UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

Stream-Based Verification with JUnit

Strombasierte Verifikation mit JUnit

Masterarbeit

Im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

Vorgelegt von
Denis-Michael Lux

Ausgegeben und betreut von
Prof. Dr. Martin Leucker

mit Unterstützung von
Malte Schmitz, M.Sc.
Dipl.-Inf. Daniel Thoma

Lübeck, den 14. Juni 2019

Selbstständigkeitserklärung

Der Verfasser versichert an Eides statt, dass er die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt hat.

Die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Ort/Datum	Unterschrift des Verfassers

Contents

Abstract

In this thesis a concept and the associated implementation of a JUnit 5 test engine is presented and explained, which integrates the testing framework JUnit, the JUnit runtime verification library jUnit^{RV}, the stream runtime verification tool TeSSLa and a mocking framework adapted to these tools. An integration of these tools has the advantage that by simultaneously executing and monitoring test cases, the respective specification can not only be used for verification, but can also influence the execution of the application code as a disruptive factor in order to investigate additional aspects of the tested software. This integration paves the way for monitoring traditional unit tests during their execution using monitors synthesized from LTL or TeSSLa specifications and simultaneously influencing application code execution using mock objects connected to these monitors. On the one hand the strengths of LTL and TeSSLa based verification can be combined and on the other hand existing JUnit test cases can retrospectively be monitored and influenced by monitors with little effort.

Zusammenfassung

In dieser Abschlussarbeit wird ein Konzept und die dazugehörige Implementierung einer JUnit 5 Test-Engine, welche das Testing-Framework JUnit, die JUnit Runtime Verification Library jUnit^{RV}, das Stream Runtime Verification Werkzeug TeSSLa und ein auf diese Werkzeuge abgestimmtes Mocking-Framework miteinander vereint, dargelegt und erklärt. Eine Integration der genannten Werkzeuge bringt den Vorteil mit sich, dass durch die gleichzeitige Ausführung und Überwachung von Testfällen die jeweilige Spezifikation nicht nur zur Verifikation dient, sondern daneben auch die Ausführung des Anwendungscodes als Störfaktor beeinflussen kann, um so zusätzliche Aspekte der getesteten Software untersuchen zu können. Dies öffnet die Türen dafür, herkömmliche Unit-Tests während der Ausführung mittels Monitoren, die aus LTL oder TeSSLa Spezifikationen synthetisiert werden, zu überwachen und gleichzeitig mittels diesen Monitoren verknüpften Mock-Objekten Einfluss auf die Ausführung des Anwendungscodes zu haben. Dadurch lassen sich zum einen die Stärken von LTL und TeSSLa basierter Verifikation vereinen und zum anderen können bestehende JUnit Testfälle mit geringem Aufwand nachträglich durch Monitore überwacht und beeinflusst werden.

Introduction

Software systems are becoming an increasingly important part of everyday life. Especially in embedded systems, software ensures that entire systems in areas such as aerospace, medicine and automotive work in a controlled and reliable manner. A very important aspect in the planning and commissioning of such systems is functional safety. Particularly, in system development according to the international standard IEC 61508 the use of unit testing is indispensable, but also techniques as *runtime verification* (RV) in conjunction with unit testing can be a reasonable alternative to conventional verification techniques like model checking [CGP99] and theorem proving [BC10].

As mentioned by [Mye11], the rule of thumb in software engineering is that in a typical programming project about 50 percent of the time elapsed and more than 50 percent of the total cost is spent on testing the developed program or system. Examples such as the *Therac 25* incidents [LT93], where patients were harmed or died due to radiation overdoses, or the *Pentium FDIV bug* [Pra95], where missing values in a look-up table led to incorrect floating-point division results, show that testing in software engineering is essential to ensure the reliability of programs or systems and to avoid unexpected costs. The high investment of time and resources for testing requires the use of appropriate testing techniques during development. The RV technique mentioned above complements traditional techniques such as testing. The difference to other verification approaches is that RV, as the name suggests, is performed during the runtime and can therefore react to incorrect software behavior on detection. Both the testing and the RV technique are equally powerful and thus allow the mutual exchange, but both techniques have different strengths. Checking the behavior of components, for example, is easier with RV, while the use of unit testing to check the system state is less complicated. Thus, to make a selection of verification techniques for a software project, the respective tools as well as their interfaces, their performance, their properties and their compatibility with other tools must be considered. Another important aspect is that RV and unit testing can be performed simultaneously and independently. The consideration of this aspect in the design and implementation of RV and unit testing frameworks allows a simple monitoring of existing test cases with the help of RV, whereas a test case to be monitored can easily be executed as a simple test case. The main objective of this thesis is to integrate the *JUnit* Testing-Framework, the RV library *jUnit*$^{\mathrm{RV}}$, the *stream runtime verification* (SRV) tools related to the *Temporal Stream-based Specification Language* (TeSSLa) and a mocking framework tailored to these tools, into a JUnit testing framework, which can be used in conjunction with the conventional JUnit 5 testing frameworks.

The following chapters introduce the new testing framework *JUnitSRV*. During design and implementation, it is taken into account how unit testing is generally practiced and how the

TeSSLa specification language works. Therefore, in chapter 1 testing, in particular *unit testing* and related techniques such as *assertions*, *mock objects* and *stubs* will first be explained and discussed. Chapter 2 then introduces the SRV language TeSSLa, starting with a general explanation of RV. Then, SRV in the form of TeSSLa is presented using examples. The insights from chapters 1 and 2 are used in chapter 3 to explain the concepts for implementing the JUnitSRV framework. Chapter 4 discusses the most important aspects of implementing these concepts. Therefore, the jUnit$^{\mathrm{RV}}$ library is introduced first. The jUnit$^{\mathrm{RV}}$ library was developed for the JUnit 4 framework, which was the latest version at the time of deployment. However, because the JUnitSRV framework is designed for the latest features of JUnit, it is based on JUnit 5. The architecture of JUnit was completely revised between versions 4 and 5. One of the challenges in the development of JUnitSRV is therefore the retention of the functions of the jUnit$^{\mathrm{RV}}$ library. In order to extend the JUnit 5 infrastructure, it is necessary to provide a custom test engine. This test engine uses the jUnit$^{\mathrm{RV}}$ library for code instrumentation. A new monitor type for the TeSSLa specifications is also designed and implemented. One of the biggest features of JUnitSRV is a lightweight mocking framework. By using the external library *Javassist*, JUnitSRV is able to create mock objects at runtime by creating classes as *bytecode* on the fly, which are then passed directly to the *java virtual machine* (JVM) when needed. The difference between conventional mocking frameworks such as *Mockito* [Moc19] and the built-in JUnitSRV mocking framework is the ability to configure the behavior of these mock objects via a TeSSLa monitor. Chapter 5 gives code examples for three scenarios. Each scenario highlights one of the main features of the JUnitSRV framework.

⋆ ⋆ ⋆ ⋆ ⋆

I would like to thank my supervisor Prof. Dr. Martin Leucker for the opportunity to write this thesis at the Institute for Software Engineering and Programming Languages. Another special thanks goes to Daniel Thoma for the supportive discussions about the implementation that emerged from this work. I would also like to thank Malte Schmitz for his efforts in reading this thesis and making many helpful comments. Without the many comments of my supervisors, the thesis today would not look the way it does.

At this point I would also like to thank my family and the Brauer family for their support during my studies. My special thanks go to Susan Brauer, who supported me personally and financially especially during the last months of my studies and especially in the completion of this thesis. Without her support, this thesis would probably never have come about.

1 Unit Testing and Mocking in JUnit

In software development, testing large programs with many instructions and many classes can be a challenge. The *unit testing* [Mye11] technique presented in this chapter is a strategy for overcoming this problem. It examines the behavior of individual components, also referred to as *units* of a system. The following sections cover unit testing and other related techniques such as *assertions*, *mock objects*, and *stubs*. In addition, the *JUnit* framework is introduced to show how these techniques can be implemented and applied in the context of Java. The methods in the following sections are illustrated by small listings that refer to a working example composed of the three units presented in listing 1.1.

```
public interface Account {
  int getBalanceInCents();
  void setBalance(int cents);
  String getName();
  String getEMail();
  String getPhoneNumber();
}

public interface ATM {
  void withdraw(int cents, Account account) throws NotEnoughMoneyException;
  void deposit(int cents, Account account);
}

public interface NotificationService {
  void send(String message, Account account);
}
```

Listing 1.1: An example application that is comprised of the three modules `Account`, `ATM` and `NotificationService`. The `Account` is the technical implementation on the server side that stores customer information such as balance, customer id, e-mail address and phone number. The `ATM` is the device with which a customer can change his balance by withdrawing and depositing cash. The `NotificationService` is responsible for informing customers of any withdrawal or deposit of money on their `Account`.

All interfaces listed in listing 1.1 are *modules* or *units* of the application they form. According to the *Dependency-Inversion Principle* (DIP) reintroduced by [Mar03], these modules are defined as interfaces to separate implementation details from abstractions. This separation decouples high-level implementations from low-level details, increasing code reusability and facilitating unit testing. The example is a simplified model of a financial institution consisting of the three modules `Account`, `ATM` and `NotificationService`. The `Account` is the implementation of a server-side object that stores customer information such as balance, customer id, email address, and phone number. The `ATM` is the device that allows customers to change their balance on the corresponding account object by withdrawing and depositing cash. `NotificationService` is a service that notifies customers each time money is withdrawn or deposited to the appropriate account. Because the `NotificationService` object receives an `Account` instance, it has access to the customer's email address or phone number and can decide what type of notification to perform. This service can, for example, send noti-

fications by SMS or e-mail or not send notifications at all. In the following sections, concrete implementations for these interfaces are presented if required.

1.1 Unit Testing

There are many techniques for testing software, including *load testing* [Jia15], which generally evaluates the system under load; *performance testing* [Mol09], which focuses on meeting non-functional requirements such as required response times and expected user numbers; *conformance testing* [TKS03], which focuses on determining whether an application under test conforms to its protocol specification; and *acceptance testing* [Mye11], where the customer of the program determines whether the program meets the original requirements by comparing both. The purpose of all these testing strategies is to demonstrate that the program or system is not in compliance with its specification and is therefore behaving incorrectly. One of the most common testing methods in software development is *unit testing*, also called *module testing* [Mye11]. Unit testing compares the behavior of the unit or module with the functional or interface specification by which it is defined. As [Mye11] explains, unit testing is the process of testing individual subroutines, classes, or procedures of the program. This means that the program is not tested as a whole, but the test concentrates on the smaller components of the program. After testing these building blocks, individual units dependent on them can be tested, which is also referred to as *integration testing* or *interaction testing* [MS01]. In *object-oriented programming* (OOP), a unit can be either a single method, a class, or even a subsystem. In [Lin05] a unit is described as a natural abstraction unit, i.e. the class or its instantiated form: the object. In the following, the currently tested unit is referred to as the *system under test* (SUT) [PL05]. As mentioned by [Mye11], the motivation of unit testing is to facilitate the task of debugging, since errors that occur during testing are known to be present in the tested unit. In addition, [Mye11] mentions that unit testing provides the ability to test modules in parallel, thereby increasing performance.

In the following sections, the words *test case* and *test suite* are often used in the context of unit testing. According to the definitions of [PL05], a test case is a structure of input and expected output behavior, while a test suite is a set of test cases. Test suites can include assumptions about the environment and configuration of the SUT, among other aspects.

1.2 The JUnit Testing Framework

The testing framework used in the following sections and chapters is an open source testing framework for automated object-oriented unit testing in Java and is known as JUnit[1]. The JUnit 4 framework was first developed by Kent Beck and Erich Gamma and is now being

[1] `https://junit.org/junit5/`

further developed as a SourceForge[2] project. As explained by [Lin05; Lin17], one of the JUnit 5 developers, the JUnit framework is from a historical point of view a descendant of *SUnit*, a similar framework for Smalltalk. In 2009, the latest version of JUnit 4 (4.7) was released, which was replaced by JUnit 5 (5.0.0) in 2017. During the development of JUnit 5, the entire JUnit 4 architecture was revised and rebuilt with a view to greater flexibility and the provision of Java 8 language features. The main objectives of the design were to separate the aspects of JUnit as a tool and JUnit as a platform. In addition, JUnit 4 and 5 should coexist and run simultaneously to facilitate test migration. Finally, JUnit 5 should be freely combinable with other frameworks and extendable by custom testing frameworks.

In the following, unit testing is presented using a practical example with JUnit 5. For this purpose, an implementation of the `ATM` interface shown in listing 1.1 is used. Listing 1.2 describes the implementation of an off-site `ATM`. The `NotificationService`, which informs customers about withdrawals and deposits on their accounts, is passed as a parameter to its constructor and thus created outside this class.

```
public class OffSiteATM implements ATM {
  public NotificationService service;

  public OffSiteATM(NotificationService service) {
    this.service = service;
  }

  public void withdraw(int cents, Account account) throws NotEnoughMoneyException {
    if (cents <= 0 || account == null)       throw new IllegalArgumentException();
    if (account.getBalanceInCents() < cents) throw new NotEnoughMoneyException();
    account.setBalance(account.getBalanceInCents() - cents);
    service.send("Balance decreased by " + (cents/100.0) + " EUR", account);
  }

  public void deposit(int cents, Account account) {
    if (cents <= 0 || account == null)       throw new IllegalArgumentException();
    account.setBalance(account.getBalanceInCents() + cents);
    service.send("Balance increased by " + (cents/100.0) + " EUR", account);
  }
}
```

Listing 1.2: An implementation for the `ATM` interface. `OffSiteATM` uses a `NotificationService` to inform customers about transactions. The `OffSiteATM.withdraw` method checks whether there is enough balance on the existing account to withdraw the specified amount of money. It then reduces the balance by the amount withdrawn and finally informs the customer of the change. With the `OffSiteATM.deposite` method, the balance is increased by the specified amount and finally the customer is informed about the change.

A brief description of the SUT, i.e. the implementation described in listing 1.2, is as follows: the `ATM`'s task is to store transactions for customers on their `account`. On the physical device, the customer enters the amount of money he wants to withdraw. This transaction is then applied by the `ATM` object to the corresponding `account` instance. If there is not enough money on the `account`, the `ATM` displays a message to the user. When depositing money, the user gives cash to the device, which then stores the amount on the appropriate `account`

[2] `https://sourceforge.net`

instance. The customer should not be able to enter negative numbers at the `ATM`. For each transaction, the `NotificationService` should send a notification of the transaction to the customer.

Two types of information are required to design test cases: a specification for the unit being tested and the source code of that unit [Mye11]. The specification is usually a description of the input and output parameters of the unit and its function, similar to the above specification.

```
@DisplayName("Test Suite: OffSiteATM")
@TestInstance(Lifecycle.PER_CLASS)
class OffSiteATMTest {
  private NotificationService service;
  private Account account;
  private OffSiteATM atm;

  @BeforeAll // setup phase
  void setUpBeforeAll() {
    service = new SMSNotificationService();
  }

  @BeforeEach // setup phase
  void setUpBeforeEach() {
    account = new PrivateCustomerAccount();
    atm = new OffSiteATM(service);
  }

  @Test
  void testDeposit() {
    // exercise phase
    atm.deposit(1050, account);
    // verification phase
    assertEquals(1050, account.getBalanceInCents());
  }

  @Test
  void testDepositNegative() {
    assertThrows(IllegalArgumentException.class, () -> atm.deposit(-100, account));
    assertEquals(0, account.getBalanceInCents());
  }
}
```

Listing 1.3: A JUnit 5 test class for the `OffSiteATM` class. The `@DisplayName` annotation sets a custom name for the class in the summary of test results. All test cases share the instance variables `service`, `account` and `atm`. The method annotated with `@BeforeAll` is executed once before all test cases are executed. The method annotated with `@BeforeEach` is executed once before each test case. Methods annotated with `@Test` are test cases.

Listing 1.3 illustrates a JUnit test class that is responsible, as the name suggests, for testing the `OffSiteATM` class. This test class contains two test cases intended to test the `OffSiteATM.deposit` method. A JUnit test is defined as a regular non-private Java class that can contain test cases. According to [JUn19a], a test case is a non-private, non-abstract instance method that returns no value and is directly annotated or meta-annotated with `@Test`, `@RepeatedTest`, `@ParameterizedTest`, `@TestFactory`, or `@TestTemplate`. A JU-

nit test typically runs through a four-phase sequence: setup, exercise, verify, and teardown [Fow07]. In listing 1.3, the two methods `OffSiteATMTest.setUpBeforeAll` and `OffSiteATMTest.setUpBeforeEach` belong to the setup phase. The setup can partly be done in these methods or in the test cases themselves. However, by using the methods annotated with `@BeforeAll` and `@BeforeEach`, the code duplication for constructing the SUT can be reduced, making the test cases simpler. During the setup phase, the SUT is set to a predefined initial state by creating all *collaborator objects* [Fow07] on which the SUT depends, such as the `Account` and `NotificationService` objects, and connecting them to the SUT. Methods annotated with `@BeforeAll` are executed once before any test case is executed. By default, the test class is reinstantiated for each test case. Therefore, the method annotated with `@BeforeAll` must be static. However, annotating `OffSiteATMTest` with `@TestInstance(Lifecycle.PER_CLASS)` configures the JUnit framework to instantiate the test class only once to execute all its test cases, so methods annotated with `@BeforeAll` do not need to be static. Assuming the `NotificationService` is stateless, it can be instantiated once and then used in each test case, allowing it to be instantiated in the `OffSiteATMTest.setUpBeforeAll` method. Methods annotated with `@BeforeEach` are executed before each test case. In the example, this method is used to create a new `OffSiteATM` instance for each test case. The test case can then perform the tested task on the new instance without worrying about side effects. Next, calling `atm.deposit` in both test methods is the exercise phase. In this phase, the SUT performs the task to be tested. The assert statements represent the verification phase. In the verification phase, the test case checks whether the task performed worked correctly. Finally, the teardown phase is responsible for the clean-up work. In JUnit, methods annotated with `@AfterAll` and `@AfterEach` are part of the teardown phase. Similar to the `@BeforeAll` and `@BeforeEach` counterparts, these methods are executed after one or all test cases are executed. There is no explicit teardown phase in listing 1.3 because the garbage collector implicitly performs the teardown process. Methods that are not annotated are ignored by JUnit, but can be used as auxiliary methods for test cases.

```
@DisplayName("Test Suite: OffSiteATM")
class OffSiteATMTest {
  @Nested
  class DepositTests {
    @Test
    void test() {
      Account account = new PrivateCustomerAccount();
      new OffSiteATM(new SMSNotificationService()).deposit(1000, account);
      assertEquals(1000, account.getBalanceInCents());
    }
  }
}
```

Listing 1.4: An example of a nested test class in JUnit 5. The `DepositTests` test suite is defined as a subsuite of the `OffSiteATMTest` class and contains a single test case. Nested classes allow developers to logically group test cases. In this example all test cases concerning `OffSiteATM.deposit` are bundled in `DepositTests`.

In the context of unit testing, test suites are also important to give test cases a structure. This structure usually comes from the units or properties to be tested. Test suites are basically classes in the context of JUnit because they bundle test cases. In addition to the test classes, JUnit offers the `@Nested` Annotation with which test cases can be logically grouped within test classes. Listing 1.4 is an example of such a nested class. If the higher-level class, in this example `OffSiteATM`, contains `@BeforeAll` and `@BeforeEach` methods, they are also applied to the test cases of the nested classes.

1.3 Assertions

In the JUnit framework, *assertions* are static auxiliary methods that support the assertion of conditions during the verification phase and are accessible via the `Assertions` class. Assertions are used to define constraints for the SUT. These constraints pass if the condition is met, or fail if the condition is violated. In general, the constraints for the SUT are derived from its specification and formulated either as a boolean expression or in the form of two objects. One object defines the expected value, while the other object represents the actual value. Either the boolean constraint or the pair of actual and expected objects are passed to one of the static assertion methods in the `Assertions` class for evaluation. A failed assertion indicates that the SUT is not acting as required by the specification. If an assertion method fails, an `AssertionFailedError` exception with detailed information is thrown and further execution of the test case is aborted.

```
@DisplayName("Test Suite: OffSiteATM")
class OffSiteATMTest {
  @Test
  void test() {
    Account account = new PrivateCustomerAccount();
    OffSiteATM atm = new OffSiteATM(new SMSNotificationService());
    atm.deposit(1000, account);
    assertEquals(1000, account.getBalanceInCents(), "Balance is not correct");
  }
}
```

Listing 1.5: An example for `Assertions.assertEquals`. The first parameter is the expected value, which is `1000`. The second parameter is the actual value provided by `Account.getBalance`. If the assertion fails, the message is passed as a third parameter.

The examples in this section illustrate various assertion methods provided by the JUnit 5 `Assertions` class. Most assertion methods follow the same pattern. The first parameter is the expected value as an object and the second parameter is the current value as an object. Both objects are compared with the `Object.compareTo` method. If the passed objects are primitive types, they are compared directly with the compare operator. Complex objects are compared using the `Object.equals` method. For each assertion method of `Assertions`, there is an overload that receives either a `String` or a `Supplier<String>` object that can provide detailed information about the test in addition to the expected and actual objects

```
@DisplayName("Test Suite: OffSiteATM")
class OffSiteATMTest {
  private Account account;
  private OffSiteATM atm;

  @BeforeEach
  void setUp() {
    account = new PrivateCustomerAccount();
    atm = new OffSiteATM(new SMSNotificationService());
  }

  @Test
  void testWithdrawExceptionV1() {
    try {
      atm.withdraw(100, account);
      fail("No NotEnoughMoneyException thrown");
    } catch(NotEnoughMoneyException ex) { /* empty catch */ }
  }

  @Test
  void testWithdrawExceptionV2() {
    assertThrows(NotEnoughMoneyException.class, () -> atm.withdraw(100, account));
  }
}
```

Listing 1.6: An example of an assertion that a checked or an unchecked exception was thrown during the exercise phase. In `OffSiteATMTest.testWithdrawExceptionV1`, the exercised method is surrounded by a try-catch block. If the method throws an exception, the try block is aborted and the catch block is executed. If no exception is thrown, the `Assertions.fail` method that causes an `AssertionFailedException` with the specified text is invoked. The `OffSiteATMTest.testWithdrawExceptionV2` method uses the `Assertions.assertThrows` method to achieve the same result.

in the case of a failed assertion. The test case presented in listing 1.5 again refers to the `OffSiteATM` example from listing 1.2 and serves as an example of a test case which verifies the assertion that the `OffSiteATM.deposit` method works as intended.

For the `ATM.withdraw` method, not only the functionality of the method is important, but also the exception handling must be asserted. There are two ways of asserting that a method throws an exception. Listing 1.6 shows both approaches in detail, which also work with unchecked exceptions. In the first approach, the method that should throw the exception is executed in a try-catch block. Immediately after the method call, the `fail` method is called with a message. The catch block remains empty. If an exception is thrown, the `fail` method is never called and the empty catch block prevents the test case from failing. The second approach uses the corresponding static method `Assertions.assertThrows` of the `Assertions` class.

In addition to equality assertions and exception handling, the `Assertions` class offers several convenience assertion methods that help simplify test cases. For example, there are methods to check whether two objects are actually the same. It can be checked whether an object is `null` or that a boolean condition is `true` or `false`. The execution time can also be checked with a timeout duration, which should not be exceeded during the exercising of the tested

task of the SUT. A complete list of possible assertions can be found in [JUn19a], the official JUnit documentation.

1.4 Mocks and Stubs

Unit testing usually involves testing the SUT in isolation to ensure that any errors that may occur during the exercise phase are caused by the SUT and not by one of its collaborators. There are several techniques that can be used to isolate the SUT from its collaborators. This section presents the techniques of *mocking* and *stubbing* [Mes07]. The purpose of both techniques is to replace concrete implementations of collaborator objects, since a unit often needs other units to work at all. Sometimes these required units are expensive to construct, e.g. databases or network components. Because collaborator objects are not of interest when testing the SUT, all necessary collaborator objects are replaced by objects that pretend to be the collaborator needed to make the SUT work. To explain the concept of mock objects and stubs, the example from Listing 1.1 is used again.

Listing 1.7 contains a simple test case that tests whether the `OffSiteATM.deposit` method works as intended. Thus, in this example, the SUT is an instance of `OffSiteATM`. Both, an `Account` instance and a `NotificationService` instance are used as collaborator objects to make `OffSiteATM.deposit` work. The `Account` and `NotificationService` instances are needed for two reasons. Firstly, `OffSiteATM.deposit` obviously cannot be called without an instance of both and secondly, the `Account` instance is needed for verification because `OffSiteATM` has no state. This means in particular that after exercising the SUT the verification is performed on the `Account` instance to check if the SUT has worked correctly. As described by [Mes07], this type of test utilizes the *state verification* technique. Besides the `Account`, the `NotificationService` is also important to decide if the functionality of `OffSiteATM.deposit` is correct. Like `OffSiteATM`, the `NotificationService` is stateless. Therefore, it is not possible to check its state after the SUT has been exercised, as there is no data to assert against. In order to decide whether a notification is sent correctly, i.e. the right customer receives exactly one notification, it would be necessary to actually send a real e-mail or SMS or some other type of notification that is implemented. However, this would not be a practical test as the overhead is too high. Checking that the SUT makes the right calls, i.e. calls in the right order, with the right parameters and as often as expected on the collaborator, is called *behavior verification* [Mes07].

When applying state verification, stubs are the technique of choice to isolate the SUT from its collaborators. In [Mes07], a stub is described as a test-specific object that is a substitute for a real object to provide the SUT with its desired indirect inputs. Listing 1.8 is an exemplary implementation of an `AccountStub`.

```
@DisplayName("Test Suite: OffSiteATM")
class OffSiteATMTest {
  @Test
  void testDeposit() {
    Account account = new PrivateCustomerAccount();
    OffSiteATM atm = new OffSiteATM(new SMSNotificationService());
    atm.deposit(1000, account);
    assertEquals(1000, account.getBalanceInCents());
  }
}
```

Listing 1.7: An example of state verification to check if the SUT, an instance of `OffSiteATM`, is working as intended. The collaborators are the `Account` and `SMSNotificationService` instances required to call `OffSiteATM.deposit`. After exercising the SUT, the status of the `Account` collaborator is used to check whether `OffSiteATM.deposit` worked correctly.

```
class AccountStub implements Account {
  private int balance;
  public int getBalanceInCents() { return balance; }
  public void setBalance(int cents) { this.balance = cents; }
  public String getName() { return "John Doe"; }
  public String getEMail() { return "john.doe@mail.com"; }
  public String getPhoneNumber() { return "1234556789"; }
}
```

Listing 1.8: An `AccountStub` implementation. The `AccountStub` contains the balance as a state, which can be changed via getters and setters. The other methods provide dummy values. A stub can be created without using third-party libraries. Usually it is only used in tests and not in the actual application.

A stub is a class that implements the interface of the unit it is intended to replace during test case execution. Since Java supports polymorphism, this object can be used as if it were of the same type as the implemented interface or inherited class. The `AccountStub` contains as a state solely the balance value, which can be changed via a getter and setter. The remaining methods return dummy values. As this example demonstrates, it is not necessary for the developer to rely on third-party libraries to use stubbing during testing. The behavior of stub objects is predefined and therefore static. Stubbing can also be used to reduce the complexity of the setup phase. For example, if the account also required dependencies, an entire subsystem would have to be instantiated during test case execution. A stub can be defined in such a way that it does not need a dependency to be created.

Unlike stubs, a mock object is used when behavior verification is required. A mock object is defined as a test-specific object on which the SUT depends and which checks whether it is used correctly [Mes07]. The following examples use the Mockito framework [Moc19] to show how mocking is performed in JUnit test cases.

The example in Listing 1.9 shows how mocking can be used during unit testing. The example also includes the `AccountStub` created in Listing 1.8. Typically, the setup phase is divided into two parts called data and expectation parts [Fow07]. First, the mock objects are created in the data part to isolate the SUT from its collaborators. The second part is the expectation part, where developers can define the behavior of the mock objects. Since the `NotificationService.send` method does not return a value and the mock object does not do anything with the passed data, this part is missing in the example. After the setup phase

```
@DisplayName("Test Suite: OffSiteATM")
class OffSiteATMTest {
  @Test
  void testDeposit() {
    // setup phase
    Account account = new AccountStub();
    NotificationService serviceMock = Mockito.mock(NotificationService.class);
    OffSiteATM atm = new OffSiteATM(serviceMock);
    // exercise phase
    atm.deposit(1000, account);
    // verification phase
    assertEquals(1000, account.getBalanceInCents());
    Mockito.verify(serviceMock, times(1)).send("Balance increased by 10.0 EUR", account);
    Mockito.verifyNoMoreInteractions(serviceMock);
  }
}
```

Listing 1.9: An example of how to mock classes or interfaces. In the `OffSiteATMTest.testDeposit` test case, the setup phase also creates a `NotificationService` mock object. The static method `Mockito.mock` receives the class object of the class to be mocked and returns a mock object for this class. The `serviceMock` is then passed to the `OffSiteATM` object. Afterwards, the SUT can then be exercised. During the verification phase, the verification for the `serviceMock` instance is added. The `NotificationService.send` method should only be called once with the specified parameters.

the SUT is exercised using the stub and the mock object. Finally, the verification is done. As in the setup phase, the verification phase has two aspects. The assertion checks if the `Account` object holds the correct balance. The new aspect is the verification of the mock objects. The first verification statement checks whether the method `NotificationService.send` is called exactly once with the specified arguments on the `serviceMock` object. Assuming the `NotificationService.send` method would be called twice within `OffSiteATM.deposit`, the verification statement would fail due to a `TooManyActualInvocations` exception. It is possible to verify further constraints for the `serviceMock` instance here. The last verification statement is the `Mockito.verifyNoMoreInteraction`, which ensures that the `serviceMock` instance does not experience more than the specified interactions. As emphasized in this example, the difference between mock objects and stubs lies in the way the verification is performed. Stubs are useful in state verification, while mock objects are used to ensure that the SUT interacts with the mock object as intended which is also referred to as behavior verification.

2 Stream Runtime Verification

This chapter introduces *runtime verification* (RV), in particular *stream runtime verification* (SRV). According to [Con+18], the main objective of software verification is to check whether a program meets its specification. In contrast to software testing, RV concentrates more on behavior verification of the program than on state verification. In RV, only a single run of the system is considered when checking whether this system violates its specification [LS09]. Typically, during RV, a property to be checked is specified as a logical formula, for example, using *linear temporal logic* (LTL), and then synthesized into a monitor that can then evaluate a run. SRV takes a different approach. A set of input streams is related incrementally to a set of output streams [BS14; Con+18; DAn+05]. This approach allows quantitative measurements in addition to monitoring correctness properties. The following sections introduce the TeSSLa language for SRV. First, RV in general is discussed. Then, the concept of stream processing is introduced. Finally, TeSSLa is presented together with some examples.

2.1 Runtime Verfication

Software verification is essentially a matter of checking whether a program corresponds to its specification [LS09]. This means in particular that the SUT is compared with its specification. Software verification can be performed using various techniques such as *theorem proving* [BC10], *model checking* [CGP99] and testing [Mye11]. As described in chapter 1, software verification can be separated into state and behavior verification. State verification checks whether the SUT is in the correct state after it has been exercised with certain tasks. It is not of interest whether it has performed the correct method calls in the correct order to get into the correct state as long as the final state is correct. Behavior verification checks whether the SUT is functioning as intended while exercising its tasks. For behavior verification, it is important that the SUT components behave exactly as specified, that is, that the sequence and respective number of method calls are correct, and that no unexpected interaction takes place between the components. This section focuses on behavior verification, in particular RV. In contrast to traditional verification techniques such as theorem proving and model checking, this is a relatively new, lightweight approach to software verification [Con+18].

Definition 2.1 ([LS09])**.** Runtime verification is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a *run* of a SUT *satisfies* or *violates* a given correctness property.

According to [LS09], a run of a system is a possibly infinite sequence of system states. In this context, a state of the system describes current variable assignments or the sequence of its emitting or performing actions. While formally a run can be considered as a possibly infinite

word or *trace*, the *execution* of a system is a *finite* prefix for a run and formally a finite trace. When executing a program, only its executions can be observed, but this restricts the corresponding evolving run as a prefix. While verification techniques such as model checking are more interested in checking whether all possible runs of the system meet the given correctness properties, the executions of the SUT are the primary object analyzed in the RV setting. In RV, the verification of the correctness properties of an execution is typically performed using *monitors*. To its simplest form, a monitor decides whether the current execution of the system satisfies a certain correctness property by giving either `true` or `false` as a *verdict*.

Definition 2.2 ([LS09])**.** A monitor is a device that reads a finite trace and yields a certain verdict.

Monitoring may be conducted in two ways: First, a monitor can be used to check the current execution of a system. This approach is called *online monitoring*. In this setting, the monitor is designed to process execution traces incrementally and efficiently. On the other hand, a monitor can process recorded executions, also known as *offline monitoring*.

In addition to the behavior of monitors, their creation is also of interest. During RV, a monitor is typically generated automatically from a high-level specification, also known as a *synthesized* monitor. Since RV has its roots in model checking, a variant of LTL is often used as a specification. The jUnit^{RV} library, which is described in detail in chapter 4 in section 4.1, synthesizes monitors from LTL properties. The next section introduces *stream runtime verification*, which uses a different form of specification to synthesize monitors.

2.2 Stream Processing

This section introduces *stream runtime verification*, which was established by the specification language LOLA [DAn+05]. As described in [Con+18], in contrast to traditional RV, SRV takes a different approach by correlating a set of input streams with a set of output streams step by step. According [BS14], in the context of SRV, specifications explicitly declare the dependencies between input streams, which represent the observable behavior of the system, and output streams, which describe error reports and diagnostic information. These dependencies can relate the data value of current events of an output stream to past, current, or future data values of events of the same or other streams. This approach enables not only monitoring of correctness properties, but also statistical measurements that are useful for system profiling and coverage analysis.

The following explanation refers to the example in listing 2.1. The example is a specification written in TeSSLa. The visualization for this specification and the corresponding execution of the SUT is shown in figure 2.2. The SUT is an `ATM` instance from the example in listing 1.1.

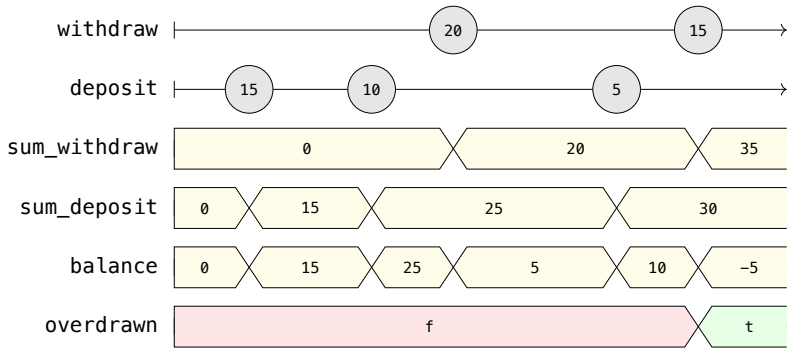


Figure 2.2: An example of the application of SRV to the execution of an `Account` instance. The diagram consists of a set of input and associated output streams defined in the TeSSLa specification of Listing 2.1. The two input streams `withdraw` and `deposit` are event streams over the domain of integers. In contrast to the streams `sum_withdraw`, `sum_deposit`, `balance` and `overdrawn`, both streams do not continuously deliver events for each timestamp. The last four streams do not produce events but signals.

Since this chapter only deals with SRV in general, the details about TeSSLa are explained in Section 2.3.

```
in withdraw: Events[Int]
in deposit: Events[Int]

def sum_withdraw := sum(withdraw)
def sum_deposit := sum(deposit)
def balance := sum_deposit - sum_withdraw
def overdrawn := balance < 0

out *
```

Listing 2.1: A TeSSLa example specification. The specification relates the output streams `sum_withdraw`, `sum_deposit`, `balance` and `overdrawn` to the input streams `withdraw` and `deposit`. Both streams `sum_withdraw` and `sum_deposit` recursively sum all withdrawals and deposits. The `balance` stream calculates the difference between all withdrawals and deposits and thus the account balance. The `overdrawn` stream indicates whether the account is overdrawn. The last line of the specification ensures that all streams are output by the TeSSLa interpreter.

Figure 2.2 illustrates the input streams `withdraw` and `deposit` as well as the corresponding output streams `sum_withdraw`, `sum_deposit`, `balance` and `overdrawn` for the execution of the SUT. Discrete events are represented by circles containing the value of the corresponding event. If a stream continuously provides an event at any time, this is interpreted as a signal. A signal is divided into several segments which represent the points in time at which the same value is supplied by the stream. Both input streams provide integer events with different and unstable frequencies. The output stream `balance` calculates the current balance on the account by calculating the difference between the two streams `sum_deposit` and `sum_withdraw`. New events on the `deposit` or `withdraw` stream cause the `sum_withdraw` and the `sum_deposit` stream to be recalculated. The `overdrawn` output stream provides boolean events where `false` indicates that the balance on the account is zero or more and `true` that the balance is negative. As the example shows, the evaluation of a stream depends on the streams it needs for its calculations. An example of this is the `balance` stream, which is

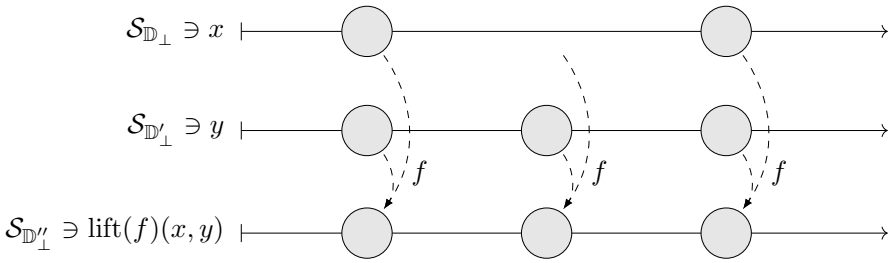


Figure 2.3: A visualization of lifting an arbitrary function f to streams, i.e. $f : \mathcal{S}_{\mathbb{D}_\perp} \to \mathcal{S}_{\mathbb{D}'_\perp}$. The $\mathrm{lift}(f)(x,y)$ operation depends on the streams x and y and is evaluated whenever one of these streams supplies a new event. As the input stream are not synchronized, the function f can only be used if both streams deliver events simultaneously or if f is able to process empty values. If the second event is considered, f cannot produce an event, because an event is not provided at the same time on both the x and y streams.

updated whenever the `sum_deposit` or `sum_withdraw` stream supplies new events. If only one of these streams produces a new event, the last event seen on the other stream is used for the calculation. This particular mechanism is explained in more detail below. Streams that deliver events irregularly and not simultaneously with other streams are called *non-synchronized* or *asynchronous streams* [Con+18]. Thus, the `withdraw` and `deposit` streams are non-synchronized. In SRV, calculation with missing data values of events on a stream, as shown in figure 2.2, is not supported by default. To compensate missing events on a stream by the last seen event the lift operator and two auxiliary operators are required.

In the specification in listing 2.1, binary operators such as the $-$, $+$ and $<$ operator are used for calculations between data values of stream events and literal values. These operators are defined over the data domain $\mathbb{D}$, which is $\mathbb{Z}$ in the example. In order for the operators to be applied to streams, they must be *lifted* [Con+18] from the data domain to streams. A non-synchronized stream over a time domain $\mathbb{T}$ and a data domain $\mathbb{D}$ is a sequence $\mathcal{S}_{\mathbb{D}} := (\mathbb{T} \times \mathbb{D})^+$. Since not all non-synchronized streams provide events simultaneously, the data domain $\mathbb{D}$ is extended by the missing value $\perp$, i.e. $\mathbb{D}_\perp := \mathbb{D} \cup \{\perp\}$. To lift functions such as $f : \mathbb{D}_\perp \times \mathbb{D}'_\perp \to \mathbb{D}''_\perp$, that operate on the data domain, to a stream, the lift operations is used:

$$\mathrm{lift} : (\mathbb{D}_\perp \times \mathbb{D}'_\perp \to \mathbb{D}''_\perp) \to (\mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{D}'} \to \mathcal{S}_{\mathbb{D}''}).$$

The lift operation applies a given function f to the current events of the streams specified as arguments. The result of the calculation of f can then be provided as an event on an dedicated output stream or provided as input to another function. Especially for non-synchronized streams, the lifted function f may not be able to calculate a result if one of the dependent streams does not supply an event at the current time. Figure 2.3 illustrates this issue for f. To solve this problem, the lift operation must be adjusted. For this purpose, the *merge* [Con+18] operation is introduced using the lift operation. The merge operation processes streams event-oriented, where two streams are merged into one and the first stream is preferred if

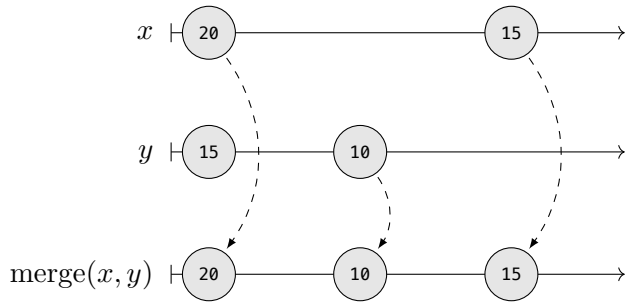


Figure 2.4: An example of merging two non-synchronized streams x and y with the merge operator, i.e. $\mathrm{merge}(x, y)$. The merge operation forms a new stream in this example. Whenever one of the two input streams x or y delivers a new event, this event is provided on the merged stream. If both input streams provide an event, the event of the stream that is the first argument of the merge operation is preferred.

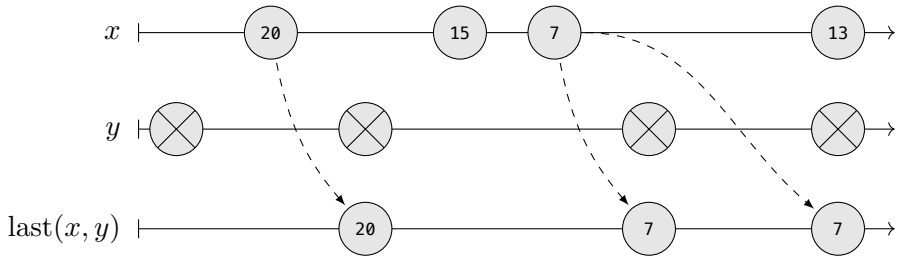


Figure 2.5: Relating to the last value occurring strictly before an event on an other stream. The input stream x provides event over the integer domain whereas the input stream y provides unit event that do not hold any value. The last operation takes two arguments. The first argument is a stream of which the last value is evaluated that occurred strictly before the event currently happened on stream y if available. Hence, the events on the stream which is the second argument serves as triggers for the evaluation of the last operation.

two events occur simultaneously on both streams. The merge operation is defined as follows:

$$\mathrm{merge}(x, y) = \mathrm{lift}(f)(x, y)$$

$$f : \mathbb{D}_\perp \times \mathbb{D}_\perp \to \mathbb{D}_\perp$$

$$f(a, b) = \begin{cases} b & \text{if } a = \perp \\ a & \text{else} \end{cases}$$

Figure 2.4 illustrates the behavior of the merge operation for the x and y integer streams. In the first step, both streams provide an event. Due to the behavior of the f function, the value of stream x is preferred to the value of stream y. In the following two steps, only one of the two streams supplies an event, which is then used as the new event for the merged stream.

The second operation besides the merge operation, which is necessary to fully allow the lifting of non-synchronized streams, is the *last* [Con+18] operation. The $\mathrm{last} : \mathcal{S}_\mathbb{D} \times \mathcal{S}_{\mathbb{D}'} \to \mathcal{S}_\mathbb{D}$ operation supplies the last event of a stream that occurred strictly before the current event of

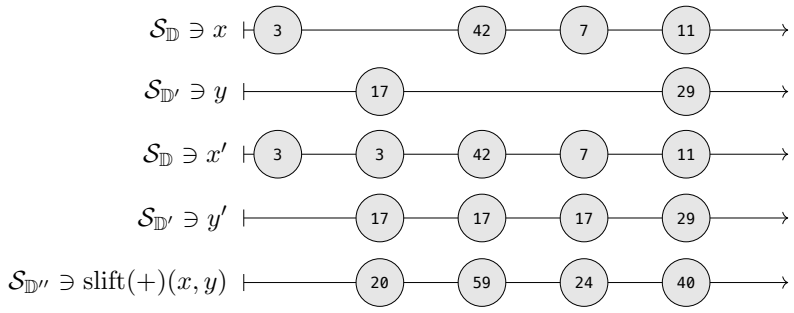


Figure 2.6: The slift operation lifts an arbitrary function f from the data domain to streams. In this case, the function to be lifted to streams is the $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ operator. To avoid the problem of missing events on streams x and y, two new streams x' and y' are constructed each with intermediate results. Both the stream x' and y' use the last operation to ensure that they both always deliver an event at the same time. If one of the streams cannot provide an event, but the other does, the last event of that stream is provided. In addition, the function f is wrapped into another function f', which is only applied to the events of the streams x' and y' if both streams supply an event. In the example, at the first event of the x' stream the y' stream does not provide an event, therefore the $+$ operator does not evaluate in this case.

another stream. This operation can be applied to two streams over arbitrary data domains, as shown in figure 2.5. In this example, the x stream is an integer stream, while the y stream provides unit events. Each time the y stream provides an event, the $\mathrm{last}(x, y)$ operation produces the last event that occurred on the x stream strictly before the y event.

Now that the necessary auxiliary operations are available, an improved version of the lift operation can be defined. This operation allows an arbitrary function f to be lifted to streams by a combination of the lift, merge and last operation. If not all streams on which the function f depends supply an event, missing events on streams are replaced by the last event that occurred on that stream. This new operation is called *slift* [Con+18] operation and is defined as follows:

$$
\begin{aligned}
\mathrm{slift} : (\mathbb{D} \times \mathbb{D}' \to \mathbb{D}'') &\to (\mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{D}'} \to \mathcal{S}_{\mathbb{D}''}) \\
\mathrm{slift}(f)(x, y) &= \mathrm{lift}(f')(x', y') \\
x' &= \mathrm{merge}(x, \mathrm{last}(x, y)) \\
y' &= \mathrm{merge}(y, \mathrm{last}(y, x)) \\
f'(a, b) &= \begin{cases} f(a, b) & \text{if } a \neq \bot \wedge b \neq \bot \\ \bot & \text{else} \end{cases}.
\end{aligned}
$$

The slift operation example is given in figure 2.6. The slift-operation uses the lift-operation, which processes the x and y streams indirectly referring to the x' and y' streams. The x and y streams are converted into the streams x' and y' respectively. The difference between x and x' is that at each event on the y stream the last event on the x stream is taken over by using the last operation. The result is then merged with the original x stream to ensure that the last event is only overtaken if there is no new event available at the current time.

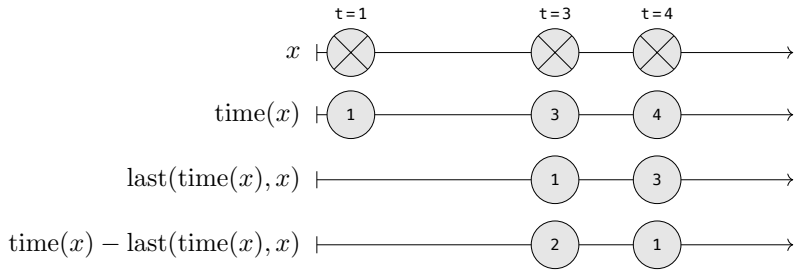


Figure 2.7: The time operator gives access to the timestamp of each event. The timestamp refers to a global clock and is unique to for its stream. The time operator produces new events that carry the timestamp as a data value. This allows all operators for data values to be applied to these events as well. In the example the stream $\mathrm{time}(x)$ produces new events containing the time as its data values. These values are then used by the last operator to refer to the timestamp of second last event occurred on x. Finally, some calculation is done on the events provided by the intermediate streams.

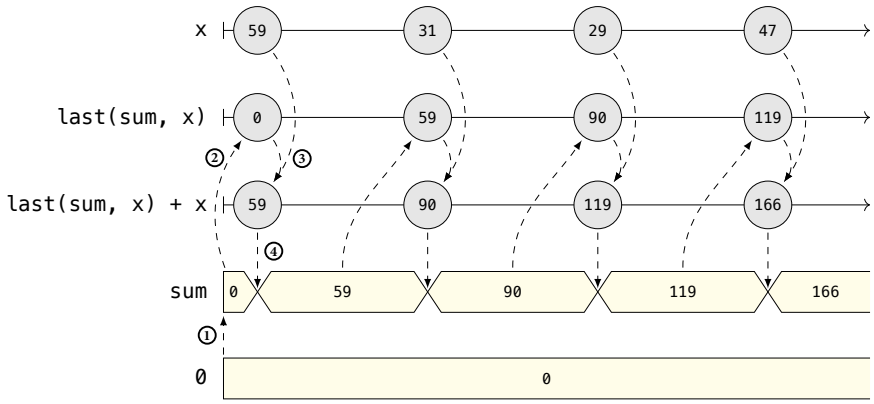


Figure 2.8: An illustration for the recursive evaluation of the `sum` stream from listing 2.9. The streams `last(sum, x)` and `last(sum, x) + x` are streams that contain intermediate results for the `sum` stream but are not explicitly declared in the corresponding specification. The dotted arrows represent the source and target of events. The ① step represents the initial configuration for the `sum` stream. Steps ②, ③ and ④ represent the calculation that is performed for each new event on `x`. Step ② is the reference to the last `sum` value, which is then added to the new event data value of `x` in step ③. Finally, the value calculated in step ③ is provided by the `sum` stream in step ④. The constant stream `0` supplies the literal 0 to the other streams.

The same procedure is also applied to the y stream. In addition, the function f is wrapped with a function f'. This function applies f to its arguments only if both arguments provide events. If one of the arguments does not supply an event, f' does not evaluate, as shown in the example. The stream x' produces an event with a value of three, but the y' stream produces no event, so the $+$ operator cannot be used. This way, the slift operation is now able to lift arbitrary functions with arguments of arbitrary data types to streams. Using the last known event of a stream to perform operations on multiple streams is a concept called *signal semantics* [Con+18].

Beside the previously introduced operators there are also other operators in the context of stream runtime verification. The events of all non-synchronized streams must have a global order, but do not have to occur simultaneously. The order of all events is determined by their time of occurrence. Thus, each event carries the time of its occurrence as a timestamp data

value that enables all operators to apply operations to these timestamps. To access an events timestamp, the $\mathrm{time} : \mathcal{S}_{\mathbb{D}} \rightarrow \mathcal{S}_{\mathbb{T}}$ operation [Con+18] can be used. Figure 2.7 shows an example of the time operation applied to the unit events of the x stream.

Lifting a function from the data domain to streams and accessing previous events are two of the three basic principles of stream processing. The last principle that has not yet been introduced is *recursive* equations [Con+18]. To write recursive equations, the last-operation is used in conjunction with the merge-operation. By using the last operation, the last event of a stream can be accessed, while the merge operation can supply an initial event from another stream to a recursive equation. An example of the application of a recursive operation is the TeSSLa specification in listing 2.1. The two streams `sum_withdraw` and `sum_deposit` sum up the data values from the events of the `deposit` and `withdraw` streams, respectively, using the `sum` operation. This operation is specified in listing 2.9. The `sum` operation calculates the sum of the data values of all events of a stream recursively. In listing 2.9 the sum of all integers on the `x` stream is calculated.

```
in x: Events[Int]
def sum: Events[Int] := merge(last(sum, x) + x, 0)
out sum
```

Listing 2.9: A TeSSLa specification example defining a recursive equation for summing up values on a given input stream `x`. The `last(sum, x)` subexpression of the `sum` stream is a self-reference to the last event occurred on the stream, which is then added up by the current event data value. If no last value is available, the merge operation supplies the 0 event.

Figure 2.8 is the corresponding graphical representation of the stream processing performed on the input stream `x`. The three additionally visualized streams `last(sum, x)`, `last(sum, x) + x` and `0` represent intermediate results. The merge operation on the `sum` stream ensures that the signal on the stream is initialized with a value of 0. After the `x` stream has provided its first event, the merge operation on the `sum` stream takes over the last event of the `last(sum, x) + x` stream. The `last(sum, x) + x` is recalculated with each new event on the `x` stream.

2.3 Temporal Stream-based Specification Language (TeSSLa)

In the following, TeSSLa [Con+18], a specification language tailored for SRV of cyberphysical systems, where timing is a critical issue, is presented. In contrast to traditional SRV frameworks such as LOLA [DAn+05], which process event streams without taking timing information into account, TeSSLa is able to operate non-synchronized due to its native support of time-stamped events. This allows efficient processing of streams with sparse and fine-grained event sequences. The TeSSLa implementation used in the following chapters enables both offline and online monitoring, i.e. the monitoring of running applications as well as the monitoring of pre-existing traces. An example of a TeSSLa specification is shown

in listing 2.1 and serves as an example for the discussion in this section. The example refers to the `Account` interface of listing 1.1 in chapter 1.

SRV is based on the three basic principles: stream processing, previous events and recursive equations. These concepts are now being investigated in TeSSLa. As listing 2.1 and the corresponding visualization in figure 2.2 emphasize, TeSSLa natively supports the processing of non-synchronized streams and thus fulfills the first of the three basic principles mentioned. Recursive equations and access to previous events are tied together in TeSSLa. As with existing SRV approaches, TeSSLa relates a set of input streams to a set of output streams via mutually recursive equations. Listing 2.9 is an example of such a recursive equation where a stream refers to its previous events by using the last operation in conjunction with the merge operation, which produces an initial value. Internally, TeSSLa only supports discrete events, but operators that follow signal semantics can be expressed with the slift operator introduced in the previous section. The operators $+$, $-$ and $<$ used in the example in listing 2.1 are automatically lifted to signal semantics.

In addition to the basic operations, TeSSLa offers a number of useful additional operations. The *time* [Con+18] operation, which provides access to timestamps of events with the objective of accessing the global order of events and performing calculations with these timestamps, the *count* [Con+18] operation, which counts the number of events supplied on the given stream, the *filter* [Con+18] operation, which can be used to filter events on a stream depending on a boolean condition, and the *delay* [Con+18] operator, which allows events to be generated at certain times using the *const* [Con+18] operation. The const operator generates a new stream that maps each event of the input stream to an event at the same time that has a constant data value on the resulting output stream. Listing 2.10 shows a TeSSLa specification with all these operations. The most common TeSSLa keywords are: the keyword `in` that defines an input stream, the `def` keyword that defines a stream with intermediate results that can later be used as an output stream, the `out` keyword that defines output streams, and the `as` keyword that allows naming of temporary streams.

```
in deposit: Events[Int]
def timeout_duration := const(3, deposit)
def high_condition := default(deposit > 100, false)
out deposit
out time(deposit) as time
out timeout_duration
out high_condition
out filter(deposit, high_condition) as high_deposit
out delay(timeout_duration, deposit) as deposit_too_late
out count(deposit) as num_deposits
```

Listing 2.10: A TeSSLa specification with its auxiliary operators. The ***default*** operation produces a default event if the first argument cannot provide an event. Since the comparison in the first argument requires an event on the `deposit` stream, the `high_condition` stream would not be initialized. Unlike listing 2.1, most output streams are not declared using the `def` keyword. Instead, they are declared inline and renamed using the `as` keyword.

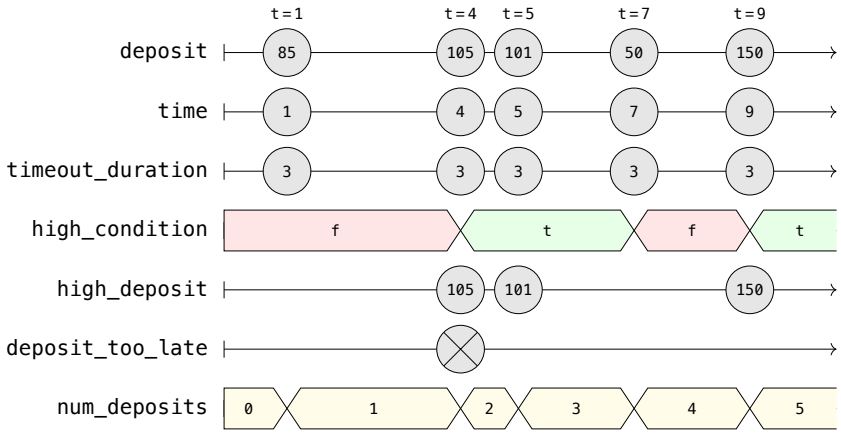


Figure 2.11: A visualization of the specification in listing 2.10 that processes a ***deposit*** stream. The `time` operation generates a stream that immediately outputs events that contain the time stamp of the events of the input stream as a data value. The `time_duration` stream maps each `deposit` event to an event containing the constant integer value 3. The stream `high_condition` checks whether the data value of the current event provided by `deposit` exceeds 100. If this is the case, the signal from `high_condition` is true, otherwise it is false. The `high_deposit` stream filters the data values of events from the `deposit` stream using the signal from `high_condition`. If `high_condition` is true, the event `deposit` is provided by the `high_deposit`. The `deposit_too_late` outputs unit events if the time span between two `deposit` events is greater than or equal to the value provided by the current `timeout_duration` event. If a new event occurs at `deposit` before the timeout is reached, the timeout is reset. Finally, the stream `num_deposits` counts the number of events provided by `deposit`.

As described in [Con+18], the TeSSLa compiler analyzes the TeSSLa specification and performs a static type check. Finally, the specification is translated into flat TeSSLa. In addition, TeSSLa provides a macro system, which means that more complex functions based on the basic operations offered by TeSSLa can be specified. This makes TeSSLa more flexible and powerful as it allows the creation of application-domain specific standard libraries. The defined macros are *expanded* by the compiler, i.e. they are transformed so that the resulting specification uses the built-in slift operator and constant values are implicitly converted into constant signals. As an example, the stream `sum_withdraw` of the listing 2.1 can be rewritten using macros. Listing 2.12 shows a specification in which the macro `sum` calculates the sum of all data values provided by the integer events from its input stream `withdraw`.

```
def sum(a: Events[Int]) := s where {
  def s: Events[Int] := merge(last(s, a) + a, 0)
}

in withdraw: Events[Int]
def old_sum_withdraw: Events[Int] := merge(last(old_sum_withdraw), withdraw, 0)
out old_sum_withdraw
out sum(withdraw) as new_sum_withdraw
```

Listing 2.12: The macro `sum` defines a function that calculates the sum of the data values provided by the integer events of the input stream `a`. The macro calculates the same result as the recursion in the `old_sum_withdraw` stream.

Both, the compiler and the interpreter, which are both the same application, are written in Scala and can therefore be used directly from Java applications. As a result, TeSSLa allows

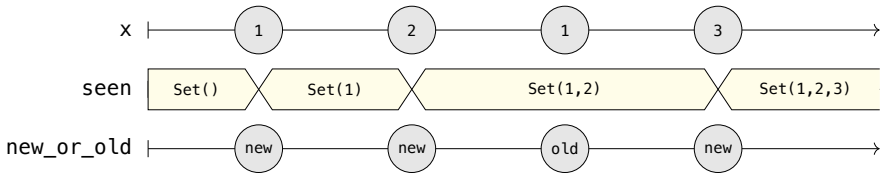


Figure 2.14: A visualization of a system run for the specification in listing 2.13. The integer events on stream `x` are stored in the set provided on stream `seen`. For each event on the stream `x`, a string is output on the stream `new_or_old` indicating whether it is an event with a new data value, or whether such an event has already been seen.

the use of Java data structures as data values for its events. Listing 2.13 is an example of a specification that uses the complex data type `Set` to track the processed integer events.

```
in x: Events[Int]

def seen: Events[Set[Int]] := merge(Set_add(last(seen, x), x), Set_empty[Int])
def new_or_old := if Set_contains(last(seen, x), x) then "old" else "new"

out *
```

Listing 2.13: A TeSSLa specification that records all processed integer events using recursion and the complex data type `Set`. In the recursion, an empty `Set` is initialized using the merge operation. In the next steps, a new object can be added to the existing set using `Set_add`. In the stream `new_or_old` it is now possible to check with `Set_contains` whether the current event was already contained in the set before.

A graphical representation of a system run for the listing 2.13 specification is shown in figure 2.14. By sharing the same JVM between application, compiler and interpreter, it would be possible to use application-specific data types as event data values for streams within TeSSLa in the implementation presented in the chapter 4.

3 TeSSLa-Based Monitoring and Mocking in JUnit

This chapter introduces concepts for instrumenting and monitoring application code and how to create and use mock objects. These concepts are illustrated by listings and partially by *Unified Modeling Language* (UML) sequence diagrams. First, the instrumentation of the application code from the existing jUnit^{RV} library is explained. Then, some approaches to TeSSLa monitors, their advantages and disadvantages as well as their attachment to test classes and test cases are presented in detail. Finally, the last section of this chapter introduces a concept for the creation and usage of mock objects. Mock objects can either be attached to TeSSLa monitors or a special handler can be used to change their behavior.

The approaches presented in the following sections refer to the terms *application context* and *test context*. Code executed in the test context is intended to test the functionality of the application code. The executed application code, on the other hand, is the application context. Therefore, code executed in the test context cannot directly modify the code executed in the application context and hence has only limited influence on the control or data flow of the application. For example, the application code cannot access the instance variables of the test class that is executing it. In addition, the test code cannot change the return value of a particular method that has no parameters. The bottom line here is that switching between application and test contexts is not straightforward. The following sections introduce techniques that allow data exchange between the test and the application context. In particular, monitoring requires data to be extracted from the execution of the application code. This data is then processed by monitors. In the other direction, mock objects can inject data processed by monitors from the test context into the application context to influence control or data flow.

3.1 Application Code Instrumentation

As explained in chapter 2, RV examines the behavior of a system with respect to given constraints or properties. Behavior in this context means e.g. the sequence of called methods and constructors, accesses to instance variables, thrown exceptions and so on. In Java, this information cannot easily be obtained from executed programs. To get such information it is necessary to modify the code to be monitored, for example by using special runtime annotations, or logging libraries. All these approaches require, however, to have access to the system to be monitored on the one hand and on the other hand that all classes to be monitored must be changed manually for test purposes. As this approach prevents, for example, that third party software, which may only be available in precompiled form, can be monitored and existing projects can only be subsequently monitored with much effort, it is necessary to develop an automated procedure that provides the information necessary for monitoring.

For this purpose the so-called *instrumentation* is used. In this process, components in the form of bytecode are manipulated in such a way that the corresponding information, also referred to as *events*, can be obtained and processed in a publish-subscribe manner from an external component when methods of the manipulated class are executed. By this approach it is on the one hand possible to monitor third-party software and on the other hand existing code can be subsequently monitored with very little effort.

The software instrumentation explained in the following is already given by the implementation of the jUnit^{RV} library and is therefore only summarized to get a picture of its functionality. The concept of instrumentation must also consider the following requirements. First, events that occur during application code execution should be processed by TeSSLa monitors in a blocking manner. This means, that if an event occurs, program execution should be halted until all monitors have finished processing the event. The reason for this is that the implementation of mock objects, which will be introduced later, can react to changes in the system state. The next requirement is that a TeSSLa monitor should immediately stop the execution of the test case to which it is attached with an assertion error if it detects a specification violation. Finally, it should be possible to instrument the application code without explicitly changing it. This requirement allows developers to instrument third-party code that may not be available. In addition, instrumentation does not lead to changes in the application code. Furthermore, this design separates the test and application contexts, since events and monitors are not specified in the application code, but in test classes.

Before a concept for instrumentation is presented, a list of events that may occur when executing application code is provided. The events *called*, *calling*, *returned* and *returning* for class constructors and methods are considered for instrumentation. In addition, the *calling* and *returning* of instantiations, *thrown exceptions*, *field accesses* and *assignment*, *class instantiation* and *class loading* should also be instrumentable. The selection of these events results from the *Javassist* library [Chi19], which is used for the modification of the bytecode, and thus is also responsible for the instrumentation.

The requirements mentioned above, as well as the aspect of reusability of the software, lead to an event implementation as Java objects, that can later be used by monitors or mock objects. Therefore, all events mentioned above are modeled as classes inheriting from a base class `Event`. Each event holds information about the class and the method or field it is associated with. This allows developers to create events as instance variables within JUnit test classes, as shown in listing 3.1. The test engine can use these `Event` objects to perform instrumentation before executing the test code.

For convenience, the explicit instantiation of an `Event` object can be replaced by a static method invocation. In the actual implementation, events can depend on additional data for internal use, which can be derived from the class name or the method or field name passed

```
class OffSiteATMTest {
  private Event calledEvent = new Called("ATMExample/ATM", "deposit", ...);
  private Event returnedEvent = new Returned("ATMExample/ATM", "deposit", ...);

  @Test
  void testDeposit() {
    Account account = new AccountStub();
    OffSiteATM atm = new OffSiteATM(new SMSNotificationService());
    atm.deposit(1000, account);
    assertEquals(1000, account.getBalanceInCents());
  }
}
```

Listing 3.1: An example for the declaration of events in a test class. The two events `calledEvent` and `returnedEvent` contain information about the method to be instrumented as well as the type of instrumentation. Both events lead to an instrumentation of the `ATM.deposit` method. The object `calledEvent` causes an event to occur when the method `ATM.deposit` is called, while the object `returnedEvent` causes an event to occur when the method is exited.

through constructor parameters. By using a static method for this purpose, the developer does not have to generate these values manually. Therefore, instantiating `Event` objects using static methods makes errors due to incorrect parameters less likely and reduces code complexity. Listing 3.2 demonstrates how to instantiate events using a static method invocation.

```
class OffSiteATMTest {
  private Event calledEvent = called("ATMExample/ATM", "deposit");
  private Event returnedEvent = returned("ATMExample/ATM", "deposit");

  @Test
  void testDeposit() {
    Account account = new AccountStub();
    OffSiteATM atm = new OffSiteATM(new SMSNotificationService());
    atm.deposit(1000, account);
    assertEquals(1000, account.getBalanceInCents());
  }
}
```

Listing 3.2: By using the static methods `called` and `returned` the declaration of `Event` objects can be simplified. The result of this notation is the same as in listing 3.1.

As described in the instrumentation requirements, the processing of events should be blocking. The easiest way to achieve this is to append or prepend code to methods, constructors, or field accesses. If a method is instrumented by a `called` Event, code is inserted at the beginning of the instrumented method to generate and publish an event. Depending on what `Event` it is, code is injected in different locations of the class. The tricky part of this solution is that the code injection has to be done after loading the class and before passing it to the JVM to make it possible to instrument third-party components as well. Since such code is often only available in precompiled form, this means in particular that the injection has to be applied to bytecode. Figure 3.3 illustrates the execution of an instrumented method. During program execution the method `ATM.deposit` is called. Before the body of the method is executed, the monitor instance `m` consumes the `Called` event of the `ATM.deposit` method. After the monitor has consumed the event, the execution of the `ATM.deposit` method continues.

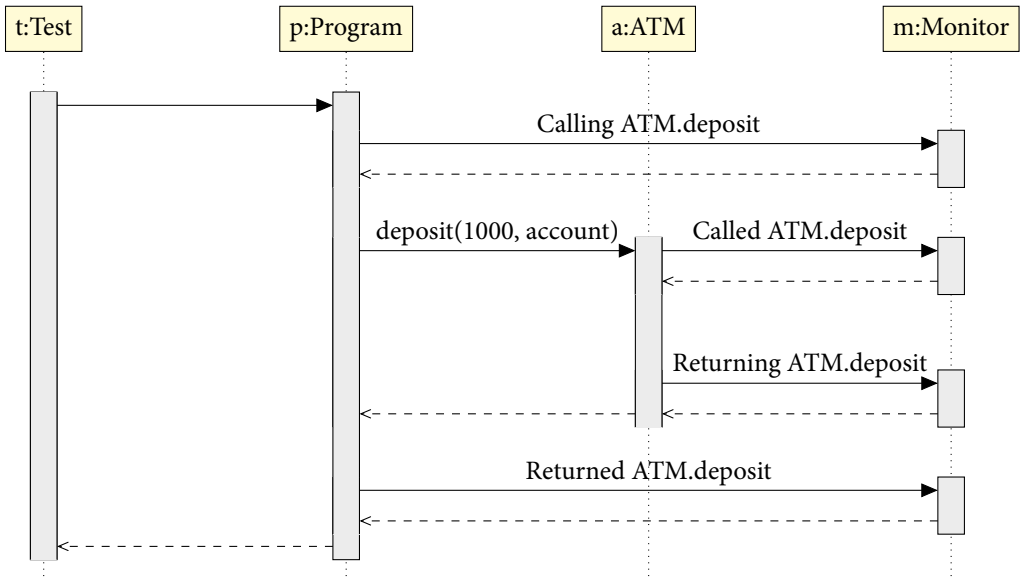


Figure 3.3: An UML sequence diagram showing how the monitor is involved in the execution process of the instrumented `ATM.deposit` method. Before the method is called, the `Calling` event is consumed by the monitor. After the monitor has processed this event, the `ATM.deposit` method starts executing. Directly after the invocation a `Called` event is created and consumed by the monitor. After this event is finished, the actual method body is executed. Finally a `Returning` event is processed by the monitor. After the method `ATM.deposit` is finished, the `Returning` event is published.

Before the `ATM.deposit` method terminates, it produces a `Returned` event, which the monitor consumes again. The `ATM.deposit` method then returns immediately after the monitor finished consuming the `Returned` event.

One of the main advantages of this approach is that not only can events be monitored during the execution of the instrumented code, but detailed information about the event, such as the method parameters or return values, can also be retrieved. Since only those methods and variable accesses that are really needed are instrumented and not all possible ones, this approach has a positive effect on the performance of the instrumentation implementation. Furthermore, it is possible to cause an instrumentation of the application code with little effort by simply declaring an instance variable. Thus, the instrumented class does not have to be changed manually.

3.2 Monitors for Test Cases and Test Suites

Hitherto monitors have been synthesized from LTL formulas by the jUnit^{RV} library and constructed and simulated as state machines. Since the TeSSLa interpreter performs the simulation, it is the responsibility of a TeSSLa monitor to translate the event data into streams, which are then forwarded to the interpreter and give a verdict based on the output streams. Because events provide a variety of information, a monitor must be designed in such a way that arbitrary input streams can be generated from this information. For example, it must

be possible for the monitor to convert an event of a method call into a stream that supplies integer events that contain the first parameter of this method as a data value. On the other hand, the judgement of the output streams is another interesting aspect, because the events of the output streams contain data values of different types. For example, a TeSSLa specification can be written in such a way that the events of all output streams contain string values. It is at the monitor to decide whether it should fail or continue based on these streams. An additional requirement for a TeSSLa monitor is that it must be described in the test context in order to maintain the separation of test and application context. In summary, this means that TeSSLa monitors serve as a communication link between the test code and the TeSSLa interpreter, which fundamentally distinguishes them from LTL-based monitors. In the following some approaches for TeSSLa monitors are presented and discussed.

Since the subsequent implementation uses a custom test engine written for the JUnit 5 library, loading this test engine through the JUnit 5 platform requires the use of Java's *service provider interface* (SPI). This results in the first approach to how monitors can be defined and executed. As with custom test engines, a monitor can be defined as a service provider that is loaded at runtime by a service loader. Therefore, each monitor that can be used in the test code must be defined as a class which inherits from a base monitor class that defines the service. The service provider classes must be grouped together with a manifest file, which contains all loadable classes, in a jar library. The test engine can then load these monitors at runtime. This approach has the advantage that monitors are easy to manage. To add a new monitor, a new class file must be added to the jar library and a line of code must be appended to the manifest file. The actual testing framework does not have to be changed, only a new version of the jar library containing this monitor has to be provided. Annotations are now used in the test code to attach monitors to test classes or test methods. If a monitor is attached to a test class, all its test cases are monitored by this monitor. During test case execution, the test engine ensures that the correct monitors from the list of available monitors are found, instantiated, and attached.

```
@Monitors({"WithdrawMonitor"})
class OffSiteATMTest {
  private Event calledDeposit = called("ATMExample/ATM", "deposit");

  @Monitors({"DepositMonitor"})
  @Test
  void testDeposit() {
    Account account = new AccountStub();
    OffSiteATM atm = new OffSiteATM(new SMSNotificationService());
    atm.deposit(1000, account);
    assertEquals(1000, account.getBalanceInCents());
  }
}
```

Listing 3.4: Both monitors `WithdrawMonitor` and `DepositMonitor` were loaded as services and attached to the test class `OffSiteATMTest` and the test method `OffSiteATMTest.testDeposit` respectively. Both monitors consume the `Called` event `calledDeposit` defined in the test class.

```
@Monitor(
  spec="spec.tessla",
  called={
    @Called(method="deposit", clazz=ATM.class, argTypes={int.class, Account.class},
        stream="deposit"),
    @Called(method="withdraw", clazz=ATM.class, argTypes={int.class, Account.class},
        stream="withdraw")
  },
  returned={
    @Returned(method="deposit", clazz=ATM.class, argTypes={int.class, Account.class},
        stream="deposit_return")
  },
  outputStreams={"balance", "error"},
  logStreams={"balance"}
)
class OffSiteATMTest {
  @Test
  void testDeposit() {
    Account account = new AccountStub();
    OffSiteATM atm = new OffSiteATM(new SMSNotificationService());
    atm.deposit(1000, account);
    assertEquals(1000, account.getBalanceInCents());
  }
}
```

Listing 3.5: An example of how to declare a monitor using annotations. The `spec` attribute defines the name and path to the TeSSLa specification. The `called` attribute declares all `Called` events while the `returned` attribute declares all `Returned` events. With `outputStreams` the output streams which the monitor receives from the TeSSLa interpreter can be defined. The `logStreams` can be used to specify streams whose events are printed on the console.

Listing 3.4 demonstrates how to attach monitors to test classes and test cases. The monitor `WithdrawMonitor` is responsible for all test methods within the test class `OffSiteATMTest`, while `DepositMonitor` is only responsible for the `OffSiteATMTest.testDeposit` test case. Both monitors are loaded by the Java SPI, which means that they only have to be used in the test classes. The `calledDeposit` event is used to instrument the application so that the attached monitors can consume these occurring events.

There are, however, two reasons why this approach is inappropriate. Managing monitors as services means that the declaration of monitors is separated from the declaration of events. The TeSSLa interpreter consumes different input streams, these input streams can be composed of several events. The monitor must know these declared events to generate streams from them. Therefore, the necessary events already defined in the test code must be redefined in the monitor class. Duplication in the event declaration leads to a higher error probability and also increases the amount of code that must be produced, especially if the number of events to be considered is large. The second disadvantage is that the bundling of monitors in jar libraries before they can be used makes the project structure more complex. Monitors are created in a separate submodule or even an external project and must be compiled before the test code is executed and then linked to the actual project, resulting in a more complex project structure.

A less dynamic and thus less complex approach, which is more test code oriented, is moving the declaration of monitors from external classes into annotations, as shown in listing 3.5. All relevant attributes such as name or path of the specification file, events as input streams, output streams and logging of streams are declared as annotation parameters.

The main advantage of this approach is that the monitor declaration can be done locally. No additional files are required and additional effort for loading the monitor during test code execution can be avoided. The problem is that annotations, which are a special type of interfaces, cannot inherit from each other. Therefore, no `input` attribute can be declared, which then gets both `@Called` and `@Returned` annotation objects inheriting from a `@Input` base annotation. This also means that as new event types are added, new attributes must be used in `@Monitor`, making the declaration code more complex. In addition, only primitive types, `String` instances, `Class` instances, enum types, and annotation types can be assigned to the annotation parameters. For each event that may occur during program execution, the monitor object must know how to translate the received event data into streams. This could be solved by using translation objects or lambda expressions that process such translations. However, this approach is not feasible due to the type limitation of the attribute parameters. Hence, this approach does not allow dynamic translation of events into streams. Finally, the event declaration is moved from the test class to the annotation. Thus, it is no longer possible for the instrumentation implementation to recognize these events. This leads to the fact that the monitor now has to ensure independently that the instrumentation is performed. In turn, this makes the simultaneous use of `TesslaMonitor` objects and LTL-based monitors more complicated.

In the last approach, the monitor declaration is similar to the event declaration. A class `TesslaMonitor` is provided by a library. During instantiation, the class `TesslaMonitor` receives the name or path of the specification file as well as the input and output streams, which are passed to the constructor via parameters. In addition to the `TesslaMonitor` class the classes `InputStream` and `OutputStream`, as well as `Transformer` and `Condition` are also provided by this library. In the `InputStream` declaration, the constructor gets the name of the stream, a `Translator` instance that translates these events into streams, and the events passed as variadic arguments. The declaration of output streams is simple as well, since the corresponding streams are returned by the TeSSLa interpreter and therefore only the name of the stream and a `Condition` object must be specified. The `Condition` object is responsible for judging the associated output stream and giving a verdict based on them. The declaration of such a monitor is done at class level and allows developers to pass the same events previously declared for application code instrumentation to the input streams. An example of this is shown in listing 3.6.

```
@Monitors({"depositMonitor"})
class OffSiteATMTest {
  private Event depositCalled = called("ATMExample/ATM", "deposit");
  private InputStream in = new InputStream("deposit", new IntStreamfromParamTranslator(1),
     depositCalled);
  private OutputStream out = new OutputStream("balance", new Condition());
  private TesslaMonitor depositMonitor = new TesslaMonitor("spec.tessla", in, out);

  @Test
  void testDeposit() {
    Account account = new AccountStub();
    OffSiteATM atm = new OffSiteATM(new SMSNotificationService());
    atm.deposit(1000, account);
    assertEquals(1000, account.getBalanceInCents());
  }
}
```

Listing 3.6: An example of the declaration of a `TesslaMonitor` object. The `depositCalled` event already created for the instrumentation can now be reused for the monitor. To declare the monitor, the `InputStream` and `OutputStream` objects must be created first. The `InputStream` object contains three parameter types: the stream name, a `Translator` object that translates events into streams, and the events to be considered by the translator. The `OutputStream` object gets the name of the stream provided by the TeSSLa interpreter and the `Condition` object which judges that stream and makes a verdict. The monitor `depositMonitor` is now created using all these objects and attached to the test class by using the `@Monitors` annotation.

For simplicity, the code required to declare a monitor can be reduced by providing a functional interface that can be used optionally. listing 3.7 provides an example of such a functional interface.

```
@Monitors({"depositMonitor"})
class OffSiteATMTest {
  private Event depositCalled = called("ATMExample/ATM", "deposit");
  private TesslaMonitor depositMonitor = TesslaMonitor.forSpec("spec.tessla").
      .inputStream("deposit", new IntStreamfromParamTranslator(1), depositCalled)
      .outputStream("balance", new Condition());

  @Test
  void testDeposit() {
    Account account = new AccountStub();
    OffSiteATM atm = new OffSiteATM(new SMSNotificationService());
    atm.deposit(1000, account);
    assertEquals(1000, account.getBalanceInCents());
  }
}
```

Listing 3.7: An example of the declaration of a `TesslaMonitor` by a functional interface. Instead of declaring `InputStream` and `OutputStream` separately, as done in listing 3.6, the declaration is made by methods on a `TesslaMonitor` instance.

This solution ensures that events declared for application code instrumentation can be reused for monitor declaration. In contrast to the pure annotation approach, developers can not only declare streams, but also specify how occurring events can be converted into streams by `Translator` instances or lambda expressions. Another important point is that monitors are declared in the test code, which means that no additional project or subproject is required to declare monitors, reducing the complexity of the project structure and setup.

The implementation, which will be presented later, follows the last presented concept. In addition to the concept of the programmatic monitor declaration, the implementation details of the monitoring are also important. The following steps are carried out in the specified sequence when the test methods of a test class are executed. Before the test code is executed, the entire test code is instrumented based on the events declared in the test class. Then, all monitors from the test class are collected, instantiated and finally added to the instrumented methods as listeners. Now, as shown in figure 3.3, the instrumented code will notify the monitor by an event during execution. The monitor then takes all its `InputStream` instances corresponding to the specified event and creates stream events with their translators, which are readable by the TeSSLa interpreter. The application code continues its execution after the interpreter has consumed and processed these stream events. Since all this is done in a single thread, the instrumented method is halted until the interpreter and then the monitor returns, which is in line with the original requirements.

3.3 Class and Interface Mocks

The last feature that the implementation will provide later is the mocking of classes and interfaces. As there are already several mocking libraries, such as Mockito [Moc19], there is a need to discuss why such a feature needs to be reimplemented. A TeSSLa monitor offers the possibility to get detailed information about the state of the system. The idea now is to connect monitors to mocks to control their behavior. As already explained in section 3.2, the execution of instrumented methods is stopped until the monitors and the TeSSLa interpreter have finished event processing. This allows Mocks to wait for the interpreter's response, in the form of output streams, to generate a return value. However, this behavior cannot be transferred to mock objects of external libraries like Mockito, which is why a separate mocking framework has to be implemented. There are requirements that the upcoming concept for mock objects must fulfill. It should be possible to mock classes, abstract classes, and interfaces. Any method within a mock object can either be linked to a monitor, controlled by an invocation handler, or simply return a default value. A mock object can be connected to different monitors at the same time. In addition, developers should be able to control the return value of methods of a mock object by attaching an invocation handler to those objects. If no invocation handler is attached to a Mock object and no monitor is associated with a particular method, that method returns a default value. If a method is connected to a monitor and an invocation handler at the same time, the values provided by the monitor are preferred.

Since monitors are connected to test methods by annotations and mocks can be connected to monitors specified in method annotations, the context in which mocks are created is the method context. In addition, a mock may require configuration, such as connecting to a

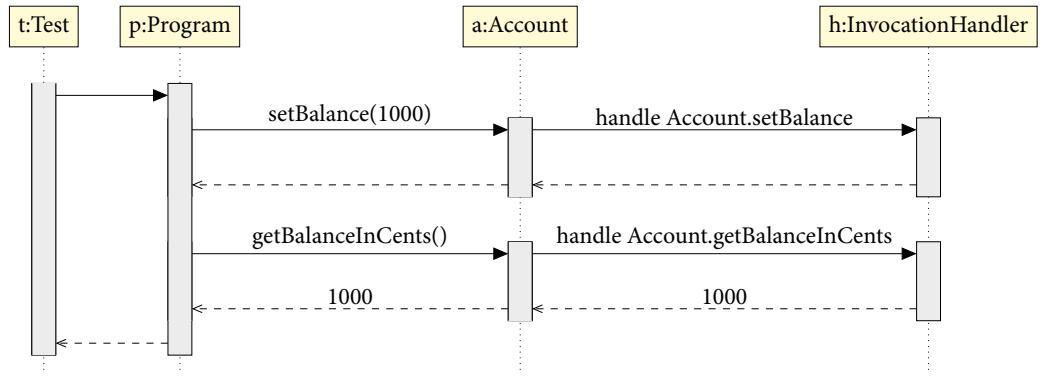


Figure 3.9: A UML sequence diagram that illustrates the method call for instances created by a `Mock<T>` object. The program is started by a JUnit test. The instance `a` is of type `Account` and was created by a `Mock<Account>` object and linked to a `InvocationHandler`. Every method call on `a` is now forwarded to the handler which generates the return values for this object. The purpose of `a` is simply to act as a middleware between the currently tested and verified code and the handler that specifies how `a` actually behaves.

particular monitor or adding a custom handler. A mock object for the `CustomClass` class is created in listing 3.8. After instantiating the mock object, a custom invocation handler is added to the mock. Finally, an instance of the mock is created using the `Mock.newInstance` method.

```
class OffSiteATMTest {
  @Test
  void testDeposit() {
    // Mock setup
    Mock<Account> accountMock = Mock.create(Account.class);
    accountMock.setHandler(new DefaultValueInvocationHandler());
    Account account = accountMock.newInstance();
    // SUT setup
    OffSiteATM atm = new OffSiteATM(new SMSNotificationService());
    // Exercise phase
    atm.deposit(1000, account);
    // Verification phase
    assertEquals(1000, account.getBalanceInCents());
  }
}
```

Listing 3.8: An example of how to create a mock object within a test class. First, the `Mock.create` method is used to create a `Mock<T>` object of the class or interface that is passed as an argument. Then the `Mock` object is configured. After the configuration of the `Mock` object, an instance of the actual class or interface can be created with the help of the `Mock.newInstance` method. Afterwards, this instance can be used as if it were a conventional instance of the mocked class or interface.

As the example of listing 3.8 emphasizes, there is a difference between a mock object of a certain class and its instances. Since instances of the class or interface to be mocked do not provide an interface for configuration, this must be done with an intermediate result. This intermediate result is an instance of the class `Mock<T>`. One advantage is, that `Mock` objects have to be configured only once, but afterwards any number of configured instances can be created. The class `DefaultValueInvocationHandler` could implement an interface `InvocationHandler`, which could be similar to the interface shown in figure 3.10.

```
public interface InvocationHandler {
  Object handle(Object instance, Method method, Object[] args, Class<?>[] argTypes);
}
```

Listing 3.10: An example of the `InvocationHandler` interface. The `InvocationHandler` interface provides only the `InvocationHandler.handle` method. This method gets four parameters. The first parameter is the instance on which the actual method was called. The second parameter is the method that was called. The second-last parameter contains an array of types of parameters passed to the actual method, while the last parameter contains the corresponding values.

The method `InvocationHandler.handle` is notified as soon as a method is called on the mocked instance. It can then calculate a return value based on the given information, e.g. the method called on the mocked instance, and returns that value. The mocked instance then takes the return value provided by the `InvocationHandler.handle` method and returns it as its own return value. This behavior is shown in figure 3.9. The object `a` of the class `Account` was instantiated by a `Mock<Account>` object. When an `InvocationHandler` is attached to the mock, it forwards all method calls to that handler. The instance `a` now behaves like a middleware. This middleware is required because during RV the currently verified code is executed in the application context while the handler is executed in the test context. The instance `a` establishes the connection between these two contexts. Furthermore, the `InvocationHandler` offers the possibility to return values to mocked objects, which are based on the output streams of a TeSSLa monitor. Listing 3.11 contains an example of how monitors can be connected to monitor output streams.

```
@Monitors({"depositMonitor"})
class OffSiteATMTest {
  // Instrumentation & Monitor setup
  private Event depositCalled = called("ATMExample/ATM", "deposit");
  private TesslaMonitor depositMonitor = TesslaMonitor.forSpec("spec.tessla").
      .inputStream("deposit", new IntStreamfromParamTranslator(1), depositCalled);

  @Test
  void testDeposit() {
    // Mock setup
    Mock<Account> accountMock = Mock.create(Account.class);
    accountMock.connectToStream("getBalanceInCents", "balance", "depositMonitor");
    Account account = accountMock.newInstance();
    // SUT setup
    OffSiteATM atm = new OffSiteATM(new SMSNotificationService());
    // Exercise phase
    atm.deposit(1000, account);
    // Verification phase
    assertEquals(1000, account.getBalanceInCents());
  }
}
```

Listing 3.11: This example creates a mock object for the `Account` class. As in listing 3.8, the intermediate object `Mock<Account>` is first created using `Mock.create`. This object is then configured. Unlike in listing 3.8, the mocked instance is now connected to a monitor with `Mock.connectToStream`. The method `Account.getBalanceInCents` is bound to the `balance` stream of the `depositMonitor`. The monitor `depositMonitor` does not have to declare the stream `balance` explicitly, because the TeSSLa specification contains this stream and therefore the TeSSLa interpreter automatically passes it on to the monitor, even if the monitor does not declare a `OutputStream` object on it.

The `connectToStream` method has three parameters. The first parameter contains the name of the method to be connected to an output stream. The second parameter is the output stream that provides the values for calculating the return value, while the third parameter is the name of the monitor that provides that stream. The monitor to which the mocked object is connected does not have to explicitly declare the stream to which the method is connected, because if the TeSSLa specification contains this stream, the TeSSLa interpreter automatically forwards it to the monitor, even if the monitor does not declare an `OutputStream` object on it.

4 The JUnitSRV Framework

This chapter introduces the *JUnitSRV* framework powered by the *SRVTestEngine*. The JUnitSRV framework consists of the existing jUnit^{RV} library which provides RV and instrumentation functionalities that can be used during testing in conjunction with a new monitor implementation utilizing the TeSSLa interpreter for verification. In addition, it provides a lightweight mocking framework that enables mock objects to be connected to monitors to adjust their behavior at runtime. The jUnit^{RV} library uses LTL-based specifications that are checked by a dedicated monitor during test execution. Whenever such a monitor detects specification violations, the execution of the test case currently running is stopped with a `AssertionFailedException`. In contrast to jUnit^{RV}, which is based on the Runner architecture of JUnit 4, the JUnitSRV framework uses JUnit 5. Due to the fact that jUnit^{RV} is integrated into JUnitSRV, the functions provided by the jUnit^{RV} library can also be used in JUnit 5 test cases. This makes it possible to monitor the application code simultaneously using LTL and TeSSLa based monitors, combining the strengths of both monitor types. The implementation of the monitors and the mocking framework corresponds to the concepts presented in chapter 3.

In the following sections, the complete implementation is not presented, as it is too complex to be explained in detail here. However, in some aspects of the implementation interesting problems arose which had to be solved. Therefore, for each main component of the implementation a detail is explained, which required special strategies for a solution of the problem. In the following, the jUnit^{RV} library is explained in general, as it is partly used for instrumentation and monitoring of test cases. Then, the implementation of the TeSSLa monitors is described, which utilizes the interfaces from the jUnit^{RV} library. Afterwards, the JUnit 5 platform is introduced, which uses the JUnitSRV framework for test case execution of monitored test cases. Finally, the custom mocking framework is presented, which allows developers to connect mock objects to TeSSLa monitors to control their behavior.

4.1 The jUnit^{RV} Library

This section introduces the jUnit^{RV} library and shows the implementation of the concept of instrumentation presented in chapter 3 section 3.1 and the concept of monitoring presented in chapter 3 section 3.2. Therefore, the JUnit 4 Framework is introduced first. Then, the internal architecture based on [Lin17], one of the JUnit 5 developers, is discussed, followed by the design and implementation of jUnit^{RV}, which is described in [DLT13].

The jUnit^{RV} library is designed and implemented on the basis of the JUnit 4 framework. In JUnit 4, the Runner architecture provided by the framework is used to customize the execution of test cases. An UML diagram containing the `Runner` class and all its parent and child

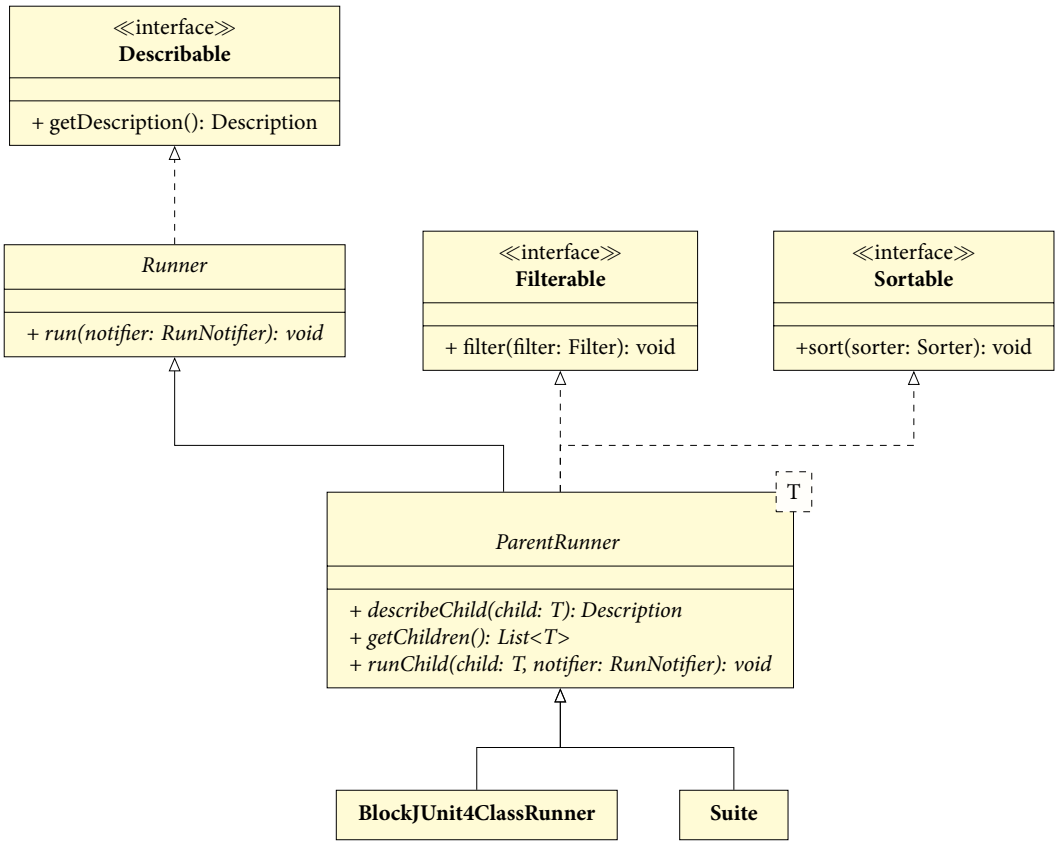


Figure 4.1: The core modules of the JUnit 4 `Runner` architecture presented as a class diagram. For reasons of simplicity, the classes and interfaces contain only their abstract methods. The classes `BlockJUnit4ClassRunner` and `Suite` do not contain any abstract methods and are therefore presented as simple classes without details in this diagram.

classes is represented in figure 4.1. All information is taken from the JUnit 4 documentation of [JUn19b].

The important class in the hierarchy of figure 4.1 is the `Runner` class. As the name suggests, the `Runner` class is responsible for controlling the execution process of test classes. A `Runner` can describe itself using the inherited method `Describable.getDescription`, which returns a `Description` object. This `Description` object contains information that is exported and can be used by other tools, such as an *integrated development environment* (IDE), to display test results. The `Runner.run` method is a generic method that executes either a test suite or a test class. Unlike in the context of JUnit 5, the class `Suite` represents test suites, so JUnit 4 explicitly distinguishes between test classes and test suites. For custom `Runner` implementations, the `Runner` class is often too generic. To facilitate the creation of custom `Runner` classes, the abstract class `ParentRunner` is also provided by JUnit 4. The `ParentRunner` class is a more specific version of `Runner` that can have multiple children. Tests are hierarchically structured in the context of JUnit in the form of trees. For example, a test suite may contain other test suites that may contain multiple test classes. Finally, these test classes

can contain several test methods. Each test suite and test class can be executed by a different `Runner` object. The generic class `ParentRunner` also provides auxiliary methods for retrieving a list of child elements via `ParentRunner.getChildren`, which can then be executed via `ParentRunner.runChild`. The JUnit 4 framework also provides two non-abstract subclasses of `ParentRunner`, the `BlockJUnit4ClassRunner` class and the `Suite` class. The `BlockJUnit4ClassRunner` is the default `Runner` class used when no other `Runner` class is explicitly attached to the test class by the annotation `@RunWith`.

The jUnit^{RV} library has its own `Runner` class, the `RVRunner`, as well as classes for event and monitor declarations to enable runtime verification during the execution of test cases. Therefore, the library includes an `Event` class that contains information about the context in which the associated event in the application code occurred. The instructions that can trigger an `Event` correspond to those already mentioned in chapter 3 section 3.1. Declaring such events causes the jUnit^{RV} library to instrument the application code. In the following, code examples are presented to illustrate the workflow of using the jUnit^{RV} library. The working example of a data service from [DLT13] is used for this purpose. This data service is described by the interface `DataService`, which is specified in listing 4.2.

```
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;
}
```

Listing 4.2: The DataService interface from [DLT13]. The `DataService` is a middleware that allows a client to communicate with a database or store data directly on disk. Before writing data using `DataService.modifyData` or reading data using `DataService.readData`, the user must connect to the service via `DataService.connect`. After writing data, possible changes must be saved by transmitting this data with the help of `DataService.commit`. Finally the connection must be disconnected at the end of the session with `DataService.disconnect`.

The `DataService` can be considered as a middleware between an application and a persistence system responsible for storing the transmitted data. Saving the data can be done by using a database or by document oriented storage. A user can connect to the `DataService` through the `DataService.connect` method to read data from the information system using `DataService.readData`, or to change data permanently using `DataService.modifyData`. All transactions performed during a session must be stored in the information system by `DataService.commit`. At the end of the session the user disconnects from the service via `DataService.disconnect`.

Two things are necessary to monitor this `DataService`. First, a property must be defined that should be checked during the execution of the SUT. Second, the events required for the property must also be defined. If data was changed during the session, the client must instruct the `DataService` to commit the changes before the user signs off and disconnects

from the service, otherwise local changes would be lost. The jUnit^{RV} library synthesizes monitors from LTL, so the property must be formulated as a LTL formula:

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

In the corresponding JUnit test case there must be events objects that correspond to the methods `DataService.modify`, `DataService.disconnect` and `DataService.commit` in order for the monitor to consume them. As mentioned in section 3.1, this can be achieved by using the `Event` class and the static auxiliary methods of the `SimpleSyntax` class from the jUnit^{RV} library.

```
private static String dataServiceQName = "myPackage/DataService";
private static Event modify = called(dataServiceQName, "modifyData");
private static Event committed = returned(dataServiceQName, "commit");
private static Event disconnect = called(dataServiceQName, "disconnect");
```

Listing 4.3: The declaration of events in jUnit^{RV}. First, the fully qualified class name of the class `DataService` is declared as a string instance variable `dataServiceQName`. Unlike in Java, a forward slash is used instead of a period between the namespace components. The declaration of all necessary events is done by the auxiliary methods `SimpleSyntax.called` and `SimpleSyntax.returned`. Both auxiliary methods require the fully qualified class name as the first parameter and the name of the method that triggers the event as the second parameter.

The fully qualified name of the class to be instrumented is first stored as a string in an instance variable. In contrast to Java, a forward slash is used instead of a period between the namespace components. Then, `Event` objects are declared that are to be consumed later by the monitor. The declaration of all necessary events is done by the auxiliary methods `SimpleSyntax.called` and `SimpleSyntax.returned`. Both auxiliary methods require the fully qualified class name as the first parameter and the name of the method that triggers the event as the second parameter. These events can be used to create the actual monitor using the above LTL formula. Listing 4.4 contains the declaration of a `LTL4Monitor` that is synthesized from both the LTL expression created by the `SimpleSyntax.Always`, `SimpleSyntax.Until` and `SimpleSyntax.implied` methods, and the events declared in listing 4.3.

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

Listing 4.4: A `LTL4Monitor` that monitors the $\texttt{Always}(\texttt{modify} \Rightarrow \neg\texttt{disconnect Until committed})$ property while executing the test cases to which the monitor is attached. The monitor consumes the events `modify`, `committed` and `disconnect`. The LTL formula is constructed using the methods `SimpleSyntax.Always`, `SimpleSyntax.implies` and `SimpleSyntax.Until`.

The LTL formula shown above is constructed using the methods `SimpleSyntax.Always`, `SimpleSyntax.implies` and `SimpleSyntax.Until` and then passed to the monitor constructor. Since the `RVRunner` extracts and instantiates the monitors from the test class before it is instantiated itself, the monitor instance variable must be static. This in turn means that the events must also be static. The declared `FLTL4Monitor` can now monitor test cases by

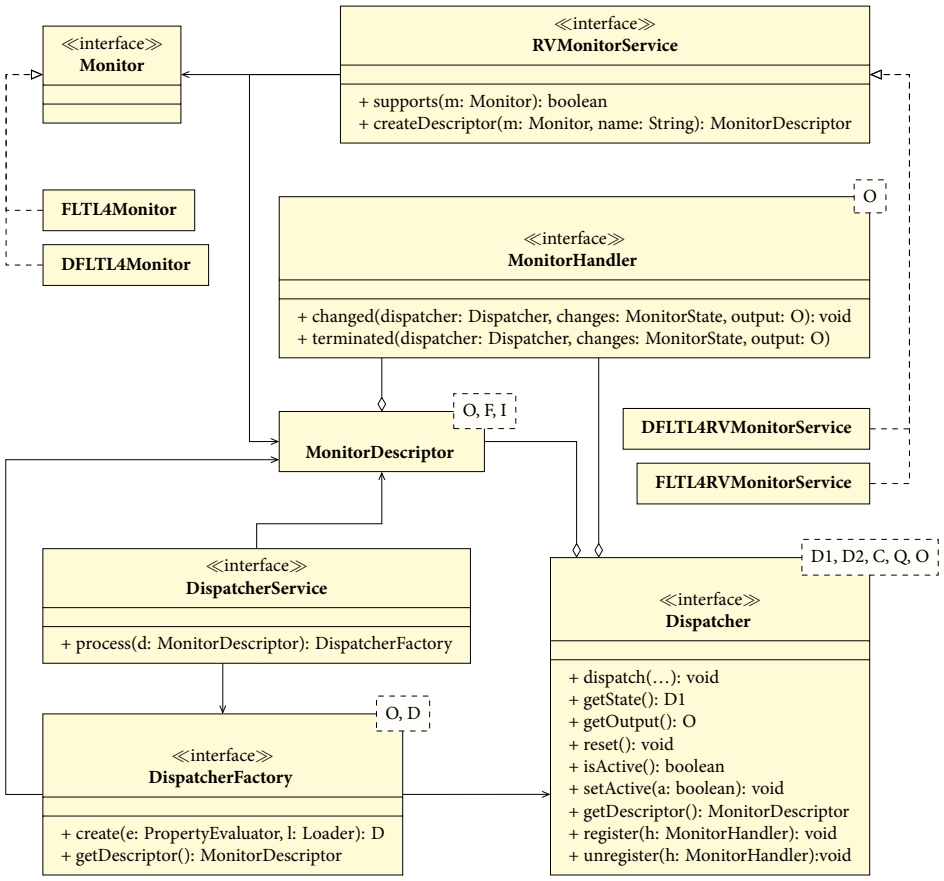


Figure 4.5: The context of the classes `Monitor`, `MonitorDescriptor` and `Dispatcher`. The `Monitor` class only serves as a container for the information needed to monitor and preserve the underlying formalism. The same information is contained in the `MonitorDescriptor` used by the `RVRunner`. The dispatcher is compiled into the bytecode of the monitored application classes and processes the occurring events according to the formalism of the `Monitor` class belonging to the `MonitorDescriptor`.

attaching it to test classes or test methods using the `@Monitors` annotation. When a monitor is attached to a test class, all test cases of that class are monitored by it.

During test execution, the declared events must be monitored. The sequence of events that occur during execution is the trace, which is processed and verified by the formalism passed to the actual monitor. To use the monitoring of the jUnit^{RV} library, it provides three main modules, the `Monitor` class, the `MonitorDescriptor` class, and the `Dispatcher` class. Listing 4.5 illustrates the context of all these classes. An instance of `Monitor` is declared within the test class and contains information about the monitored events and the formalism used for verification, i.e. a LTL formula. The `FLTLMonitor` in the example of listing 4.6 receives the events it depends on in the form of the LTL formula passed. The `RVRunner` extracts all `Monitor` instances by collecting the monitor names from the `@Monitors` annotations on the test class and its test cases using the *Reflection API* [Ora19]. The instance variables of the test class are then accessed by name to obtain the monitors that appear in the `@Monitors` annotation of that class or test case. Each monitor is instantiated by a `RVMonitorService`. The

```
@Monitors({"commitBeforeDisconnect"})
@RunWith(RVRunner.class)
public class DataServiceTest {
  private static String dataServiceQName = "myPackage/DataService";
  private static Event modify = called(dataServiceQName, "modifyData");
  private static Event committed = returned(dataServiceQName, "commit");
  private static Event disconnect = called(dataServiceQName, "disconnect");
  private static Monitor commitBeforeDisconnect = new FLTL4Monitor(
      Always(implies(modify, Until(not(disconnect), committed))));

  @Test
  public void test() {
    DataService service = new MyDataService("http://myserver");
    service.connect("12345");
    service.modifyData("value", new IntData(42));
    service.commit();
    service.disconnect();
  }
}
```

Listing 4.6: An exemplary test class that declares events on the `DataService` class and provides them to a `FLTL4Monitor` instance in the form of a LTL expression. The monitor is attached to all test cases of the `DataServiceTest` class because `DataServiceTest` is annotated with `@Monitors`. The values passed to `@Monitors` are the names of the monitor variables declared in the test class that are to be considered for monitoring.

`RVMonitorService` is loaded via Java's SPI. It decides whether a `Monitor` object is supported, and it can create a `MonitorDescriptor` from the `Monitor` instance. Among other information, the `MonitorDescriptor` contains the information that the corresponding `Monitor` object contains and is only used by internal classes of the jUnit^{RV} library. In the next step, DispatcherServices are loaded via the SPI. The task of these services is to create `Dispatcher` instances based on the `MonitorDescriptor` they contain. Finally, the `Dispatcher` instances are compiled into the bytecode of the instrumented methods. This bytecode manipulation is done via the Javassist library [Chi19]. Depending on the type of event, the invocation of `Dispatcher.dispatch` is appended to or prefixed to the instrumented method of the SUT. Now, when an instrumented method is called, the context information is collected and passed to the `Dispatcher.dispatch` invocation. To perform bytecode manipulation, a custom *ClassLoader* is required to intercept the loading process. A `ClassLoader` is responsible for loading class files at runtime when a instruction creates a new instance of a particular class. In this case, the JVM requests this class from the `ClassLoader`, if not already present. If a class was loaded by a particular `ClassLoader`, the same `ClassLoader` is responsible for loading dependent classes as needed. Since the `RVRunner` executes the test class, it can load its test class with a custom `ClassLoader` and load all application classes that are supposed to be tested.

In order to extend the jUnit^{RV} library with user-defined monitor types, implementations for the interfaces in 4.5 must be provided via the SPI during test case execution. These functions enable the implementation in the following sections to outsource the monitoring process to the jUnit^{RV} library and ensure that the additional functions of the JUnitSRV framework are consistent with the jUnit^{RV} library.

4.2 TeSSLa-Based Monitors

This section presents the implementation of the TeSSLa monitor concepts from chapter 3 section 3.2. As already described in the section 4.1, the jUnit^{RV} library supports dynamic extensions for monitors by using Java's SPI. Therefore the new monitor services of JUnitSRV are discussed first and then their internal structure and classes are introduced. Finally, examples for different monitors using listings are shown.

Figure 4.7 is a simplified overview of the services and associated classes implemented by the JUnitSRV framework. The classes and interfaces in this UML class diagram do not contain any attributes or methods, otherwise the diagram would have become too complex. In this diagram, boxes represent classes and circles interfaces. The meaning of arrows still corresponds to that of conventional UML diagrams. As usual in the UML standard, italic font still means that it is an abstract component. The services necessary to recognize `TesslaMonitors` during the transformation of test cases are the `TesslaMonitoringService`, which is responsible for identifying `TesslaMonitor` instances, and the `TesslaMonitorDispatcherService`, which is responsible for creating `Dispatcher` instances for the instrumented classes. In addition to these services, a custom `Monitor` implementation the `TesslaMonitor` is also provided by the JUnitSRV framework. The abstract class `TesslaMonitorAttribute` serves as a marker class used by `TesslaMonitor` to describe an attribute that configures the monitor. As described in section 4.1, the `TesslaMonitor` is a container that contains the configuration for monitoring behavior and is not used during actual monitoring, while the `TesslaMonitorDispatcher` is embedded in the application code. By implementing all these interfaces and providing the necessary services, the code instrumentation of the jUnit^{RV} library is able to support the new `TesslaMonitors` for test case monitoring.

While the instrumented application code is being executed, `Dispatcher.dispatch` is called. The `Dispatcher` can then process the event and calculate a result that is passed to the attached handlers. The monitors provided by the jUnit^{RV} library are based on automatons, which means that each event results in a new state in the monitor automaton. However, the `TesslaMonitor` does not use any automatons for monitoring, but the TeSSLa interpreter which receives discrete events on certain streams and correlates these input streams with the resulting output streams. The monitor must give a verdict on the basis of these output streams. For this purpose, the JUnitSRV framework requires two mechanisms: `InputStream` objects, which refer to `StreamTransformer` objects, and `OutputStream` objects, which refer to `StreamCondition` objects. Events processed by a `Dispatcher` contain information such as the instance on which a particular method was called, the arguments of that method, the method itself, the return value, and some other information in the form of a `StreamEvent` object. The `InputStream` is responsible for preparing this information and translating it into streams. For example, an integer event stream should provide events that represent

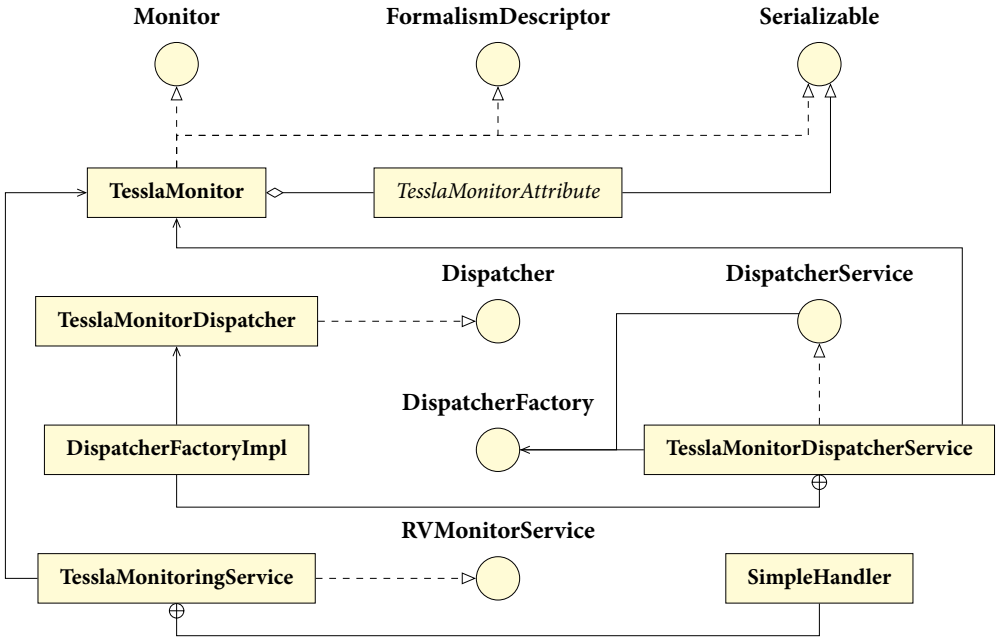


Figure 4.7: A simplified hierarchical overview of the `TesslaMonitor`, its `TesslaMonitoringService`, its `TesslaMonitorDispatcher`, its `TesslaMonitorDispatcherSerivce`, and the `TesslaAttribute` class. The meaning of the arrows continues to correspond to that of conventional UML diagrams. Concrete classes are represented as boxes, while interfaces are represented as circles. The displayed interfaces are provided by the jUnit$^{\mathrm{RV}}$ library.

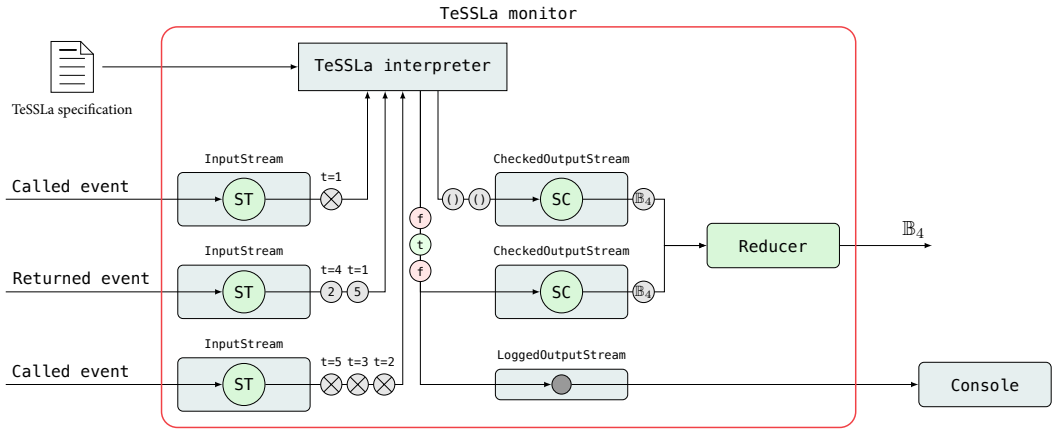


Figure 4.8: An overview of the structure of the TesslaMonitor. The left side of the monitor shown in the picture shows the input for a `TesslaMonitor`. The `InputStream` instances use a `StreamTransformer` instance to translate instrumentation events into streams. The result of the stream transformation is passed as stream events over to the TeSSLa interpreter, which calculates a step from these events. The resulting output streams are passed to the corresponding `OutputStream` objects. Each `CheckedOutputStream` instance uses an instance of the `StreamCondition` class to calculate a $\mathbb{B}_4$ value. These values are collected and aggregated by the `Reducer` of the monitor to calculate a final verdict for this step. The instances of the `LoggedOutputStream` class ensure that the corresponding streams are printed on the console. The right side of the monitor represents the output of the monitor.

one of the arguments of a particular method. A `StreamTransformer` is used to perform this transformation. The `StreamTransformer` retrieves the `StreamEvent` and calculates a resulting value. The `InputStream` then uses this resulting value and feeds the TeSSLa interpreter with it. After the interpreter has finished calculating the output stream events for this step, the `OutputStream` objects come into action. There are two types of output streams. The `CheckedOutputStream`, which is taken into account when calculating a verdict, and the `LoggedOutputStream`, which is used to output the events of the output stream, provided by the TeSSLa interpreter, to the console to enable debugging. Each `CheckedOutputStream` uses a `StreamCondition` to decide whether a condition is violated or not. The resulting value is a $\mathbb{B}_4$ value. Since a monitor can use multiple `CheckedOutputStream` objects, a so-called `Reducer` is used to collect and aggregate the verdicts of each `CheckedOutputStream` after each step. JUnitSRV offers the `ConjunctionReducer`, which is the default `Reducer`, and the `DisjunctionReducer`. The first calculates the conjunction of all `CheckedOutputStream` verdicts, while the second calculates the disjunction. The structure of a TesslaMonitor is shown in the figure 4.8. All components that are colored green in this graphical representation are freely configurable by the developer. Since the developer has the possibility to influence data that goes to the TeSSLa interpreter and can also process data that comes back from it, the developer can dynamically adapt a `TesslaMonitor` to almost any use case. To show how a `TesslaMonitor` is used within JUnit, the working example from chapter 1 is used again.

Listing 4.9 contains both a JUnit test case and a TeSSLa specification which is used for SRV during test case execution. In contrast to conventional test cases, monitored test cases are annotated using the annotation `@MonitoredTest` instead of `@Test`. The reason for this is explained in section 4.4. The TeSSLa specification receives as input stream the deposits made using the `ATM` class and defines constraints on that stream to be met during monitoring. If the constraints are not met, a unit event is generated on the output stream `error` together with a message on the `error_message` stream. The `TesslaMonitor` provides a functional interface for its configuration. For example, the `TesslaMonitor.forSpec` method is used to identify the specification used for stream processing during verification, while the `TesslaMonitor.inputStream` method specifies an input stream that is passed to the interpreter, and the `TesslaMonitor.outputStream` method specifies an output stream used for the final verdict. Finally, the `TesslaMonitor.logAllOutputStreams` method takes care of printing all output streams from the TeSSLa interpreter on the console. Both objects `transformer` and `condition` are used by the object `InputStream` and `CheckedOutputStream` respectively. There are two arguments for the `ATM.deposit` method. The first argument is the amount of money to be deposited. This value is translated by the `transformer` object into stream events on the `deposit` stream. Whenever the `deposit` method is called on a class that implements the `ATM` interface, this translator creates an event that is then passed

```
@DisplayName("Tessla Monitored Tests")
class MonitoredOffSiteATMTest {
  private static final String atmQName = "ATMExample/ATM";
  private static final Event calledDeposit = called(atmQName, "deposit");
  private static final StreamTransformer<TesslaInt> transformer = (StreamEvent event) -> {
    return new TesslaInt((Integer) event.getArgs()[0].getValue());
  };
  private static final StreamCondition<TesslaUnit> condition = (TesslaUnit event) -> {
    return Boolean4.FALSE;
  };
  private static final Monitor exampleMonitor = TesslaMonitor.forSpec("spec.tessla")
      .inputStream(new InputStream<>("deposit", transformer, calledDeposit))
      .outputStream(new CheckedOutputStream<>("error", TesslaUnit.class, condition))
      .logAllOutputStreams();

  @Monitors({"exampleMonitor"})
  @MonitoredTest
  void testDepositSRV() {
    // Exercise Phase
    atm.deposit(2000, account);
    // Verification Phase
    assertEquals(2000, account.getBalanceInCents());
  }
}
```

```
-- TeSSLa specification
in deposit: Events[Int]
def condition: Events[Bool] := deposit > 1000
def error: Events[Unit] := if condition then ()
def error_message: Events[String] := if condition then "deposit exceeded 1000 cents"
out *
```

Listing 4.9: A simple example of the declaration of a `TesslaMonitor`. The declaration of `Event` objects corresponds to that of the jUnit^{RV} library. Next, a `StreamTransformer` is declared that extracts the first argument from `ATM.deposit` calls and translates it into integer events that can be processed by the TeSSLa interpreter. Then, a `StreamCondition` is declared which causes the monitor to fail when an event is consumed on the `error` stream. Finally, a `TesslaMonitor` with all components is declared. Below the JUnit test class is the corresponding TeSSLa specification, which creates a unit event on the `error` stream when the input stream `deposit` returns an event containing an integer data value exceeding 10. All streams, including the input stream, are then considered as output streams.

to the TeSSLa interpreter. The interpreter processes all stream events and provides a set of associated output streams. Then, the object `condition` processes the events on the output stream for which the `OutputStream` to which it belongs is responsible. In the case of the monitor from listing 4.9 the `condition` object checks whether an event occurs on the `error` stream. If this is the case, it will always return false, so the monitor will always fail when an event occurs on the `error` stream.

Since the JUnitSRV framework uses the Scala interface of the TeSSLa interpreter, special types are used to represent primitive values such as integers. Therefore, the JUnitSRV framework provides wrapper classes for these types. In the example the two types `TesslaInt` and `TesslaUnit` are used. This approach has two advantages. First, although the interpreter and the test code are executed in the same JVM, arbitrary types cannot simply be passed to the interpreter, since the wrapper classes serve as restrictions on possible types during the monitor declaration. Second, the type of objects passed to and received from the TeSSLa interpreter

```
@DisplayName("Tessla Monitored Tests")
class MonitoredOffSiteATMTest {
  private static final String atmQName = "ATMExample/ATM";
  private static final Event calledDeposit = called(atmQName, "deposit");
  private static final Monitor exampleMonitor = TesslaMonitor.forSpec("spec.tessla")
      .inputStream("deposit", intEventsFromArg(1), calledDeposit)
      .failOnEvent("error")
      .logAllOutputStreams()
      .reportOn("error_message");

  @Monitors({"exampleMonitor"})
  @MonitoredTest
  void testDepositSRV() {
    // Exercise Phase
    atm.deposit(2000, account);
    // Verification Phase
    assertEquals(2000, account.getBalanceInCents());
  }
}
```

Listing 4.10: A simplified example for monitoring test cases. The monitor declaration is simplified in contrast to the monitor declaration in listing 4.9 by using the class `Convenience` and its static auxiliary methods. Both, the objects `transformer` and `condition` are created by the class `Convenience`. The method `Convenience.intEventsFromArg` generates a stream transformer, which translates the argument at the position specified by the parameter of the `Convenience.intEventsFromArg` method from the instrumented method into an integer, that is then passed on as a stream event to the TeSSLa interpreter. The `TesslaMonitor.failOnEvent` method causes the monitor to fail when an event is consumed on the `error` stream. If this is the case, the message in the `error_message` stream is used as an error report. In addition, all output streams from the specification are printed to the console using the `TesslaMonitor.logAllOutputStreams` method.

is typed and thus the responsibility for type checking shifts from the TeSSLa interpreter to the SRVTestEngine. This allows early detection of errors in the use of monitors, and data transformations during test case execution.

In order to facilitate the usage of the interface of JUnitSRV, the JUnitSRV framework provides the class `Convenience`, which contains some static methods, that allow to create certain objects like, e.g., `InputStream`, `OutputStream`, `StreamTransformer` and `StreamCondition`. The monitor declaration in listing 4.9 can be simplified by using the class `Convenience`. The `Convenience.intEventsFromArg` method returns a `StreamTransformer` similar to the `transformer` object in the listing 4.9. It receives the index of the argument to be translated into an event. Using the `TesslaMonitor.failOnEvent` method, a `CheckedOutputStream` object can be created that behaves exactly the same as the `condition` object from listing 4.9. Unlike monitors based on LTL formulas, monitors synthesized from TeSSLa can specify multiple conditions that can fail, and therefore in some cases a detailed message with information about the failed condition is required to determine the cause of the error. For this purpose, the `TesslaMonitor` can be configured using the `TesslaMonitor.reportOn` method to display the string that occurs on the stream passed by name as an argument. In the example of listings 4.9, the monitor displays detailed information about an error using the stream `error_message`.

4.3 The JUnit Platform

This section introduces the *JUnit Plaform*. First, it explains how the JUnit Plaform architecture allows developers to extend functionality through custom frameworks. Then the responsibilities of these custom test engines are explained.

As shown in figure 4.12, the architecture of JUnit 5 is divided into the JUnit Platform, which is the link between the client application and the actual testing frameworks, and the respective testing frameworks. By default, the JUnit 5 framework offers the two testing frameworks *JUnit Vintage* and *JUnit Jupiter*. While the JUnit Vintage framework is responsible for executing old JUnit 4 test cases on the new JUnit 5 platform, JUnit Jupiter executes JUnit 5 test cases. During the test case execution all testing frameworks are considered at the same time. The JUnit Platform also allows the usage of custom testing frameworks during test case execution. A test engine contains the business logic by which the JUnit tests are performed. Each testing framework needs its own test engine, a class that implements the `TestEngine` interface. The JUnit Platform uses the SPI to load `TestEngine` implementations during runtime and then use it for test case execution. Therefore, the class name of the `TestEngine` implementation of each testing framework must be added to the `META-INF/services/org.junit.platform.engine.TestEngine` file in the corresponding jar file. Using this file, the JUnit Platform service loader can determine custom implementations of the `TestEngine` interface using Java's SPI. Listing 4.11 contains the interface `TestEngine`.

```
public interface TestEngine {
  String getId();
  TestDescriptor discover(EngineDiscoveryRequest discoveryRequest, UniqueId uniqueId);
  void execute(ExecutionRequest request);
}
```

Listing 4.11: The JUnit `TestEngine` interface. Custom `TestEngines` that run test cases during testing must implement this interface and must be provided to the Javas SPI. A `TestEngine` has a unique id which is returned by `TestEngine.getId`. The is also responsible to extract all its test cases from the given `EngineDiscoveryRequest` and returns the resulting tree-based structure as a `TestDescriptor`. Besides the discovery process a `TestEngine` can also execute a `TestDescriptor` via its `TestEngine.execute` method.

The JUnit Platform also offers the Runner class `JUnitPlaform`. This class allows tests to be run on the JUnit Platform in a JUnit 4 environment. Also of great importance is the JUnit Platform Launcher, which is used by the actual client application to perform testing. In most cases, the client application is an IDE, such as IntelliJ, Eclise or NetBeans. However, it is also possible to interact with JUnit via a *command line interface* (CLI). The JUnit Platform Launcher requires a `TestPlan` for the actual execution, which is generated by the implementation of the `Launcher.discovery` method. The `Launcher` class forwards the discovery and execution requests to any attached `TestEngine`. A `TestEngine` is responsible for two tasks. First, it receives an instance of `EngineDiscoveryRequest`, which gives

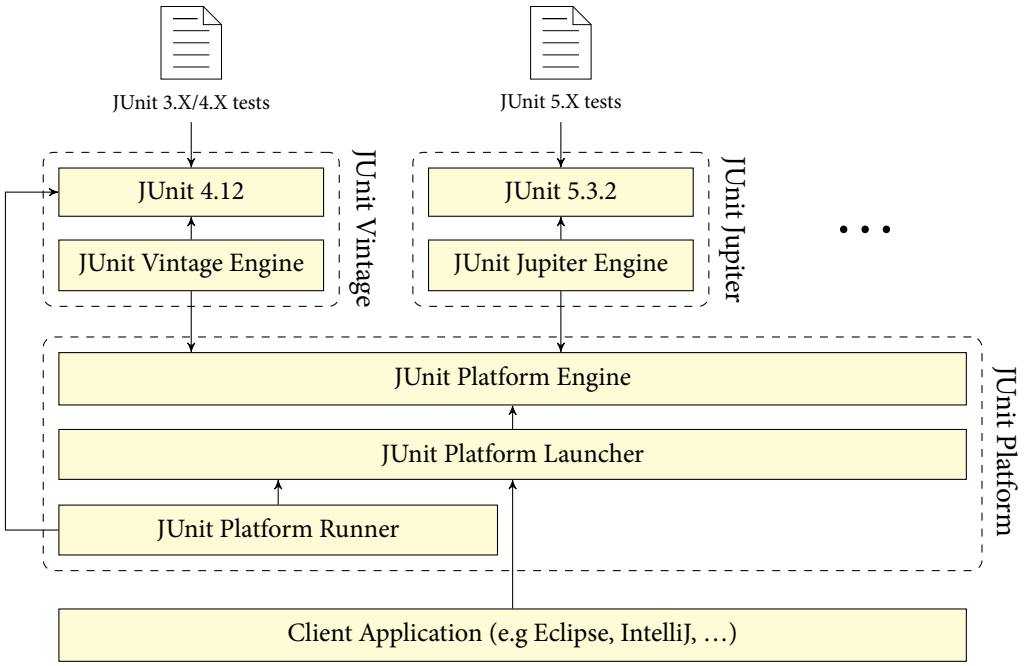


Figure 4.12: The layers of a project that uses JUnit 5. The client application is an application that can run JUnit tests such as Eclipse or IntelliJ. This application communicates with the JUnit Platform to discover and run certain tests. The detection and execution requests are passed to all connected testing frameworks and the results are post-processed by the JUnit Platform before being passed to the client application. The client application can then present a representation of the results.

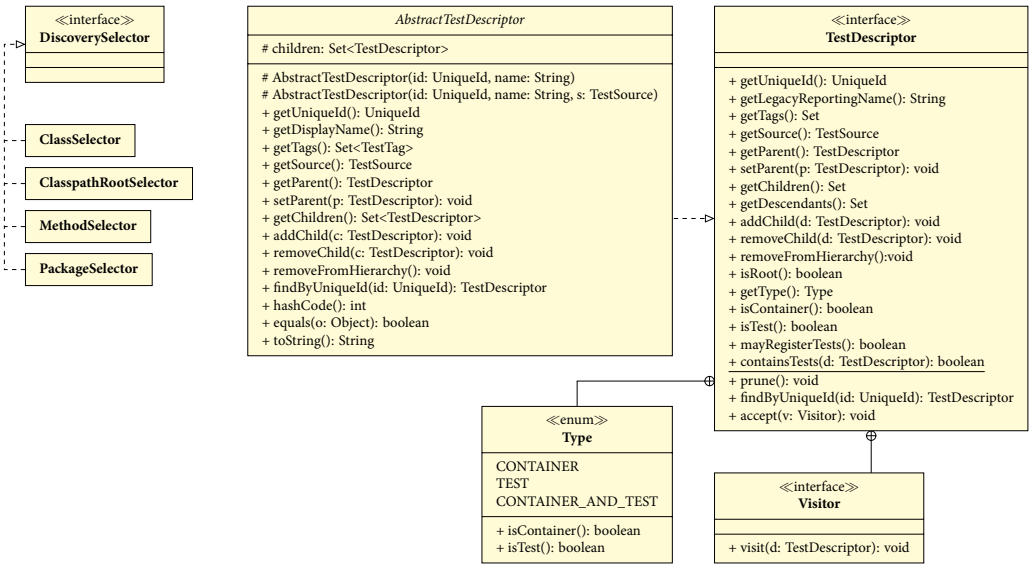


Figure 4.13: The class hierarchy of the `TestDescriptor`. A `TestDescriptor` contains information about the corresponding test. Each test suite, test class and test method is described by a `TestDescriptor`. Each `TestDescriptor` has subdescriptors, i.e. a test class can consist of several test cases. To simplify the definition of user-defined `TestDescriptors`, the JUnit Platform provides the abstract class `AbstractTestDescriptor`, which has already implemented most interface methods.

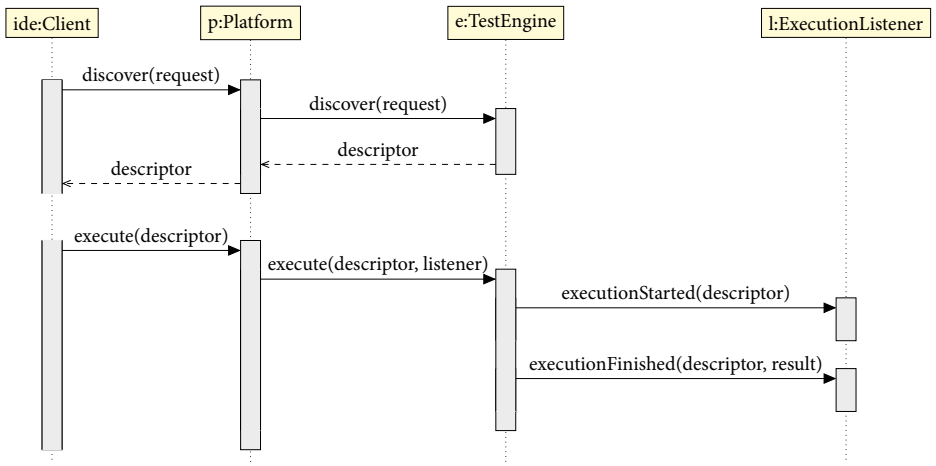


Figure 4.14: The discovery and execution process of test cases in JUnit 5. First, the client application sends a discovery request to the JUnit Platform to extract test cases from the specified selector. The JUnit Platform forwards the request to all loaded `TestEngines` to create a test descriptor. At a later stage, the client can perform tests that are detected by an execution request. The JUnit Platform redirects this request to all `TestEngines` that support the given test cases. The `TestEngine` can then execute the test cases from the descriptor. During execution, it notifies the appropriate `ExecutionListener` when test cases have been started, terminated, or failed.

the `TestEngine` access to the information needed to recognize tests and containers. The `EngineDiscoveryRequest` consists of selectors and a filter. For example, a selector can contain the name of a Java class, the path to a file or directory, and possibly other details. The corresponding class hierarchy is shown in figure 4.13 and describes all `DiscoverySelectors` used by the `SRVTestEngine`. While each `MethodSelector` and `ClasspathRootSelector` provides the current class with its `getJavaClass` method, the test classes must be extracted from `PackageSelector` and `ClasspathRootSelector` using auxiliary methods provided by the JUnit framework. It is the responsibility of the `TestEngine` to find all relevant tests and containers from the provided `DiscoverySelector` objects. The filter, on the other hand is responsible for providing information about which resources should be considered by the `TestEngine` and which resources should be ignored. The `TestEngine.discover` method is responsible for creating a representation of tests from the selectors and filters of the `EngineDiscoveryRequest`. This test representation is a tree containing only containers and tests. A container is a structure that can contain other containers or tests, i.e. a test class, while a test is an executable test case, i.e. a method annotated with `@Test`. Containers represent nodes and tests represent leaves in the corresponding tree. Each container and each test must be translated from the `TestEngine` into a `TestDescriptor`, which is shown in figure 4.13. The hierarchy of the `TestDescriptors` contains all test containers and tests of the provided selectors as well as additional data that the test to be executed requires.

In addition to the discovery process of test cases, the execution of test cases is also performed by `TestEngine` instances. The `TestEngine.execute` method receives an instance of `ExecutionRequest`, which includes the `TestDescriptor`, that was previously created by

the TestEngine.discover method; an instance of EngineExecutionListener, which is a listener that is notified when tests are started, stopped, or skipped; and some configuration parameters in the form of an ConfigurationParameters object. Figure 4.14 visualizes the process of detection and execution of all detected tests. Since both processes are decoupled, the client application can decide when tests are detected and when they are executed. For example, tests can be discovered by an IDE to visualize the test structure in a special view, but they can also be discovered immediately before execution.

4.4 The SRVTestEngine

In this section the SRVTestEngine that powers the JUnitSRV framework is introduced. The SRVTestEngine implements the TestEngine interface shown in section 4.3 and adapts the behavior of the RVRunner class introduced in section 4.1. Since the application code instrumentation of the jUnit^{RV} library should be used within the SRVTestEngine, the test class transformation applied by the RVRunner to the test classes must be performed by the SRVTestEngine. As mentioned in the previous sections, the JUnit 5 architecture does not provide Runner classes. In particular, the JUnit 5 architecture supports two methods for intercepting the execution of test cases. First, by using *extensions*. Extensions are basically callbacks that are called at different stages of the test case execution. Since extensions are designed to be called at different stages and the executed test class cannot be preprocessed by them, they cannot load test classes with a user-defined class loader, which is essential for the instrumentation performed by the jUnit^{RV} library. Therefore, a custom TestEngine is the last and only option that provides the ability to instrument application code during the class loadindg process and thus enable RV during test case execution.

Figure 4.15 illustrates the main classes that form the actual SRVTestEngine. The interface TestEngine and the abstract class AbstractTestDescriptor are the same as in figure 4.13. The SRVTestEngine.discover creates a TestDescriptor tree structure of all tests extracted from the selectors of the specified EngineDiscoveryRequest. Unlike JUnit Jupiter, a test case executed by the SRVTestEngine contains additional information such as the connected monitors. Therefore, the SRVDescriptor inherits the AbstractTestDescriptor. Thus, the SRVDescriptor is a conventional TestDescriptor and can be used by the JUnit Platform as such, in addition the SRVTestEngine can add further information to it, which can be used during test case execution. During execution, the TestDescriptor tree is translated into a SRVExecutionNode tree. This is necessary because the SRVTestEngine does not use test classes as they are to execute its test cases. Each test class is transformed exactly the same as the RVRunner transformed these classes in the context of the jUnit^{RV} library. A representation of the data structures created by the JUnit Platform and the SRVTestEngine is shown in figure 4.16. First, the JUnit Platform passes an ExecutionRequest to the SRVTestEngine,

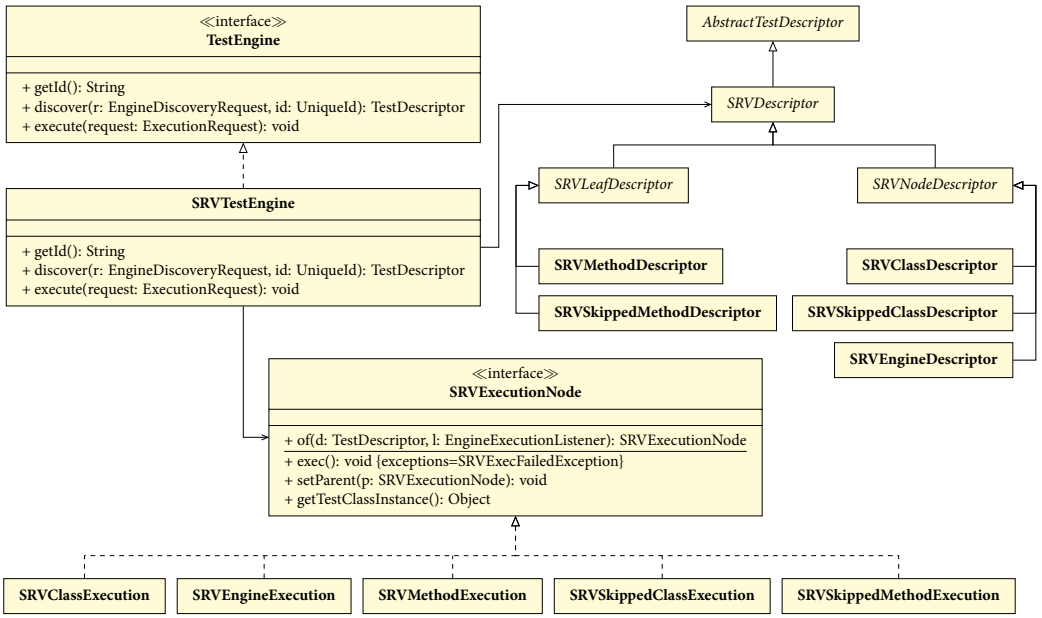


Figure 4.15: The structure of the `SRVTestEngine`. A UML diagram containing all classes important with regard to the `SRVTestEngine`. The `SRVTestEngine` implements the interface `TestEngine`. By combining this implementation with Java's SPI it can be loaded from the JUnit Platform at runtime. The `SRVTestEngine` uses its own test descriptors, the `SRVDescriptors`, to add additional information to them. By using the composite design pattern, a tree-like structure is created from the `DiscoveryRequest`. Before the test case is executed, an additional tree structure of `SRVExecutionNode` objects is generated from the `SRVDescriptor` tree structure and linked to it. An example of this process is shown in Figure 4.16.

which then translates it into a tree structure of `TestDescriptors` during the discovery process. By using the `SRVDescriptor` class, additional information can be appended to these descriptors to increase performance during test case execution. When the JUnit Platform executes test cases, the tree structure of `SRVTestDescriptors` is passed back to the `SRVTestEngine`. Then, it translates the descriptors into a tree of `SRVExecutionNodes`. Each node corresponding to a `SRVClassDescriptor` transforms the corresponding test class using the `TestTransformation` from the jUnit^{RV} library. During the transformation, monitors are extracted and instantiated for later use. The `SRVClassExecution` node then creates an instance of the transformed test class to be used when executing its test cases.

The `SRVExecutionNode` and its implementing classes apply the *composit* pattern [Gam+95]. According to [Gam+95] the composite pattern is used to assemble objects into a tree structure representing part-whole hierarchies. Any class that implements the `SRVExecutionNode` interface is also treated as a `SRVExecutionNode`. It can be executed, which triggers the execution of all child `SRVExecutionNodes` and it contains contextual information about the executing unit as well as the corresponding `SRVDescriptor` and an `ExecutionListener`. The `ExecutionListener` is used to inform the JUnit Platform that the execution of a `TestDescriptor` node has been started, stopped, or skipped. The `SRVClassExecution` class holds a reference to the corresponding `SRVClassDescriptor` that provides access to attached monitors. It also creates the current instance of the transformed test class. Within the

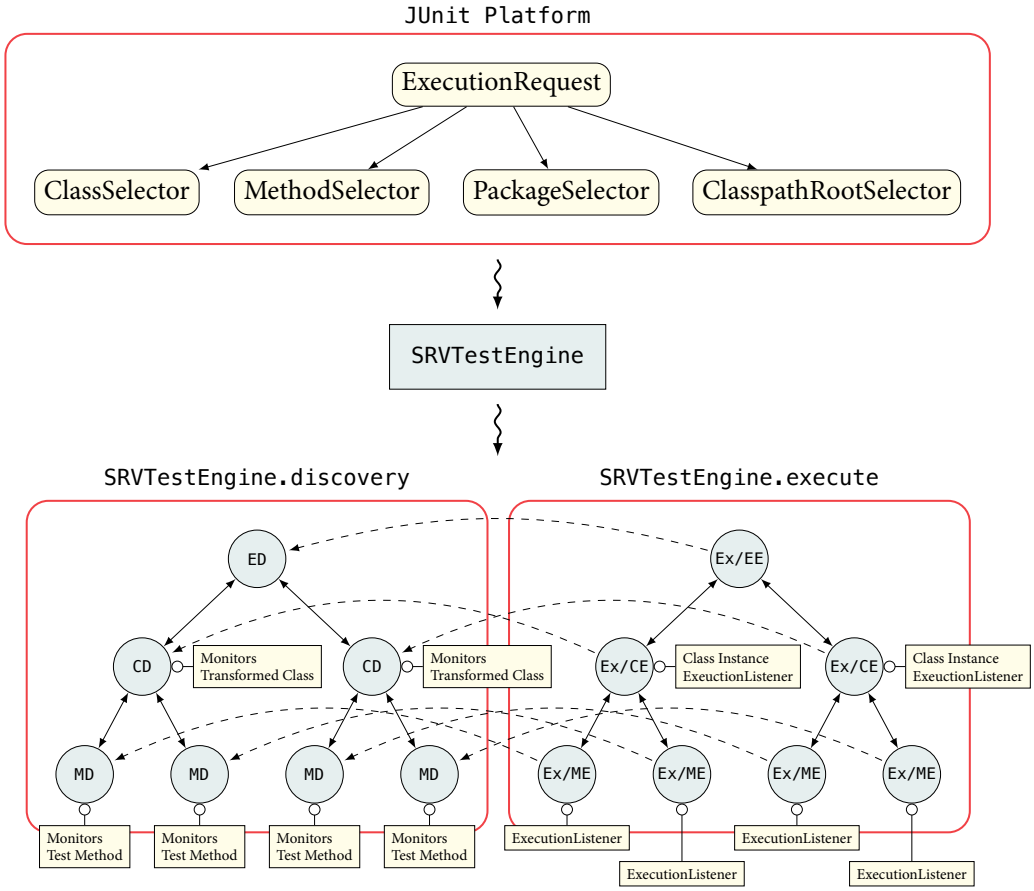


Figure 4.16: A graphical representation of the data structures used in test discovery and test execution process by the `SRVTestEngine`. The client application starts the test discovery by providing all the resources to be considered for the discovery process. These resources can be classes, methods, packages, or a classpath. During the search for tests, the `SRVTestEngine` creates a tree structure that represents the parent-child relationship of test classes, nested classes, and their test cases. Additional information such as a transformed version of the test class is also appended to the nodes and leaves of the tree. When test cases are executed, a congruent representation is created from `SRVExecutionNodes` of the `TestDescriptor` tree. Each node at the same position in both trees is connected.

`SRVClassExecution.exec` method the methods attached with `@BeforeAll`, `@BeforeEach`, `@AfterAll` and `@AfterEach` and the child nodes of type `SRVMethodExecution` are executed.

When test cases are discovered, an `EngineDiscoveryRequest` is passed to each `TestEngine` loaded via the SPI. Using this `EngineDiscoveryRequest`, the `TestEngine` determines which test cases should be considered during execution. The jUnit^{RV} library uses the annotation `@Monitors` to identify the test cases to be monitored during execution. However, this approach is not applicable by the `SRVTestEngine` because a test case annotated with both annotations `@Monitors` and `@Test` would be executed by both the JUnit Jupiter Framework and the `JUnitSRV` Framework. Therefore, a special annotation is required that distinguishes JUnit Jupiter test cases from JUnitSRV test cases so that when a test case is considered by one framework, the other ignores it. For this purpose, the `SRVTestEngine` provides the an-

```
class OffSiteATMTest {
  private static final String notificationServiceQName = "ATMExample/NotificationService";
  private static final Event calledSend = called(notificationServiceQName, "send");
  private static final Monitor sendCalledOnce = TesslaMonitor
    .forSpec("ATMExample/spec.tessla")
    .inputStream("send", calledSend)
    .failOnEvent("error")
    .reportOn("error_message");

  @Monitors({"sendCalledOnce"})
  @MonitoredTest
  void testDepositSRV() {
    // Setup Phase
    NotificationService service = new SMSNotificationService();
    Account account = new AccountStub();
    OffSiteATM atm = new OffSiteATM(service);
    // Exercise Phase
    atm.deposit(1000, account);
    // Verification Phase
    assertEquals(1000, account.getBalanceInCents());
  }
}
```

```
-- TeSSLa specification
in send: Events[Unit]
def cnt: Events[Int] := count(send)
def condition: Events[Bool] := cnt > 1
def error: Events[Unit] := if condition then ()
def error_message: Events[String] := if condition then "Method 'send' called more than 0"
out *
```

Listing 4.17: A test class with JUnitSRV test cases. The TeSSLa monitor created with the `calledSend` event is declared and attached to the `OffSiteATMTest.testDepositSRV` test case using the `@Monitors` and `@MonitoredTest` annotations. The monitor uses the specification displayed below the test class. It ensures that the NotificationService.send method is called only once. If this restriction is violated, the monitor fails with the message provided by the `error_message` stream.

notation `@MonitoredTest`. This annotation is also associated with an extension that prevents test cases from being executed by JUnit Jupiter or JUnitSRV when annotated with both `@Test` and `@MonitoredTest`. Listing 4.17 contains a complete test class containing a single test case executed by the JUnitSRV framework and ignored by the JUnit Jupiter framework. The listing also contains a TeSSLa specification that defines the behavior of the monitor `sendCalledOnce`.

4.5 TeSSLa-Based Mock Objects

One of the main features of the JUnitSRV framework is the mocking of interfaces and classes. In particular, the JUnitSRV framework allows mocking of classes, abstract classes, and interfaces at runtime and allows these mocks to connect to monitor streams. In contrast to other mocking frameworks like Mockito [Moc19] the mock behavior is controlled by so-called `InvocationHandlers`. These handlers are responsible for calculating return values for methods based on their arguments. With this handler mechanism, mock objects are highly configurable and can be adapted to almost any application. To connect a mock to a stream of a

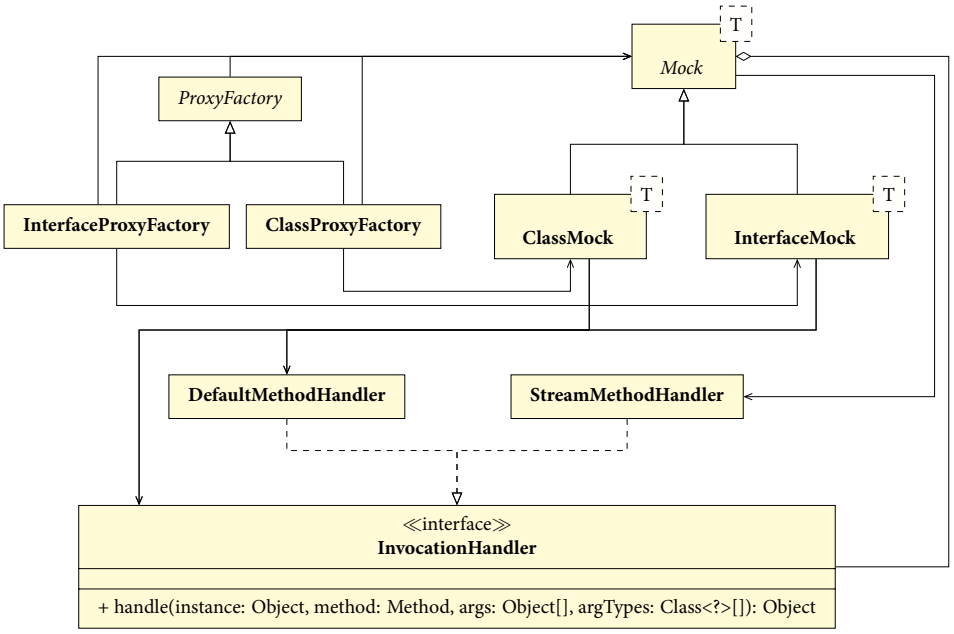


Figure 4.18: The class hierarchy for the mocking framework and all related classes. The factory classes are responsible for creating proxy classes for the classes or interfaces that should be mocked. The resulting proxy classes are wrapped in a generic `Mock` class. Mocked classes are represented as `ClassMock` objects, while mocked interfaces are represented as `InterfaceMock` objects. The implementations of the `InvocationHandler` class are responsible for describing the behavior of mock objects.

TeSSLa monitor, a special handler is used internally, which can also be used in conjunction with `InvocationHandlers`.

Figure 4.18 outlines the main classes that allow classes and interfaces to be mocked. One of the two main components is the `ProxyFactory` and its two subclasses `ClassProxyFactory` and `InterfaceProxyFactory`. These classes apply the *proxy pattern* and the *abstract factory pattern* [Gam+95], and are responsible for creating *proxy classes* for classes or interfaces to be mocked. The instantiation of classes and interfaces is fundamentally different, e.g. classes must be instantiated with the provided constructors and interfaces cannot be instantiated at all. When mocking classes or interfaces during testing, the use of constructors that require arguments for instantiation should be avoided. This makes it possible to create objects without having to create their dependencies beforehand. The problem with this approach is that classes may not provide standard constructors that can be used. In the context of Java, so-called *default constructors* are only added to a class if it does not have any explicitly declared constructors. However, this cannot be required of any class to be mocked. Furthermore, interfaces do not provide constructors and implementations for methods at all. To solve this problem, the instances `ClassProxyFactory` and `InterfaceProxyFactory` create classes at runtime using the Javassist library [Chi19]. Whenever these factory classes are prompted to create a proxy class, they generate the bytecode of classes that inherit or implement the specified class or interface. In the case of classes, the factory scans the class to be mocked for the default constructor. If this constructor does not exist, it is created in the proxy class.

Then, each method is overwritten to provide an implementation. To allow configuration of the behavior of the implemented methods, a `DefaultInvocationHandler` instance is attached to the proxy class, which is then called in each method. A proxy class as created by `ClassProxyFactory` for the class `OffSiteATM` from listing 1.2 is the `OffSiteATMMock0` class in listing 4.19.

```
public class OffSiteATMMock0 extends OffSiteATM {
  private InvocationHandler invocationHandler6459966968329040635;

  public OffSiteATMMock0() {}

  public OffSiteATMMock0(NotificationService service) {
    super(service);
  }

  public void withdraw(int cents, Account account) throws NotEnoughMoneyException {
    Object[] args = new Object[] {cents, account};
    Class<?>[] sig = new Class[] {int.class, Account.class};
    Method m = MethodUtils.getMethod(this.getClass(), "withdraw", sig);
    this.invocationHandler6459966968329040635.handle(this, m, args, sig);
  }

  public void deposit(int cents, Account account) {
    Object[] args = new Object[] {cents, account};
    Class<?>[] sig = new Class[] {int.class, Account.class};
    Method m = MethodUtils.getMethod(this.getClass(), "deposit", sig);
    this.invocationHandler6459966968329040635.handle(this, m, args, sig);
  }
}
```

Listing 4.19: A proxy class for the class OffSiteATM. The name of the proxy class is generated by appending the term ***Mock*** followed by the number of already created proxy classes to the actual name of the class or interface to be mocked. A proxy class needs a handler that defines its behavior for method calls. The handler name is generated with a random hash to prevent instance variable shadowing with the instance variables of the parent class. Each method of the parent class or the implemented interface is implemented or overwritten, so that the call is passed to the `InvocationHandler` instance instead of being calculated directly in the method.

As shown in listing 4.19, a default constructor is added to the class. This constructor is not available in the `OffSiteATM` class. Then, an instance variable `InvocationHandler` is added. The name of this variable is formed by the word *invocationHandler* followed by a random hash value to prevent *variable shadowing* with the instance variables of the base class. Within all overridden methods, this handler is then called with context information such as the current method, parameters and their types, and the instance on which the method was originally called. Proxy classes for interfaces are created analogously, with the difference that the respective methods are not overwritten but only implemented there and the proxy class implements the interface instead of extending it. The class `MethodUtils` is an auxiliary class that is provided by the JUnitSRV framework and returns an `Method` instance for the specified class, name and parameter types, if the corresponding method exists.

```
class AccountMockTest {
  @MonitoredTest
  void test() {
    Mock<Account> accountProxy = MockFactory.mock(Account.class);
    accountProxy.setHandler((instance, method, args, sig) -> {
      switch (method.getName()) {
        case "getBalanceInCents": return 10000;
        case "getName": return "John Doe";
        case "getEMail": return "john.doe@email.com";
        case "getPhoneNumber": return "12345";
        default: return DefaultValue.of(method.getReturnType());
      }
    });
    Account accountMock = accountProxy.newInstance();
    assertEquals(10000, accountMock.getBalanceInCents());
    assertEquals("John Doe", accountMock.getName());
    assertEquals("john.doe@email.com", accountMock.getEMail());
    assertEquals("12345", accountMock.getPhoneNumber());
  }
}
```

Listing 4.20: An example of how to create a mock object for the `Account` class. The method of the class `MockFactory.mock` passes the request either to the `ClassMockFactory` or to the `InterfaceMockFactory`, based on the argument obtained. The proxy class created by the factory is wrapped by a `Mock` instance. A custom handler is then attached to the proxy class. Finally, the mock object is created from the proxy class. The Assert statements after instantiation check whether the handler returns the correct return values for all methods.

After Javassist has created these proxy classes, they are passed to the JVM and can then be instantiated. Before instances of these proxy classes are created, they can be configured, since the associated handler is not initialized by the initializer or constructor. The `ClassProxyFactory` and `InterfaceProxyFactory` objects create instances of the `ClassMock` and `InterfaceMock` classes, respectively. Instances of `ClassMock` or `InterfaceMock` can instantiate proxy classes via their method `Mock.newInstance`. This and the `Mock.setHandler` method allows the proxy objects to be configured to use custom `InvocationHandlers`. If no custom `InvocationHandler` is passed to the `Mock`, it uses an instance of the `DefaultInvocationHandler` class. If a `Mock` object is connected to a `TesslaMonitor`, the `StreamMethodHandler` is used internally instead. Figure 4.20 illustrates the creation of a proxy and its corresponding mock object for the `Account` class from listing 1.1. The class `MockFactory` decides via the method `MockFactory.mock` which factory should create the proxy class. Since `Account` is an interface, the `InterfaceProxyFactory` is used to create the proxy class and the corresponding `Mock<Account>` object. A custom handler is then attached to the proxy class, returning a value based on the current method. Finally, the `Mock<Account>` object creates an instance of the proxy class that is assigned to the `accountMock` variable of type `Account`. This `Account` instance can then be used as a normal `Account` instance thanks to the polymorphism, although it is actually the proxy class. The assertions at the end of the test case check whether the attached handler returns the correct values. To use the mocking feature of the JUnitSRV framework, the Class Loader, which also performs the instrumentation, is required. For this reason the call of the `MockFactory.mock` method only works in JUnitSRV test cases, i.e. in test cases annotated with `@MonitoredTest`. However, no moni-

```
class AccountMockTest {
  private static String accountQName = "ATMExample/Account";
  private static Event calledSetBalance = called(accountQName, "setBalance");
  private static Monitor mockMonitor = TesslaMonitor.forSpec("spec.tessla")
      .inputStream("balance_stream", intEventsFromArg(1), calledSetBalance);

  @Monitors({"mockMonitor"})
  @MonitoredTest
  void test() {
    // Setup phase
    Mock<Account> accountProxy = MockFactory.mock(Account.class);
    accountProxy.setHandler((instance, method, args, sig) -> {
      switch (method.getName()) {
        case "getName": return "John Doe";
        case "getEMail": return "john.doe@email.com";
        case "getPhoneNumber": return "12345";
        default: return DefaultValue.of(method.getReturnType());
      }
    });
    accountProxy.connectToStream("getBalanceInCents", "balance_stream", "mockMonitor");
    Account accountMock = accountProxy.newInstance();
    // Exercise Phase
    accountMock.setBalance(1000);
    // Verification Phase
    assertEquals(1000, accountMock.getBalanceInCents);
    assertEquals("John Doe", accountMock.getName())
  }
}

-- TeSSLa specification
in balance_stream: Events[Int]
out *
```

Listing 4.21: An example of the declaration of a mock object which is controlled by both a monitor output stream and an `InvocationHandler`. The TeSSLa specification for the monitor is specified below the class and takes only one input stream `balance_stream`, which contains the values passed to the `Account.setBalance` method. These will also be output as an output stream. The mock object now connects the `Account.getBalanceInCents` method with this output stream to generate return values. The other methods get their return values from the attached `InvocationHandler`.

tor has to be attached to such a test case. If no monitor is attached to such a test case, it is executed as a conventional test case.

Besides an user-defined `InvocationHandler`, a `Mock` object can also be connected to an output stream of a `TesslaMonitor`. This function allows mocks to retrieve return values for methods from the last provided events of a particular stream. It allows the dynamic behavior of mocks based on the overall state of the system tracked by the TeSSLa specification. In addition, it is a more elegant way to configure proxy classes. Listing 4.21 shows a mock object for the class `Account`. The `Account.getBalanceInCents` method uses as its return values the last seen value of the `balance_stream` provided by the `mockMonitor`. The input streams for the underlying specification are the values passed to the `Account.setBalance` method. In addition, an user-defined `InvocationHandler` is also used, which generates the return values of the other methods. Thus, one part of the methods is provided with return values of the monitor attached to the test case in which the mock object lives, while the other part gets its values from an user-defined `InvocationHandler`. To ensure that the return values of

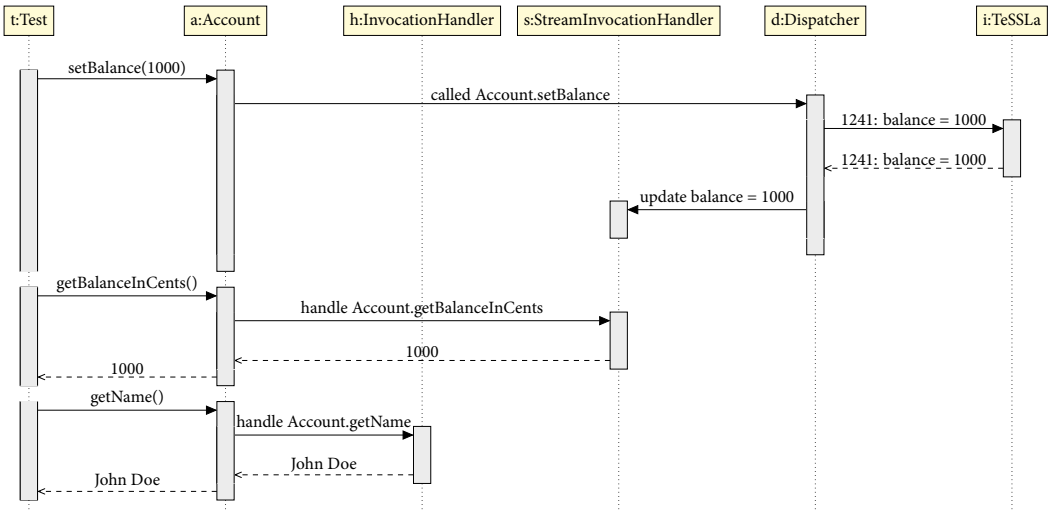


Figure 4.22: A visualization of the interactions between the components involved into mocking and instrumentation. The test case invokes the methods of the `Calculator` class. The `Calculator.add` method is instrumented and its invocation produces events that are consumed by the TeSSLa interpreter. The `Calculator.sum` method is connected to an output stream of the TeSSLa specification. Once the interpreter processed the `add` event it will update the `StreamInvocationHandler`. If the `Calculator.sum` method on instance `a` is invoked later it can access this value to return it. The `Calculator.sub` method is controlled by a custom `InvocationHandler`.

the monitor stream are always available, the `StreamMethodHandler`, which is used internally, must take care to remember the output stream values. The reason for this is the fact that an output stream can supply a value at a certain point in time, but the associated mock object does not use this value until much later.

Figure 4.22 is an UML sequence diagram that shows how the various components from the test case in listing 4.21 interact during the exercise phase. Once an instrumented method performs the action that was instrumented, the corresponding dispatcher is notified. The dispatcher instance then translates the events into stream events using the corresponding `StreamTransformer` objects of the respective `InputStream` instances. After the TeSSLa interpreter has completed the calculation for the input events, it will inform the listening dispatchers of any changes. These dispatchers can now inform all `StreamInvocationHandlers` about changes so that they can save the output stream values. If a mock object later calls a method associated with a stream, it can obtain the value previously calculated by the TeSSLa interpreter from the `StreamInvocationHandler`.

5 Case Study

This chapter presents three examples that use the JUnitSRV framework to apply RV and SRV in specific contexts. These three examples have been chosen to showcase a particular strength of the JUnitSRV framework in each one. First, the ATM example from chapter 1, which also served as a working sample throughout this thesis, is used to show how test cases are generally monitored. Then, the section 5.2 presents an example project based on the example presented in [DLT13] that uses both LTL and TeSSLa based monitors and shows how both types of monitors can synergize with each other. Finally, in the section 5.3 an application is monitored using mock objects. For each of these examples a TeSSLa specification and if necessary a LTL specification specifying the properties to be monitored is shown. In addition, the event and monitor declarations as well as the declaration of test cases are shown. Finally, a passing and a failing execution of one of the test cases is presented by means of diagrams.

5.1 The ATM Example

In this section, the application consisting of the components shown in listing 1.1 on page 3 is performed. In particular, the implementation of the class in listing 1.2 on page 5 is examined. For this purpose instances of the classes `Account` and `NotificationService` are replaced by stubs. While the `Account` stub can be checked by assertions during the verification phase, the TeSSLa specification checks the behavior of the `NotificationService` stub. Specifically, a `TeSSLaMonitor` is used to check whether the `Notification.send` method is called only once during each of the `ATM.deposit` and `ATM.withdraw` methods. Besides the monitoring tests, a conventional JUnit Jupiter test is used to check if exceptions are thrown correctly.

Since the JUnit Platform considers all loaded test engines for test case execution, it is possible to use traditional JUnit 5 test cases and monitored test cases simultaneously. The specification used for the `TesslaMonitor` is shown in figure 5.1. The monitored implementation is given in listing 1.2 on page 5. The declaration of events and monitors is done in listing 5.2. Finally, the figures 5.4 and 5.5 show a passing and a failing test case execution from the `testWithdraw` test case shown in listing 5.3.

The TeSSLa specification first defines input streams for `send`, `deposit` and `withdraw` that correspond to the method `NotificationService.send`, `ATM.deposit` und `ATM.withdraw`. Then, the two streams `deposit` and `withdraw` are merged into one stream to indicate when the counter for the `send` stream should be reset. As long as no event occurs on the merged stream, the events of the `send` stream are counted. The error condition is that if this counter is ever greater than one, the method `NotificationService.send` is invoked multiple times in one of the `ATM.deposit` or `ATM.withdraw` methods.

```
-- A macro that counts the events on the stream of the first argument and whose counter is
-- reset when events that appear on the stream of the second argument.
def count_until_reset[A,B](values: Events[A], reset: Events[B]) := count where {
  def count: Events[Int] := default(
    if default(time(reset) > time(values), false)       then 0
    else if default(time(reset) == time(values), false) then 1
    else last(count, values) + 1, 0)}

-- The declaration of input streams that are made up of instrumentation events in the
-- application code.
in send: Events[Unit]
in deposit: Events[Int]
in withdraw: Events[Int]

-- All intermediate streams that are computed using the input streams and the defined macro
def deposit_withdraw := merge(deposit, withdraw)
def cnt_send := count_until_reset(send, deposit_withdraw)

-- The definition of the error condition as well as the message shown to the developer in
-- case the condition is false
def condition := cnt_send > 1
def error := if condition then ()
def error_message := if condition && default(time(deposit) > time(withdraw), false)
    then "Too many send calls during ATM.deposit"
    else if condition && default(time(withdraw) > time(deposit), false)
    then "Too many send calls during ATM.withdraw"

-- the definition of all output streams.
out error
out error_message
```

Listing 5.1: The TeSSLa specification checks whether the `NotificationService.send` method is called more than once in the `ATM.deposit` or `ATM.withdraw` methods. The `count_until_reset` macro counts the events on the stream of the first argument and resets the counter if an event has occurred on the stream of the second argument. The three input streams represent `NotificationService.send`, `ATM.deposit` and `ATM.withdraw` events. If the method `NotificationService.send` is called more than once in `ATM.deposit` or `ATM.withdraw`, a unit event is provided on the `error` stream together with a message on the `error_message` stream.

To use the TeSSLa specification in the test class, the monitor must be defined. The following assumes that the tested class `OffSiteATM` is declared in the `ATMExample` package. The declaration of the events and monitors is straightforward and is done as in the previous chapters.

```
  private static final String serviceQName = "ATMExample/NotificationService";
  private static final String atmQName = "ATMExample/ATM";
  private static final Event calledSend = called(notificationServiceQName, "send");
  private static final Event calledDeposit = called(atmQName, "deposit");
  private static final Event calledWithdraw = called(atmQName, "withdraw");
  private static final Monitor sendCalledOnce = TesslaMonitor.forSpec("spec.tessla")
      .inputStream("send", calledSend)
      .inputStream("deposit", intEventsFromArg(1), calledDeposit)
      .inputStream("withdraw", intEventsFromArg(1), calledWithdraw)
      .failOnEvent("error")
      .reportOn("error_message");
```

Listing 5.2: The declaration of the events, generated by the instrumented application code, and the monitor for the TeSSLa specification in listing 5.1. The events `calledSend`, `calledDeposit` and `calledWithdraw` are created by the methods `NotificationService.send`, `ATM.deposit` and `ATM.withdraw` and then translated by the monitor `sendCalledOnce` into the streams `send`, `deposit` and `withdraw`.

The monitor declaration for the specification in listing 5.1 is shown in listing 5.2. First, the fully qualified names for both interfaces `ATM` and `NotificationService` are stored in variables. Then, all associated method call events are created and passed to the monitor as configuration parameters. The monitor fails if the specification provides a unit event on the `error` stream. If this is the case, an error message is output, provided by the `error_message` stream. Finally, the test cases are also of interest.

```
@Test
void testExceptions() {
  assertThrows(IllegalArgumentException.class, () -> atm.deposit(-100, null));
  assertThrows(IllegalArgumentException.class, () -> atm.deposit(-100, account));
  assertThrows(IllegalArgumentException.class, () -> atm.deposit(100, null));
  assertThrows(IllegalArgumentException.class, () -> atm.withdraw(-100, null));
  assertThrows(IllegalArgumentException.class, () -> atm.withdraw(-100, account));
  assertThrows(IllegalArgumentException.class, () -> atm.withdraw(100, null));
  atm.deposit(100, account);
  assertThrows(NotEnoughMoneyException.class, () -> atm.withdraw(101, account));
}

@Monitors({"sendCalledOnce"})
@MonitoredTest
void testDeposit() {
  atm.deposit(100, account);
  assertEquals(100, account.getBalanceInCents());
}

@Monitors({"sendCalledOnce"})
@MonitoredTest
void testWithdraw() throws NotEnoughMoneyException {
  atm.deposit(100, account);
  atm.withdraw(50, account);
  assertEquals(50, account.getBalanceInCents());
}
```

Listing 5.3: The three test cases that are executed to check the `OffSiteATM` class. The first test case is a JUnit Jupiter test case which checks the exceptions thrown within `ATM.deposit` and `ATM.withdraw`. The last two test cases are JUnitSRV test cases that monitor the behavior of the SUT according to the TeSSLa specification and monitor declaration during execution.

The first test case is executed by the JUnit Jupiter test engine due to the annotation `@Test`. It should test whether exceptions are thrown if the arguments passed to `ATM.deposit` and `ATM.withdraw` are invalid. The last two test cases confirm that the methods `ATM.deposit` and `ATM.withdraw` work as intended by asserting against a `Account` stub. While exercising the SUT, the monitor `sendCalledOnce` checks whether the method `NotificationService.send` is called more than once in `ATM.deposit` and in `ATM.withdraw`. The execution of the monitored test case `testWithdraw` is visualized in figure 5.4.

As the visualization and test summary in figure 5.4 shows, all test cases are passed successfully. In the implementation, each call to `ATM.deposit` and `ATM.withdraw` leads to a single call to `NotificationService.send`. Therefore, the `condition` stream does not return a `true` value. As a result, no event is provided on the `error` stream. If the SUT would call

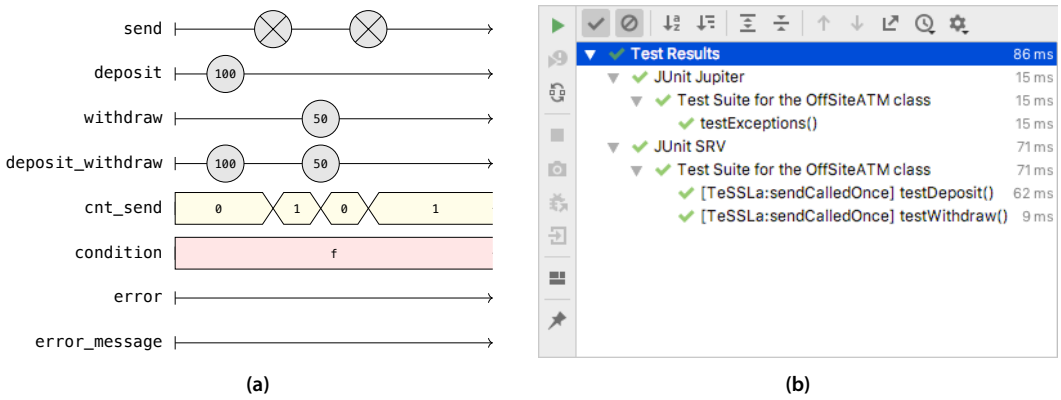


Figure 5.4: A visualization of the streams that are filled with events by the `sendCalledOnce` monitor during the execution of the `testWithdraw` test case is shown in figure 5.4a. Figure 5.4b shows the corresponding test case execution summary within the IntelliJ IDE. As shown in the figure, both the JUnit Jupiter and the JUnitSRV test cases were executed successfully.

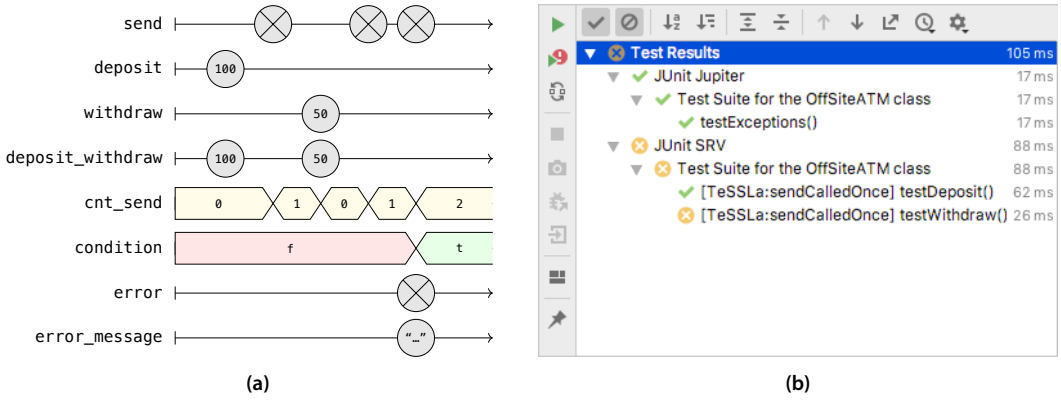


Figure 5.5: A visualization that corresponds to the one in figure 5.4 with the difference that in the `ATM.withdraw` method the `NotificationService.send` method is now called twice. As a result, the events that appear on each stream change so that finally a unit event on the `error` stream and a message on the `error_message` stream appears.

the `NotificationService.send` method twice, the corresponding test case would fail. This scenario is visualized in figure 5.5.

5.2 The Data Service Example

This section uses the example `DataService` from listing 4.2 on the page 39, which originally comes from [DLT13]. In this case it is examined whether the method calls on instances of the class `DataService` correspond to a predefined specification. Unlike the example in the previous section, the following example focuses on combining the capabilities of LTL and TeSSLa monitors. Both monitors have their own strengths and weaknesses. While LTL is useful for defining properties over an execution path, especially the system state in the future, the specifications of TeSSLa can focus on quantitative measurements. In the following, the LTL monitor checks whether the method calls on the SUT are in the correct order, while

the TeSSLa monitor checks whether the relative execution time of certain methods is within predefined limits. Defining the LTL property in TeSSLa would result in a relatively complex specification, so this property is checked by a LTL monitor. On the other hand, it is impossible to define the TeSSLa measurement property in LTL. Therefore, this example shows the synergy of LTL and TeSSLa monitors when used simultaneously.

```
-- A macro that sums up the events on the arguments stream.
def sum(values: Events[Int]) := s where {
  def s: Events[Int] := merge(last(s, values) + values, 0)}

-- A macro that sums up the time elapsed between an event that occurred on the stream of the
-- first argument and the subsequent event that occurred on the stream of the second
-- argument.
def sum_time_difference[A, B](a: Events[A], b: Events[B]) := std where {
  def dif := time(a) - time(b)
  def std := sum(if dif > 0 then dif else 0) / 1000000}

-- A macro that calculates the relative proportion of the event value on one stream from the
-- event value on the other stream.
def rel(event_time: Events[Int], overall_time: Events[Int]) := r where {
  def r := if overall_time == 0 then 0 else 100 * event_time / overall_time}

-- The declaration of input streams that are made up of instrumentation events in the
-- application code.
in modify: Events[Unit]       -- modify invoked
in modified: Events[Unit]     -- modify returned
in commit: Events[Unit]       -- commit invoked
in committed: Events[Unit]    -- commit returned
in disconnect: Events[Unit]   -- disconnect invoked
in disconnected: Events[Unit] -- disconnect returned

-- First, calculate the elapsed time during the modify, commit and disconnect operations.
def modify_time := sum_time_difference(modified, modify)
def commit_time := sum_time_difference(committed, commit)
def disconnect_time := sum_time_difference(disconnected, disconnect)

-- Then, calculate the overall elapsed system time for these operations.
def all_time := modify_time + commit_time + disconnect_time

-- Then, calculate the relative values of elapsed time during these operations.
def modify_rel_t := rel(modify_time, all_time)
def commit_rel_t := rel(commit_time, all_time)
def disconnect_rel_t := rel(disconnect_time, all_time)

-- Finally, define the error condition and throw an error unit along with an error message
-- in case the condition is violated.
def condition := disconnect_rel_t > commit_rel_t || disconnect_rel_t > modify_rel_t
def error := if condition then ()
def error_message := if condition then "disconnect took longer than commit and modify"

-- Consider all input and intermediate streams as output streams.
out *
```

Listing 5.6: The TeSSLa specification that calculates how much relative execution time is required within each method `DataService.modify`, `DataService.commit` and `DataService.disconnect`. The time is calculated using the two macros `sum` and `sum_time_difference`. It then checks whether the method `DataService.disconnect` takes more time than `DataService.commit` or `DataService.modify`. If this is the case, both a unit event on the `error` stream and a message on the `error_message` stream are provided.

Listing 5.6 contains the specification in which the maximum relative time portion of the `DataService.disconnect` method is defined. First, the specification receives called and returned events on the streams for the methods `DataService.modify`, `DataService.commit`, and `DataService.disconnect`. The elapsed time between the call event and the return events of the respective method is then calculated and added up. Then, the elapsed time of all methods is added to the total elapsed time. Finally, the total elapsed time is used to calculate the percentage of each method. If the relative time portion of the `DataService.disconnect` method is greater than for the `DataService.modify` or `DataService.commit` methods, the error condition is met and a unit event is provided on the `error` stream and a message on the `error_message` stream.

First the LTL monitor and the events necessary for its declaration are declared. Figure 5.7 shows the `commitBeforeClose` monitor that ensures that `DataService.commit` is called at least once before a client calls `DataService.disconnect` if `DataService.modify` was called. The interface `DataService` is declared in the package `DataServiceExample`. The monitor and therefore the LTL formula used for the synthesis of this monitor is the same as in listing 4.4 on page 40.

```
private static final String dataServiceQname = "DataServiceExample/DataService";
private static final Event modify = called(dataServiceQname, "modifyData");
private static final Event committed = returned(dataServiceQname, "commit");
private static final Event disconnect = called(dataServiceQname, "disconnect");
private static final Monitor commitBeforeClose = new FLTL4Monitor(
    Always(implies(modify, Until(not(disconnect), committed))));
```

Listing 5.7: First, the fully qualified name of the class `DataService` is saved as a variable. Then, events for calling the methods `DataService.modify` and `DataService.disconnect` are defined. An event is also defined for returning from the `DataService.commit` method. Finally, these events are used to construct a monitor to which a LTL formula is passed that uses the declared events.

Additional events are required to declare the TeSSLa monitor. Listing 5.8 contains the missing event declarations for the `time` monitor. The monitor is then declared using the events and the corresponding input streams are configured immediately. In addition, the monitor is configured to fail if an event is present in the `error` stream. If this is the case, the message is reported to the developer in the `error_message` stream.

```
private static final Event modified = returned(dataServiceQname, "modifyData");
private static final Event commit = called(dataServiceQname, "commit");
private static final Event disconnected = returned(dataServiceQname, "disconnect");
private static final Monitor time = TesslaMonitor.forSpec("spec.tessla")
    .inputStream("modify", modify).inputStream("modified", modified)
    .inputStream("commit", commit).inputStream("committed", committed)
    .inputStream("disconnect", disconnect).inputStream("disconnected", disconnected)
    .failOnEvent("error")
    .reportOn("error_message");
```

Listing 5.8: First the missing events for the `time` monitor are declared. Then, the declaration of the `time` monitor is done with the help of the events declared before and the events from listing 5.7. In addition, the monitor is configured to fail if an event is present in the `error` stream. If this is the case, the message is reported to the developer in the `error_message` stream.

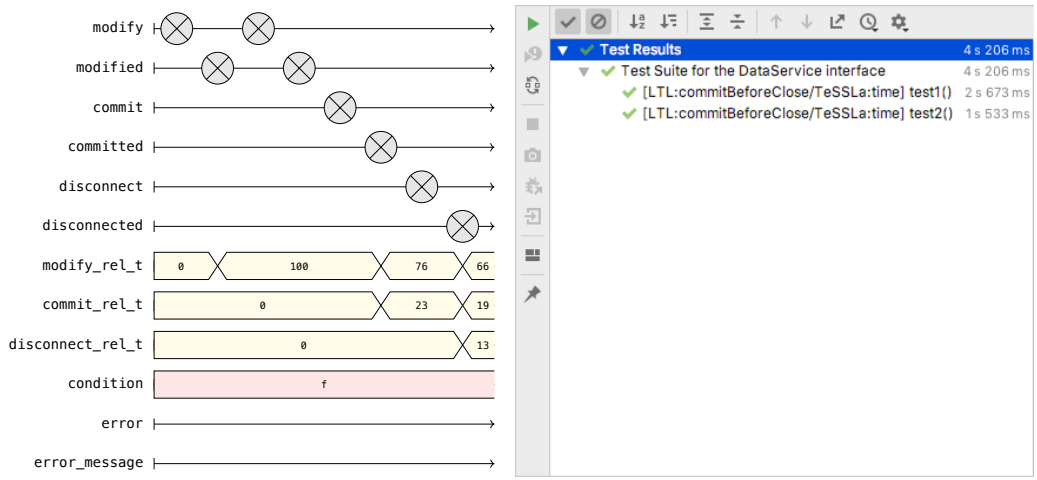


Figure 5.9: The visualization of the execution of the test case `test2`. Figure 5.9a shows the visualization of the TeSSLa streams. The corresponding representation of the results in the IntelliJ IDE is shown in Figure 5.9b. Both test cases pass because neither the TeSSLa monitor nor the LTL monitor fails.

Finally, test cases are required for a `DataClient`. The `DataClient` is the SUT in these test cases. The test cases aim to monitor the communication between the `DataClient` and the `DataService`. Only JUnitSRV test cases are considered here. In addition to such test cases the SUT would have to be checked by conventional JUnit Jupiter test cases.

```
@Monitors({"commitBeforeClose", "time"})
@MonitoredTest
void test1() {
  client.authenticate("daniel");
  client.addPatient("Mr. Smith");
  client.switchToUser("ruth");
  client.getPatientFile("miller-2143-1");
  client.setPhone("miller-2143-1", "012345678");
  client.exit();
}

@Monitors({"commitBeforeClose", "time"})
@MonitoredTest
void test2() {
  client.authenticate("daniel");
  client.addPatient("Mr. Smith");
  client.addPatient("Mr. Smith");
  client.exit();
}
```

Listing 5.11: Two test cases, which cause a communication between the `DataClient` and `DataService` by means of different method calls on an instance of the `DataClient` class. This communication is monitored by the two attached monitors `commitBeforeClose` and `time`.

The visualization in Figure 5.9 shows the execution of the test case `test2`. Both monitors `commitBeforeClose` and `time` check the execution of the test case. Since the implementation of the `DataClient` is correct, the TeSSLa monitor and the LTL monitor do not fail and both test cases pass. By making the `DataService.disconnect` method more time-consuming, the `time` monitor will fail because the time spent executing `DataService.disconnect` is

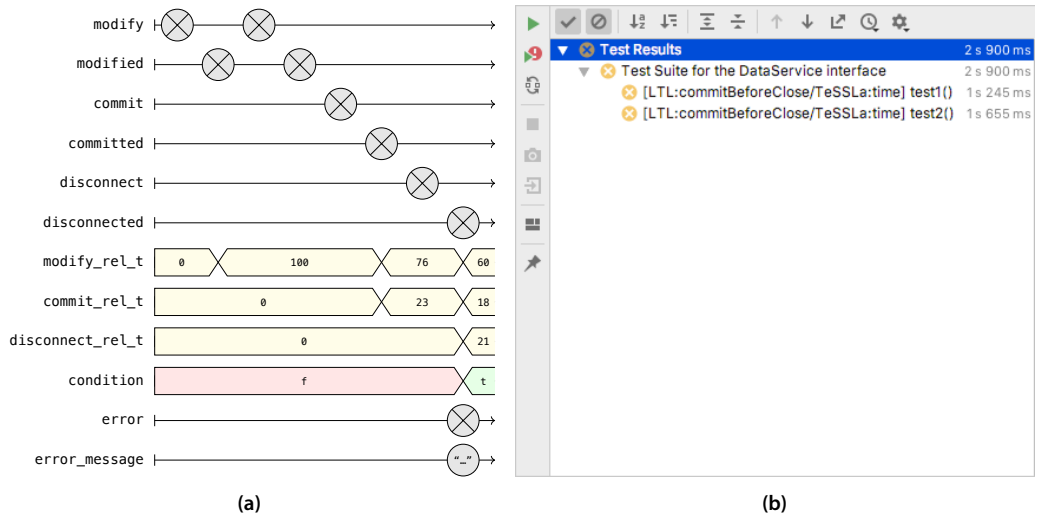


Figure 5.10: A variant of the test case execution `test2` where the `DataService.disconnect` method consumes more time than the `DataService.commit` method. As a result, the `time` monitor will fail after receiving the `disconnected` event. Figure 5.17a shows the corresponding visualization of the corresponding streams and figure 5.17b shows the test case summary from the IntelliJ IDE.

more than the elapsed time in `DataService.commit`. However, since the method call order remains the same, the monitor `commitBeforeClose` does not fail. This case is shown in figure 5.10.

5.3 The Autonomous Robot Example

In this section the mocking feature of the JUnitSRV framework is demonstrated by an example. The example consists of an autonomous robot vehicle that uses a bumper as a sensor to determine whether it has collided with an obstacle. It is assumed that the control of the vehicle in such a case will move away from the obstacle by driving in the opposite direction. If the bumper is still pressed after a certain time, the vehicle detects a defect and stops immediately. Using the JUnitSRV framework, the bumper is replaced by a mock object controlled by a TeSSLa specification. The scenario takes place within a simulation in a test case. In the following, the interfaces of the components are shown first. Then, as in the previous examples, the TeSSLa specification is created and explained. The monitor that monitors the application during test case execution is subsequently created. The monitor is an important component in this case, as the mock object in the test cases is connected to this monitor to be controlled by the TeSSLa specification. These test cases are specified at the end. Finally, as in the previous examples, an execution of the test case is visualized.

Listing 5.12 shows the modules used within an autonomous robot vehicle. The `Clock` class is used to determine when certain actions are performed within a vehicle. For example, a `Clock` is used to decide whether to still drive forward or backward. Also the check of the sensors is triggered by `Clock.tick` events. A `Component` represents a module in the system which

```
public interface Clock {
  void tick(int time);
}

public interface Component {
  int consumedTimeInMilliseconds();
}

public interface AutonomousVehicle extends Component, Clock {
  void start();
  boolean isDriving();
}

public interface Bumper extends Component {
  boolean isPressed();
}

public interface Motor extends Component {
  void moveForward();
  void moveBackward();
  void stop();
  int getDirection();
}
```

Listing 5.12: The example application consists of several modules. At the top level are the modules `Clock` and `Component`. The `Clock` class ensures that actions are triggered within the system at a certain point in time, while the `Component` class corresponds to a component within the system that can perform actions. The class `AutonomousVehicle` is the entire robot vehicle as such. The `Bumper` class represents the sensor used by the vehicle to determine if it has collided with an obstacle. Finally, the class `Motor` ensures that the vehicle can be moved.

can perform actions. The method `Component.consumedTimeInMilliseconds` supplies the time that this component needs for its action. The class `AutonomousVehicle` represents the complete vehicle as such. The only sensor class in this application is the `Bumper` class which checks if the vehicle has collided with an obstacle. Finally, the `motor` ensures that the vehicle can move.

```
-- The declaration of input events supplied by the monitor
in t: Events[Int]
in stopped: Events[Bool]

-- The output stream that are used by the mock object to determine its behavior
def bumper_pressed := t >= 150 && t <= 750

-- The error condition that makes the monitor fail
def error := if stopped then ()
def error_message := if stopped then "The motor should not stop!"

-- The definition of output streams
out *
```

Listing 5.13: The TeSSLa specification gets three information from the monitor. The time of the simulation, the call event of the method `Bumper.consumedTimeInMilliseconds` and whether the vehicle was stopped due to an error. The specification provides two output streams that control the mock object. The output stream `bumperReactionTime` gives the bumper the time it takes to perform its action, while the stream `bumperPressed` indicates whether the bumper was pressed or not. Finally, the specification provides an output stream `error` that provides an event when the vehicle is stopped due to an error. A message about the error is provided on the stream `error_message`.

Listing 5.13 shows the TeSSLa specification used to monitor the application code and control the mock object. The input streams are `t`, the simulation time of the test case and the `stopped` stream, which contains a truth value indicating whether the `AutonomousVehicle` stopped due to a defect. The specification checks if the simulation time is within a certain interval. If this is the case, a `true` value is provided on the output stream `bumper_pressed`, which triggers the `Bumper`, otherwise it provides `false`. Depending on how the interval is defined in this specification, it can be checked whether the vehicle stops automatically if the maximum duration for a pressed `Bumper` is exceeded. Finally, the specification generates a unit event on the `error` stream and a corresponding message on the `error_message` stream when the vehicle has stopped itself. An event on the `error` stream signals the monitor to fail.

```
private static final String componentQName = "RobotExample/Bumper";
private static final String clockQName = "RobotExample/Clock";
private static final String motorQName = "RobotExample/Motor";
private static final Event calledTick = called(clockQName, "tick");
private static final Event calledConsumedTime = called(componentQName,
    "consumedTimeInMilliseconds");
private static final Event calledStop = called(motorQName, "stop");
private static final Monitor vehicleBehavior = TesslaMonitor.forSpec("spec.tessla")
    .inputStream("bumperProcessTime", calledConsumedTime)
    .inputStream("t", intEventsFromArg(1), calledTick)
    .inputStream("stopped", boolLiteral(true), calledStop)
    .failOnEvent("error")
    .reportOn("error_message");
```

Listing 5.14: The monitor declaration for an autonomous robot vehicle. First the events for the instrumentation of the methods `Bumper.consumedTimeInMilliseconds`, `Clock.tick` and `Motor.stop` are provided. The monitor is then created using these events. The three events are used to create input streams that are passed to the specification. The monitor is additionally configured to fail at an event on the `error` stream using the information from the `error_message` stream message as a reason.

Listing 5.14 shows the declaration of the monitor for the TeSSLa specification from listing 5.13. The declaration is only slightly different from the previous use cases. First, the fully qualified names of the components to be instrumented are declared. In this case the classes are `Clock`, `Bumper` and `Montor`. The monitor is then constructed with the help of these events. The difference lies in the declaration of the two input streams `t` and `stopped`. For the input stream `t`, the first parameter of the method `Clock.tick`, which receives the simulation time, is used as the stream event, whereas for the `stopped` stream, when `Motor.stop` is called, `true` is always output on the stream. Therefore, calling the `Motor.stop` method will cause the monitor to fail.

Next, a test case is required that allows the monitoring of the application code. In contrast to the previous use cases, however, a simulation and a mock object are now used here, since the behavior of the system as such is to be assessed and manipulated by the TeSSLa specification. Listing 5.15 is a test case which uses the `simulate` method to run a simulation of the application. The test case executes the simulation for one second in the model time. The important thing in the simulation method is creating and using the mock object. The

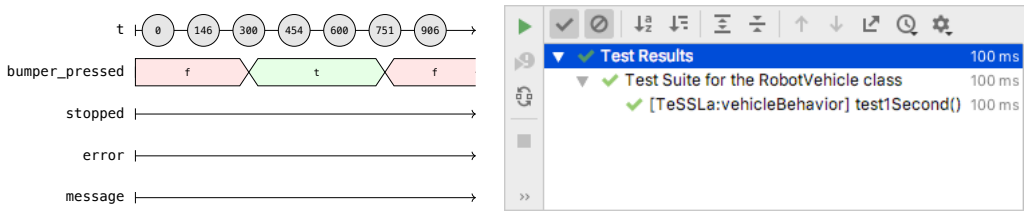
mock object `bumper` is created using the `MockFactory.mock` method, then configured using an `InvocationHandler` and finally connected to the monitor `vehicleBehavior`. If the method `Bumper.consumedTimeInMilliseconds` is called, an user-defined handler ensures that a value of about 50 milliseconds, with a deviation of 5 milliseconds in both directions, is returned. This random part ensures that the sensors in the simulation are not checked regularly, but with a slight irregularity.

```
@Monitors({"vehicleBehavior"})
@MonitoredTest
void test1Second() {
  simulate(1);
}

private void simulate(int tInSeconds) {
  // Setup
  Random r = new Random();
  InvocationHandler handler = (instance, method, args, argTypes) ->
      method.getName().equals("consumedTimeInMilliseconds")
          ? 50 + r.nextInt(11) - 5
          : defaultValue(method.getReturnType());
  Bumper bumper = MockFactory.mock(Bumper.class)
      .connectToStream("isPressed", "bumper_pressed", "vehicleBehavior")
      .newInstance(handler);
  AutonomousVehicle vehicle = new RobotVehicle(bumper, new ServoMotor());
  int maxTime = tInSeconds * 1000;

  // Simulation
  vehicle.start();
  while (time <= maxTime) {
    time += vehicle.consumedTimeInMilliseconds();
    vehicle.tick(time);
  }
}
```

Listing 5.15: The test case for the robot example including the simulation method used. The logic of the system execution takes place in the simulation method. The test case is responsible for starting the simulation. In addition, the previously declared monitor is attached to the test method.

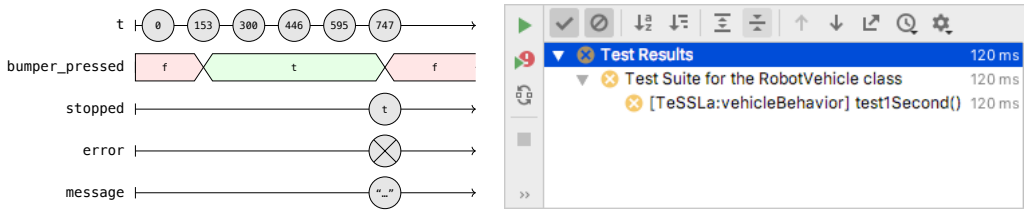
The simulation method recreates the required units for each simulation. Then, the current time in the simulation is stored in a variable. This variable is incrementally increased until the time for the simulation has elapsed. The difference between the simulation steps in the model time results from the time required during one step by the `AutonomousVehicle`. In order to obtain this time, the required times of all individual components, such as the motor and sensor, are summed up. After this time elapsed the method `Clock.tick` on the instance of `AutonomousVehicle` is called to cause a re-evaluation of all sensors and possible reaction to the measurement.

The figures 5.16 and 5.17 show a successful and a failed execution of `test1Second` method. The vehicle simulated in this test case initiates an emergency stop after `500` milliseconds. In figure 5.16 the `Bumper` is only pressed for a period of `451` milliseconds, so the vehicle does not initiate an emergency stop. On the other hand, in figure 5.17 there is a test case execution in



(a) (b)

Figure 5.16: The execution of the test case `test1Second`. On the `t` stream the specification gets the simulation time. At the event with the value `300` the value `true` is provided for the mock object on the `bumper_pressend` stream, so the bumper is pressed. The event with the value `751` ensures that the bumper is in accordance with the specification no longer pressed. The difference of these times is `451` and is thus below the limit of the vehicle, which is why it does not perform an emergency stop and the monitor does not fail.



(a) (b)

Figure 5.17: With this execution of the test case `test1Second`, the components finish processing their tasks faster in the respective steps, which leads to the `Bumper` being pressed faster. Since the specification does not release the bumper when the event with the value `747` appears, the `Bumper` is now pressed `594` milliseconds. Thus, the vehicle initiates an emergency stop, which the monitor recognizes and consequently fails.

which the `Bumper` is pressed for `594` milliseconds. Thus, the vehicle detects a potential defect and initiates an emergency stop. The monitor recognizes this and fails. Since the monitor fails and therefore also the test case, the simulation is not continued despite remaining time. Depending on how the interval is selected in the TeSSLa specification, the mocked `Bumper` object can change from the pressed state to the unpressed state and vice versa at certain times.

6 Discussion and Conclusion

The purpose of this thesis was to combine JUnit, the jUnit^{RV} library, TeSSLa and a mocking framework tailored to these tools into a single tool. The concepts introduced in chapter 3 were used as a template and were applied in the use cases presented in chapter 5 in a real context. As these use cases illustrate, the JUnitSRV framework offers all features presented in the concepts. It enables the instrumentation of the application code without having to actively change it. It is also possible to use LTL and TeSSLa based monitors simultaneously when executing unit tests. Because the JUnitSRV framework is based on the JUnit Platform, not only monitored test cases can be used, but it is also possible to declare and execute conventional JUnit test cases alongside monitored test cases simultaneously within the same test suites. This allows developers to integrate JUnitSRV into existing software projects with little effort. One of the main features that distinguishes JUnitSRV from all existing testing frameworks is the ability to replace instances of classes and interfaces in the application code with mock objects that can be controlled by a monitor. This type of mocking allows additional aspects such as self-healing properties of a system to be investigated more elegantly and dynamically while being exercised in a test case. The declaration of the monitors is highly configurable as developers can control the transformation of instrumentation events into streams as well as evaluate the output streams themselves. In addition to the extensive configuration options, the JUnitSRV framework also offers predefined auxiliary methods that allow a simplified monitor declaration for common use cases. Like the monitors, the mock objects can also be configured using two different approaches. On the one hand a user-defined call handler can be used and on the other hand a mock object can be controlled by monitors. It is possible that individual methods are controlled by a call handler, while other methods of the same mock object are controlled by different monitors.

Although the implementation meets the original requirements, there are still some improvements that can be made through additional development time. First, the `SRVTestEngine` offers only the basic functionality of a test engine. Test classes and test cases are recognized correctly and can be executed afterwards. However, apart from the test cases, only the methods for setup and teardown, i.e. methods that are annotated with `@BeforeAll`, `@BeforeEach`, `@AfterAll` and `@AfterEach`, are executed. More extensive mechanisms such as the Extension system of JUnit Jupiter are not implemented. Templated test cases and test case factories are also missing. To integrate these features either additional development time is necessary or the JUnit Jupiter test engine can be utilized to detect and execute test cases using the *adapter pattern* [Gam+95]. However, this option generates dependencies between JUnitSRV and JUnit Jupiter which have to be maintained regularly and requires a deep understanding of the JUnit Jupiter implementation. Another point that is currently not considered is the parallel execution of test cases. To enable a parallel execution of test cases, some aspects of the

jUnit^{RV} library and some components of the JUnitSRV framework have to be restructured or redesigned as this aspect has not been considered as a requirement. Regarding the mocking framework, the interface can be adapted to be more similar to the interface of the Mockito framework. This would make it easier for potential developers to transfer expertise in the use of the Mockito framework to the build-in JUnitSRV mocking framework.

List of Abbreviations

CLI *command line interface* 48

DIP *Dependency-Inversion Principle* 3

IDE *integrated development environment* 38

JVM *java virtual machine* 2

LTL *linear temporal logic* 13

OOP *object-oriented programming* 4

RV *runtime verification* 1

SPI *service provider interface* 29

SRV *stream runtime verification* 1

SUT *system under test* 4

TeSSLa *Temporal Stream-based Specification Language* 1

UML *Unified Modeling Language* 25

List of Listings

List of Figures

References

[BC10] Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development*. 1st. Springer Publishing Company, Incorporated, 2010.

[BS14] L. Bozzelli and C. Sánchez. *"Foundations of Boolean Stream Runtime Verification"*. In: *Runtime Verification*. Ed. by B. Bonakdarpour and S. A. Smolka. Cham: Springer International Publishing, 2014, pp. 64–79.

[CGP99] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[Chi19] S. Chiba. *Javasssist, Java bytecode engineering toolkit since 1999*. 2019. URL: `http://www.javassist.org` (visited on 06/06/2019).

[Con+18] L. Convent et al. *"TeSSLa: Temporal Stream-Based Specification Language"*. In: *Formal Methods: Foundations and Applications*. Ed. by T. Massoni and M. R. Mousavi. Cham: Springer International Publishing, 2018, pp. 144–162.

[DAn+05] B. D'Angelo et al. *"Lola: Runtime Monitoring of Synchronous Systems"*. In: *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*. Burlington, Vermont: IEEE Computer Society Press, June 2005, pp. 166–174.

[DLT13] N. Decker, M. Leucker, and D. Thoma. *"jUnit^{RV}–Adding Runtime Verification to jUnit"*. In: *NASA Formal Methods*. Ed. by G. Brat, N. Rungta, and A. Venet. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 459–464.

[Fow07] M. Fowler. *Mocks Aren't Stubs*. 2007. URL: `https://www.martinfowler.com/articles/mocksArentStubs.html` (visited on 06/06/2019).

[Gam+95] E. Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[Jia15] Z. Jiang. *"Load Testing Large-Scale Software Systems"*. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2015, pp. 955–956.

[JUn19a] JUnit Team. *Class Assertions*. 2019. URL: `https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html` (visited on 06/06/2019).

[JUn19b] JUnit Team. *JUnit 4.12 API*. 2019. URL: `https://junit.org/junit4/javadoc/4.12` (visited on 06/06/2019).

[Lin05] J. Link. *Softwaretests mit JUnit: Techniken der testgetriebenen Entwicklung*. 2nd. Heidelberg: dpunkt, 2005.

[Lin17] J. Link. *JUnit 5 Design und Architektur eines Frameworks*. 2017. URL: https://johanneslink.net/downloads/JUnit5-Architektur.pdf (visited on 06/06/2019).

[LS09] M. Leucker and C. Schallhart. *"A brief account of runtime verification"*. In: *The Journal of Logic and Algebraic Programming* 78.5 (2009), pp. 293–303.

[LT93] N. G. Leveson and C. S. Turner. *"An Investigation of the Therac-25 Accidents"*. In: *Computer* 26 (July 1993), pp. 18–41.

[Mar03] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[Mes07] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Signature Series. Upper Saddle River, NJ: Addison-Wesley, 2007.

[Moc19] Mockito. *Tasty mocking framework for unit tests in Java*. 2019. URL: https://site.mockito.org (visited on 06/06/2019).

[Mol09] I. Molyneaux. *The Art of Application Performance Testing*. 1st. O'Reilly Media, Inc., 2009.

[MS01] J. D. McGregor and D. A. Sykes. *A Practical Guide to Testing Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[Mye11] G. J. Myers. *The Art of Software Testing*. 3rd. Hoboken, New Jersey: John Wiley & Sons, Inc., 2011.

[Ora19] Oracle Corporation. *The Java Reflection API*. 2019. URL: https://docs.oracle.com/javase/tutorial/reflect/ (visited on 06/06/2019).

[PL05] A. Pretschner and M. Leucker. *"Model-Based Testing – A Glossary"*. In: *Model-Based Testing of Reactive Systems*. Ed. by M. Broy et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. Chap. 20, pp. 607–609.

[Pra95] V. Pratt. *"Anatomy of the Pentium Bug"*. In: *TAPSOFT '95: Theory and Practice of Software Development*. Ed. by P. D. Mosses, M. Nielsen, and M. I. Schwartzbach. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 97–107.

[TKS03] V. Trenkaev, M. Kim, and S. Seol. *"Interoperability Testing Based on a Fault Model for a System of Communicating FSMs"*. In: *Proceedings of the 15th IFIP International Conference on Testing of Communicating Systems*. TestCom'03. Sophia Antipolis, France: Springer-Verlag, 2003, pp. 226–242.