KitCAT - A Framework for Converged Application Testing

Venkita Subramonian
AT&T Labs Research
180 Park Avenue
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 require sophisticated development tools and techniques and as a result a variety of technologies and standards have been developed or are in the process of development to address the complexity of application development and deployment. For example, application servers implementing the SIP servlet standard [18] [20] aids the application developer with development of SIP based applications and this, in conjunction with HTTP servlet standard [19], aids the development of converged applications based on the HTTP [22] and SIP [23] protocols.

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 application developers to do system level functional testing of converged applications. This paper makes the following contributions - (1) describes a conceptual model for testing converged applications (2) introduces a simple programming model for test case writers to exercise and test converged applications and (3) describes our converged application testing framework KitCAT that is a concrete implementation of (1) and (2).

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 [23], HTTP [22], RTP [26], email. 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 SIP session could initiate web server logic in the application that ultimately updates a web browser. For example, a conferencing application could be responsible for displaying the list of current participants in a conference in real-time.

Creating and running converged applications is a non-trivial task. Typically such applications require non-trivial and often complex coordination among multiple protocol sessions in the application. Efforts are underway to incorporate standard programming models and mechanisms in application servers that ease the task of developing and running converged applications. For example, the SIP servlet standard [18, 20] defines the notion of an application session (**javax.servlet.sip.SipApplicationSession**) that can contain multiple protocol sessions and in effect act as a standard way to correlate and coordinate between multiple protocol sessions. Currently, there are several *converged* application servers with multiple protocol containers that are capable of hosting converged applications - *e.g.,* SailFin, BEA WebLogic, 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 application developers to do system-level testing of converged applications.

1.1 Challenges

An important challenge to testing converged applications is to be able to deal with multiple protocols and provide appropriate abstractions for simulating these protocol endpoints and achieve coordination among them. These protocols may have different endpoint characteristics. For example, 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 a converged application testing framework and the test writer presented with an easy-to-use programming model for testing converged appli-

cations.

To choose an appropriate programming model for a converged application testing framework, there are two important challenges that need to be resolved - (1) concurrency and coordination among protocol endpoints (2) asynchronous arrival of protocol messages from the system under test (SUT) when the test case is not explicitly blocked waiting for messages. (1) is important from the perspective of testing since we need to be able to simulate multiple endpoints accessing the system under test concurrently or in a pre-determined sequence. For example, load testing could involve several endpoints accessing the SUT concurrently, whereas functional testing could involve coordination of multiple endpoints accessing the SUT in a pre-determined order. The testing framework needs to support the notion of multiple protocol endpoints accessing the SUT concurrently as well as achieve coordination among endpoints. (2) is an important factor that needs to be considered in the context of protocols where there is a possibility of messages arriving in the background when the test itself is not blocked waiting for incoming messages - *e.g.,* SIP re-INVITE or a BYE sent from the SUT. The testing framework needs to handle such messages and provide opportunity for the test case to respond to these messages. The test framework should thus support both types of message arrivals from the SUT - (1) synchronous, when the test case is blocked waiting for a message that it expects - *e.g.,* an HTTP response to an HTTP request that the test case had previously sent and (2) asynchronous, when the arrival of a message is not necessarily anticipated by the test case - *e.g.,* a SIP re-INVITE or BYE from the SUT.

1.2 Solution Exploration

There are different solutions addressing the challenges above. Concurrency can be achieved by having an endpoint execute in its own thread or process. This is the approach taken in SIPp [7] (where each endpoint runs as an operating system process) and TTCN-3 [9] (where each endpoint can be mapped to a Parallel Test Component). The advantage here is that each endpoint can run independent of other endpoints - block/resume execution without affecting execution of other endpoints. But the disadvantage is that coordination of endpoint executions poses a major challenge requiring the test case to deal with synchronization of threads/processes (or parallel test components in TTCN-3) which is difficult to implement in practice and hence undesirable from the perspective of a test writer. Another solution is to have a single thread of execution for a test case hosting multiple endpoints. This makes the coordination of the endpoint executions easier than in the previous approach. Multiple endpoints share the same thread of execution and hence thread/process synchronization mechanisms are not necessary for coordination. Moreover, this approach provides a more deterministic test execution in the absence of multiple threads within a test case.

To process messages arriving from the SUT, the test writer may be provided with a blocking or a non-blocking API in the programming model. From the perspective of the test writer, a blocking API is easy to understand since there is a sequential flow of execution control as opposed to a non-blocking API which typically involves flow of execution control to call-back event handlers intended for processing messages arriving in the background. It is desirable to encapsulate the handling of such message arrivals within the testing framework and provide the test writer with a blocking interface to process these messages in a deferred manner. This approach has been documented in the Half-Sync/Half-Async [25] architectural pattern which is being used in many real-world software systems. This allows the test writer to handle multiple protocol messages in a uniform fashion - by explicitly blocking to process messages - irrespective of the nature of their arrival.

Our overall objective was to find a framework that enables us to test converged applications that involve both web and VoIP events. We identified a brief set of capabilities/requirements that we wanted such a testing framework to satisfy:

- Multiple test endpoints simulating SIP user agents can be hosted in one single operating system process to facilitate easy coordination among them.
- Provide a programming model with blocking primitives to handle protocol messages. The test framework itself should hide the synchronous/asynchronous nature of message arrival.
- Support for media exchange - this is especially useful for testing systems that involve user interactions with a interactive voice response system, where we need to be able to simulate DTMF key presses by the user.
- Support for HTTP client to simulate web based interactions with the SUT.

We took a survey of the existing test frameworks that are available for web as well as SIP testing. We realized that although there were many different frameworks (*e.g.,* HtmlUnit, HttpUnit, Canoo Web test, Watij, Floyd, Imprimatur, Funkload, Watir, Selenium, actiWate and many more) - both free and commercial products - available for testing web based applications, there were only a few (SIPp [7], SipUnit [8]) freely available testing frameworks for SIP based applications. Furthermore, even among the few that were available for SIP testing, none of them satisfied all the above requirements. For example, SIPp does not have support for testing web applications and it is not possible to host multiple endpoints within the same SIPp test script. SipUnit does not have support for media exchange. These limitations in the existing testing tools motivated us to create a testing framework - KitCAT (Kit for Converged Application Testing - that aims to satisfy the above requirements.

The rest of this paper is structured as follows. Section 2 gives an overview of KitCAT highlighting its main features and the overall approach we have taken towards testing converged applications. Section 3 describes the execution model and semantics of testing primitives used in our framework. Section 4 illustrates a few example usages of KitCAT. Section 5 gives an overview of the implementation details in KitCAT and real-world usage experience of this initial implementation is presented in Section 6. We present related work in Section 7 and finally, Section 8 offers conclusions and describes future work.

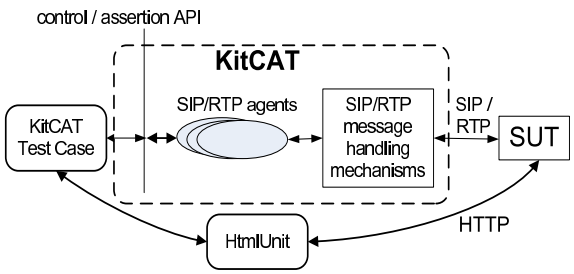

Figure 1: Overall approach in KitCAT to test converged applications

2. KITCAT FRAMEWORK OVERVIEW

Figure 1 shows the overall approach used in KitCAT to test converged applications. A test case in KitCAT uses multiple protocols - *e.g.,* SIP, RTP, HTTP - to interact with the system under test (SUT). An *agent* is the basic entity in KitCAT used to simulate a SIP/RTP end user of the SUT and thus responsible for sending and receiving SIP/RTP messages. In the rest of the discussion we use the term *agent* to indicate a SIP/RTP test agent. A test case creates KitCAT *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 make assertions on the state of agents to determine whether agents are in expected states during the course of a test. An agent 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 agents within a single process thus facilitating the coordination among the callers and callees.

Apart from providing agents for SIP/RTP interaction, KitCAT allows the usage of other testing frameworks based on HTTP and that are already being used to test web based applications - *e.g.,* HttpUnit [5], HtmlUnit [4]. The idea is to leverage the features of these existing web test frameworks with KitCAT so that we are able to test converged applications involving both web and VoIP. We compared web testing frameworks and found that HtmlUnit has a rich set of features to aid with web testing. These features include abstractions for a web browser, HTML pages, elements within a page such as forms, anchors, *etc.* . Moreover, HtmlUnit has support for Javascript and AJAX. It also provides support for XPath [10] based search within an HTML document node. Although HtmlUnit has been the desired web testing framework for our internal use with KitCAT, we don't restrict other web testing frameworks for use with KitCAT.

The following are features that are supported by KitCAT 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.
- A state-machine based approach to 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.

KitCAT allows the test writer to write test cases in a manner where the test case determines *when* agents send and process received messages. The test case uses command primitives (*e.g.,* call, answer, end) to direct agents to send appropriate SIP messages to the SUT. KitCAT implements the mechanisms necessary to receive and buffer incoming SIP messages in the background without intervention from the test case and provides a blocking interface to the test writer to process these messages later in a deferred manner. It is important to note that the act of receiving a SIP message is distinct from the act of processing that message. Receiving a message does not result in state transition of an agent, whereas processing a message might cause a state transition of an agent along with associated actions. KitCAT also provides the infrastructure necessary for supporting the *agent* abstraction and takes care of demultiplexing the incoming messages to the appropriate agent as well as sending SIP messages to the SUT.

KitCAT also allows the test writer to send DTMF key sequences to the SUT. It converts the string sequence into appropriate RFC2833-based [27] RTP packets and send 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). Finally, agents can be directed to send an RTP stream stored in a file. Agents can 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.

We use a state-machine-based approach to specify agent behavior. Agents go through state transitions and perform associated actions such as sending a SIP message. 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 [23]. The state-machine-based approach enables us to achieve the appropriate level of control over agent behavior (explained further in Section 3).

KitCAT primitives can make assertions on agent states. Moreover, they can also make queries based on the history of messages sent/received by each agent. We have adopted a style of assertions (hamcrest [16, 15] assertions) in KitCAT,

that enables more expressive assertions leading to increased readability of test cases. The assertion primitives are extensible allowing the user to create custom assertions that are tailored to the user's testing requirements. Section 5 describes these assertions in detail.

KitCAT integrates well with JUnit [21] 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 [2] environment using a JUnit plugin.

Following are some of the key agent primitives that are provided in KitCAT - **setProxy, call, answer, cancel, end, sendResponse, sendDTMF, playAudio**. Most of these primitives serve as instructions to specific agents to send SIP/RTP messages. We now proceed to describe the runtime execution model that the test writer needs to be aware of when writing KitCAT test cases and the semantics of the above primitives.

3. EXECUTION MODEL AND SEMANTICS

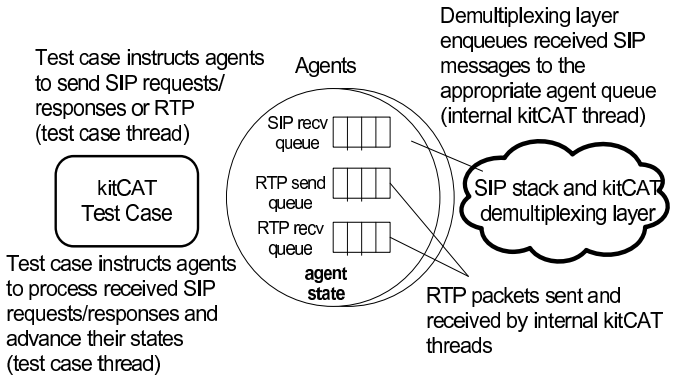

Figure 2: Execution Model in KitCAT

Figure 2 shows the execution time components in KitCAT. Figure 3 provides an informal description of the semantics of the core primitives in KitCAT. The test writer should be aware of this execution model and semantics when writing test cases. Following are the key components in the model:

1. the test case
2. the agents in the test case; and
3. the demultiplexing layer which dispatches incoming messages from the SUT to the appropriate agent.

A KitCAT test case has its own thread of execution in the context of which it performs all testing operations. This includes interacting with a SUT using SIP, HTTP and RTP. In this execution model, we leave out HTTP since HTTP operations are typically blocking operations - the test case thread sends a HTTP request (using HtmlUnit) and blocks waiting for a response [1].

[1]Even though AJAX requests typically occur in a background thread other than the browser thread of execution, HtmlUnit provides a way to send the AJAX requests in the same thread of execution as the test case thread and block waiting for the response

All SIP messages to the SUT are sent in the context of the test case thread. SIP messages are sent in a blocking manner from within a test case *i.e.*they are not queued in any buffers within KitCAT. Messages are sent using the JAIN-SIP API provided by the SIP stack. An agent provides the necessary encapsulation for creating and parameterizing outgoing SIP messages. All SIP messages received from the SUT are received in the context of an internal KitCAT thread in the demultiplexing layer[2] These received SIP messages are then enqueued to the appropriate agent's queue(SIP recv queue in Figure 2). The receipt and subsequent queueing of SIP messages to agent queues are both done in the context of the demultiplexing layer thread. Using a different thread for enqueueing received SIP messages allows the test case to process the received messages subsequently in a blocking manner.

While the SIP messages are sent in a blocking manner, RTP packets (see semantics of sendDTMF, playAudio in Figure 3) are sent in a non-blocking manner. These primitives return immediately after enqueueing RTP packets onto the agent queue (RTP send queue in Figure 2). The RTP packets are sent in the background by internal KitCAT threads without blocking the execution of the test case thread. Sending the RTP packets in a background thread prevents unnecessary blocking of the test case thread, especially since RTP packets have to be paced appropriately for sending. Moreover, this may be necessary in situations where the test case has to simulate concurrent IVR interactions each of which may have time-sensitive behavior. Received RTP packets are enqueued in appropriate agent's buffer (RTP recv queue in Figure 2). The test case can then process these RTP packets subsequently in a blocking manner.

An agent does not have its own thread of execution. It always executes in the context of the test case thread of execution. An agent can thus be viewed as a purely passive data structure encapsulating the state information for a particular SIP end point. An agent also provides the necessary abstraction so that the test case can carry out appropriate tasks on the agent. Examples of these tasks include making a call to another agent or SUT, answering an incoming call, processing the next SIP message in the queue, *etc.* .

Of the three components described above, the test writer directly deals only with the test case and the agents. The demultiplexing layer is hidden from the test writer. We now proceed to informally describe the semantics of some key primitives in KitCAT.

All primitives shown in Figure 3 are executed in the thread context of the test case thread and an agent does not have any thread context of its own. Primitives such as call, answer, end and cancel result in the appropriate SIP message's being sent as well as in advancement of the agent state. These primitives are blocking primitives and return only af-

[2]For reasons of clarity, we omit implementation details here. Even though we talk of a single logical thread of execution for the demultiplexing layer, the actual implementation could involve more than one thread.

```
Agent
{
  processNextSIPMsg()
  {
    msg = dequeue SIP message from head
          of SIP message queue
    advanceState(0, msg)
  }

  advanceState(cmd, msg)
  {
    advance state according to agent
     state machine logic (e.g. Figure 4)
  }

  call/answer/end/cancel
  {
    cmd = create command with necessary
          parameters e.g. request-uri
    advanceState(cmd, 0)
  }

  sendDTMF(dtmf)
  {
    packetize dtmf keys using RFC 2833 and
    enqueue them to the RTP send queue
  }
}

processSIP(timeout)
{
 do
 {
   if all agent queues are empty
      block until atleast one of them is
      non-empty or timeout, whichever is earlier

      for each agent with a non-empty SIP message queue
        do
          agent.processNextSIPMsg()
        until agent queue is empty
 }
 until timeout happens
}
```

Figure 3: Informal semantics of primitives in Kit-CAT

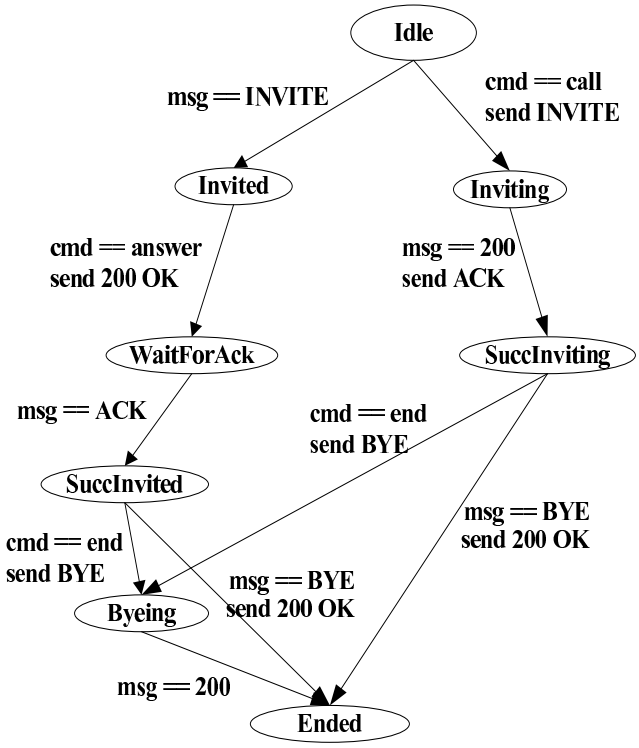

Figure 4: Simplified machine illustrating command execution from test case and SIP message processing

ter the outgoing SIP message is handed over to the SIP stack for sending. As part of execution of these primitives, a command is created and passed to the agent state machine logic. The agent state machine logic executes this command appropriately based on the current state of the agent and the type of the command.

While primitives such as call, answer, *etc.* operate on an agent, there is one important primitive **processSIP** that applies to multiple agents. The **processSIP** primitive provides the test writer with an interface to process received SIP messages in a blocking manner. The **processSIP** primitive returns only after the specified duration has expired; until then it keeps processing received messages. Note that these are SIP messages that have been received and enqueued by the demultiplexing layer thread.

The **processSIP** primitive takes a timeout parameter and runs until the expiry of the timeout interval. During this interval, previously received (and queued) SIP messages are processed (**processNextMsg**) and agent states are also advanced (**advanceState**). Note that the processSIP primitive shown here instructs *all* agents to process their queued SIP messages. We are considering an extension of this primitive that specifies a subset of agents to process their queued SIP messages. This allows finer control over processing of queued SIP messages. Note also that the order in which the agents queues are processed is arbitrary. But this also can be extended by parameterizing the processSIP primitive with a user-specified policy that determines the order,

if necessary. Another possible extension of this primitive is to parameterize it with a condition upon the satisfaction of which the execution returns.

Apart from message queues, an agent keeps track of its current state so that it can take appropriate action when SIP messages are exchanged. For example, Figure 4 shows a very simplified state machine showing a sample set of state transitions an agent can go through. Along with state transitions, the state machine could execute actions within the context of an agent. Primitives such as call, answer, *etc.* are converted to commands for execution by the state machine logic. Execution of a command (indicated by the *cmd* in Figure 4) typically results in actions such as sending a SIP message and finally a change in state. Agent state machine logic is also invoked as part of execution of **processSIP** during which received messages (indicated using *msg* in Figure 4) are processed.

In Figure 4, an agent is initially in the **Idle** state. In this state, if the test case instructs an agent to make a call, the state machine logic is invoked with a command "call" (*cmd*=call). The state machine logic (*cmd* == call) then sends out an INVITE message and then moves to the **Inviting** state. On the other hand, if the test case calls *processSIP* while the agent is in the **Idle** state, and a previously received INVITE is found in the agent's SIP message queue, then that message is dequeued and presented to the state machine logic (*msg*=INVITE). The machine advances to the **Invited** state. SIP response messages are also dealt with in a similar manner.

Note that some transitions take place without the test case explicitly instructing the agent to perform them - *e.g.,* sending an ACK to the 200 OK response in the **Inviting** state (although this assumes a processSIP call by the test case in which the 200 OK response was processed). This is in contrast with the INVITE request being sent in the **Idle** state or the 200 OK response being sent in the **Invited** state. In both these states the test case has to explicitly instruct the agent to send the INVITE (using call command) or send the 200 response (using answer command). In other words, the transitions that are guarded with a command check (cmd == answer, cmd == call *etc.*) means that there is intervention required from the test case to proceed further. Thus by changing the state machine appropriately, the granularity of control over agent state transitions can be altered. KitCAT comes with a default agent state machine (a substantially extended version of the one in Figure 4), where we limit test case intervention to tasks like initiating a call, answering a call and ending a call. Nevertheless, the framework aims to provide the necessary extensibility so that a user can plug in a different state machine to achieve a different level of granularity of control. For example, a user-specified state machine might require test case intervention to send an ACK to a 200 OK response to an INVITE. Providing the right balance between user-instructed and implicit sending of SIP messages can be done only with more experience with different use cases.

The benefit of the above model is that the test case determines when queued SIP messages are processed by using the **processSIP** primitive. The test case executes sequentially instructing agents to send SIP/RTP messages, and process received (and enqueued) SIP messages. It also checks assertions on the agent states. The test writer has control over when SIP messages are sent/processed and when agent states are advanced. This makes the task of writing assertions easier since the test writer knows when to expect changes in the states of agents and when not to.

4. EXAMPLES

We now describe some examples that illustrate the use of KitCAT in testing converged applications. These example test cases illustrates the usage of the primitives described in sections 2 and 3. For reasons of clarity, the examples interleave code extracts with psuedo-code whereever necessary.

4.1 Simple Call Setup and Teardown

Figure 5 shows a KitCAT test case for simple call setup and teardown. The SUT in this example acts as a SIP proxy, proxying the call to the agent specified in the request URI of the incoming INVITE. The objective of the test case is to make sure that the callee (Alice) receives a call from the caller (Bob) and that they are connected. The test case then proceeds to assert that media is exchanged between the two agents. Finally, the test case initiates a BYE from Alice to Bob and makes sure that the call ended properly. In this subsection, we focus on the structure of a KitCAT test case. We subsequently describe the run-time execution model snapshots for this example in section 4.2.

As illustrated in Figure 5, a test case in KitCAT allows direct expression of the intention of the test writer at a high level. As part of every test case, the underlying SIP infrastructure has to be initialized with the SIP listen port (Line 2 in Figure 5). Possible optimizations include implicitly choosing a default SIP listen port if none is specified, but those details are avoided here for purposes of clarity. The test case then creates two agents - Alice and Bob - Bob acting as a caller and Alice acting as a callee (Lines 9,10) . An agent name has to be specified as a parameter to the agent creation factory method. Also, we do not distinguish between the caller and callee agents - they are just roles played by an agent. The KitCAT infrastructure uses the agent name to demultiplex initial (*e.g.,* initial INVITE) incoming requests to the appropriate agent. The test case then specifies the IP address and SIP port for the SUT so that SIP messages can be sent to the SUT (Line 14)[3].

The test case then instructs agent Bob to make a call to agent Alice (Line 17). This results in KitCAT sending a INVITE request to the SUT. The request URI for this INVITE request contains the SIP URI (which is derived from the agent name, the local IP address and the SIP listen port) for Alice. For a single second duration, received SIP messages are processed (Line 19). This example assumes that the initial dialog gets established within a second. This delay should be adjusted based on the application behavior and test environment. The processSIP primitive must not be replaced by a Thread.sleep() or equivalent, since processSIP primitive allows the framework to process received SIP messages. Once the SIP messages are processed, the

[3]KitCAT internally inserts a ROUTE header so that SIP messages are sent to the SUT by the SIP stack

```
 1 //init kitCAT with SIP listen port
 2 init(23456);
 3
 4 //Assuming that current host IP address
 5 //is x.y.z.w, this will create agents
 6 //with SIP URI -
 7 //Alice@x.y.z.w:23456
 8 //Bob@x.y.z.w:23456
 9 SIPAgent alice = createAgent(``Alice'');
10 SIPAgent bob = createAgent(``Bob'');
11
12 //specify the IP:port for the SUT. initial
13 //INVITE from Bob will be sent to the SUT
14 bob.setProxy(sut_ip_port);
15
16 //send INVITE with URI Alice@x.y.z.w:23456
17 bob.call(alice);
18
19 processSIP(1000);
20
21 assert alice is in Idle state
22 assert bob is in Inviting state
23
24 //send 200 OK response to the INVITE
25 alice.answer();
26
27 processSIP(1000);
28
29 assert alice is connected to Bob
30
31
32 alice.clearMediaBuffer();
33 bob.playAudio(``bobspeak.raw'');
34
35 processSIP(2000);
36
37 assert alice has incoming media
38
39 processSIP(2000);
40
41 bob.clearMediaBuffer();
42 alice.playAudio(``alicespeak.raw'');
43
44 processSIP(2000);
45
46 assert bob has incoming media
47
48 processSIP(5000);
49
50 //send BYE
51 bob.end();
52
53 processSIP(1000);
54
55 assert bob is in Ended state
56 assert alice is in Ended state
```

Figure 5: Call setup and teardown example

test case checks whether agents Bob and Alice are in the appropriate states and (Line 21,22). Section 5.1 describes assertion mechanisms in KitCAT in more detail.

The test case then instructs agent Alice to answer the call (Line 25) . After SIP messages are processed using another processSIP call (Line 27), an assertion is made to check whether the agents Alice and Bob are in the appropriate states and connected (Line 29). The test case now tests for media connectivity between the two agents (Lines 32-46) . The *playAudio* primitive sends a RTP stream to the peer agent and the peer agent can be tested for receipt of RTP packets. Finally, the test case initiates a BYE from Alice to Bob and then makes sure both agents are in the **Ended** state (Line 51).

4.2 Simple Call Setup and Teardown - Runtime Execution Model

While the focus of the section 4.1 was on the structure of a KitCAT test case, the focus of this section is on the runtime execution model of KitCAT (described in section 3) in the context of the call setup and teardown example discussed in section 4.1.

Figure 6 shows the approximate runtime execution model for the test case in Figure 5. The actual timing and order of arrival of events may vary from that shown in Figure 6. But that does not have an impact on the properties being tested, provided the test case takes into consideration the message transmission delays in the test environment by calling the **processSIP** primitive with sufficiently large delay values.

Figure 6 shows three timelines - one each for the agents Bob and Alice and another one for the SUT. On the left hand side, we show the state changes of the agents as the test case is executed. At the beginning of the test (1), Bob and Alice are both in the **Idle** state. Once Bob calls Alice, an INVITE is sent out which reaches the SUT. The SUT sends a 100 Trying response to Bob and then proxies the INVITE to Alice. Bob's state changes to **Inviting** (2), whereas Alice's state remains **Idle**. At this point in execution, two SIP messages have been received by the KitCAT infrastructure - 100 Trying response for Bob and INVITE request for Alice. Both of these messages are queued in the respective agent queues for SIP messages and these messages will be processed only during the execution of a **processSIP** primitive, which is either currently under execution or subsequently executed.

At the end of execution (3) of the first **processSIP** primitive, Bob's state is still **Inviting** and Alice's state has advanced to **Invited** since the INVITE message has been processed by Alice. As seen in figure 4, in the **Invited** state, an agent can be instructed to send a 200 OK response (with SDP) using the *answer* primitive. Once agent Alice answers the call (4), the agent's state advances to **WaitForAck**. The 200 OK response from Alice is enqueued to Bob's SIP message queue and since a **processSIP** is already under execution, that response is processed and Bob's state is advanced to **SuccInviting** (5). As part of processing the response, the SUT sends an ACK to Alice which is enqueued in Alice's queue. Since the processSIP is still under execution, the ACK gets processed by Alice and Alice's state is advanced to **SuccInvited** (6). The SIP dialog has been es-

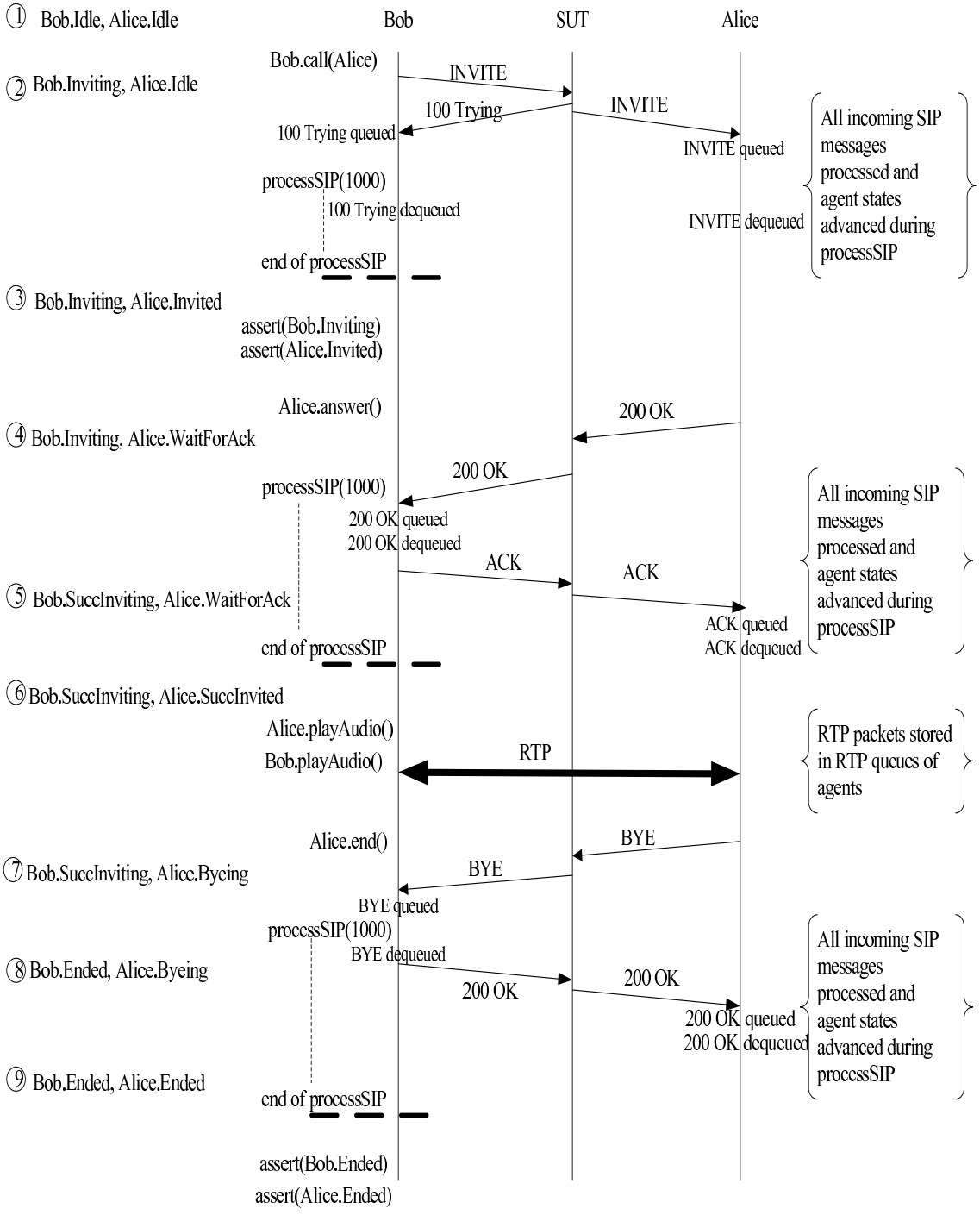

Figure 6: Execution model snapshots for a simple call setup and teardown

tablished at this point and the agents start exchanging RTP packets, which are stored in the RTP queues of the individual agents and later sent using internal KitCAT threads. Similarly, RTP packets from peer agents are received by internal KitCAT threads and stored in agent buffers. An agent can check for incoming media by determining whether its RTP receive buffer is empty or not. Finally, Alice ends the session by sending a BYE. When the BYE is sent, Alice's state is advanced to **Byeing** (7). The BYE is enqueued in Bob's queue and processed; as a result, Bob's state advances to **Ended** (8) and a 200 response to the BYE is sent to Alice. Eventually this response gets processed by Alice and the agent state advances to **Ended**.

4.3 Simple Call Setup and Teardown - Extension to Web

We now demonstrate how a converged application can be tested using KitCAT. Consider that the previous example SUT is extended to include logging of calls that go through the SUT. These call logs are then retrieved through a website as shown in Figure 7. The web page has an HTML form with a table to show the logs and a button to clear off the logs. The objective here is to test both the web and call control aspects of the SUT.

Figure 7: Calls logged on a website

Figure 8 shows an example of how testing of converged applications is achieved in KitCAT.

The sample code extract in Figure 8 illustrates the usage of HtmlUnit within a KitCAT test case. The first part of the test clears the call logs. The appropriate web page is accessed using HtmlUnit. The web page document is then traversed using HtmlUnit APIs. In this case, the form showing the call logs is obtained first and the clear button pressed to clear the logs.

The SIP test (in section 4.1) is now run and the necessary assertions performed. Using HtmlUnit, the call log web page is accessed again and a search performed on the web page document using XPath [10] search expressions. In this example test case, we simply verify the presence of a table row containing both the strings Alice and Bob. HtmlUnit offers a rich API to navigate an HTML document and the code extract above shows only a very minor subset.

4.4 Do Not Disturb application

The example in this subsection demonstrates the ease with which KitCAT can be used to send DTMF key sequences.

```
//clear the call logs
WebClient webClient = new WebClient();
HtmlPage page =
  (HtmlPage) webClient.getPage("www.xyz.com/calls");
HtmlForm form = page.getFormByName("CallLogs");
form.submit();

//previous SIP/RTP test goes here

//verify whether call log is updated
page =
  (HtmlPage) webClient.getPage("www.xyz.com/calls");
form = page.getFormByName("CallLogs");
String xpath =
  "//table//tr[contains(.,'Alice')" +
  " and contains(.,'Bob')]";
List l = form.getByXPath(xpath);

assert list l is not empty
```

Figure 8: Call setup and teardown example - Extension to Web

This capability is useful for testing IVR based applications. The Do Not Disturb (DnD) application discussed here allows a subscriber to indicate to a caller that the subscriber should not be disturbed unless absolutely necessary. When a caller calls a callee who is subscribed to DnD and has turned it on, the caller hears an announcement from a media server - "The subscriber has indicated not to be disturbed. Please press 1 if you would like to leave a message. Press 2, if you would like to get connected to the callee". Based on the caller's choice, the caller can either leave a voice message or get connected to the callee. Furthermore, the callee has the capability to browse a text version of the voice messages using a web interface. The callee also uses a web interface to turn the DnD feature on or off.

Figure 9 shows a KitCAT test case for the DnD example. As part of the test setup, the DnD feature can be turned on for Alice using a web testing framework such as HtmlUnit [4]. The test then proceeds to establish a call with Alice and then chooses the option to leave a message. The caller plays some audio and then ends the call. Once the call is ended, the appropriate web site can be accessed using the web test framework and appropriate assertions can be made to check that there was a new voice mail for Alice.

Figure 10 shows another test case for the DnD feature where the caller chooses to talk to the callee (option 2 in the IVR). In this case, we verify that the caller gets connected to the callee and that they are able to exchange media.

The above examples shows how we can leverage already existing web testing frameworks like HtmlUnit in KitCAT. We have done this in the context of other non-trivial converged applications within AT&T and the experience has been really positive so far (see Section 6). These examples also demonstrates the simple media exchange capabilities (including DTMF keys) in KitCAT that are very helpful for testing applications that expect a DTMF based response from the user. The RTP packets that arrive at an agent are saved to a disk file, which can later be converted to audio files. We have found this feature to be helpful in debugging

```
//do necessary setup in SUT using HtmlUnit
//so that DoNotDisturb is on for Alice.

SIPAgent alice = createAgent(``Alice'');
SIPAgent bob = createAgent(``Bob'');
bob.setProxy(sut_ip_port);
bob.call(alice);
processSIP(3000);

assert bob is in SuccInviting state
assert alice is in Idle state
assert that bob has incoming media
  from media server

//we want to leave a message
bob.sendDTMF(``1'');

processSIP(3000);

bob.playAudio(``voicemail.raw'');
processSIP(5000);

bob.end();
processSIP(2000);
assert bob is in Ended state
assert alice is in Idle state

//use HTMLUnit to make sure that we
//can see a new message for Alice
//through the web
```

Figure 9: Do Not Disturb scenario1

failures.

5. IMPLEMENTATION OVERVIEW

An initial implementation of KitCAT has been completed. KitCAT uses the JAIN-SIP [17] compliant SIP stack implementation from NIST [6] and ECharts [13, 14, 11, 12] for implementing the agent state machine that is packaged with KitCAT. From an architecture perspective, we have used the Half-Sync/Half-Async [25] concurrency pattern to decouple the asynchronous nature of SIP message arrivals in KitCAT from the (deferred) synchronous processing of these messages. The JAIN SIP API provides a call-back programming abstraction to receive SIP messages and a blocking programming abstraction to send SIP messages. We have created an adapter layer that demultiplexes incoming SIP messages to the appropriate agent ECharts machine. For initial requests, the demultiplexing strategy uses the agent name in the user field of a SIP request URI to deliver the message to the right agent. For subsequent requests/responses, we use a combination of CallId and To/From-tag (from the SIP To/From header) to identify which agent the request/response needs to be delivered to. Because we use the agent name for demultiplexing initial requests, the test writer must make sure that agent names are unique.

For RTP processing, we use two internal KitCAT threads - one for sending RTP packets from different agents over to peer RTP ports and another to receive RTP packets destined for multiple agents. The single receiver thread uses the Reactor [24, 25] pattern (implemented using **java.nio.Selector**) to demultiplex messages from multiple RTP ports to the appropriate agent queues.

5.1 Assertion mechanisms

```
.....same as in scenario 1...

bob.call(alice);
processSIP(3000);

assert bob is in SuccInviting state
assert alice is in Idle state

assert that bob has incoming media
  from media server

//we want to get connected to callee
bob.sendDTMF(``2'');
processSIP(3000);

assert alice is in Invited state
assert bob is in Inviting state

alice.answer();
processSIP(2000);

assert alice and bob are connected
...

bob.end();
processSIP(2000);
assert alice is in Ended state
assert bob is in Ended state
```

Figure 10: Do Not Disturb scenario2

Our goal in KitCAT is to provide simple assertion primitives that the test writer can use to assert on agent states and other states in the test system. 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.

A nice solution uses the **assertThat** assertion [16, 15] available as part of the Hamcrest framework in JUnit 4.4 [21]. 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)));
```

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. The following extract illustrates a few more types of assertions in KitCAT.

```
//is and has are just syntactic sugar
assertThat(bob, is (idle()));
assertThat(bob, is (disconnected()));
assertThat(bob, has (incomingMedia()));
```

6. EXPERIENCE USING KITCAT

We have used KitCAT in functional as well as load testing of a non-trivial conferencing application and the results are quite encouraging. The conferencing application is a converged application that was created within AT&T and deployed in production with a large number of users. We use KitCAT to run periodic sanity tests - every 5 minutes 24 x 7 - to check the health of the application. As part of the sanity test, we simulate users joining a conference and perform basic checks on both the web as well as VoIP functions. Even though the test is a functional test, it exercises all the application components as well as the supporting infrastructure - SIP application server, web server, media server and PSTN gateway. The results from these tests are regularly monitored by the production support staff. These tests have been very effective in indicating problems and enabling the production support team to take proactive measures to mitigate adverse user experience.

We have created an application specific wrapper over HtmlUnit for testing web functions of the conferencing application. This provides the test writer with a high-level application-aware interface for writing web function tests. The source code for the sanity test contains about 200 lines of Java code excluding logging and comments.

KitCAT is being used for load testing the above conferencing application as well. Load testing involves each of these endpoints interacting with an IVR system and subsequently joining a conference. These load tests are being used to gain more confidence in the application under load. We have successfully run load tests upto 1200 SIP/RTP endpoints. The limit of 1200 is because of limitations in the SUT rather than from KitCAT itself. As part of future work, we would like to try extending this limit and study the effects of increasing load - memory, resource consumption, *etc.* - on the load tester itself.

We have started to use KitCAT for functional regression testing of the above converged application as well as other reusable call-control features.

7. RELATED WORK

Two areas are closely related to our work with KitCAT: (1) web testing and (2) SIP testing. A variety of different frameworks assist testing in each of these areas, but we could not find an ideal tool which would enable us to the combine the features of all these frameworks for converged application testing. The failure to do so provided the motivation for our efforts on KitCAT.

Web testing:. A number of frameworks are available for web testing - *e.g.,* HtmlUnit [4], HttpUnit [5], Canoo Web Test [1]. 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. For example, we are using HtmlUnit in KitCAT test cases to test a non-trivial converged application.

SIP testing:. Compared to the number of frameworks available for web testing, there are only a few frameworks that are available to do functional testing of SIP based applications. SIPp [7] is a open source tool that can test SIP applications. But it is cumbersome to do functional testing of SIP applications since the test scripts are XML based and most often too low-level. Moreover, simulating multiple user agents involves writing multiple scripts and running multiple OS processes. In contrast, KitCAT offers a higher-level programming model for test writers to do functional testing and we have successfully used KitCAT for load testing purposes also - all within a single process. But KitCAT uses the call model used by the JAIN SIP stack and hence there are things that you can do with SIPp that you cannot do with KitCAT - *e.g.,* sending an ill-formed SIP message.

SipUnit [8] is another open source tool that can be used to test SIP applications. One major deterrent for us to use SipUnit was the lack of support for media in SipUnit and it is not possible to send DTMF key tones to test interactive applications using SipUnit. KitCAT provides a simple primitive to do this. Moreover, KitCAT offers a simpler programming model, uses a state-based approach to testing and an extensible assertion mechanism. We believe that SipUnit offers the user a lower-level API which KitCAT currently does not offer. For example, a test writer can construct a JAIN SIP message and send it using SipUnit, whereas this is not currently supported in KitCAT.

Commercial products that are available for performing application testing are Hammer G5 [3] (from Empirix) and TTCN-3 [9] 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.

8. CONCLUSIONS AND FUTURE WORK

Converged applications are gaining increasing significance in the context of convergence of IP and telephony. Efforts are underway to ease the task of developing, deploying and testing converged applications. In this paper, we have presented the KitCAT framework and its features to test converged applications. We have presented illustrative examples that demonstrate the simplicity of writing test cases using KitCAT. The primitives are simple, yet powerful for testing a wide variety of applications. We have used KitCAT in combination with HtmlUnit to successfully build functional as well as load tests for a non-trivial converged application within AT&T. The experience so far has been positive and encouraging.

There are limitations to the current implementation of KitCAT some of which we plan to address as part of future work. It does not offer an API that allows a test case to create a JAIN SIP message and send it. Currently we do not support unit testing of SIP servlets in the sense that we expect the SIP servlet application to be deployed in a servlet container. We would also like to consider the feasibility of an approach where the servlets can be unit-tested during development in a non-container environment.

As part of our future work, we would also like to make enhancements to KitCAT in the following areas. Enhancements to semantics involve adding the ability to specify policies to govern the semantics. For example, we may want a user-specified policy to restrict processing of SIP messages only for a subset of agents. We plan to add capability to plug in user-specified state machine and providing necessary abstractions to write a custom state machine. We also plan to support more types of SIP messages (*e.g.,* SUBSCRIBE/NOTIFY) in KitCAT. Finally, we would also like to research the possibility of test case generation from formal models of SIP UAC and UAS such as the one described in [28].

Acknowledgments

The author would like to gratefully acknowledge the valuable contributions of research colleagues - Gregory Bond, Eric Cheung, Hal Purdy, Thomas Smith and Pamela Zave. The RTP subsystem in KitCAT uses a modified version of previously written code by Eric Cheung. The author would also like to thank the development teams of the following open source libraries - NIST SIP stack, ECharts, HtmlUnit, Hamcrest, JUnit and the many other open source libraries that they depend on. Finally, thanks to my colleagues - Gerald Karam, Karrie Hanson and Don Henderson - for help with an internal converged application which has served as a real-world use case for KitCAT.

9. REFERENCES

[1] Canoo Web Test. http://webtest.canoo.com.
[2] Eclipse. http://www.eclipse.org/.
[3] Hammer G5. http://www.empirix.com.
[4] HtmlUnit. http://htmlunit.sourceforge.net/.
[5] HttpUnit. http://httpunit.sourceforge.net/.
[6] NISTSipStack. https://jain-sip.dev.java.net/.
[7] SIPp. http://sipp.sourceforge.net/.
[8] SipUnit. http://www.cafesip.org/projects/sipunit/.
[9] TTCN-3. http://www.ttcn-3.org/.
[10] XPath. http://www.w3.org/TR/xpath20/.
[11] G. W. Bond. Timed transition activation semantics in Statecharts. Technical Report TD-6H4L5S, AT&T, 2005. Available from: `http://echarts.org`.
[12] G. W. Bond. An introduction to ECharts: The concise user manual. Technical Report TD-6NKLR2, AT&T, 2006. Available from: `http://echarts.org`.
[13] 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`.
[14] G. W. Bond and H. H. Goguen. ECharts: From lab to production. Technical Report TD-6FSMWT, AT&T, 2005. Available from: `http://echarts.org`.
[15] *Hamcrest*. Available from: `http://code.google.com/p/hamcrest`.
[16] *Hamcrest Tutorial*. Available from: `http://code.google.com/p/hamcrest/wiki/Tutorial/`.
[17] *JAIN(tm) SIP Specification*. Java Community Process, 2003. Available from: `http://jcp.org/aboutJava/communityprocess/final/jsr032/`.
[18] *JSR 116: SIP Servlet API Version 1.0*. Java Community Process, 2003. Available from: `http://www.jcp.org/aboutJava/communityprocess/final/jsr116`.
[19] *JSR 154: Java Servlet 2.4 Specification*. Java Community Process, 2003. Available from: `http://jcp.org/en/jsr/detail?id=154`.
[20] *JSR 289: SIP Servlet Version 1.1*. Java Community Process, 2008. Available from: `http://jcp.org/en/jsr/detail?id=289`.
[21] *JUnit*. Available from: `http://www.junit.org/`.
[22] 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/.
[23] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol RFC 3261. 2002.
[24] D. C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching. In *Proceedings of the* 1^{st} *Annual Conference on the Pattern Languages of Programs*, pages 1–10, Monticello, Illinois, Aug. 1994.
[25] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.
[26] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-time Applications. *RFC 1889*, Jan. 1996.

[27] H. Schulzrinne and S. Petrack. Rtp payload for dtmf digits, telephony tones and telephony signals. *RFC 2833*, May 2000.

[28] P. Zave. Understanding SIP Through Model-Checking. IPTComm 2008. Available from: `http://research.att.com/~pamela/under2.pdf`.