KitCAT - A Framework for Converged Application Testing

Venkita Subramonian
AT&T Labs Research

Contents

1 Introduction **1**

2 Installation **3**

3 Getting Started **5**
3.1 Generating the test project 5
3.2 Testing in a Command Line Environment 6
3.2.1 Running Test Using Shell Script 7
3.2.2 Running Test Using Ant 9
3.3 Testing in a Eclipse environment 9

4 Structure of a KitCAT Test Case **14**

5 Writing Test Cases **17**
5.1 A simple two way call test 17
5.2 Sending media 19
5.3 Testing a converged application 20

6 Logging **23**

1 Introduction

KitCAT is a JUnit [7] based testing framework to facilitate functional testing of converged telecom applications. A KitCAT test case uses the KitCAT

API (Java) to interact with the system under test (SUT) using SIP/RTP protocols. In conjunction with other web testing APIs like that offered by HtmlUnit [1], users can write functional tests for converged applications. A single KitCAT test case can create, control and coordinate multiple SIP/RTP as well as browser endpoints involved in functional testing.

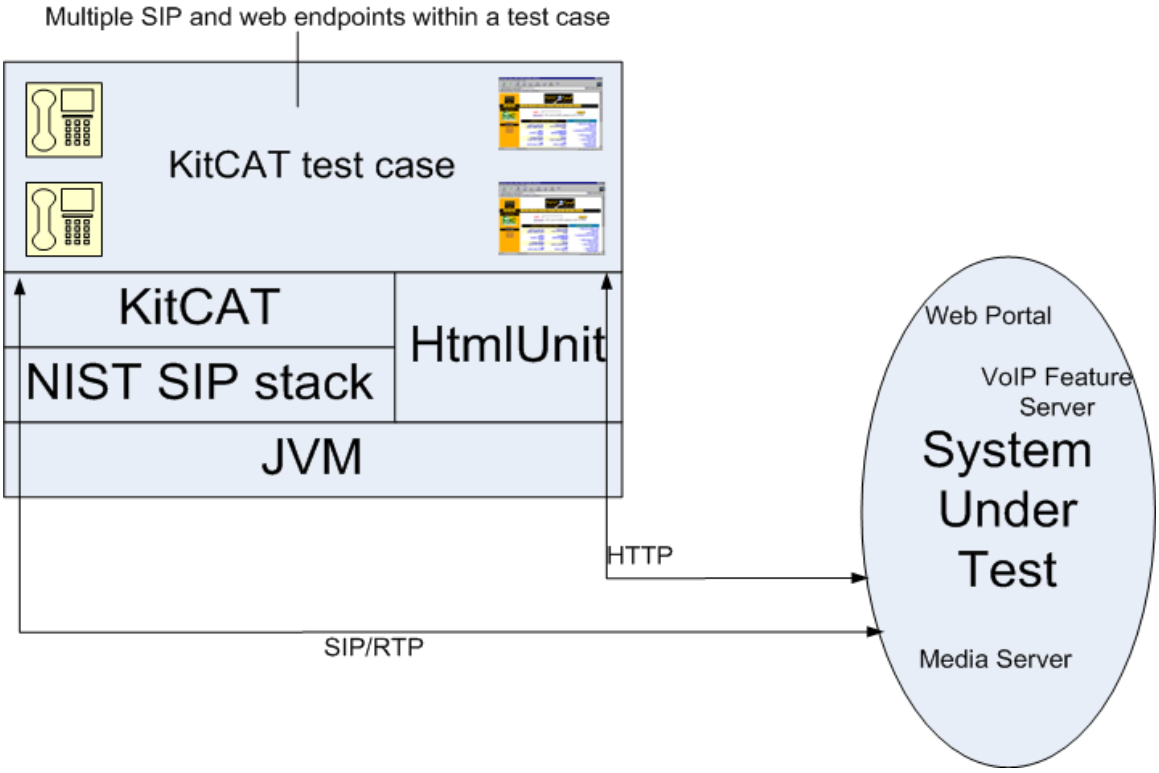

Figure 1: Testing architecture using KitCAT

Figure 1 shows how KitCAT is typically used for functional testing of converged applications that have both web as well as VoIP functions. KitCAT uses the JAIN SIP [6] compliant NIST SIP stack [2] underneath for SIP interaction with the system under test (SUT). A test *agent* in KitCAT is used to simulate a SIP/RTP end user of the SUT and thus responsible for sending and receiving SIP/RTP messages. A test case creates *agents* which are then used to communicate with the SUT. An agent also maintains a state which is advanced appropriately as SIP messages are sent and received by

that agent. A test case can control and coordinate execution of agents as well as assert the execution states of agents based on messages sent/received by them. An agent can act as either a caller initiating SIP messages to the SUT or a callee receiving SIP messages from the SUT. A KitCAT test case hosts both caller and callee agents within a single process thus facilitating the coordination among the callers and callees.

KitCAT offers the following main features for testing converged applications:

- A simple sequential execution model for the test writer. The test writer does not have to worry about receiving SIP/RTP messages that arrive asynchronously.
- All test endpoints are hosted in one single process. This facilitates coordination among agents within a test case. Agents can share common data within a test case since they run within the same process address space.
- High level primitives to control test agents (*e.g.,* call, answer). These primitives are simple enough for the test writer to express the most common forms of call-control such as initiating a call, answering a call and ending a call.
- Helper classes for test writer to manipulate headers in messages that are sent by agents.
- A state-machine based approach for specifying agent behavior.
- Assertion primitives to check assertions on agent states and messages sent/received by agents. These assertion primitives are extensible so that the test writer can write custom assertions suited to the application under test.
- Ability to send/receive RTP stream to/from remote peer.
- Ability to send DTMF key sequence specified as a string.

2 Installation

The prerequisites for using KitCAT are -

- JDK 1.5 or up

- ant 1.7.x or any development environment like Eclipse
- NIST SIP/SDP API and implementation jars (build 1.2.86 or up). KitCAT has been tested with NIST distribution 1.2.86.

The command line instructions described in this manual are intended for Unix environments, but are also applicable in a Windows environment. Appropriate batch command files are provided where necessary. **Note: For the batch files to work properly in Windows, you should enable the delayed environment variable expansion - add /V:ON parameter to your command prompt window executable (cmd.exe /V:ON)**

The following step-by-step instructions can be used to guide you through the installation process.

1. Download and uncompress the KitCAT distribution. The **lib** directory in the distribution contains all the KitCAT related jar files that need to be in the classpath when building or running KitCAT test cases. The **doc** directory contains the javadoc and other documentation for KitCAT. The **tools** directory contains some tools to facilitate getting started with a KitCAT test case. The **examples** directory contains example tests that show usage of the KitCAT API.

2. Download the NIST SIP/SDP distribution (build 1.2.86 or up) from the NIST SIP stack download site (`http://download.java.net/communications/jain-sip/nightly/jain-sip-sdp/`). Copy the NIST distribution to the **lib** directory in the KitCAT distribution.

3. To test the installation, from the KitCAT installation directory, do the following -

```
$$ cd examples
$$ ./testinstall.sh
0    [main] INFO  KitCAT.Tester  - TwoWay ctor called
7    [main] INFO  KitCAT.Tester  - running test setup
7    [main] INFO  KitCAT.Tester  - testing a simple two way call
8    [main] INFO  KitCAT.Tester  - Running test simpletest
544  [main] INFO  KitCAT.Tester  - Alice sends INVITE to Bob
656  [main] INFO  KitCAT.SipStack  - <message
from=''135.207.26.97:23456''
to=''135.207.26.97:23456''
```

```
time=''1225738423927''
isSender=''true''
transactionId=''z9hg4bkf13fff7727eb86bba5352e3701fd0f59''
callId=''a9a32a0b0a1b4b0bba89d7d9ab92f1af@135.207.26.97''
firstLine=''INVITE sip:Bob@135.207.26.97:23456 SIP/2.0''
.................
....more logs.....
.................
39321 [main] INFO  KitCAT.Tester  - Install test succeeded!!
39322 [main] INFO  KitCAT.Tester  - running test tear down
```

The above test sets up a simple test (see `TwoWay.java(method=simpletest` under **examples** directory) in the KitCAT distribution) call between two agents, exchanges media and hangs up. The test case also makes appropriate assertions to ensure that the call was successful. The output log messages are those from the test case, KitCAT framework as well the NIST SIP stack. If the installation is successful, then you should see the test run terminate as shown above with the message "Install test succeeded". A batch file has been provided to run the above test on a Windows platform. You should see similar log output on Windows as well.

3 Getting Started

KitCAT is a Java library and hence any tools that are used for Java based developement and testing can be used to build and run KitCAT test cases. The jar files in the KitCAT distribution (under **lib** directory) should be in the classpath for building and running KitCAT test cases. Moreover, the log4j settings should be available to KitCAT during runtime. This can be done by including the path to log4j.properties in the classpath.

3.1 Generating the test project

To get started, we have provided simple command line tools with KitCAT to generate a skeleton test case and associated files for building and running the test case. Before following these instructions, an environment variable **KITCAT_HOME** should be set to the KitCAT installation directory. For example,

```
$$ echo $KITCAT_HOME
/home/venkita/demo/KitCAT_1.0-beta
```

Switch to a directory where the test case and associated files need to be generated. Run the test generation utility. A sample interaction is shown below.

```
$$ $KITCAT_HOME/tools/testgen.sh
Enter test directory name (e.g. KitCATTest) : test1
Enter package name (e.g. com.xyz.tests) : mytest
Enter test class name (e.g. MyTest): MyTest
Please fill in test parameter values in etc/test.properties file.
```

The following directory structure is created from the above run.

```
test1/
     build.properties     - used for ant-based build
     build.xml            - used for ant-based build
     .classpath           - used with Eclipse
     .project             - used with Eclipse
     testdriver.bat       - used to run the test on Windows
     testdriver.sh        - used to run the test from a shell
     etc/
        log4j.properties  - log4j settings
        test.properties   - test parmeters
     src/
        mytest/
              MyTest.java - Skeleton test case
```

The above utility generates a simple test case skeleton and the associated files to build and run the test case. Now we can work with this test project either from a command line environment or eclipse environment.

3.2 Testing in a Command Line Environment

To build and run the test project from a command-line, the generated **ant** based build scripts can be used. Switch to the newly generated test directory and do the following:

```
$$ ant jar
```

The generated test case uses a `test.properties` file to get test parameters such as the location of the SIP application server, HTTP app server and the listen port for the SIP stack. You should fill in the parameters appropriately. To begin with you must fill in the **SipStackListenPort** parameter. If you forget this step you'll get an exception when running the generated test case.

Now that we have built the test case, we illustrate two ways to run the test case from the command line. Both these methods use a JUnit test runner to run the test. The first method provides the tester with a shell script that uses a command line based JUnit runner directly and the second method uses an ant script which uses a JUnit runner provided by ant (version 1.7.x or higher). The ant based JUnit runner outputs results in a web page which is easier to view and navigate in a unit testing scenario. The non-ant based JUnit runner is useful in scenarios where a KitCAT test case is embedded as part of a larger system test scenario which may involve a number of other non-KitCAT tests. Moreover, this run method can be used with ant versions prior to 1.7.x, that don't have support for JUnit ant tasks.

3.2.1 Running Test Using Shell Script

First, set the **SipStackListenPort** property in etc/test.properties to a valid port number (eg. 12345). From the command line, the test case can be run as follows:

```
$$ ./testdriver.sh

JUnit version 4.4
.0    [main] INFO  KitCAT.Tester  - running test setup
2    [main] INFO  KitCAT.Tester  - Running test simpletest
530  [main] INFO  KitCAT.Tester  - caller sends INVITE to callee
648  [main] INFO  KitCAT.SipStack  - <message
from=''135.207.26.97:12345''
to=''135.207.26.97:12345''
time=''1226511787438''
isSender=''true''
transactionId=''z9hg4bke6461ec6f03a147d4fec54f15f6796ce''
callId=''bbf99b9226466153d6e53958059d61ea@135.207.26.97''
firstLine=''INVITE sip:callee@135.207.26.97:12345 SIP/2.0''
```

```
>
<![CDATA[INVITE sip:callee@135.207.26.97:12345 SIP/2.0
Call-ID: bbf99b9226466153d6e53958059d61ea@135.207.26.97
CSeq: 1 INVITE
From: ‘‘caller’’ <sip:caller@135.207.26.97>;tag=callernwPort
To: <sip:callee@135.207.26.97:12345>
Via: SIP/2.0/UDP 135.207.26.97:12345;branch=z9hG4bKe6461ec6f03a147d4fec54f15f6796ce
Max-Forwards: 70
Contact: <sip:caller@135.207.26.97:12345>
Content-Type: application/sdp
Content-Length: 180
.................................
.........more logs here.............
.................................
6802 [UDPMessageChannelThread] INFO  KitCAT.SipStack  - <message
from=’’135.207.26.97:12345’’
to=’’135.207.26.97:12345’’
time=’’1226511793597’’
isSender=’’false’’
transactionId=’’z9hg4bk7c7003c7accf3448177e18f993d7f138’’
callId=’’bbf99b9226466153d6e53958059d61ea@135.207.26.97’’
firstLine=’’SIP/2.0 200 OK’’
>
<![CDATA[SIP/2.0 200 OK
Via: SIP/2.0/UDP 135.207.26.97:12345;branch=z9hG4bK7c7003c7accf3448177e18f993d7f138
CSeq: 2 BYE
Call-ID: bbf99b9226466153d6e53958059d61ea@135.207.26.97
From: ‘‘caller’’ <sip:caller@135.207.26.97>;tag=callernwPort
To: <sip:callee@135.207.26.97:12345>;tag=calleenwPort
Content-Length: 0

]]>
</message>

8794 [main] INFO  KitCAT.Tester  - Test passed
8795 [main] INFO  KitCAT.Tester  - running test cleanup
13804 [main] INFO  KitCAT.Tester  - ====================

Time: 13.911

OK (1 test)
```

The output from the run includes the output from KitCAT framework as well as the SIP messages log output from the NIST SIP stack.

3.2.2 Running Test Using Ant

We have also provided an ant script for running a test case using the JUnit task in ant. First, set the **SipStackListenPort** property in etc/test.properties to a valid port number (eg. 12345).

```
$$ ant test
```

A **report** directory is generated under which the test results can be found in html format. For example, the above run generated the web page in Figure 2.

3.3 Testing in a Eclipse environment

To build and run the test project from an Eclipse environment, the generated Eclipse related files (`.classpath` and `.project`) can be used. The following instructions enable you to get started with KitCAT test cases quickly in an Eclipse environment.

1. Create a user library for KitCAT as shown in Figure 3. From Eclipse, do **Window–Preferences** and then **Java–Build Path–User Libraries**. Create a new library called **KitCAT**. Add all the jar files from the **lib** directory in the KitCAT distribution.

2. Import the generated project from section 3.1 into Eclipse (see Figure 4. From Eclipse, do **File–Import** and select **Existing Projects into Workspace**. In the subsequent screen, select **Select root directory** and fill in the path to the test project directory generated in Section 3.1. Figure 5 shows the project structure in Eclipse after the project is imported.

3. Set the **SipStackListenPort** property in test.properties to a valid port number (eg. 12345).

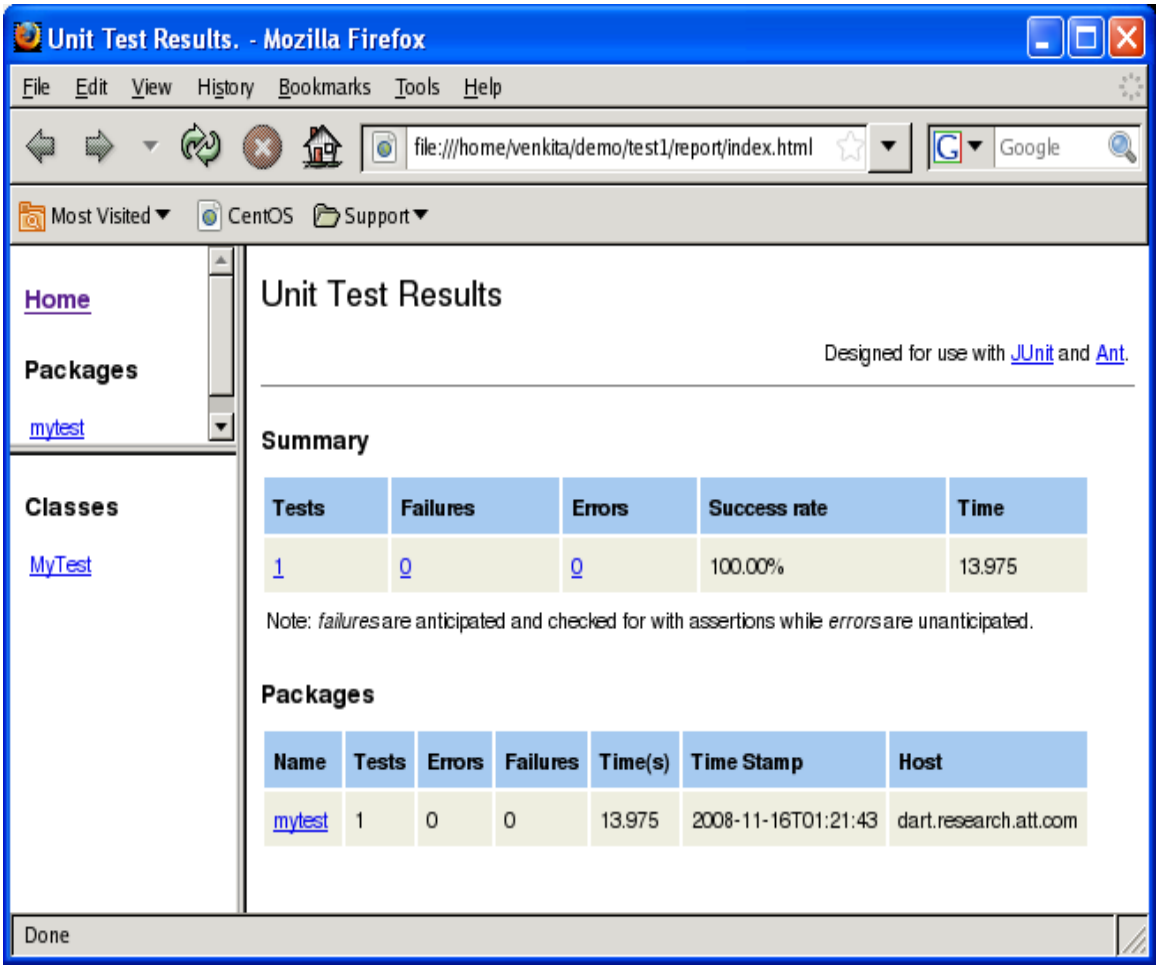

Figure 2: Test run results from ant based run

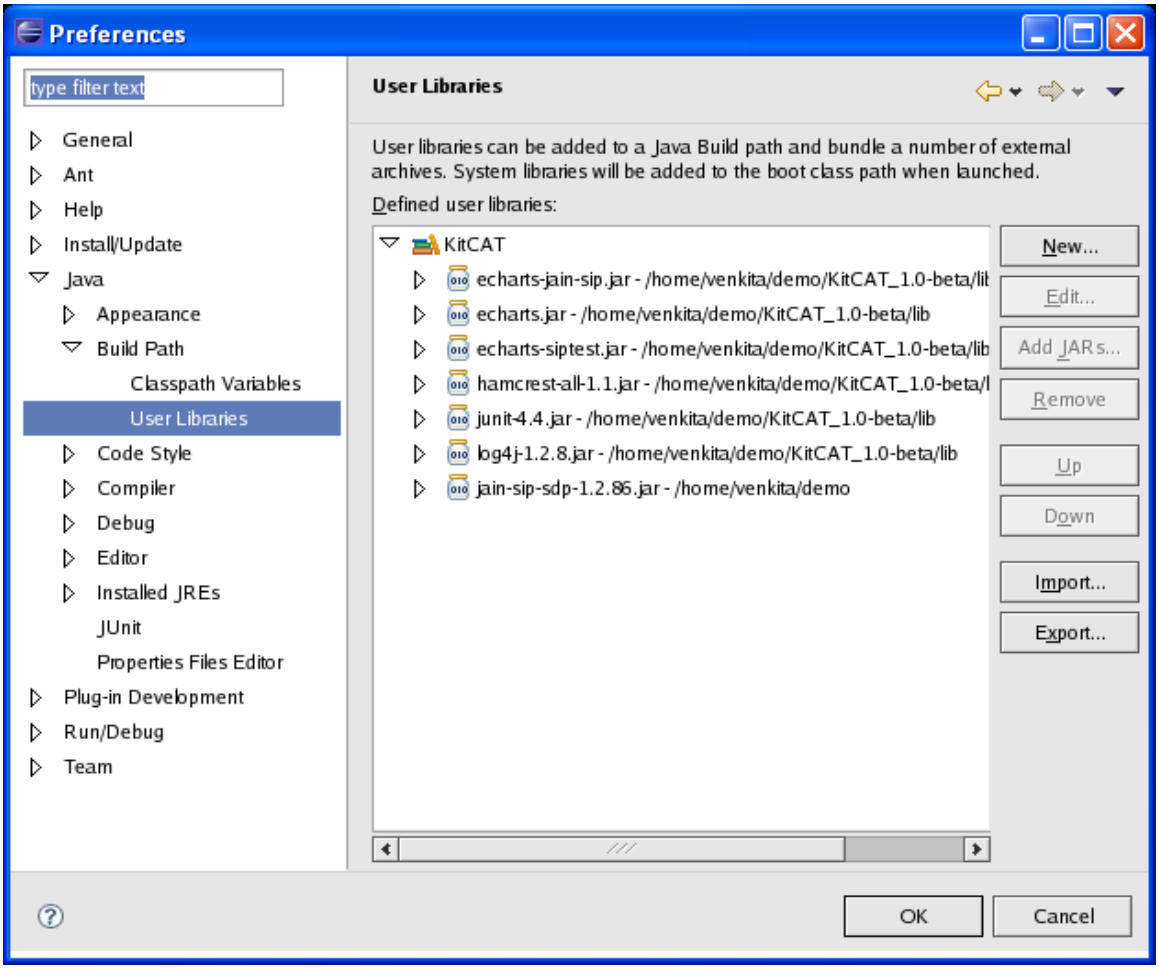

Figure 3: Creating Eclipse user library for KitCAT

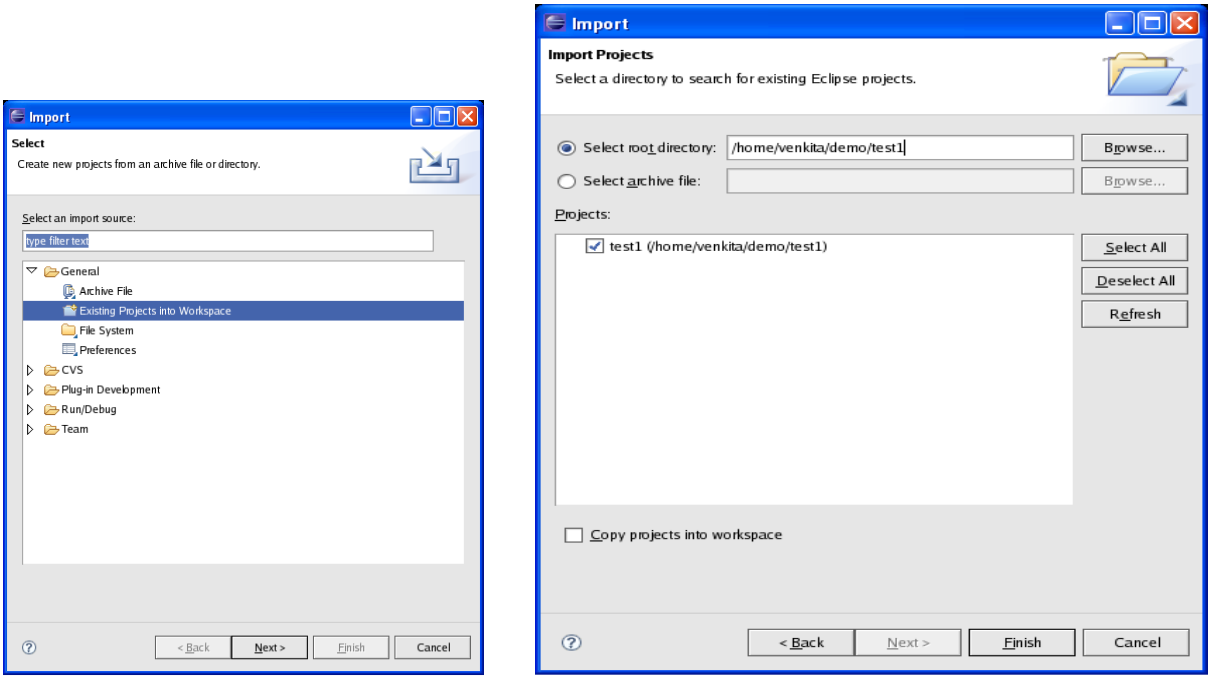

Figure 4: Importing generated project in to Eclipse

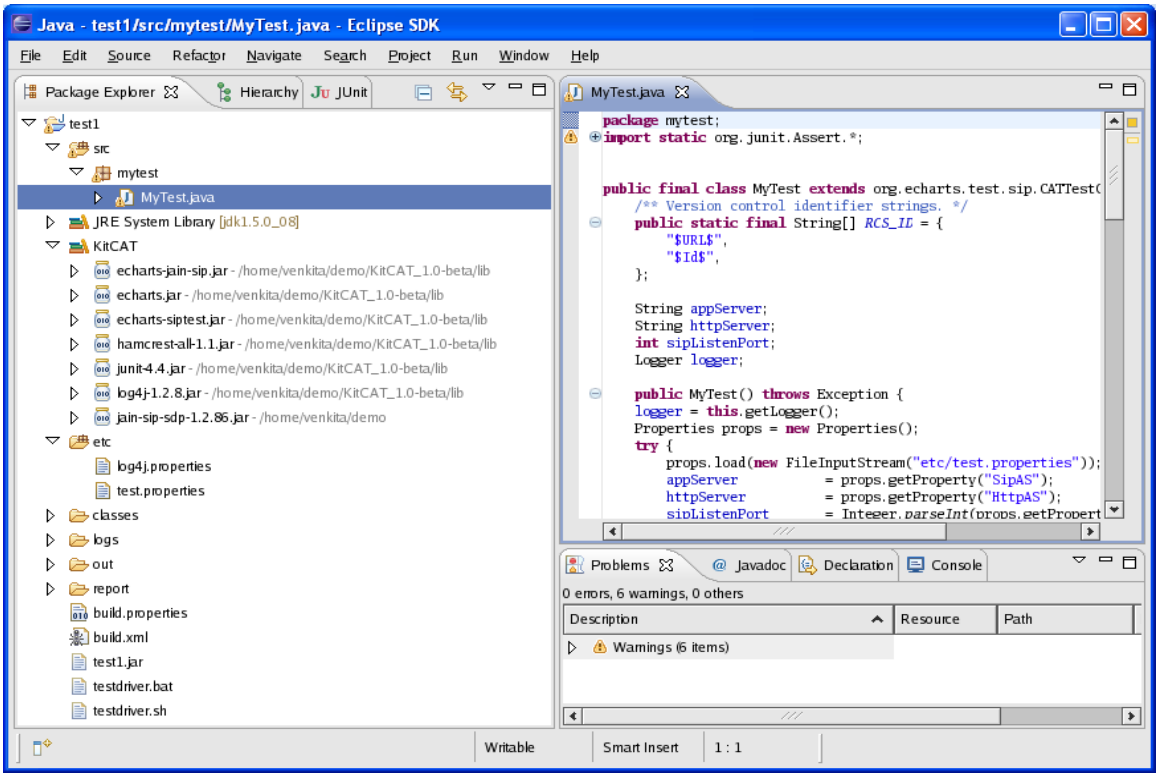

Figure 5: Sample test project structure in Eclipse

4. Run the test case just as you would run any JUnit 4.4 test case in Eclipse. The test case should pass and you should not see any errors in the Eclipse console window. You should see log messages as in the command line run.

4 Structure of a KitCAT Test Case

A KitCAT test case is a JUnit test case that uses the KitCAT library to simulate SIP/RTP endpoints. Each test agent in KitCAT contains a SIP and RTP endpoint. The behavior of a test agent is governed by a state machine within KitCAT. A KitCAT test case hosts multiple such agents whose execution is controlled and coordinated by the test case. A test case has three different types of statements that either query or affect the state of test agents:

- User commands
- processSIP
- Assertions

Command	**Effect on agent**
call	sends a INVITE with SDP
answer	sends a 200 OK response to an INVITE
sendResponse	sends a response to a request
info	sends a INFO message
register	sends a REGISTER message
reinvite	sends a mid-dialog re-INVITE
cancel	sends a CANCEL
end	sends a BYE
playAudio	queues RTP stream for transmission
sendDTMF	queues RTP packets with DTMF events for transmission

Table 1: User commands in KitCAT

User Commands: A user command causes a test agent to send a SIP message or schedule RTP packets for transmission. Table 1 shows some of the user commands currently available in KitCAT.

processSIP primitive: A KitCAT test case presents the test writer with a sequential execution model to execute a test case. There are no call-back functions to handle the asynchronously arriving SIP messages. Instead, the KitCAT runtime framework handles these asynchronous SIP messages in background threads and buffers them in internal KitCAT buffers. It then provides a synchronous (blocking) interface for the test case to process these messages using the **processSIP** API. This primitive causes all buffered messages for all agents to be processed by the agents until a specified time expiry occurs. If all messages are processed by all agents, then this primitive causes the test case execution to block until either a newly arrived message is buffered for processing or a timeout occurs whichever is earlier. Processing of messages involve execution of the agent state machines causing their states to possibly change. Sometimes, this could cause agents to send further SIP messages in response to the processed message. For example, by default an agent sends an ACK in response to a 200 OK to an INVITE. Whenever there is a need to introduce a delay in the test case, it is highly recommended to use the **processSIP** primitive instead of **Thread.sleep()**. This ensures that SIP messages are still being processed during the delay. **processSIP is the only primitive where arriving SIP messages are processed.**

Assertions: Assertions in KitCAT query agents' states to check whether the test case execution caused the expected state changes in agents. An assertion statement itself does not cause any side-effects on an agent. During our implementation, we tried providing specific assertion primitives for carrying out specific types of assertions. For example, there could assertions based on states of agents (assertState), relating the states of multiple agents (assertAgentsConnected), about messages exchanged by agents (assertMsgRecvdByAgent), and so on. As you can see, the problem with this approach is that we end up with many assertion primitives and a new type of assertion would require a change to the KitCAT framework. We soon realized the need for a more flexible and reusable assertion mechanism that will allow us to create different types of assertions easily. Assertions in KitCAT are designed to use the **assertThat** primitive that is part of JUnit 4.4. For

more information about the **assertThat** primitive, see [5, 4, 7].

The logic of the **assertThat** primitive is roughly as follows -

```
assertThat(java.lang.Object subject,
           org.hamcrest.Matcher matcher) {
  //Matcher objects implement matches method
  if (!matcher.matches(subject))
    throw assertion error
}
```

It takes in a subject which is of type **java.lang.Object** and a matcher object of type **org.hamcrest.Matcher**, which checks for specific properties on the subject. This simple primitive allows us to create a variety of matchers for specific purposes. For example, the following assertion checks that two agents are connected by making sure that they are in the appropriate states and also that they have exchanged SDPs.

```
assertThat(bob, is (connectedTo(alice)));
```

Matcher factory method	**Condition check**
idle	agent has not received any message
invited	agent has received INVITE
connected	agent is in a SIP dialog
connectedTo	agent is in a SIP dialog with a specified agent and SDP's have been exchanged between them
disconnected	agent's SIP dialog has terminated
recvdMessage	agent received a SIP message
sentMessage	agent sent a SIP message

Table 2: Sample assertion condition API in KitCAT

Here, **connectedTo** is a static factory method that creates a matcher to check whether the two agents are connected. This assertion mechanism helps with the readability of a test case, and also provides a facility for the test writer to write new matchers that are not already available as part of KitCAT. KitCAT provides convenient-to-use matcher factory methods (see Table 2) to create appropriate matcher objects.

5 Writing Test Cases

5.1 A simple two way call test

A simple test case is shown below. This example along with other example test cases are available in the examples directory as part of the KitCAT distribution (see **examples/src/TwoWay.java**). These examples illustrate usage of the various primitives and assertion conditions possible in KitCAT. For detailed discussion about the execution model and semantics of KitCAT primitives see [9].

This test case creates two SIP endpoints (test agents) and tests whether a call can be established between the two agents. The test case also checks media interaction between the two agents.

```
@Test public void simpletest() throws Exception {
      try {
            logger.info("testing a simple two way call");

            //initialize the test framework.
            init(listenPort);

            //create test agents
            SIPAgent alice = createAgent("Alice");
            SIPAgent bob = createAgent("Bob");

            //Alice --INVITE--> Bob
            alice.call(bob);

            //Both test agents process any incoming SIP messages
            //for 2 seconds.
            processSIP(2000);

            //JUnit (Hamcrest) assertThat API to assert
            //test conditions. Matchers are available in
            //KitCAT to test for various conditions.
            //Here, we make sure agent bob received an
            //INVITE.
            assertThat(bob, invited());

            //Bob --200 OK--> Alice
            bob.answer();
```

```
            //Both test agents process any incoming SIP messages
            //for 2 seconds. By default, a KitCAT test agent
            //receiving a 200 OK sends an ACK to its peer.
            processSIP(2000);

            //Here connectedTo static method creates a matcher
            //that tests whether INVITE-200-ACK sequence
            //has occured and the SDPs have been exchanged.
            assertThat(alice, connectedTo(bob));

            //user-defined function to check media flow
            processSIPAndCheckMedia(alice, bob);

            processSIP(2000);

            //Alice --BYE--> Bob
            alice.end();

            //Both test agents process any incoming SIP messages
            //for 2 seconds. By default, a KitCAT test agent
            //receiving a BYE sends a 200 OK to its peer.
            processSIP(2000);

            //disconnected static method creates a matcher
            //that tests whether the BYE-200 sequence has
            //occurred and the call is torn down.
            assertThat(alice, disconnected());
            assertThat(bob, disconnected());

            logger.info("install test succeeded!!");
      } catch(Exception e) {
            logger.error("install test failed!!", e);
            throw e;
      } catch(Error e) {
            logger.error("install test failed!!", e);
            throw e;
      }
}
```

It must be noted that sending of SIP messages occur as soon as a user command (*e.g.,* . call, end, info) is invoked on a test agent. For example, when the statement **alice.call(bob)** is executed, it immediately results in a SIP `INVITE` being sent to Bob without **processSIP** being called. But, agent Bob processes this `INVITE` only when a **processSIP** primitive is called.

Assertion statements don't have any side-effects and hence do not affect the state of the test agents.

5.2 Sending media

In the above test case, there is a user-defined method to check media connectivity between two agents.

```
void processSIPAndCheckMedia(SIPAgent a, SIPAgent b)
        throws Exception {
     logger.info("checking media connectivity between "
             + a.getName() + " and " + b.getName());
     processSIP(3000);
     b.clearMediaBuffer();
     a.playAudio("etc/howdoin.raw");
     processSIP(2000);
     assertThat(b, has (incomingMedia()));
     a.clearMediaBuffer();
     b.playAudio("etc/hellobaby.raw");
     processSIP(2000);
     assertThat(a, has (incomingMedia()));
}
```

Such methods are reusable across multiple test cases. The extract below illustrates use of some of the media related primitives in KitCAT. The **playAudio** primitive queues RTP packets for transmission by a background KitCAT thread. **KitCAT uses G.711 μ-law codec with 20ms for RTP packetization**. Each KitCAT agent stores received RTP packets to a disk file (*testmethodname_agentname*.audio.raw) under the **out** directory - *e.g.,* simpletest_Alice.audio.raw contains the audio stream received by Alice during this test run. The **clearMediaBuffer** primitive is used to clear internal agent data structures that keep track of RTP packets received.

Currently KitCAT does not process audio files with data format headers in the media file (*e.g.,* .wav file). Any media file should be converted to a *raw* format (header-less data only) using a utility like **sox** [3]. An example interaction is shown below.

```
#to convert from wav format to raw format
$ sox -V hellobaby.wav -b -c 1 -U -r 8000 hellobaby.raw
```

```
sox: Detected file format type: wav

sox: Chunk fmt
sox: Chunk data
sox: Reading Wave file: Microsoft PCM format, 1 channel, 11000 samp/sec
sox:         11000 byte/sec, 1 block align, 8 bits/samp, 23760 data bytes
sox: Input file hellobaby.wav: using sample rate 11000
        size bytes, encoding unsigned, 1 channel
sox: Output file hellobaby.raw: using sample rate 8000
        size bytes, encoding u-law, 1 channel
sox: resample opts: Kaiser window, cutoff 0.800000, beta 16.000000

#to convert from raw format to wav format
$ sox -V -t raw -c 1 -r 8000 -b -U simpletest_Bob.audio.raw bob.wav
sox: Input file simpletest_Bob.audio.raw: using sample rate 8000
        size bytes, encoding u-law, 1 channel
sox: Input file simpletest_Bob.audio.raw: comment ‘‘simpletest_Bob.audio.raw’’

sox: Writing Wave file: Microsoft U-law format, 1 channel, 8000 samp/sec
sox:         8000 byte/sec, 1 block align, 8 bits/samp
sox: Output file bob.wav: using sample rate 8000
        size bytes, encoding u-law, 1 channel
sox: Output file: comment ‘‘simpletest_Bob.audio.raw’’

sox: Finished writing Wave file, 19520 data bytes 19520 samples
```

5.3 Testing a converged application

In Section 5.1, we discussed a test case, which dealt with SIP/RTP test endpoints. We now illustrate how we can integrate web testing in a KitCAT test case. Even though we show a very simple example, the concepts can be and have been applied to write test cases to do functional testing of non-trivial converged applications.

Although KitCAT is not tied to any web testing frameworks, we have been successful using HtmlUnit [1] for functional testing of web applications and highly recommend using it. An easy way to get started would be to download (`http://htmlunit.sourceforge.net/`) the HtmlUnit related jar files on to the **lib** directory in the KitCAT installation. An alternative way would be to create a User Library in Eclipse for HtmlUnit (using the procedure described for creating KitCAT User Library in Section 3.3) and then including this

user library in the Java build path for this project.

The following example shows a test case for unit testing a simple call forwarding application. This call forwarding application has an extremely simplified web interface that has been developed for purposes of unit testing functionality. In this test, we set up call forwarding for a subscriber through the web interface. The call is set to be forwarded to another agent in the same test case. The test case then makes sure that a call to the subscriber is indeed forwarded to the new location. As part of the call forwarding application, there is a simple jsp page that is used to setup the forwarding parameters. The **setForwardingUser** method uses HtmlUnit to invoke this jsp page (`http://hudson:28501/ucfTest/set.jsp?fwdUser=forwardedTo&fwdHost=dart&fwdPort=12345`) provided by the application. This directs the application to forward all calls to the subscriber to the SIP address `sip:forwardedTo@dart:12345` which is the address of the test agent `forwardedTo`. The test case now initiates a call to the subscriber and asserts that the agent designated to receive the forwarded calls gets it instead of the subscriber.

```
@Test public void testForwarding() throws Exception {
   try {
        int sipListenPort = 12345;
        String thisMachineIP = ‘‘dart’’;
        this.init(sipListenPort);

        SIPAgent caller = createAgent("caller");
        SIPAgent subscriber = createAgent("subscriber");
        SIPAgent forwardedTo = createAgent("forwardedTo");

        //sets up call forwarding through the web. after
        //this setup, all calls to subscriber should get
        //forwarded to forwardedTo@<thisMachineIP>:<sipListenPort>

        setForwardingUser("forwardedTo", thisMachineIP, sipListenPort);
        processSIP(1000);

        //host and port where the converged app is running
        caller.setProxy(‘‘hudson:28501’’);

        caller.call(subscriber);
        processSIP(2000);

        assertThat(subscriber, is(idle()));
        assertThat(forwardedTo, has(recvdRequest("INVITE")));
```

```
        forwardedTo.sendResponse(180);
        processSIP(2000);

        assertThat(caller, has(recvdResponse(180)));

        forwardedTo.answer();
        processSIP(2000);

        assertThat(caller, is(connectedTo(forwardedTo)));

        forwardedTo.end();
        processSIP(2000);

        assertThat(forwardedTo, is(disconnected()));
        assertThat(caller, is(disconnected()));

   } catch(Exception e) {
        logger.error("testing failed ", e);
        throw e;
   }
   catch(Error e) {
        logger.error("testing failed ", e);
        throw e;
   }
}

//Invoke the jsp page provided by the application, passing the
//information about the new location to forward the call to.
private void setForwardingUser(String user, String ip, int port)
     throws IOException {

     String callFwdSetupString =
      ``http://hudson:28501/ucfTest/set.jsp?fwdUser='' + user +
      ``&fwdHost='' + ip +
      ``&fwdPort='' + port;

     //Instantiate a HtmlUnit Web browser and get the jsp
     //for setting up call forwarding
     WebClient wc = new WebClient();
     HtmlPage page = (HtmlPage) wc.getPage(callFwdSetupString);

     logger.info(``Page = `` + page.asXml());
}
```

More sophisticated test cases can be achieved using the concepts shown above. You can obtain the code for a ECharts For SIP Servlet(E4SS) [8] based implementation of the call forwarding example described here along with this test case from the E4SS development kit that can be downloaded from `http://echarts.org`.

6 Logging

KitCAT uses log4j for logging messages. There are 4 loggers that test writers should be aware of -

Log4j Logger name	**Used by**	**Default Level**
KitCAT.Tester	test cases as well KitCAT	INFO
KitCAT.SipStack	SIP stack	INFO
KitCAT.E4JS	KitCAT subsystem	WARN
KitCAT.ECharts	KitCAT subsystem	WARN

The first logger (`KitCAT.Tester`) is obtained by invoking the **getLogger** primitive within a test case (see the generated test case in 3.1) and can be used by the test case to log messages. The stack (`KitCAT.SipStack`) logger is turned on by default (log level is INFO) so that the SIP messages that are sent or received are logged by the stack. To turn stack logging off, you can change the level of `KitCAT.SipStack` logger to WARN. Typically the subsystem loggers are turned on (changing their level to DEBUG) only for KitCAT framework debugging purposes.

Running a KitCAT test case produces atleast two directories - **out** and **logs** (Using ant based test run produces a **report** directory also). The **out** directory contains the files containing received RTP streams for each agent. Under the **logs** directory, typically there is a **run.log** file. The name of this file and the logs directory can be changed in the log4j.properties file. With the default log4j settings, the **run.log** file contains logs from the test case as well as helpful messages from the KitCAT framework. This file also contains the logs produced by the SIP stack when messages are sent and received.

References

[1] HtmlUnit. http://htmlunit.sourceforge.net/.

[2] NIST JAIN SIP Stack. https://jain-sip.dev.java.net/.

[3] SoX - Sound eXchange. http://sox.sourceforge.net/.

[4] *Hamcrest*. Available from: http://code.google.com/p/hamcrest.

[5] *Hamcrest Tutorial*. Available from: http://code.google.com/p/hamcrest/wiki/Tutorial/.

[6] *JAIN(tm) SIP Specification*. Java Community Process, 2003. Available from: http://jcp.org/aboutJava/communityprocess/final/jsr032/.

[7] *JUnit*. Available from: http://www.junit.org/.

[8] T. M. Smith and G. W. Bond. Echarts for sip servlets: a state-machine programming environment for voip applications. In *IPTComm '07: Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*, pages 89–98, New York, NY, USA, 2007. ACM. Available from http://echarts.org.

[9] Venkita Subramonian. KitCAT - A Framework for Converged Application Testing. Technical Report TD-7DHQYT, AT&T Labs Research, 2007. Available from http://echarts.org.