ALLSEEN ALLIANCE

How to Write AllSeen Alliance Self-Certification Test Cases

September 25, 2014

Contents

1 Introduction 3

1.1 Purpose 3
1.2 Scope 3
1.3 References 3
1.4 Acronyms and terms 3

2 Overview 4

2.1 AllJoyn validation framework 4
2.1.1 ValidationTestCase/ValidationBaseTestCase classes 4
2.1.2 ValidationSuite/ValidationTest annotations 4
2.1.3 ValidationTestContext class 4

3 Adding a new test suite 5

3.1 Prepare for your new test suite 5
3.2 Modify POM files 5
3.3 New test case class 7
3.4 New test suite class 9
3.5 Compiling your new test classes 10
3.6 Running your new test cases 10

4 Useful techniques 11

4.1 Handling asynchronous events 11
4.1.1 Create a handler class that will handle the event 11
4.2 Determine if an AllJoyn service/interface is implemented 12

1 Introduction

1.1 Purpose

This guide helps you write tests to verify that a device/application functionally behaves according to AllJoyn™ service framework interface definitions. It details each step in the process to write a validation test using the AllJoyn validation framework and JUnit 3 framework.

1.2 Scope

This document is intended for software engineers and assumes familiarity with the AllJoyn SDK and Java programming language.

1.3 References

The following documents are available on the AllSeen Alliance wiki.

- *Test Case Specification Template*
- *AllSeen Alliance Self-Certification User Guide*
- *AllSeen Alliance Self-Certification Developer Guide*

1.4 Acronyms and terms

Term	Definition
AllJoyn service frameworks	A collection of full-feature implementations using the AllJoyn framework that provides specific functionality. These are building blocks that can be combined together to build interoperable devices and applications.
AllJoyn base services	This includes the Config, Control Panel, Notification, and Onboarding interfaces that support the relevant service frameworks.
DUT	Device Under Test. The device, typically embedded, that runs the service to be validated.
POM file	Project Object Module file. Maven uses a POM XML file to compile and link your Validation Test app. It uses another to deploy your Validation Test to the test device, and to execute your Validation Test.
Test device	Android device, typically phone or tablet, which runs the Validation Tests against the service running on the DUT.

2 Overview

The AllJoyn framework is open-source software that allows for proximity peer to peer over various transports. The AllJoyn service frameworks are a collection of full-feature implementations using the AllJoyn framework that provide specific functionality. These are building blocks that can be combined together to build interoperable devices and applications.

The AllJoyn service framework implementations must conform to the published service framework interface specifications. This document goes over the steps to write tests to verify that an implementation behaves according to AllJoyn service framework interface specifications and the steps involved in executing the tests.

2.1 AllJoyn validation framework

The AllJoyn validation framework provides the set of Java interface definitions and classes required to write a test which can verify an AllJoyn service/interface implementation. It defines the API for plugging in the test cases/suites.

2.1.1 ValidationTestCase/ValidationBaseTestCase classes

- A class containing a test must extend *junit.framework.TestCase* and also implement *org.alljoyn.validation.framework.ValidationTestCase*.
- Optionally, the test case class can extend *org.alljoyn.validation.framework.ValidationBaseTestCase*.

ValidationBaseTestCase provides an implementation of methods specified in *ValidationTestCase*.

2.1.2 ValidationSuite/ValidationTest annotations

- The test case class must be annotated with *org.alljoyn.validation.framework.annotation.ValidationSuite* annotation to indicate that it's a test case class.
- The methods implementing the tests must be annotated with *org.alljoyn.validation.framework.annotation.ValidationTest* annotation.

The annotations provide the mapping from a particular test case class and method to the test suite name and test case name.

2.1.3 ValidationTestContext class

The *org.alljoyn.validation.framework.ValidationTestContext* class facilitates communication between a test case and the validation framework. A test case can retrieve test parameters from the *ValidationTestContext* as well as provide information about the test case to the *ValidationTestContext*.

3 Adding a new test suite

This section details the steps for writing a sample test case class and adding the test to a sample test suite. This section assumes that the compliance tests repository has been cloned under $VAL_DIR.

3.1 Prepare for your new test suite

1. From the $VAL_DIR\tests\java\components\validation-tests\HEAD folder, copy the validation-tests-suites folder and paste to a new folder. Rename it to represent your service, e.g., validation-tests-suites-lighting.

   This path will be referred to as $SVC_DIR in this document.

2. Delete the 'test' folder from $SVC_DIR\src.
3. Delete all existing folders from $VAL_DIR\tests\java\components\validation-tests\HEAD\$SVC_DIR \src\main\java\org\alljoyn\validation\testing\suites.
4. Create a new folder in $SVC_DIR \src\main\java\org\alljoyn\validation\testing\suites, e.g., 'lamp'

   In section 3.3, this new folder will be where you place your new class files.

5. Delete all existing folders and files from $VAL_DIR\tests\java\components\validation-tests\HEAD\$SVC_DIR \src\main\resources\introspection-xml.
6. Place the XML file for your new service in this folder (introspection-xml).

3.2 Modify POM files

Complete the following steps to modify the POM files to build the modules.

1. Locate the POM XML file in $SVC_DIR.
2. Modify the ***second*** <artifactId> tag, which is currently 'validation-tests-suites', to be $SVC_DIR. For example:

```
<parent>
      <groupId>org.alljoyn.validation.validation-tests</groupId>
            <artifactId>validation-tests-suites-lighting</artifactId>
            <version>1.0.0.09-SNAPSHOT</version>
</parent>
<artifactId>validation-tests-suites-lighting</artifactId>
…..
```

3. Locate the POM XML file in the …\validation-tests\HEAD folder.
4. Modify the "modules" section in it to include the new module you built, for example:

```
<modules>
        <module>validation-tests-simulator</module>
        <module>validation-tests-suites</module>
```

```
            <module>validation-tests-suites-lighting</module>
            <module>validation-tests-utils</module>
            <module>validation-tests-utils-android</module>
            <module>validation-tests-it</module>
        </modules>
```

5. Locate the POM XML file in the …\validation-tests\HEAD\validation-tests-it folder.
6. Modify the maven-remote-resources-plugin section in it to add a new resource bundle for your service:

```
<plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-remote-resources-plugin</artifactId>
      <version>1.5</version>
      <configuration>
      <resourceBundles>
            <resourceBundle>org.alljoyn.validation.validation-
tests:validation-tests-suites:${project.version}</resourceBundle>
            <resourceBundle>org.alljoyn.validation.validation-
tests:validation-tests-suites-
lighting:${project.version}</resourceBundle>
      </resourceBundles>
      <attached>false</attached>
      <outputDirectory>${project.basedir}/target/shared</outputDirectory
>
      </configuration>
      <executions><execution><goals><goal>process</goal></goals></execut
ion></executions>
</plugin>
```

7. Add a dependency tag for your new module, for example:

```
<dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <scope>provided</scope>
</dependency>
<dependency>
      <groupId>org.alljoyn.validation.validation-tests</groupId>

      <artifactId>validation-tests-suites-lighting</artifactId>
      <version>${project.version}</version>
</dependency>
<dependency>
      <groupId>org.alljoyn.validation.validation-tests</groupId>
      <artifactId>validation-tests-suites</artifactId>
      <version>${project.version}</version>
</dependency>
```

3.3 New test case class

1. In the folder you created in section 3.1, step 4, create a class that extends *org.alljoyn.validation.framework.ValidationBaseTestCase* and annotate the class with *org.alljoyn.validation.framework.annotation.ValidationSuite* annotation.
2. Specify a name for the testGroupId to be used when creating an *org.alljoyn.validation.framework.ValidationTestGroup* object.

   ```
   @ValidationSuite(name = "Sample-v1")
   public class SampleTestSuite extends ValidationBaseTestCase
   ```

   The expected naming convention for testGroupId includes the name of the service framework and its version (for example, Config-v1). This value must uniquely identify the test suite that includes one or more test cases.

   The test cases for a test suite can also be contained in multiple Java classes. In that case, all the classes for the test suite must contain this annotation and the same name.
3. Implement a *setUp()* method in which the parameters passed in to the test are read. Any steps that are common to all tests in the class should also be implemented in this method.

   Since JUnit 3 framework is used, the *setUp()* method is invoked before executing each test method.

   ```
   @Override
   protected void setUp()
   {
      super.setUp();
      testParameterValue =
      getValidationTestContext().getTestParameter(testParameterName);
      ....
      ....
   }
   ```
4. Create a *tearDown()* method to perform cleanup after a test completes execution.

   Since JUnit 3 framework is used, the *tearDown()* method is called after executing each test method.

   ```
   @Override
   protected void tearDown()
   {
      super.tearDown();
      ……
   }
   ```
5. Implement the test method and annotate it with the *org.alljoyn.validation.framework.annotation.ValidationTest* annotation.
6. Specify a test method name to be used later when creating an *org.alljoyn.validation.framework.ValidationTestItem* object.

The expected naming convention for test method name is the name of the service framework, its version, and a test case ID (for example, Config-v1-01). This must uniquely identify the test.

```
@ValidationTest(name = "Sample-v1-01")
public void testSample_v1_01_Sample()
{
       …...
}
```

- If some information about the test execution needs to be captured, use *org.alljoyn.validation.framework.ValidationTestContext* to add a note in the test.

  The ValidationTestContext interface extends *org.alljoyn.validation.framework.TestCaseNoteListener* interface which defines the *addNote()* method.

  ```
  getValidationTestContext().addNote("Information to be captured");
  ```

- If the test requires some user interaction, use the *org.alljoyn.validation.framework.ValidationTestContext* object to wait for user input.

  The ValidationTestContext interface extends *org.alljoyn.validation.framework.UserInputHandler* interface which defines the *waitForUserInput()* method. The *org.alljoyn.validation.framework.UserInputDetails* class is used to interact with the user and the *org.alljoyn.validation.framework.UserResponse* object will contain results of the user operation.

  ```
  @ValidationTest(name = "Sample-v1-01")
  public void testSample_v1_01_Sample()
  {
     …
     UserResponse userResponse =
        getUserResponse(getValidationTestContext());
     …
  }

  private UserResponse getUserResponse(UserInputHandler userInputHandler)
  {
     String[] messageArray = { "Test Msg 1", "Test Msg 2", "Test Msg 3" };
     UserInputDetails userInputDetails = new UserInputDetails("Select the
        message(s) received", messageArray);

     return userInputHandler.waitForUserInput(userInputDetails);
  }
  ```

3.4 New test suite class

1. In the folder you created in section 3.1, step 4, create a class that implements *org.alljoyn.validation.framework.ValidationTestSuite*.

   This class serves the following purposes:

   - It determines if the test cases present in a ValidationSuite should be run against a device based on the received About announcements.
   - It determines the set of test cases to be executed. One test suite class must be created for each ValidationSuite.

     ```
     public class SampleTestSuiteManager implements ValidationTestSuite
     ```

2. Implement the *getApplicableTests()* method. This method has to iterate over the About announcements from the device and return applicable test groups.

   ```
   @Override
   public List<ValidationTestGroup>
       getApplicableTests(AllJoynAnnouncedDevice allJoynAnnouncedDevice)
   {
       for (AboutAnnouncement aboutAnnouncement :
           allJoynAnnouncedDevice.getAnnouncements())
       {
           if (addTestGroupForApplication(aboutAnnouncement))
           {
               testGroups.add(createTestGroup(aboutAnnouncement));
           }
       }

       return testGroups;
   }
   ```

   - The addTestGroupForApplication() method should have the logic to determine if the test suite is applicable based on the About announcement details.
   - The createTestGroup() method should have the logic to create an org.alljoyn.validation.framework.ValidationTestGroup object.

     ```
     private ValidationTestGroup createTestGroup(AboutAnnouncement
     aboutAnnouncement)
     {
           Class<? extends ValidationTestCase> validationTestCaseClass =
                 SampleTestSuite.class;
           ValidationSuite validationSuite =
                 validationTestCaseClass.getAnnotation(ValidationSuite.class
                 );
           String testGroupId = validationSuite.name();
           ValidationTestGroup testGroup = new
                 ValidationTestGroup(testGroupId, aboutAnnouncement);
           addTestItemsToGroupFromAnnotations(testGroup,
                 validationTestCaseClass);

           return testGroup;
     ```

```
}
```

NOTE

The addTestItemsToGroupFromAnnotations() method should take care of creating *org.alljoyn.validation.framework.ValidationTestItem* objects and adding them to *ValidationTestGroup* object.

```
private void addTestItemsToGroupFromAnnotations(ValidationTestGroup
        testGroup, Class<? extends ValidationTestCase>
        validationTestCaseClass)
{
        Method[] methods = validationTestCaseClass.getMethods();
        List<Method> testMethods = new ArrayList<Method>();

        for (Method method : methods)
        {
                ValidationTest validationTest =
                        method.getAnnotation(ValidationTest.class);
                Disabled disabled = method.getAnnotation(Disabled.class);

                if ((validationTest != null) && (disabled == null))
                {
                        testMethods.add(method);
                }
        }

        Collections.sort(testMethods, sampleTestComparator);

        for (Method testMethod : testMethods)
        {
                ValidationTestItem testItem = new
                        ValidationTestItem(validationTest.name(),
                        validationTestCaseClass.getName(),
                        testMethod.getName(), validationTest.timeout());
                testGroup.addTestItem(testItem);
        }
}
```

The *Collections.sort(testMethods, sampleTestComparator)* sequences the tests as per the values specified in the order field of *ValidationTest* annotation.

3.5 Compiling your new test classes

Refer to the *AllSeen Alliance Self-Certification Developer Guide* for instructions.

3.6 Running your new test cases

Refer to the *AllSeen Alliance Self-Certification User Guide* for instructions.

4 Useful techniques

This section provides helpful techniques for writing test cases.

4.1 Handling asynchronous events

The test cases execute synchronously in a single thread. Sometimes a test case must wait for other events to occur before test execution can proceed. The following design provides a technique for waiting for something to happen, and retrieving details on what happened.

4.1.1 Create a handler class that will handle the event

The handler class will maintain an internal optionally bounded blocking queue that holds the events. The handler class must expose a public method that is invoked by the event publisher. When an event is received from the publisher, the event is added to the queue.

The handler class must also expose another method for event receivers to get the event. Whenever this method is invoked, the event is polled from the queue and returned to the receiver. The same instance of handler class is used by both the event publisher and event subscriber.

```
public class EventHandler
{
    private LinkedBlockingDeque<Object> eventQueue = new
        LinkedBlockingDeque<Object>();

    public void handleEvent(Object object)
    {
        eventQueue.add(object);
    }

    public Object waitForNextEvent(long timeout, TimeUnit unit) throws
        InterruptedException
    {
        return eventQueue.poll(timeout, unit);
    }
}
```

The *waitForNextEvent()* method will time out if no event is received within the timeout duration specified in the call. The reason for waiting only a specific length of time is that it allows the test case to fail sooner if the expected event doesn't occur in the expected time (versus waiting for the test case to time out).

If the expectation is that a particular event will happen within a particular time frame, the test case should be coded to wait only that long before having an assertion fail.

4.2 Determine if an AllJoyn service/interface is implemented

Create a utility method that will accept an AboutAnnouncement method and the AllJoyn feature interface name.

```
public boolean supportsInterface(AboutAnnouncement aboutAnnouncement,
    String expectedInterfaceName)
{
   boolean interfaceSupported = false;
   for (BusObjectDescription busObjectDescription :
      aboutAnnouncement.getObjectDescriptions())
   {
       for (String interfaceName : busObjectDescription.interfaces)
       {
          if (interfaceName.equals(expectedInterfaceName))
          {
             interfaceSupported = true;
             break;
          }
       }
   }

   return interfaceSupported;
}
```