jUnit^{RV}—Adding Runtime Verification to jUnit

Normann Decker, Martin Leucker, and Daniel Thoma

Institute for Software Engineering and Programming Languages
Universität zu Lübeck, Germany

{decker, leucker, thoma}@isp.uni-luebeck.de

Abstract. This paper presents jUnit^{RV} as a tool extending the unit testing framework jUnit by runtime verification capabilities. Roughly, jUnit^{RV} provides a new annotation `@Monitors` listing monitors that are synthesized from temporal specifications. The monitors check whether the currently executed tests satisfy the correctness properties underlying the monitors. As such, jUnit's concept of plain assert-based verification limited to checking properties of single states of a program is extended significantly towards checking properties of complete execution paths.

1 Introduction

Testing is the verification technique that is most applied in practice. Yet, testing is still quite ad-hoc, time consuming and as such, expensive. Easily, testing of software systems consumes up-to 50% of total development costs in safety-critical systems.

One of the most popular testing approaches to Java code is unit testing based on the jUnit framework [1]. Unit testing is essential in test-driven development such as extreme programming but also common when following classical development models.

Runtime verification is still a rather new verification technique in which a formal correctness property is checked on the actual execution of a system under scrutiny. Typically, monitor code checking the property at hand is synthesized and interweaved with the underlying program. Then, any execution of the resulting program is checked with respect to this property.

In this paper, we present jUnit^{RV} as a tool combining the ideas of unit testing and runtime verification.[1] It allows for high-level specifications of monitors for temporal assertions within the jUnit framework. Testing temporal properties commonly leads to complicated test cases and may require modifications to the application code. In jUnit^{RV}, monitors can be annotated to single test cases to automatically check the corresponding properties during execution.

While there are several runtime verification frameworks (see [2] for a recent overview), none of the available tools provides a close integration into jUnit.

In the next section, we give a brief overview on how to use jUnit and jUnit^{RV}. Afterwards, we discuss technical issues of our tool. Section 3 details how temporal

[1] jUnit^{RV} is freely available at http://www.isp.uni-luebeck.de/junitrv.

specifications are related to program executions and Section 4 describes how monitoring is integrated into the jUnit framework.

2 jUnit$^{\mathrm{RV}}$—A quick starting guide

In this section, we introduce jUnit$^{\mathrm{RV}}$ by means of an example. We recall the ideas of jUnit, explain current limitations and show how jUnit$^{\mathrm{RV}}$ can simplify testing of so-called *temporal assertions* by means of runtime verification.

Testing and jUnit. The aim of unit testing is to check simple, individual units of a program. While jUnit is originally developed to support unit testing, it allows, in principle, for complex test scenarios and it is often used for integration testing and system testing in practice as well.

Let us explain the main ideas about jUnit based on the following, exemplifying hospital application: For every patient, the hospital personnel takes the necessary data and submits it to the central hospital information system. The information may be queried and modified later on. For this, the hospital personnel uses terminals which may be shared by different users by switching between the respective accounts.

The terminal runs a Java application which takes care of user management and modification of patient data. To access and modify patient data in the hospital information system, it uses the following (simplified) interface.

```
public interface DataService {
  void connect(String userID) throws UnknownUserException;
  void disconnect();
  Data readData(String field);
  void modifyData(String field, Data data);
  void commit() throws CommitException;
}
```

The terminal application, called client in the following, is to be tested whether it meets the following requirement: If data was modified through the interface, the client must instruct the data service to commit the changes before the user logs out since local changes would be lost otherwise. A user is logged out from the system, e.g. when the client is shut down or the user is switched, and, as such, the requirement has to be tested at different functions of the application.

Within Java's unit testing framework jUnit, test cases are specified in dedicated test classes. A test case in jUnit is a method comprising the annotation `@Test` and a sequence of method calls to be executed. Additionally, assertions are used to specify expectations to the program state at certain steps. jUnit loads a specified test class and consecutively invokes all included test cases. A typical test case for the requirement mentioned above looks as follows.

```
@Test
public void test1() {
  DataService service = new MyDataService("http://myserver.net");
  MyDataClient client = new MyDataClient(service);

  client.authenticate("daniel");
  client.addPatient("Mr. Smith");
```

```
    client.switchToUser("ruth");
    assertTrue(service.debug_committed()); // switching means logout

    client.getPatientFile("miller-2143-1");
    client.setPhone("miller-2143-1", "012345678");
    client.exit();
    assertTrue(service.debug_committed());
  }
```

The difficulty of using jUnit in this example is twofold: (i) for executing the test case above the implementation must be refactored to provide enough information to indicate whether a commit has happened. Moreover, (ii) the tester needs the information which methods actually perform a logout (`switch()` and `exit()` in our example).

Clearly, the need for complete knowledge of such information as well as the need for refactoring e.g. an interface in late development phases makes testing labor-intensive and error prone. In essence, the problem in the example above is that a requirement on the execution trace should be checked while jUnit only supports assertions to be checked in individual states of the system.

Runtime Verification and jUnit$^{\mathbf{RV}}$. Runtime verification (see [3] for a survey) aims at verifying properties on individual execution traces. To this end, *temporal assertions* may be specified, typically in terms of temporal logic formulae, and are automatically translated into a so-called monitoring code. A monitor is a program that observes the current execution and yields a verdict whether the property is fulfilled or violated.

The requirement in the example above can be stated as

$$\texttt{Always}\ (\texttt{modify} \Rightarrow \neg\texttt{disconnect Until committed})$$

meaning it is always the case, that whenever the method `DataService.modify()` is invoked, the client does not disconnect until a call to `DataService.commit()` returned successfully. The link between formal events and method calls are made explicit in jUnit$^{\mathrm{RV}}$ as follows:

```
String dataService = "myPackage.DataService";

private static Event modify = called(dataService, "modify");
private static Event committed = returned(dataService, "commit");
private static Event disconnect = called(dataService, "disconnect");
```

Note that, besides *events*, jUnit$^{\mathrm{RV}}$ also supports *propositions*. The distinction is made precise in the next section. A corresponding monitor definition within the jUnit$^{\mathrm{RV}}$ framework can be given as follows.

```
private static Monitor commitBeforeDisconnect = new FLTL4Monitor(
  Always(implies(
      modify,
      Until(not(disconnect), committed)
    )
  ));
```

jUnit$^{\mathrm{RV}}$ is in general capable to deal with different logic plug-ins but comes with a DSL for specifying temporal assertions in the temporal logic FLTL$_4$, which follows [4] and is defined formally in [2].

Individual test cases can now be monitored by just adding an annotation @Monitors together with a list of monitor names that have been defined before. For our example, we get:

```
@Test
@Monitors({"commitBeforeDisconnect"})
public void test1() {
  DataService service = new MyDataService("http://myserver.net");
  MyDataClient client = new MyDataClient(service);

  client.authenticate("daniel");
  client.addPatient("Mr. Smith");
  client.switchToUser("ruth");
  client.getPatientFile("miller-2143-1");
  client.setPhone("miller-2143-1", "012345678");
  client.exit();
}
```

3 Execution Traces and Formal Runs

jUnit^{RV} allows for specifying temporal assertions in jUnit. Such specifications can be annotated to test cases and are monitored during test execution. At every execution step, the monitor reports a (possibly preliminary) verdict. The test case fails, if the monitor reports a violation of the property during the execution.

The monitor specifications are based on temporal logic, which describes discrete sequences of observations, i.e. individual steps in time. At every such time step, atomic propositions are assumed to evaluate to either true or false. However, the actual observation that is made and the user intends to describe is the *execution trace* or run of a program.

Such a run includes e.g. method invocations and returns, variable access and variable evaluations. To use temporal logic as a tool to describe program runs, the mapping between formal semantics and program traces must be clear, intuitively as well as formally. We therefore introduce our notion of *events* and *propositions*. In first place, events serve as clock triggers to the monitors, thereby defining the discrete steps in time. Additionally, propositions characterize the current program state within such a discrete time unit. They are evaluated within the scope of an event, i.e. a specific time instant.

Events. In jUnit^{RV}, events are specified explicitly and are automatically triggered. The temporal assertion in our example above uses the events `modify`, `committed` and `disconnect`. Such events, mark specific actions of the program. The events `modify` and `disconnect` trigger as soon as the methods `modify()` and `disconnect()`, respectively, are invoked on an object of type `DataService`. The event `committed` occurs when the method `commit()` returned successfully, i.e. without throwing an exception. Each monitor is associated with a set of events and whenever one of them occurs, a time step is indicated to it.

Propositions. Within the context of a particular time step, propositions are evaluated and define the *current* observation. This evaluation defines which transition a monitor takes in the current step. In jUnit$^{\text{RV}}$, propositions are defined explicitly as follows:

```
private static Proposition auth =
    new Proposition(eq(invoke($this, "getStatus"),AUTH);
```

The proposition `auth` evaluates to true if the method `getStatus()` returns the value `AUTH`. The method is invoked on the *current* object (denoted `$this`), which is the object on which the method was invoked that *caused the current event*, i.e. the current time step.

Additional propositions are defined implicitly in terms of events: For each event there is a proposition with the same name that can be used in the temporal specification. Note that in any time step, only a single event can occur and thus the propositions implicitly defined by events exclude each other. For example, the property `modify` $\wedge$ `committed`, meaning that the events `modify` and `committed` occur at the same time, can never be true. In the data service example, the specified property only uses propositions that are defined implicitly by the corresponding events.

4 jUnit Integration

The jUnit testing framework comes with sophisticated default test case execution capabilities. Moreover, it provides the possibility to change the test execution behavior with the help of annotation `@RunWith`, which takes as argument a suitable test runner class. As jUnit$^{\text{RV}}$ has to take care of event injection and monitor execution, it provides the class `RVRunner`. To reuse most of jUnit's standard test runner, like its reporting facilities etc., `RVRunner` inherits from jUnit's test runner.

The notion of events is bound to the access of fields and invocation or return of methods in the program under test. That means that the program must be interleaved with code being executed whenever a respective method is invoked. As the classes to be tested are compiled and already loaded by jUnit, when they are about to be tested with `RVRunner`, monitoring code cannot be added to the byte code directly. For code injection, `RVRunner` uses the following idea: It creates a customized class loader that will inject corresponding code when loading classes. It then reloads all involved classes using this custom class loader, which now adds the monitoring code into the program under test. Our framework uses the Javassist library [5] that provides the functionality to manipulate the Java bytecode at load-time of Java classes. For test execution, `RVRunner` delegates to jUnit's the default implementation preserving the standard functionality.

While jUnit$^{\text{RV}}$ maintains the monitor state, recognizes events and evaluates propositions, the behavior of monitors is provided by the implementation of a single interface `Mealy` which basically represents the transition function and output labeling of some deterministic (possibly infinite-state) Mealy machine. That is, jUnit$^{\text{RV}}$ provides the current state and proposition evaluations and expects the subsequent monitor state. The implementation of a monitor construction remains independent of the state and event management. This easily allows for the integration of custom monitoring approaches.

Since all required classes are loaded by the jUnit framework, jUnit$^{\mathrm{RV}}$ can be deployed as a standard jar-archive and integrated into any common testing environment, it suffices to make jUnit$^{\mathrm{RV}}$ available through the Java class path. The tool works with common IDEs, e.g. Eclipse or Netbeans as it leverages the jUnit test integration.

A major advantage of manipulation of byte code runtime verification is, that it allows to insert event generation routines even into third party code where the sources are not available. jUnit$^{\mathrm{RV}}$ is hence also independent of the programming language of the target program as long as it is run on the JVM. Testing Scala applications or libraries, for example, is thus also possible.

Note that, in principle, manipulation of byte code must be treated with care as the tested and deployed byte code differ. However, we consider this uncritical in most practical cases. Additionally, jUnit$^{\mathrm{RV}}$ allows for deploying the instrumented application, i.e. including all modifications.

5 Conclusion

In this paper, we introduced jUnit$^{\mathrm{RV}}$ as a tool extending the unit testing framework jUnit by runtime verification capabilities. Within jUnit, test cases are specified manually together with assertions that are evaluated in the corresponding states of the system under scrutiny. Using jUnit$^{\mathrm{RV}}$, it is now possible to specify temporal assertions that specify correctness properties for complete test runs. As such, test case specification is simplified significantly in many situations. In the near future, we plan a case study with a larger number of users to investigate jUnit$^{\mathrm{RV}}$'s usability in practical applications, including scalability under larger test suites and the practical overhead.

References

1. Beck, K., Gamma, E.: Test-infected: programmers love writing tests. In Deugo, D., ed.: More Java Gems. SIGS Reference Library, Cambridge University Press (2000) 357–376
2. Leucker, M.: Teaching runtime verification. In Khurshid, S., Sen, K., eds.: RV. Volume 7186 of Lecture Notes in Computer Science., Springer (2011) 34–48
3. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. **78**(5) (2009) 293–303
4. Lichtenstein, O., Pnueli, A., Zuck, L.D.: The glory of the past. In Parikh, R., ed.: Logic of Programs. Volume 193 of Lecture Notes in Computer Science., Springer (1985) 196–218
5. Chiba, S.: Load-time structural reflection in java. In Bertino, E., ed.: ECOOP. Volume 1850 of Lecture Notes in Computer Science., Springer (2000) 313–336