not have any requirements scenarios that contain external events in AquaLush, we can design tests to simulate such occurrences and oracles to verify correct system behavior under such circumstances.

From each 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. In figure 4 this corresponds to following the top arrow 'generate' from goals, plans, and scenarios to test scenario. For example, in the scenario snippet of figure 3, the alternatives present five different paths. However, since the iteration specifies *for each sensor*, five sensors allow one test scenario to cover all paths, three to four sensors allow two test scenarios, and so forth.

When mapping the requirements scenario events to test harness functions, we look at the events for system output and input. The requirements scenario event "Failed to read sensor", for example, is divided into a system output part, which the test harness should be monitoring occurrences of, and a system input part. The system input part is the result of the event, or input to "drive" the system in cases where this is needed. In our example, the system output is "Read sensor", and maps to test harness function `SimSensorDevice.read()`, and the system input is "fail", which maps to the return statement of the test harness function SimSensorDevice.read()

The process of mapping scenario events to input and output functions can reveal whether or not the events in the scenario are testable. In section 5 we will describe instances of non-testable events (internal events) that we discovered in AquaLush's use cases. Since each event is evaluated for testability, this process

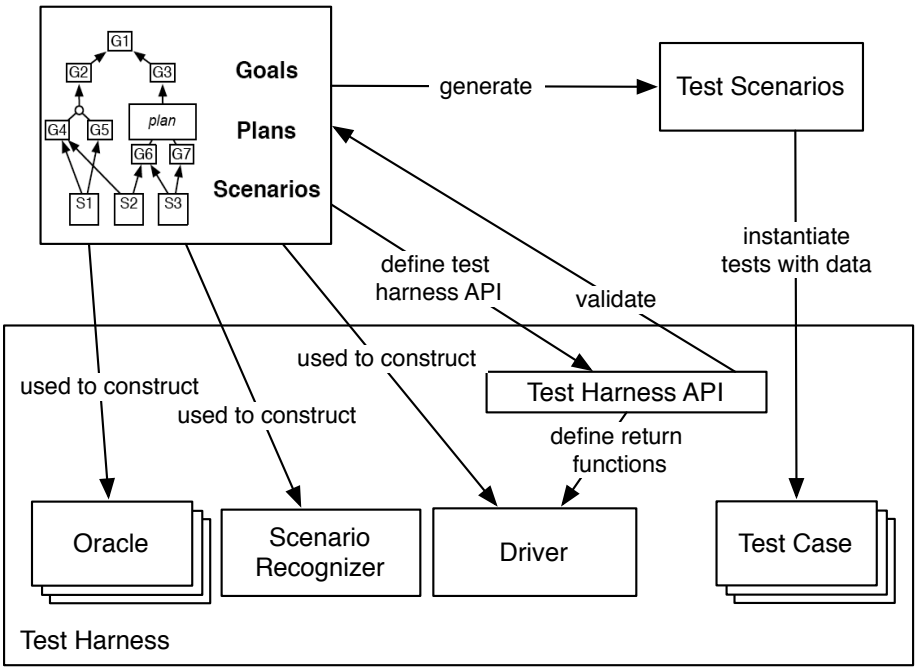

Figure 4: Test Design Flow

serves as one validation of the requirements.

The mapping of scenario events to system input and output allows us to define the test harness API (such as the functions mentioned above), which the system can hook in to later. A simulated environment is often needed when testing software systems. A simulated environment for AquaLush was one of the artifacts provide by [6], but for most projects such an environment would have to be built. We made AquaLush's simulation a part of the test harness by altering it to provide the capability of monitoring calls and providing input. The SimSensorDevice class for example, contains instances of the SensorDevice class which AquaLush will communicate with when reading moisture levels from its sensors. Although we cannot add any monitoring code to SensorDevice, because it is part of the system under test, we are able to monitor the calls to the sensors through SimSensorDevice instead.

Figure 5 shows a runtime view of the test harness. Once the different components are constructed and integrated, events flow from the system into the test harness where they are matched against the expected events given by the current test case. When an event is matched against the test case, the test case provides the next input needed to drive the system along the chosen path. The oracles use the runtime information along with requirements knowledge to compute correct results, and the runtime pre- and post-conditions in the scenario are checked against these results for correctness.

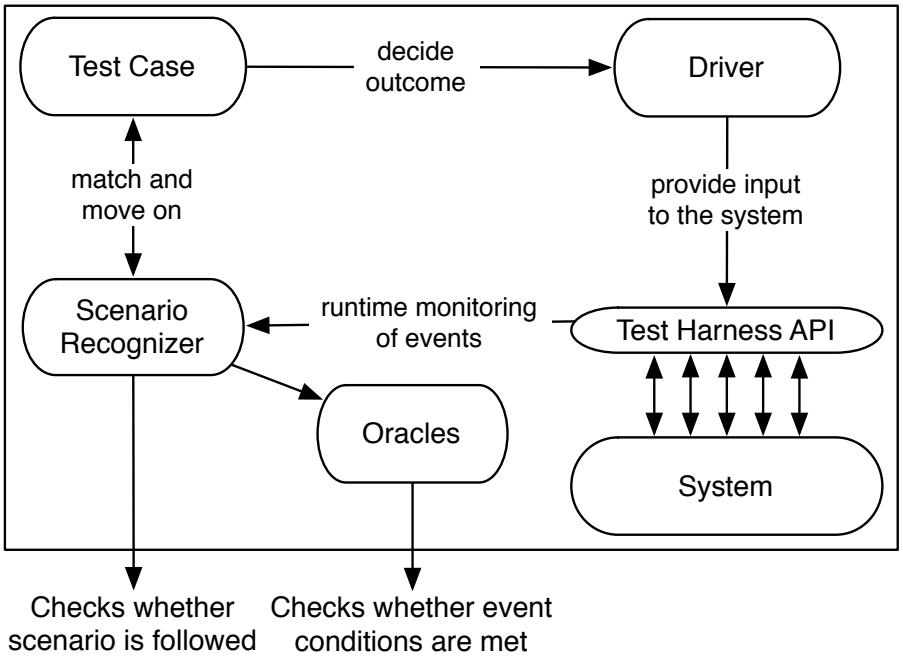

Figure 5: Runtime Flow

During test design we can also construct the rest of the test harness since it is independent of the development of the system under test. The scenario recognizer matches events coming into the test harness from the system under test to expected events in the test case. The scenario recognizer is constructed using the structure and information in the requirements scenario. Each test case is a piece of code that contains an ordered list of *event-responses*. These event-responses contain information about the type of event, particular event parameters, and a particular response to the event. The test cases are specializations of the test scenarios created by adding concrete input data for the system under test. The driver is a piece of code that uses the information from the event-response to drive the system by providing it input. The driver is constructed by using information in the requirements scenarios and the test harness API. The oracles are pieces of code that compute expected results based on the test case and test case information available at runtime, and compares this to actual results of the running system. We construct the oracles manually using the information in the requirements scenarios. For our sample system, we constructed the test harness API from the AquaLush simulation in Java, and implemented the scenario recognizer, driver, test cases, and oracles in the rule-based language JESS (Java Expert System Shell) [8].

Figure 6 shows a sample JESS rule for the driver. The left hand side of the rule states that there has to be a task for activating a zone and an event

for reading a sensor in that zone that was matched by the scenario recognizer; the number of times this sensor has been attempted to read must be below the maximum number allowed (three according to the requirements scenario); and there exists an event-response in the test case for the event and the zone which has not yet been processed, and whose outcome has the value of 'fail'. The right hand side of the rule states that if the left hand side of the rule is true, then the number of times this sensor was read will be incremented by one, the result of the sensor read will be set to -1, the status of the event will be set to processed, and the status of the event-response will be set to 'done'.

```
(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 6: JESS rule

An important part of test design is to test the test harness itself. We can do this by providing the test harness with streams of expected and unexpected events (simulating the system). This evaluation will both provide useful information about the requirements, such as whether they make sense, as well as help make the test harness robust and ready for testing.

4.2 Test Generation

Another preparatory step is to generate the test cases that give specific data for each of the test scenarios. The test scenarios correspond to particular paths through the requirements scenario, and provide information about event responses along those paths. The test scenario is basically an abstract description of the test case. For example, in the requirements scenario in Figure 3 a particular path could be chosen if the moisture level is below the critical moisture level (see precondition for Sequence 2). The test scenario corresponding to that particular 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 particular test case for this test scenario would therefore look like this: (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 instanceTypes. 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 driver keeps track of appropriate responses to each event. Figure 7 shows a test case for the the scenario snippet in figure 3 that uses four sensors and therefore covers four of the five paths of the scenario. Although test generation was manual in our study, we believe it is possible in general to automate the generation of both test scenarios and test cases.

```
(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 7: Test case snippet

4.3 Test Execution

When we execute a test, the test harness provides input to the system, when the system produces output (results or function calls to external functions) the test harness monitors these and responds. In figure 8 we demonstrate test execution by following an external call from the system to a hardware device (a sensor) through the test harness and back to the system. The test harness acts as a simulated hardware device by "picking up" the function call from the system and registering this as an event ("Read sensor"). The scenario recognizer checks with the test case whether or not this event was expected. The driver will respond to the call once the scenario recognizer has decided that the event was matched. In our example, the test case specifies a branch of the scenario that corresponds to a failed sensor read and the driver therefore returns a -1, instead of a moisture level, that the system will interpret as a failed read.

If there are mismatches between the expected and received events, the scenario recognizer will find out and alert the tester of the particular mismatch.

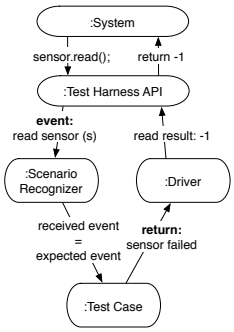

Figure 8: Test execution flow

This testing approach allows us to verify whether or not the scenarios are being followed. If they are not being followed, it helps us determine how and why.

4.4 Test Evaluation

Our testing approach evaluates whether or not the system passes test cases that are derived from the requirements. We can detect two distinct types of mismatches between the actual system behavior and the requirements. First, the scenario recognizer detects mismatches between individual pairs of system events and events expected according to the requirements scenarios. Second, we use oracles for evaluating mismatches that cannot be deduced from the event stream alone. An example of such a requirement is: "AquaLush's water usage does not exceed the zone allocation for any zone". This requirement shows up as a precondition for a sequence of events in IrrigateScenario. We cannot verify this by a single instance of an event, but we can verify that this condition is true eventually by comparing the amount of water that our test case expects each zone to use with the actual amount of water used.

4.5 Automation

From ScenarioML we will (ultimately) be able to generate both test cases that cover all branches of a scenario (reducing each iteration to a specific number of repetitions) and JESS rules for driving and evaluating the tests. The rules necessary to perform and drive the tests are independent of any particular test case, and the test case is in the form of a list of event-response elements which would be easy to generate from a more simply structured artifact, such as a test scenario. What makes the JESS rules particularly interesting (and gives hope for automation) is that there is a direct relation between the rules and requirements. That is where the power lies in terms of directly exposing requirements violations.

5 Validation Study

In the following sections we will describe our experience constructing the test harness, generating tests, and running the tests on a sample software system, AquaLush [6].

5.1 About AquaLush

AquaLush is an automatic irrigation system that controls irrigation based on soil moisture levels rather than timing. The project and its artifacts are available in [6], which provides material such as requirements, use cases, various design models, and approximately 10KLOC of Java code (including the simulator). We chose AquaLush for several reasons; it is an adequate size system with a fair amount of complexity; it was published with a full range of requirements and design artifacts, and it was developed elsewhere for a different purpose than ours.

5.2 Procedure

We constructed a goal graph from the project mission statement, stakeholder-goals list, and the needs list, and we used the requirements and use cases to produce ScenarioML scenarios. We created seven different scenarios based on seven of the eight existing use cases and related these scenarios to goals in the goal graph. We selected all the use cases except for the one that describes the simulation of the system. We excluded this use case because it does not describe the functionality of the system under test, and in fact we will use the simulation as part of our test harness. 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. It is therefore essential that this requirement is implemented correctly and tested sufficiently. IrrigateScenario contains two major event sequences where the second one itself consists of three event sequences. The first event sequence describes reading all the sensors. The system will try to read each sensor at most three times. If it cannot get a successful read within those three tries the system will report that the sensor has failed. This is the event sequence described in figure 3. In the second event sequence the system will (1) open all valves in a zone that needs irrigation, (2) read the zone's sensor once every minute until irrigation must stop (either the zone reached its critical moisture level, used up its zone allocation, or the sensor failed to get a read within three tries), and (3) close all of the zone's valves. After that, the system will open the valves in another zone that needs irrigation, if one exists. Each valve can be manipulated unsuccessfully at most three times, after which the valve is reported to have failed.

To make the existing AquaLush simulator part of the test harness, we modified its code to monitor event occurrences and provide the system with test case input rather than the input that the original simulator used. We also added

code to connect the simulator to the rest of the test harness. We then constructed test scenarios, test cases, driver, scenario recognizer, and oracles, as described in the previous section.

We ran our test cases against the released version of the system as well as several mutants with seeded faults created by MuJava [10]. A compatibility problem between MuJava and Java 1.5 forced us to convert the AquaLush system to Java 1.4. We changed each occurrence of the new enhanced for loop to the regular for loop, each occurrence of an enumerated type to a collection of a general type, and typecasts to access these objects. These were the only changes in the source code of the system under test. The results from the different testing activities (test design, test generation, test execution, and test evaluation) are reported below.

5.3 Test Design

Our testing approach relies on requirements in the form of ScenarioML scenarios. In general, we do the translation from use cases to ScenarioML scenarios in two steps. In the first step, the use cases are directly translated into ScenarioML. We will refer to these as the original scenarios. In the second step, the ScenarioML scenarios are refined by improving the requirements. We will refer to these as the refined scenarios. The second step was part of the test design.

When attempting to map scenario events from the original scenarios to output from the system, we found that several of the events in the scenario were not testable in their original form. One such event is "AquaLush places the zone on an active zone list", with the condition that the sensor of the zone was able to successfully read the moisture level and that the moisture level was below the critical moisture level (meaning that the zone needs watering). The condition plus the scenario event represent one of the original requirements. At first glance this requirement seemed appropriate. However, as we were trying to map the scenario event to an event measurable from the boundary of the system, we realized that this requirement is actually more of a design choice than a requirement. After some discussion, we concluded that the deeper requirement behind this design choice is that only zones with working sensors and moisture levels below the critical level should be irrigated. We therefore refined the scenario by deleting the non-testable internal event and replacing it with a check of the sensor and moisture level conditions as preconditions of the irrigation sequence of the scenario. This refinement of the scenario represents the true requirement that states *what* the system should do, rather than *how* it should do it. Thus we managed to clarify the original stated requirement and, at the same time, identify a requirement that is testable with our black-box approach.

Since we had access to AquaLush and its simulator, we knew that adding an item to the active zone list was not a testable event from the boundary of the system. However, when analyzing the requirement from a requirements perspective, it was clear that it should not be a requirement because it is, in fact, a design choice. In the cases where the scenario events and conditions represent true requirements, we define the test harness API to specify what should be

testable from the boundary of the system. The designers of the system should thus look at the requirements to design the system to be testable.

Another non-testable event in one of the original scenarios is "AquaLush counts the number of working valves in all active zones and adds them to the total number of working valves". This is also a description of how to do something rather that what it should do. The requirement behind this design choice is, we believe, that AquaLush does not use up more water than its zone allocation for any zone, and that the zone allocation is computed by the valve allocation multiplied by the number of working valves in the zone to be irrigated. The valve allocation is computed by the water allocation for the irrigation cycle divided by the number of working valves in the entire system. This form of the requirement states what the system should do, not how, and provides us something to test. There could be several different ways of designing the system to meet this requirement. One design choice is to use the non-testable event we described above and count the number of working valves in all active zones and add them to the total number of working valves. Another design choice could be to keep a counter of all the working valves and update this counter every time a valve fails or gets repaired. We concluded that the requirement in its original form was a premature design decision and replaced the event in the refined version of the scenario with conditions on the irrigation sequence as well as ontology definitions of water allocation, zone allocation, and valve allocation.

There is another peculiarity relating to the original scenario and the event "AquaLush counts the number of working valves in all active zones and adds them to the total number of working valves". The event was placed before the event of opening the valves in the zone to irrigate. Since the zone allocation is dependent on the number of working valves and the valves have not yet been opened, the zone allocation computed by the system could turn out to be based on faulty assumptions. In our refined scenario, since the calculation of the zone allocation is no longer an event but a condition, it is performed in terms of the current state of the world. As we will describe in section 5.6, our oracles will catch mismatches between the zone allocation used by the system and the expected zone allocation according to our interpretation of the requirements. There are several other original scenario events that turned out to be non-testable. In each case we evaluated whether or not the associated requirement really made sense as a requirement. In every instance where we judged the requirement to be badly formed, we were able to rewrite the requirement so that previously associated non-testable scenario events could be eliminated. The result was a better requirement (describing a what instead of a how) that would be testable with a black-box approach.

Checking requirements for testability is one way of validating the requirements through test design, but creating tests from the requirements could also reveal other potential problems with the requirements. The sequence of events in the original scenario failed to address what happens if all valves in a zone fail to open. We interpreted the requirement to be that the system should not attempt to irrigate a zone in which no valves opened. However, when we ran such a test case against our requirements we found a mismatch. It turns out

that the system does attempt to read the sensor in the zone at least once before moving on to open the valves in the next zone to be irrigated. This mismatch is either an implementation fault or a requirements problem. In any case, the requirement as stated by the original scenario did not offer an unambiguous interpretation. There were also several other concepts that were not well explained in the original scenarios. It is, for example, unclear what an *open valve* in AquaLush means. If *open* means that water is flowing out of the valve, what happens if the valve fails to close? The original use case says that the failure is reported to *Persistent Store* and that the use case then continues. But if the water is flowing, wouldn't the zone eventually flood? If it means that water might or might not be flowing when the valve is *open*, how come there is no information about how the system controls the water flow? There are other similar ambiguities, that are not necessarily problems, but which have the potential to be problems. Had we been testers using our requirements-based testing approach in this project, during the project development, we would have had a chance to clarify these requirements before designers, developers, and testers make their own assumptions of what they mean.

5.4 Test Generation

From IrrigateScenario we derived 6 test scenarios. We set the initial parameters of the tests to use four zones. Each zone has exactly one sensor (given in the requirements for AquaLush), and we instantiated each zone to have four valves (a zone can have from 1 to 32 valves). We set the critical moisture level to 50% and the maximum water allocation to 10000 gallons. With these settings, the 6 test scenarios were sufficient to cover all 20 branches of the requirements scenario. In the case of branch pre-conditions expressing alternative conditions, our test scenarios cover each alternative condition. For example, in the case of closing the valves after irrigation, the system can choose the branch either by reaching critical moisture level, by using up the zone allocation, or by failing to read the sensor three times.

We created 6 test cases out of the test scenarios, each instantiated with different concrete input data, for the system under test to follow (see sample test case in figure 7).

Even before running the test cases, in this situation we know that our test scenarios cover 100% of the paths in the requirements scenario. We also know that we are exercising several parts of the ScenarioML ontology. Once we have run the test cases we will also be able to report on successes or failures of particular paths, and provide an estimate of how well the system meets its requirements.

5.5 Test Execution

During test execution, we intercept system output and use this to infer occurrences of events. Once an event has occurred it is matched against the expected event in the test case. If the event is matched, the test harness will mark off

this event from the test case, provide input data to the system if necessary, and then mark the next event to be expected. If the system event does not match the expected event in the test harness, the test harness will alert us of the mismatch.

We ran the six test cases on the released system and recorded the data. We then ran the six test cases on eight different mutants that we generated by using MuJava [10]. Although we generated and tried many more mutants, we chose to report only these eight mutants because of the types of errors they contain. We do not report results from any mutants that caused the system to crash. Examples of such mutants are ones that manipulate loop counters that cause nullPointerExceptions, such as

```
for (int i = 0; ++i < storedString.length(); i++)
```

instead of

```
for (int i = 0; i < storedString.length(); i++)
```

We also decided to exclude the many mutants in which the mutation affected code that our test scenarios do not, and should not, exercise and could therefore not detect. Another type of mutant we decided to exclude from the report is one that manipulates water allocation and water usage, because we had already found a fault in the calculated zone allocation in the released system. Mutants of that sort would, therefore, not provide any new knowledge of the types of faults that our testing approach can find. Furthermore, most of those mutants were acceptable according to our requirements anyway. Since the original scenario for irrigation does not specify clearly when irrigation stops with regard to a sensor read, we allow one minute more or less from what our oracle expects. The mutants that manipulated water usage and water allocation mostly altered these values by one (+1, -1 gallon), which made the mutants uninteresting in any case.

The eight mutants we chose are shown in figure 9.

	Mutation name	Original code	Mutated code	Consequence
1	AutoCycle.start()-AOIS_7	for (int k = 0; **k** < zones.size(); k++)	for (int k = 0; **k++** < zones.size(); k++)	skips a zone when beginning irrigation
2	AutoCycle.assignZoneAllocations()-AOIS_59	for (int m = 0; **m** < zones.size(); m++)	for (int m = 0; **m++** < zones.size(); m++)	skips to assign zone allocation to some zone
3	Sensor.isFailed()-COI_3	return isFailed;	return !isFailed;	switches failed and working sensors
4	Sensor.setIsFailed()-COI_9	isFailed = newValue;	isFailed = **!**newValue;	switches failed and working sensors
5	Zone.isIrrigated()-ROR_14	if (criticalLevel **<=** sensor.read())	if (criticalLevel **>** sensor.read())	sets zone to be irrigated prematurely
6	Zone.closeAllValves()-COD_8	if (!v.isFailed())	if (v.isFailed())	attempts to close non-working valves
7	Zone.openAllValves()-COD_7	if (!v.isFailed())	if (v.isFailed())	attempts to open non-working valves
8	Zone.setCriticalMoistureLevel()-AOIS_42	criticalLevel = theLevel;	criticalLevel = **--**theLevel;	reduces the critical level by one

Figure 9: Mutations

5.6 Test evaluation

The test harness has several components for evaluating the test results. The scenario recognizer will reveal mismatches between expected and actual behavior in the form of monitoring system events and comparing these to test scenario events. The oracles exist to check things that individual system events do not reveal.

It is also worth noting that some mismatches can occur because the requirements are wrong, or because of faults in the test harness. We hope that these types of faults would be minimal when using this approach, because the approach validates the requirements early and because the test harness can be constructed without access to the system under test. We can simulate the use of the test harness by giving it event sequences and hopefully flush out most test harness faults before testing of the system.

5.6.1 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 irrigating at the time and had actually been reported as not working. The second test case we ran showed the same mismatch and it had also been determined that sensor S1 was not working during the first event sequence of the test case. This mismatch showed that the system did something it should not have done.

```
...

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.

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

...
```

We also need to detect other forms of mismatches with our oracles. In the ScenarioML ontology, we defined the water allocation to be the maximum

amount of water that can be used in an irrigation cycle (over all zones). We defined the valve allocation to be the water allocation divided by the number of working valves and the zone allocation to be the valve allocation multiplied by the number of working valves in a particular zone. We know that we need to calculate the zone allocation before a zone begins to get irrigated, because we use reaching the zone allocation as one of the possible conditions for stopping the irrigation of a zone. We therefore implemented an oracle that computes these values based on the number of working valves and the number of zones needing to be irrigated at any given time during the test. We also know that the water usage in a zone is the amount of time that the zone has been irrigating multiplied by the flow rate in the zone. Using this information our oracle could infer whether or not the system had calculated the zone allocation correctly. From looking at the original scenario, we already have several indications that this calculation might be faulty, since it was expressed as an event before the sequence of opening valves in a zone. Also, the previous mismatch of trying to read a sensor in a zone whose sensor was reported to be out of order, could potentially lead to an incorrect calculation of the zone allocation if that zone's valves were used in the calculation for the valve allocation. When we ran the third test case, a test case that uses up the entire zone allocation because the moisture level never reaches the critical moisture level in two of the four zones, the oracle detected a mismatch in the amount of water used and the amount of water the oracle expected the system to use (see test output below). The water allocation is set to 10000 gallons, and there are four zones. The system started to irrigate the second zone, in which one of the valves failed to open. The oracle calculates the valve allocation to be 10000 gallons divided by 15 working valves, then calculates the zone allocation to be the valve allocation multiplied by 3, which is the number of working valves in the zone. The zone allocation for the first zone to be irrigated is therefore 2000 gallons. The system continues to read the sensor until it has used up 3312 gallons of water, which is more than our oracle expected. Later in the event trace, we detected that the system used up less than the oracle was expecting (see second test output below). Another test case showed that our interpretation of the zone allocation requirement makes sense. In that test case, one zone is not irrigated because its sensor fails to read and two other zones are not irrigated because their moisture levels are above the critical moisture level. Our oracle correctly calculates that the last zone should have the entire water allocation (10000 gallons). The other three test cases also showed mismatches of attempting to read a failed sensor during irrigation of another zone and not calculating the zone allocation correctly.

...

```
A read sensor event for zone S2 was received as
expected while irrigating.

Start closing valves in zone S2 because zone
used up its zone allocation.
```

```
zone used: 3312 gallons of water,
zone allocation: 2000.0.

A close valve event for V06 was received as
expected.

...

...

A read sensor event for zone S4 was received as
expected while irrigating.

Start closing valves in zone S4 because zone
used up its zone allocation.
zone used: 2256 gallons of water,
zone allocation: 3320.0.

A close valve event for V15 was received as
expected.

...
```

5.6.2 Testing Mutants

We found mismatches in all eight mutants that we are reporting on in this paper. As described previously, the seeded faults that our approach did not find were either faults that crashed the system, valid implementations according to the requirements, already reported faults in the system, or not in the code that the test scenarios exercise.

The first mutant skips some zones when beginning the irrigation scenario. The mutation has altered the counter in the for loop from `for (int k = 0; k < zones.size(); k++)` to `for (int k = 0; k++ < zones.size(); k++)`. We found this fault as a mismatch between our test cases and the system (see test output below). The system output a "Read sensor" event for sensor S4 and the test case provided input that the read was successful on the first try. The system then output a "Read sensor" event for sensor S3 and the test case provided input that the attempt failed. The system then tried to read the same sensor two more times and on the third attempt the read was successful. The scenario recognizer then expected the system to try to read either sensor S1 or S2, but instead the test harness received an "Open valve" event from the system. The system had looped through what it believed to be the list of zones. However, that did not correspond to the actual list. Thus, the scenario recognizer caught a mismatch between the expected and actual sequence of events.

```
A read sensor event for S4 was received as
expected.
```

```
A read sensor event for S3 was received as
expected.
A read sensor event for S3 was received as
expected.
A read sensor event for S3 was received as
expected.

An unexpected Open valve event with parameter
V16 was detected.
This was unexpected because the system is
currently attempting to activate a zone which
means it is expecting a read sensor event.
```

The second mutant uses a similar loop fault to skip some zones when assigning zone allocations. Our scenario recognizer found this mismatch as can be seen in the test output below. The system read the sensor and determined that a zone needed irrigation. It then successfully opened the valves in that zone. The scenario recognizer was expecting to receive a "Read sensor" event next as part of the irrigation sequence, but the system output a "Close valve" event instead. Apparently, the system could not check the zone allocation for this zone and decided to close the valves and move on.

```
...

An open valve event for V09 was received as
expected.

A Read sensor event was expected but a
Close valve event occurred.
```

The third mutant switches working and non-working sensors. Our scenario recognizer found this mismatch right after the initial sequence of reading the sensors. The system output events to read the sensors and S1 failed to read three times. The mutated system however thinks that this sensor is working and therefore outputs an "Open valve" event for that zone. The scenario recognizer detects this mismatch, because the test case has already determined that sensor S1 does not work.

```
A read sensor event for S2 was received as
expected.
A read sensor event for S1 was received as
expected.
A read sensor event for S1 was received as
expected.
A read sensor event for S1 was received as
expected.
A read sensor event for S4 was received as
```

```
expected.
A read sensor event for S3 was received as
expected.

An unexpected Open valve event with
parameter V02 was detected.
This was unexpected because the system has
concluded that the zone is inactive.
```

We found similar types of mismatches for the rest of the eight mutants as well.

5.7 Results

We have demonstrated the effectiveness of our approach to requirements-based testing by identifying a number of ways in which the published version of the Aqualush system does not satisfy its requirements. The traces produced by the test harness comparing the actual system behavior with expected behavior clearly indicate the nature of the requirements violations in each of our six test cases. This means that the requirements scenario IrrigateScenario was not achieved by the system and that the high-level stakeholder goals that depend on this scenario were not satisfied.

We have further validated our approach by demonstrating that we can detect errors that result from mutations of the system. Our test cases were able to detect all the seeded errors.

Finally we have demonstrated that there is great potential benefit in the process of creating requirements-based tests early on in the development cycle. We have shown this by presenting several examples where what appeared to be valid requirements at the start revealed themselves to be, in reality, design choices. These disguised design choices were detected while trying to map the ScenarioML descriptions of the requirements onto the observable behavior of the system.

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. On the other hand we have pattern-directed rule-based programming technology as embodied by Jess. What makes the JESS rules particularly interesting is that there is a direct relation between the rules and requirements. That is where the 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.

6 Conclusion

In this work we have proposed requirements-based testing as a particularly effective specification-based testing approach. In our requirements-based testing approach, tests are directly driven by and checked against requirements in the form of scenarios. Requirements-based testing utilizes the natural symbiotic relation that exists between requirements and testing to produce software that better satisfies stakeholder requirements.

Our evaluation study has provided us with evidence that our approach is effective in revealing both potential specification problems and implementation faults. As stated in the introduction, 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 plugin 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 postconditions as constraints and search through the scenario paths with those constraints until we have enough test scenarios to cover all branches. As explained previously the number of particular instances of an instance type can impact the number of test scenarios needed to cover all branches. We will auto-generate test cases in the form of JESS event-responses from the test scenarios by either randomly selecting input data that satisfies the path, or by manually selecting data. We are also looking into ways of automating the production of test driving code, event recognizing code, and oracles from ScenarioML scenarios. We are aware that we might need to extend the ScenarioML language to be able to express information needed for testing but which would not be necessary for the requirements themselves. We are also working on improvements for executing tests and collections of tests automatically.

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

[2] 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.

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

[4] L. C. Briand and Y. Labiche. A UML-based approach to system testing. *Software and System Modeling*, 1(1):10–42, 2002.

[5] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Springer, 2000.

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

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

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

[9] A. v. Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Trans. on Softw. Eng*, 24(12):1089–1114, 1998.

[10] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system. *Softw. Test, Verif. Reliab*, 15(2):97–133, 2005.

[11] H. Muccini, M. S. Dias, and D. J. Richardson. Systematic testing of software architectures in the C2 style. In *FASE'04*, volume 2984 of *Lecture Notes in Computer Science*, pages 295–309. Springer, 2004.

[12] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jézéquel. Automatic test generation: A use case driven approach. *IEEE Trans. Softw. Eng.*, 32(3):140–155, 2006.

[13] T. O. O'Malley, D. J. Richardson, and L. K. Dillon. Efficient specification-based oracles for critical systems. In *Second California Software Symposium (CSS'96)*, Apr. 1996.

[14] C. Rolland, C. Souveyet, and C. Ben Achour. Guiding goal modeling using scenarios. *IEEE Trans. on Softw. Engineering.*, 24(12):1055–1071, 1998.

[15] A. S. Tanenbaum. In defense of program testing or correctness proofs considered harmful. *SIGPLAN Not.*, 11(5):64–68, 1976.

[16] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[17] M. W. Whalen, A. Rajan, M. P. Heimdahl, and S. P. Miller. Coverage metrics for requirements-based testing. In *International Symposium on Software Testing and Analysis (ISSTA '06)*, 2006.

[18] K. Winbladh, T. A. Alspaugh, H. Ziv, and D. J. Richardson. Architecture-based testing using goals and plans. In *ROSATEA'06*, 2006.

[19] K. Winbladh, T. A. Alspaugh, H. Ziv, and D. J. Richardson. An automated approach for goal-driven, specification-based testing. In *21st International Conference on Automated Software Engineering (ASE 2006)*, pages 289–292, 2006.

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