Towards Automated Functional Testing of Converged Applications

Venkita Subramonian
AT&T Labs — Research
Florham Park, NJ 07932, USA
venkita@research.att.com

ABSTRACT

There is a growing demand for IP based multimedia services that encompass usage of multiple user interfaces including web and telephony. The complexity of such converged applications requires sophisticated development tools and techniques. While standards such as SIP and HTTP Servlets enable the application developer to develop and deploy converged applications, there is a growing need for tools and techniques that can help with functional testing of converged applications. This paper makes the following contributions - (1) identifies key challenges including concurrency and coordination in functional testing of converged applications (2) describes our solution to address these challenges and (3) describes the impact of our solution based on experience gained from its use in functional testing of a real-world converged conferencing application.

Keywords

Testing, Telecommunications, VoIP applications, Converged applications

1. INTRODUCTION

Converged applications are applications whose input (output) events originate from (or are sent to) multiple sources and traverse over multiple protocols - *e.g.,* SIP [17], HTTP [16], RTP [18], email. These applications typically allow the end-user to interact with the application using multiple user interfaces - *e.g.,* web browser, telephone. One popular example of a converged application is the click-to-call feature that we often see in online stores. Customers can click on a button and specify a call back number and they get a voice connection to a customer service representative. In this example, an HTTP request over the web initiates some call-control logic which then initiates a voice session between the customer service representative and the customer, possibly through SIP. Similarly, a conferencing application could be responsible for displaying the list of current participants dialed-in to a conference in real-time. In this example, a SIP session initiates web server logic in the application that ultimately updates a web browser.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. IPTCOMM'09, July 7-8, 2009, Atlanta, Georgia, USA. Copyright 2009 ACM.

Creating and running converged applications is a non-trivial task. Typically such applications require non-trivial and often complex coordination among multiple protocol sessions. There are standards that focus on programming models and mechanisms to ease the task of developing and running converged applications. For example, the SIP Servlet standard [11, 12] defines the notion of an application session that can contain multiple protocol sessions and in effect act as a standard way to correlate and share information among multiple protocol sessions. Currently, there are several *converged* application servers with multiple protocol containers that are capable of hosting converged applications - *e.g.,* SailFin, Oracle Communications Converged Application Server, IBM Websphere. These servers take care of handling multiple protocols within a single application server instance. While such standards and tools enable the application developer to develop and deploy complex applications, there is a growing need for tools and techniques that can help with testing of converged applications. The focus of this paper is on functional testing of converged applications involving web browser and IP telephony interfaces.

The rest of this paper is structured as follows. Section 2 describes a motivating example which is a converged application with both web and VoIP interfaces and has non-trivial functional testing requirements. Section 3 identifies key challenges in automating functional tests for converged applications. Section 4 presents our solution to address these challenges. This solution was applied to automate functional testing for the application in the motivating example and this is described in Section 5. We discuss impact of our solution, its limitations and key lessons learnt from our experience in Section 6. Related work is described in Section 7 and Section 8 summarizes our conclusions and describes future work.

2. MOTIVATING EXAMPLE - A CONVERGED CONFERENCING APPLICATION

To motivate the need for an automated converged testing platform, a real-world application and one of its functional test scenarios is described here. The System Under Test (SUT) is a mission-critical IP-based conferencing application [27] that has been deployed internally in our company since 2007. This conferencing application provides a major

Step	Role	Protocol Endpoint	Activity
1	Host	Browser	Login into the conferencing application web portal
2	Host	Browser	Create a conference bridge through the web portal
3	Host	Browser	Parse the conference page and get the conference URL, bridge number, user id, user name, and host conference PIN
4	Participant	Browser	Log into the web portal and navigate to the conference URL obtained in step 3
5	Participant	Browser	Parse the conference page and get the conference bridge number, user id, user name, and participant conference PIN
6	Host	Phone	Dial the conference bridge number obtained in step 3
7	Host	Phone	Interact with IVR using touchtones - enter user id followed by PIN (user id, PIN obtained in step 3)
8	Participant	Phone	Dial the conference bridge number obtained in step 5
9	Participant	Phone	Interact with IVR using touchtones - enter user id followed by PIN (user id, PIN obtained in step 5)
10	Host	Browser	Verify whether conference page shows both parties joined to conference - both host and participant user names (obtained in steps 3 and 5) should appear in the participant list
11	Participant	Browser	Verify whether conference page shows both parties joined to conference
12	Host	Phone	Speak something
13	Participant	Phone	Verify whether audio heard
14	Host	Browser	End meeting
15	Host	Phone	Verify whether audio session ended
16	Participant	Phone	Verify whether audio session ended
17	Host	Browser	Parse the conference page and verify whether conference is in "Ended" status
18	Participant	Browser	Parse the conference page and verify whether conference is in "Ended" status

Table 1: Simple test scenario for the conferencing application

share of our company's tele-conferencing needs and handles millions of call minutes on a daily basis. The conferencing application is a converged application and offers a web portal interface for its users apart from the telephony interface. The application offers a number of features for the conference users, some of which are the following:

- Users with hosting privileges can create a conference through the web portal and this results in the creation of a conference web page and possible allocation of resources for a conference bridge; host and participant PINs for that conference are generated and displayed on the conference web page; the application also provides features to distribute the conference web page URL and PIN by email

- Users can join a conference bridge either through the telephone or by using click-to-call from the web portal

- When a conference is in progress, the conference web page is updated, in real-time, with the current status of that conference; participants in a conference can view the list of all participants currently joined to the conference; the web page also displays the name of the current speaker

- Using the web portal or the telephone interface, conference hosts can perform various tasks such as the following — termination of a meeting, audio recording of the meeting and muting/unmuting of participants; participants can mute/unmute themselves; both host and participants can upload documents and make them available to others

The conferencing application was developed in-house. The web components in the application run on standard Java Servlet and JavaServer Pages application servers and the SIP components run on standard SIP Servlet application servers. Media servers drive the Interactive Voice Response (IVR) subsystem and provide audio mixing for the conference bridges. PSTN-to-SIP gateways provide the adapter interface between PSTN and IP.

The scenario shown in Table 1 is a simple test specification that exercises the main components in the conferencing application and performs a functionality test. This is a representative test scenario that is used in manual testing of the conferencing application. It also constitutes a "sanity" test since it provides end-to-end testing of the most essential behavior of the application, as experienced by an end user.

To perform this test manually, a tester has to perform two *roles* — a host and a participant. Each role uses two different *protocol endpoint*s (PEs) - (i) a web browser handling the HTTP protocol and (ii) a phone[1] handling the SIP[2] protocol. A PE encapsulates protocol state and executes a state machine as it interacts with the SUT. For example, the SIP PE handles SIP communication with the SUT. A *test agent* (TA) is an instance of a PE for a particular role. TAs run concurrently with each other as well as with the SUT. For example, the browser used to login as host is a TA which is an instance of the HTTP PE and the phone used to dial into the conference bridge is a TA which is an instance of the SIP PE. In this paper, we

[1]In reality, this is a PSTN phone with the call routed through a PSTN-SIP gateway

[2]Throughout the discussion in this paper, we assume that the SIP PE also handles the RTP and SDP protocols

sometimes refer to a TA which is an instance of a SIP PE as a "SIP TA" and a TA which is an instance of a HTTP PE as a "HTTP TA" or simply "web TA".

Manual testing of the above scenario is a cumbersome task. First, the tester has to coordinate activities of four concurrent TAs — two different (HTTP and SIP) PEs each for the host and participant roles. For example, the tester first plays the role of a host and creates the conference using the web portal and later dials into the conference bridge playing the role of a participant.

Second, multiple web browsers would be necessary to run this test without test agents interfering with each other. This is because login credentials are stored as cookies which are shared across browser instances of the same type, forcing a tester to use a different browser type for each HTTP PE. For example, the tester has to login as the host through an Internet Explorer browser instance and as the participant through a Firefox browser instance. This is cumbersome when testing a scenario requiring multiple roles.

Third, test-related information needs to be correctly obtained from a certain TA and used for test activities on a different TA and this requires manual intervention. For example, in step (3) the conference page is parsed to obtain the conference user id and PIN which are then typed in later in step (7). In this case, test-related information is obtained from a TA — the web browser for the host — and used to drive another TA — the phone for the host.

This real-world example scenario motivates the need for a test tool that allows us to automate functional testing of converged applications. Automation of functional tests aids in the reduction of regression testing time between software releases.

3. CHALLENGES

Automation of the previously described example test scenario raises the following important challenges for a test tool from the perspective of functional testing:

Challenge 1: Emulate diverse PEs. To realize a converged test scenario such as the one described here, a test tool is minimally expected to have the following capabilities: (1) web browser emulation for web testing and (2) SIP user agent emulation for VoIP testing. For functional testing, emulation of PEs pose some non-trivial challenges. First, the test tool should be able to deal with characteristics of diverse protocols. In HTTP, an endpoint is exclusively either a client or a server – the client initiating an HTTP request and the server serving a response back to the client. In contrast, SIP is a peer-to-peer protocol where an endpoint could be both the originator and receiver of requests and responses. A single request from a SIP endpoint could result in multiple responses from the peer endpoint. Such differences in endpoint characteristics need to be reconciled in the test tool. Moreover, it is desirable to offer the test writer with easy-to-use high-level abstractions to drive a PE and hide most of the protocol details such as creating a SIP or HTTP message.

Second, the web PE may be required to handle executable content such as JavaScript[3] scripts in a retrieved HTML page. Moreover, the web PE may also be required to support asynchronous content-retrieval techniques such as AJAX[4]. The conferencing application uses both JavaScript and AJAX. For example, after a host or participant dials into the conference bridge in step (6) or (8), the conferencing page is updated automatically with the current list of participants without the user having to refresh the page. This automatic update is achieved using a combination of JavaScript and AJAX.

Third, the automated test may be required to interpret and extract information from an HTML page. For example, step (3) in the conferencing test scenario involves the tester parsing the conference page visually and extracting information such as host and participant PINs. The host and participant PINs are subsequently used in later steps. To support such automation, it is desirable for the web PE to provide abstractions to perform the following tasks: (i) interpret the retrieved HTML page, (ii) traverse the HTML page and search for specific content and (iii) extract the content.

Fourth, the SIP PE may be required to interact with an IVR. For the conferencing test scenario, both steps (7) and (9) involves the tester typing in the user id and PIN using touchtone keys in response to an IVR asking to key in this information. To automate these steps, the SIP PE should minimally support RTP-based media exchange to transmit the DTMF key information to the SUT.

Challenge 2: Balance reusability and customization of PEs. While a PE should provide abstractions that hide low-level protocol details, it should also be flexible enough to be customized where necessary. For example, a SIP PE instance may be required to send a 200 OK response automatically to an incoming INVITE while another PE instance may be required to send a 200 OK response only when directed by the test. Another scenario may require a special header to be added to a SIP message sent by a SIP PE. These slightly different scenarios should not require different PEs with different state machines. Instead a PE should provide enough abstractions so that it is reusable; however, individual instances of a PE should be customizable to fit the requirements of a test scenario.

Challenge 3: Presence of concurrency. Concurrency among TAs is desirable and necessary for certain types of testing such as load testing. For functional testing also, concurrency among TAs cannot be avoided because of the inherently concurrent and hence non-deterministic behavior of the SUT. However, uncontrolled concurrency among

[3]JavaScript is a scripting language and is widely used in web applications. JavaScript scripts can be embedded as part of an HTML page and are executed in the web browser (as opposed to the web server) triggered by user action or a timer.

[4]AJAX is a type of programming that allows web browsers to communicate with the web server in the background without reloading the page the user is currently viewing.

TAs makes the test execution non-deterministic, which is undesirable for functional testing. For example, there are execution dependencies among the TAs in the conferencing test scenario even though they run concurrently. Step (6) can be executed only after step (3) is completed and similarly step (8) can proceed only after step (5) is completed. Coordination of concurrent TAs is necessary to achieve this execution dependency. In manual testing, this coordination is done by the tester by executing activities in a certain order. Automating such coordination among concurrently running TAs is non-trivial and requires support from the test tool as well as the TAs.

Challenge 4: Automation of information exchange between TAs. In manual testing, passing information between TAs is done by the tester. For example, the userid and PINs are collected in step (3) of the conferencing test scenario from a web TA and subsequently passed to SIP TAs in steps (7) and (9). Automation of this manual information exchange process requires communication between concurrent TAs and hence requires careful coordination.

Challenges 1 and 2 relate to PEs and their capabilities, while challenges 3 and 4 relate to coordination of TAs while executing a test scenario.

4. TOWARDS A CONVERGED TESTING SOLUTION

4.1 Emulation of Protocol Endpoints

Our initial objective was to find an existing framework that addresses these challenges. We performed a survey of existing test frameworks that are available for automated web as well as SIP testing.

There are many tools, both free and commercial products, available for testing web-based applications [29, 10]. We found HtmlUnit to satisfy the web testing capabilities that we needed. HtmlUnit is an open-source Java-based web testing framework with a rich set of high-level primitives for navigating web pages and traversing/searching HTML page content. These primitives include abstractions for a web browser, HTML pages, elements within a page such as forms, anchors, *etc.* It provides support for XPath [30] based search within an HTML document node. It has excellent support for Javascript and AJAX also. Moreover, it has an active open source development community.

Following is a simple illustration[5] of how HtmlUnit primitives can be used to automate web page retrieval, content search and navigation.

```
WebClient browser = new WebClient();
HtmlPage homePage = browser.getPage(``http://conf.app.url'');
HtmlAnchor anchor = homePage.getAnchorByHref(
        ``UpdateAction.do?action=createAdHocMeeting'');
HtmlPage page = anchor.click();
```

The code above uses HtmlUnit API (highlighted in bold) to fetch the conferencing web page, find the link to create a specific type of meeting, called ad-hoc meeting, on the conferencing application home page. It then clicks on that link to create the meeting. HtmlUnit offers a high-level abstraction for an HTML page and a rich API to traverse and perform actions on the content.

[5]For clarity reasons, we have omitted many details that are not necessary for this discussion.

login	Login to the conferencing application web portal
createAdhocMeeting	Create a particular type of meeting by emulating a click on the appropriate link on the conferencing home page
getLoginUserName getMeetingName getBridgeNumber getPhoneUID getHostPin getParticipantPin getMeetingURL	High-level primitives to obtain information from the conference web page
endMeeting	End the meeting from the web by emulating a click on the appropriate link in the conference web page

Table 2: Conferencing application-specific web testing primitives

We have developed high-level wrapper primitives (see Table 2 for a sample) for web testing that are specific to the conferencing application. These primitives can be reused across test cases within the conferencing application, and improves test case readability. These primitives hide details of searching and traversing HTML content within retrieved web pages and enable the test writer to focus on high-level application-specific testing tasks. For example, the details related to searching for a specific anchor to create a meeting can be abstracted away in a **createAdhocMeeting** primitive.

There are many tools available for SIP testing [20] as well. However, they are limited with respect to the *functional* testing capabilities that we were looking for. For example, SIPp is a very popular and versatile tool used in SIP testing in a variety of different ways including protocol testing and load testing. It has various capabilities to assist in load testing such as dynamically adjustable call rate, dynamic display of statistics about running tests, *etc.* . However, it is cumbersome (not impossible) to use for functional testing of a converged application. In fact, originally we had partly automated the SIP part of the conferencing application test scenario using SIPp. Tests in SIPp are written in XML with a test script for each TA. The test scripts contain commands to send/receive SIP messages which are included as part of the test script. Although this is a suitable test script format for low-level testing such as SIP protocol-compliance testing, high-level primitives that hide the SIP messaging details are more suitable for functional testing. The script-per-TA approach makes it difficult to address Challenge 2 since two different TAs with slightly different behavior may require two different scripts. Moreover, to simulate keypad based IVR interaction (Challenge 1) in SIPp, one has to first perform this interaction manually and record the RTP audio session (using packet capture tools such as wireshark)

Primitives supported by SIP PE	
setProxy	Set the URI of the SUT so that all SIP messages from the PE are sent to the SUT
register	Send a REGISTER request and handle authentication
call	Send an INVITE request
answer	Send a 200 OK response with SDP to an incoming INVITE
sendResponse	Send a response to an incoming request
end	Send a BYE
cancel	Send a CANCEL request
callNoSdp	Send an INVITE request without SDP
reinvite	Send a mid-dialog INVITE request
info	Send a INFO request
sendDTMF	Send DTMF keys via RTP. uses RFC4733-defined payload format
enableAutoAnswer	Setup a PE so that an incoming initial INVITE message is responded with a 200 OK response without explicit instruction from the test case
playAudio	Play audio from a specified disk file
addMessageModifier	Allows user-specified modification (*e.g.,* add a header) to messages
Test Case Primitives	
createAgent	Create a SIP/RTP endpoint
releaseAgent	Release resources for the specified SIP/RTP endpoint
assertThat	Assert conditions on state of a PE or messages exchanged by a PE
pause	Pause test execution for a specified duration while SIP messages are processed in the background
processDeferredSIP	Process queued and newly arriving SIP messages for all agents for the specified duration. Enables fine-grained concurrency control in the test system.

Table 3: KitCAT primitives

between a VoIP agent and the SUT. This packet capture file contains the key entries by the user and is then later used as part of a SIPp script to replay the keypad entries. In the conferencing example scenario, it is difficult to capture and replay the conference PINs for IVR interaction since the PINs are generated dynamically and need to be exchanged between TAs (Challenge 4).

To address our functional test challenges, we developed the KitCAT (Kit for Converged Application Testing) framework, which is a Java-based library and released open source (`http://echarts.org/KitCAT`). Our converged test solution is based on a combination of KitCAT and HtmlUnit.

The SIP PE in KitCAT provides the necessary abstractions for exercising the SUT using SIP. It also maintains a state which is advanced appropriately as messages are sent and received. Table 3 shows some of the key primitives supported by the SIP PE in KitCAT. These are high-level primitives that hide a lot of protocol details. The SIP PE provides the infrastructure necessary for supporting these high-level primitives. It implements the mechanisms necessary to create and send/receive SIP messages.

The KitCAT SIP PE has been designed with the following goals — it is (1) reusable across tests, (2) customizable for an individual test, if necessary and (3) amenable to test coordination (explained later in Section 4.2). The behavior of the SIP PE can be customized as part of an individual test. For example, the **enableAutoAnswer** primitive (see Table 3) customizes the behavior of the PE so that it automatically sends a 200 OK response to an incoming initial INVITE.

The SIP PE in KitCAT also allows the test writer to send DTMF key sequences, specified as a string, to the SUT. It converts the string sequence into appropriate RFC4733-based [19] RTP packets and sends them to the SUT. This provides a very convenient mechanism for the test writer to interact with SUTs that involve DTMF-based interactions with a media server (*e.g.,* Interactive Voice Response systems). SIP PEs can also be directed to send audio from a disk file. They also store received RTP packets onto a disk file which can later be converted to audio files that can be heard using an audio player.

4.2 Concurrency Control

Concurrency (Challenge 3) is controlled in KitCAT using a test coordinator. There is a single instance of a test coordinator for each test and this coordinator coordinates the actions of multiple TAs. Presence of a coordinator is essential for running tests in KitCAT. The test coordinator itself executes sequentially instructing TAs to perform actions and therefore provides an intuitive way to express a functional test scenario. A test writer writes a KitCAT *test case* that typically consists of three kinds of statements - (1) coordination commands, (2) pause statements to introduce delays and (3) assertion statements to validate conditions on TAs. A coordination command instructs a TA to perform an action that could result in the TA sending messages to the SUT. A pause statement introduces delay in test execution, so that state changes may occur. The test case can also validate expected conditions on TAs based on a snapshot of TA states. A KitCAT test case thus focuses on coordination of TAs and assertion of results rather than protocol details.

Since TAs run concurrently with the coordinator, their coordination requires support from their underlying PEs.

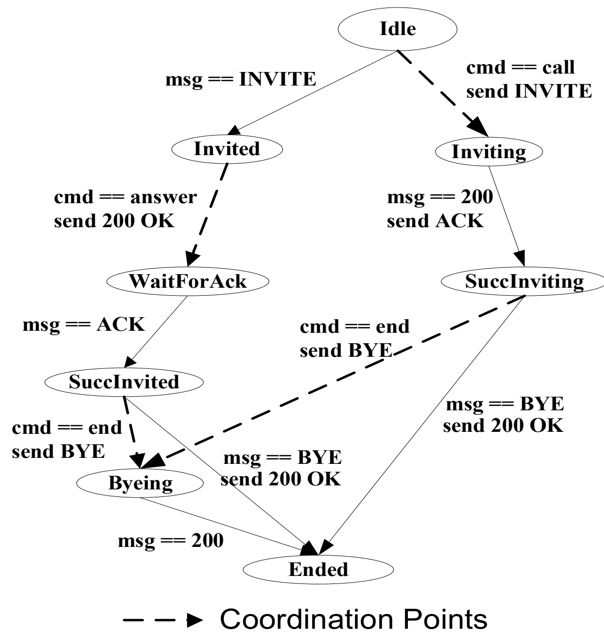

Figure 1: Simplified machine illustrating SIP PE behavior

This is done through coordination points in the PEs.

4.2.1 *Coordination Points*

Figure 1 shows a very simplified version of the KiTCAT SIP PE using a state machine representation. We use this state machine here for illustrative purposes only; the actual state machine is more comprehensive. The PE reacts to two external events — (1) arrival of SIP messages for that PE and (2) coordination commands from the test case. Both these events may cause a PE to take certain actions such as sending a SIP message, and then transition to a possibly different state. For example, in Figure 1, a SIP PE is initially in the **Idle** state. In this state, if the PE receives an INVITE message, it transitions to the **Invited** state. SIP response messages are also processed in a similar manner. These states are high-level states used for testing and control; they do not necessarily reflect the states of a SIP dialog as defined in the SIP specification [17]. PEs also send certain responses by default. For example, a BYE or INFO is automatically responded with 200 OK. However, this default behavior can be changed by special configuration commands which allow the test case to override (*e.g.,* send an error response) these default actions of the PE.

To allow coordination by a test case, there are several coordination points (shown by dotted lines) in the PE. These coordination points represent execution points in the PE where the PE may expect to receive coordination commands (*e.g.,* call, answer) from the test case. For example, in the **Idle** state, if the test case instructs a PE to make a call (using the **call** command), the PE sends an INVITE and transitions to the **Inviting** state. Similarly, in the **Invited** state, an **answer** command from the test case causes the PE to send a 200 response.

Coordination points thus allow the test case to control when certain SIP messages can be *sent* to the SUT by the PE.

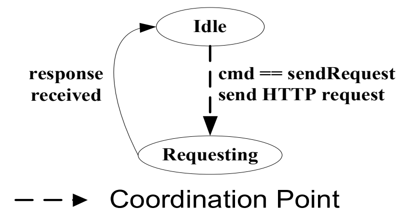

Figure 2: HTTP PE state machine

However, SIP messages are still received and processed by the PEs. While processing a received SIP request, the PE could hit a coordination point in its execution which then causes the PE to wait until it receives a coordination command from the test case. A simple KitCAT test for a B2BUA application with a caller and callee can be easily written (KitCAT primitives in bold) as shown below. Pause statements allow for PE execution and state change as well network delays.

```
SIPAgent caller = createAgent(‘‘Alice’’);
SIPAgent callee = createAgent(‘‘Bob’’);
caller.setProxy(‘‘sut.appserver.com’’);
caller.call(callee);
pause(1000);
callee.answer();
pause(1000);
assertThat(caller, connected());
assertThat(callee, connected());
pause(1000);
caller.end();
pause(1000);
assertThat(caller, disconnected());
assertThat(callee, disconnected());
```

Coordination points also allow separation of concerns between the test case and the PE - test case focuses on coordination and the PE focuses on handling protocol details. Moreover, the concept of a single test case hosting multiple TAs allows easy information exchange between TAs (Challenge 4). The test case obtains information from one TA and then subsequently passes it to another TA through the coordination commands.

So far, our discussion focused on coordination points in the SIP PE. Conceptually, the state machine in the HTTP PE is trivial (Figure 2) when compared to the one in the SIP PE. Only one coordination point is needed in the HTTP PE and this coordination point is hit when the test case invokes a HtmlUnit primitive to fetch a page. This results in an HTTP request getting sent to the SUT. With HtmlUnit, the test case in KitCAT blocks until the HTTP response is received.

4.2.2 *Deferred SIP Message Processing*

While coordination points allow control over when certain messages are *sent* from a PE, it may be necessary to control the processing of received SIP messages in certain cases such as testing a race condition. Let us imagine a scenario in the conferencing application where the meeting is ended from

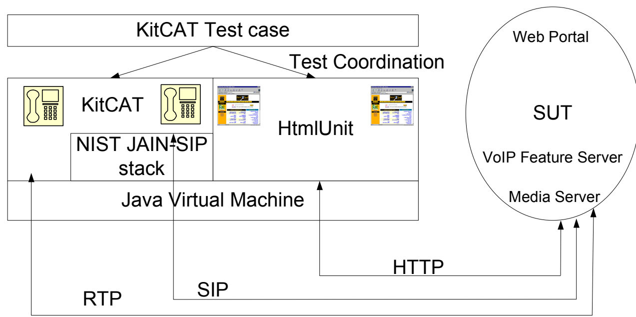

Figure 3: KitCAT architecture to test converged applications

both the web and the VoIP endpoints. The purpose of the test is to determine if the SUT still behaves correctly while simultaneously ending the meeting on the web portal and disconnecting the participants from the conference. It may be difficult to do this test manually because of the inherent concurrency involved. The web portal sends a message to the SUT causing it to send BYE messages to all the participants in the meeting. Concurrently, the participants send BYE messages to the SUT. The test case should be able to coordinate the test in such a way that the BYEs from the SUT cross over with the BYEs to the SUT.

With coordination points alone, the test case does not have control over a TA as to when it processes the BYE request received from the SUT. The PE processes the BYE request immediately upon receipt executing concurrently with the test case coordinator. It then automatically sends a 200 OK response back to the SUT. The test case cannot deterministically simulate BYEs crossing over since it has to instruct the PE to send a BYE to the SUT *before* the PE automatically responds with the 200 OK to the incoming BYE.

KitCAT provides mechanisms to control the processing of received SIP messages to address the above issue. A test case can turn on this mechanism and explicitly invoke the **processSIP** primitive to specify when TAs can process received SIP messages. All SIP messages arriving at a TA are queued for processing at a later time. These messages are processed by TAs when the test case calls **processSIP**. The test writer should be careful to invoke this primitive often so that SIP message processing can take place. The deferred SIP processing mechanism thus provides additional control over SIP TAs.

4.3 Test Environment Architecture

Figure 3 shows the test environment architecture in KitCAT to test converged applications. A test case creates TAs which are then used to communicate with the SUT. A test case can control execution of TAs and validate conditions on the state of TAs to determine whether they are in expected states during the course of a test. A TA can act as either a caller initiating calls to the SUT or a callee receiving calls from the SUT. A KitCAT test case thus hosts both caller and callee TAs within a single process thus facilitating their control.

Although a KitCAT test case imposes deterministic ordering

```
//Create SIP endpoints for host and participant
SIPAgent hostVoIP = createAgent("Host");
SIPAgent partVoIP = createAgent("Part");

//Create browser endpoint and login
hostBrowser = new TMeetBrowser();
hostHomePage = hostBrowser.login(hostId, hostPasswd);

//Create a particular kind of meeting
//using the web portal
hostMeetPage = hostHomePage.createAdhocMeeting();

//Parse HTML content to get data to
be used later in the test
hostUserName = hostMeetPage.getLoginUserName();
meetingName = hostMeetPage.getMeetingName();
bridgeNum = hostMeetPage.getBridgeNumber();
hostUid = hostMeetPage.getPhoneUID();
hostPin = hostMeetPage.getHostPin();
partPin = hostMeetPage.getParticipantPin();
meetUrl = hostMeetPage.getMeetingURL();

//Participant logs in to portal using the newly
//created conference url obtained in the previous step
partBrowser = new TMeetBrowser();
partMeetPage =
  partBrowser.loginToMeeting(
     partId, partPasswd, meetUrl);

//Get information from the web page to be used
//later in the test
partUid = partMeetPage.getPhoneUID();
partUserName = partMeetPage.getLoginUserName();

//Initiate VoIP calls to the SUT and assert
//whether the endpoints are connected to the SUT
hostVoIP.call("sip:" + bridgeNum + "@" + sipServer);
partVoIP.call("sip:" + bridgeNum + "@" + sipServer);
pause(10000);
assertThat(hostVoIP, is (connected()));
assertThat(partVoIP, is (connected()));

//Endpoints interact with initial IVR
hostVoIP.sendDTMF(hostUid + "#");
partVoIP.sendDTMF(partUid + "#");
pause(3000);
hostVoIP.sendDTMF(hostPin + "#");
partVoIP.sendDTMF(partPin + "#");

pause(20000);

//Assert endpoints are still connected to SUT
assertThat(hostVoIP, is (connected()));
assertThat(partVoIP, is (connected()));
pause(30000);

//Assert that web browser has been updated with
information about newly joined host and participant
assertThat(hostMeetPage,
            has (allOf (
                   participantJoined(hostUserName),
                   participantJoined(partUserName))));
assertThat(partMeetPage,
            has (allOf (
                   participantJoined(hostUserName),
                   participantJoined(partUserName))));

pause(3000);

//Assert audio connectivity
hostVoIP.playAudio(‘‘audio.raw’’);
pause(2000);
assertThat(partVoIP, has (incomingMedia()));

//Host ends meeting through web
hostMeetPage.endMeeting();

pause(30000);

//Assert meeting has ended on web
assertThat(hostMeetPage, has (ended()));
assertThat(partMeetPage, has (ended()));

//Assert meeting has ended on the SIP side
assertThat(hostVoIP, is (disconnected()));
assertThat(partVoIP, is (disconnected()));
```

Figure 4: Excerpt from conferencing application test case

of test activities within a single test case, multiple test cases (same or different) can still be run concurrently using standard concurrency mechanisms in Java such as threads. This makes KitCAT useful in load testing of applications. Moreover, running concurrent tests allows non-determinstism in the test environment which may be important for exposing concurrency bugs in the application under test.

KitCAT uses the JAIN-SIP [9] compliant SIP stack implementation from NIST [4] and ECharts [6, 5] for implementing the PE state machine. KitCAT integrates well with JUnit [13] which is the de-facto standard for testing Java applications. A KitCAT test case can be run using a JUnit runner and the tester can get all the advantages that one gets with using JUnit for running tests. For example, the tests can be run from within an Eclipse [1] environment using a JUnit plugin. This is very useful for functional testing during application development.

5. AUTOMATED TEST SOLUTION FOR CONFERENCING APPLICATION

The combination of KitCAT and HtmlUnit provided us with a converged test tool for functional testing of the conferencing application. Figure 4 shows an extract from the actual test case that implements the test scenario in Section 2. The actual test case mimics the scenario specification closely. The test case executes sequentially coordinating the web and SIP TAs. Protocol handling and the associated concurrency details are hidden in the TAs allowing the test writer to focus on test coordination.

Condition checks are performed on TAs using the extensible assertThat API in JUnit. This method takes in two parameters - (1) the object on which a condition has to be checked and (2) the condition to be asserted. If an assertion fails, then an exception is thrown and the test execution ends.

Figure 5 roughly shows the sequence of interactions between the test case and the converged conferencing application. This is a simplified picture and, for clarity, omits many messages that are exchanged in the real conferencing application. The sequence diagram shows the clear separation of concerns between the PEs and the test case - the PEs dealing with protocol details and the test case dealing with coordination of the test.

To test the race condition involving crossing BYEs, the test case should change as follows towards the end, apart from enabling the deferred SIP processing mode.

```
hostMeetPage.endMeeting();
hostVoIP.end();
partVoIP.end();

processDeferredSIP(30000);

//Assert meeting has ended on web
assertThat(hostMeetPage, has (ended()));
assertThat(partMeetPage, has (ended()));

//Assert meeting has ended on the SIP side
assertThat(hostVoIP, is (disconnected()));
assertThat(partVoIP, is (disconnected()));
```

Figure 6 shows the execution sequence in the deferred processing mode. The key difference from Figure 5 is that the incoming BYE from the SUT is *queued* on arrival and processed only when **processDeferredSIP** is invoked in the test case. Using the deferred processing mode thus provides the test writer with more control over execution of TAs. The BYE from the SUT (triggered by meeting termination from the web) is queued and before it is processed a BYE is initiated from the endpoint. The processing of the BYE and the resulting 200 OK response occurs only when **processDeferredSIP** is called.

6. IMPACT, LIMITATIONS AND LESSONS

The combination of KitCAT and HtmlUnit has provided us with a powerful automated test solution for converged applications such as the conferencing application. Automated sanity tests are run frequently (every 5 minutes 24×7) for our conferencing application. These functional tests are run directly against each production SIP application server so that faults on a specific server can be detected rapidly even in the presence of a load-balancer. Test calls are also placed through a PSTN gateway, which simulate users dialing in. The results of these tests are collected and published on an internal website. Apart from PASS/FAIL results for each test, this site includes SIP message logs as well as audio that were received by the SIP PEs during the test. These logs are helpful for diagnostic purposes in case of test failure. This web site is constantly monitored by our production support staff. Alarms are triggered based on reported failures so that preemptive action can be taken. The periodic automated testing has thus become a critical part of production monitoring of the conferencing software. Apart from being used as a tool for production monitoring, we have used this tool combination for regression and load testing the conferencing application before major releases of the application software.

Apart from the conferencing application, we have used this combination successfully in some internal projects - (1) unit testing VoIP features created using our open source service creation frameworks [25, 14], (2) testing a VoiceXML browser using VoiceXML [28] conformance tests. In (1), regression test suites using KitCAT and HtmlUnit have cut down unit testing efforts substantially. In (2), more than 700 tests have been developed using KitCAT. According to the personnel working on this project, it took approximately a week to manually run these tests whereas the automated test cases take about 8 hours to run.

So far, there have been over 1000 known test cases that have used this solution. These test cases are used in regression-testing applications as part of multiple internal projects. This converged testing toolset has thus provided us a powerful solution resulting in (1) increased effectiveness and (2) lowered cost due to the use of open source methods. We intend to conduct further user studies to evaluate the impact of our solution on reduction of testing time in an entire project life-cycle.

Benefits. The key benefits of our approach are the following: (1) Separation of concerns - test writer focuses on orchestration and test endpoints focus on handling protocol

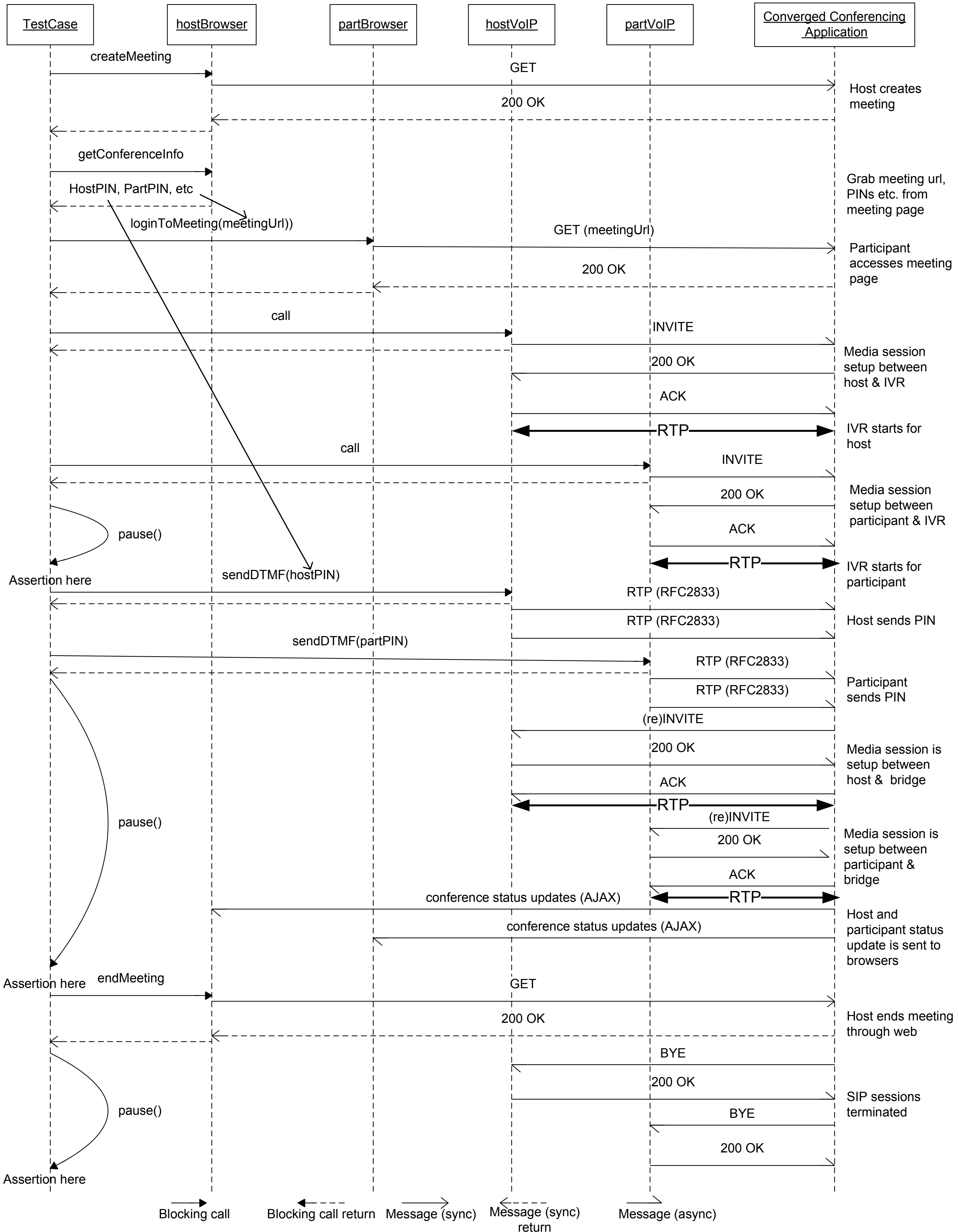

Figure 5: Approximate sequence of events in the conferencing application test case

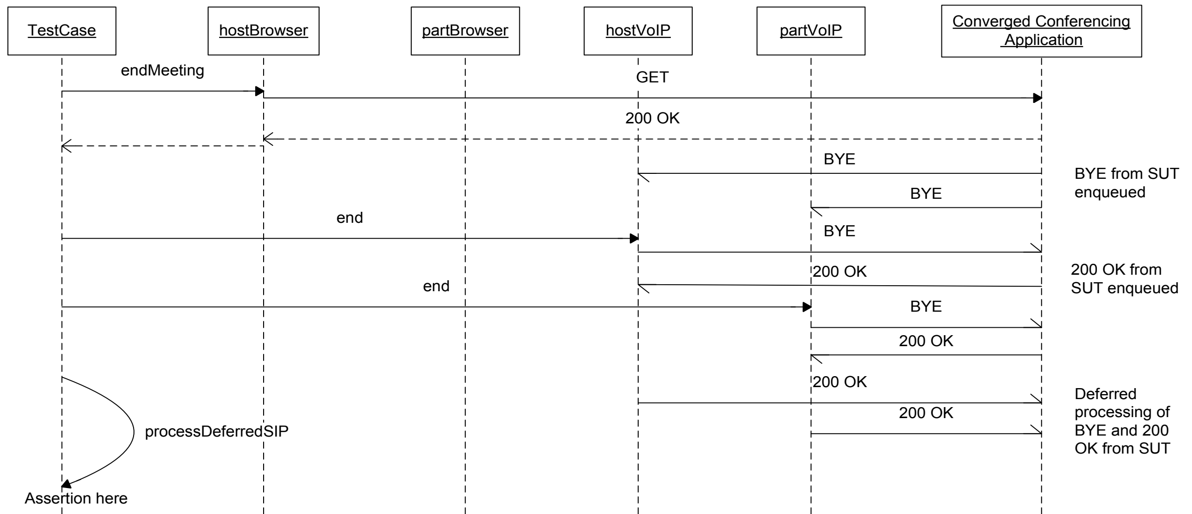

Figure 6: Testing for crossing BYEs in the conferencing application

details (2) Reusability of endpoints with customization capabilities (3) Single test case that coordinates the test and controls concurrency (4) Java based solution and hence runs on a wide variety of platforms; support tools for Java-based development are applicable to test development also; test can include other protocols/activities (*e.g.,* database access) that can be done using Java (5) Open source availability of both KitCAT and HtmlUnit; all libraries that both these frameworks depend on, are open source and freely available to use.

Limitations. Some of the limitations of our approach are the following: (1) There are several existing test tools intended for different protocols. Our approach is not a general solution to compose arbitrary test tools to form a converged test tool. KitCAT and HtmlUnit are both Java-based and that commonality helped us avoid challenges such as reconciling differences in test scripting language, semantics of execution and output format for test results. However, it is an interesting area of research to consider standardizing coordination mechanisms that work across different test tools. (2) Our approach is software-based and hence cannot be used directly to test hardware endpoints such as a media processing board connected to the PSTN. (3) Our approach is intended mainly for functional testing and may not be suitable for other types of testing such as protocol testing. (4) Some limitations with our current implementation are - (a) it is restricted to a Java-based runtime environment, (b) a test case has to introduce explicit time delays for state changes to occur in the test system; it is more desirable to wait on certain conditions to occur in the test system rather than fixed time delays (c) no support for the following SIP messages - SUBSCRIBE/NOTIFY, PUBLISH, MESSAGE, REFER, UPDATE, (d) audio codec is restricted to G.711 μ-law. We plan to address most of these implementation limitations in future releases.

6.1 Lessons Learnt

Application-specific web testing primitives aid re-usability and readability. Even though HtmlUnit provides rich content search capabilities within an HTML document, it is useful to have application-specific primitives that hides details about *how* a particular functional testing task — *e.g.,* click on a particular meeting, upload a shared document, mute a particular participant, check if a participant is shown as joined to a conference — is achieved. The advantages of developing these primitives are: (a) while these primitives may not be reusable across applications, they can be reusable across different test cases for the same application; and (b) better test case readability. Based on the current experience, we believe that the utility of higher-level wrapper primitives is best decided on an application-by-application basis.

Testing-oriented hints in web pages aid in functional testing of applications. Relying on web page content for our testing has two inherent challenges — (a) tests may rely on implementation details such as the structure of the HTML document in a web page and (b) they may be hard to write manually since it would require understanding the structure of the content within application web pages. For example, in our conferencing application tests, we use XPath [30] search expressions to determine the existence of a particular HTML image element in the document hierarchy of a meeting web page indicating that a participant has joined a conference. Such dependency may make these tests fragile in the face of implementation changes in the structure or content of the concerned HTML documents. To reduce the impact of implementation changes, it would be helpful to incorporate "hints" as part of web page content that would aid testing. For example, the meeting web page could embed a hint in the form of an invisible HTML table that contains the state information of each participant. Both the web page presentation logic and the tests would then depend on these hints. Further any changes to the hints would immediately, and possibly automatically, identify possible changes needed in the tests. This will work only if one has control over the source code for the application. We plan to explore this

topic further as part of future research.

Test generation from a higher-level test specification aids better adoption among testers. Some of the internal projects (*e.g.,* VoiceXML browser project), that use KitCAT, have been using test generation techniques, though adhoc, to generate KitCAT test cases. Further work is needed with respect to a standard way to specify these tests and generate executable code based on KitCAT and HtmlUnit. This enables testers to write higher-level test specifications, as opposed to Java code, that are translated to converged test cases.

7. RELATED WORK

Two areas that are closely related to our work are: (1) web testing and (2) SIP testing. A variety of different frameworks assist testing in each of these areas.

Web testing. A number of frameworks [29, 10] are available for web testing - *e.g.,* HtmlUnit [8], HttpUnit [3], Canoo Web Test [7]. These frameworks complement KitCAT since KitCAT does not provide any facilities for web testing, but can co-exist with these frameworks to provide a converged application functional testing tool.

SIP testing. There are many tools available for SIP testing [20] including (SIPp [21], SipUnit [24], sipsak [23], PROTOS [15] and SIPr [22]). Among these tools, SIPp, SipUnit and SIPr can be used for functional testing of SIP applications. SIPp [21] is an open source tool used for various kinds of testing — *e.g.,* performance testing, load testing, protocol testing — on SIP applications. However, it is cumbersome to do functional testing of SIP applications. The test scripts are XML based; they consist of SIP protocol messages or message templates. For functional testing, it is desirable to have high-level abstractions that hide protocol details. Moreover, simulating multiple user agents involves writing multiple scripts and running multiple OS processes. In contrast, KitCAT offers higher-level abstractions to test writers for functional testing. On the other hand, KitCAT may not be suitable for certain kinds of testing for which SIPp may be more appropriate — *e.g.,* sending an ill-formed SIP message.

We found SipUnit [24] to be the closest in terms of being able to host multiple endpoints within the same test case. However, SipUnit does not have support for RTP, which made it an unsuitable candidate to automatically interact with an IVR which was one of our requirements. Moreover, the concurrency control primitives in KitCAT allow better control over the test. On the other hand, SipUnit has support for testing Presence applications which KitCAT does not currently support.

SIPr is a Ruby-based testing tool that was released in November 2008 and may be used for converged application testing. Even though a core part of SIPr has been released open source, enhanced functionality is available only with a commercial license. We believe that concurrency control primitives in KitCAT are not natively available in SIPr. We plan to conduct further comparative studies with our approach with respect to reusability of endpoints and test coordination. SIPr provides tools that generate a test based on SIP messages from a packet capture file, while KitCAT does not support this capability.

Commercial products that are available for performing application testing are Hammer G5 [2] (from Empirix) and TTCN-3 [26] based products. Hammer G5 offers a complete and robust testing environment with various analysis tools for various VoIP signaling protocols including SIP. Hammer offers a proprietary scripting language called Hammer Visual Basic that is used to develop test scripts. It also offers a graphical environment for the user to specify test scenarios and generate these test scripts. Although Hammer offers web testing suite of products, it is not clear as to how well they co-exist with their VoIP counterparts to do functional testing of converged applications. Moreover, it is not clear whether there is any scope for leveraging existing well established web testing frameworks.

We believe the learning curve for TTCN-3 is higher and moreover, that standard forces us to create TTCN-3 based wrapper layers for the different protocol sessions and for each of these layers, we need to develop encoding and decoding adapters. We could not find any case studies where TTCN-3 has been used for testing converged applications. On the other hand, KitCAT uses a popular language (Java) and enables one to leverage already existing and well established frameworks for web testing. Finally, the converged test platform based on KitCAT and HtmlUnit is an open source solution.

8. CONCLUSIONS AND FUTURE WORK

Converged applications are gaining increasing significance in the context of convergence of web and telephony. In this paper, we have identified key challenges including concurrency encountered in automated functional testing of converged applications. We presented a solution towards automated functional testing of converged applications involving web and SIP events. We presented benefits and limitations of our approach and our successful experience in the context of a real-world converged conferencing application. Lessons learnt from our experience may be applicable to the broader community involved in converged application development, testing and deployment.

The following are interesting areas for future work that we plan to pursue. First, automatically generating tests based on analysis of a high-level application specification will be a valuable aid in service creation and testing. A KitCAT test case is typically linear and its purpose is to test one execution path among the many possible execution paths of the SUT. Based on analysis of a SIP application created using high-level service-creation languages such as StratoSIP [14], it may be possible to automatically generate KitCAT test cases covering multiple execution paths in the application. Second, we plan to work on supporting test primitives for Presence and instant messaging applications. Third, we plan to explore high-level test specification capabilities, as opposed to writing a test case in Java, in the future.

Acknowledgments

The author would like to gratefully acknowledge the valuable contributions of colleagues - Gregory Bond, Eric Cheung, Danilo Giulianelli, Gerald Karam, Hal Purdy, Thomas Smith and Pamela Zave.

9. REFERENCES

[1] Eclipse. http://www.eclipse.org/.

[2] Hammer G5. http://www.empirix.com.

[3] HttpUnit. http://httpunit.sourceforge.net/.

[4] NISTSipStack. https://jain-sip.dev.java.net/.

[5] G. W. Bond. An Introduction to ECharts: The Concise User Manual. Technical Report TD-6NKLR2, AT&T, 2006. Available from: `http://echarts.org`.

[6] G. W. Bond and H. H. Goguen. ECharts: balancing design and implementation. In M. Hamza, editor, *Proceedings of the 6th IASTED International Conference on Software Engineering and Applications (SEA 2002)*, pages 149–155. ACTA Press, 2002. Available from: `http://echarts.org`.

[7] Canoo Web Test. http://webtest.canoo.com/.

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

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

[10] Java-based web testing tools. http://java-source.net/open-source/web-testing-tools.

[11] *JSR 116: SIP Servlet API Version 1.0*. Java Community Process, 2003. Available from: `http://www.jcp.org/aboutJava/communityprocess/final/jsr116`.

[12] *JSR 289: SIP Servlet Version 1.1*. Java Community Process, 2008. Available from: `http://jcp.org/en/jsr/detail?id=289`.

[13] JUnit. http://www.junit.org/.

[14] Pamela Zave, *et al.*. Abstractions for Programming SIP BacktoBack User Agents, 2009.

[15] PROTOS Test-Suite. http://www.ee.oulu.fi/research/ouspg/protos/testing/c07/sip/.

[16] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Standards Track RFC 2068, Network Working Group, Jan. 1997. www.w3.org/.

[17] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. 2002. IETF RFC 3261.

[18] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-time Applications. *RFC 1889*, Jan. 1996.

[19] H. Schulzrinne and T. Taylor. Rtp payload for dtmf digits, telephony tones and telephony signals. *RFC 4733*, Dec. 2006.

[20] SIP testing tools. http://www.cs.columbia.edu/sip/implementations.html.

[21] SIPp. http://sipp.sourceforge.net/.

[22] SIPr. http://sipper.agnity.com/.

[23] sipsak. http://sipsak.org/.

[24] SipUnit. http://www.cafesip.org/projects/sipunit/.

[25] 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.

[26] TTCN-3. http://www.ttcn-3.org/.

[27] Venkita Subramonian, Eric Cheung, and Gerald Karam. Automated Testing of a Converged Conferencing Application. In *Proceedings of the 31st International Conference on Software Engineering (ICSE), Fourth International Workshop on the Automation of Software Test (AST)*, 2009.

[28] VoiceXML Forum. http://www.voicexml.org.

[29] Web testing tools. http://www.softwareqatest.com/qatweb1.html.

[30] XPath. http://www.w3.org/TR/xpath20/.