TestML Programmer's Guide

Touchstone Technologies, Inc.
1500 Industry Road Suite H
Hatfield, PA 19440
Tel: 267-222-8687
Fax: 267-222-8697

www.touchstone-inc.com

Table of Contents

Introduction

The WinSIP TestML engine is designed to provide the user with a comprehensive set of features and functions to enable free form scripting of SIP call flows. It supports structured concepts such as nested transactions without imposing rigid restrictions on form. Because the TestML engine is implemented alongside the WinSIP internal stack, the TestML engine can call upon the resources of WinSIP's built-in functions which can significantly reduce the minutia of complex transactions such as SDP parsing.

The TestML engine also has extended support for media processing such as setting, starting, and stopping the media streams at any point during the script. IVR functions such as sending DTMF and receiving DTMF are also supported making the engine ideal for "call and response" scenarios. Advanced features such as user data dictionaries, timers, retransmission control, and more are also included in this release.

The following document describes the concepts, structure, and elements of a TestML document. In the code examples you will often times see the symbol "**…**". This indicates that in a typical implementation, there would most likely be other code in this section. This code has been omitted from the example in the interest of space and readability usually because it is extraneous to the example being discussed.

Getting started

A word about scope

The TestML document supports the programming concept of scope. The definition of transactions, messages, templates, and other reusable components are scope aware. Therefore, if you want a transaction, template, message or other component to be accessible, be sure to define it in a “public” scope relative to where you want to access it. Likewise, if you want to “privatize” one of these components, narrow the scope to the most local logical component.

A word about data dictionaries, substitution tags, and variables

The TestML engine has a powerful underlying data dictionary and support mechanisms. This dictionary mechanism allows you to access internal variables (such as user input fields) as well as creating your own variables (e.g. anything_received). You can substitute variables as “smart text” into messages, test on the value of dictionary elements, set or increment values and so forth.

Structural overview

Blueprint

The TestML document's defining attribute is the blueprint element. All TestML scripts **must** have one and only one blueprint element. Therefore, the most basic TestML Script would consist of the following code:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE TestML SYSTEM "TestML.dtd">
<blueprint name="My Test" description="My Test SIP (UAS)">
        …
</blueprint>
```

A Blueprint **may** also include a plan element which can execute one or more session elements, implement logic such as repeat statements and conditionals and provide other control mechanisms. If no plan element is present, the sub-elements such as sessions or dialogs are executed in the order in which they appear in the script.

Plan

A plan element implements the control logic for a blueprint or session and is contained within either of these elements. The plan element is the component which is typically used to control the execution of sessions, dialogs and other test-related components. The following is an example of a script which executes two different session plans within one blueprint:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE TestML SYSTEM "TestML.dtd">
<blueprint name="My Test" description="My Test SIP (UAS)">
        …
        <plan>
                <execute type="session" id="Basic UAC"/>
                <execute type="session" id="Basic UAS"/>
        </plan>
</blueprint>
```

It is also implemented in a session element in a similar manner to execute dialogs or transactions:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE TestML SYSTEM "TestML.dtd">
<blueprint name="My Test" description="My Test SIP (UAS)">
        …
        <session name=”My Session>
                …
                <plan>
                        <execute type="dialog" id="Basic Outgoing Telephony Call" />
                </plan>
        </session>
        <plan>
                <execute type="session" id="Basic UAC"/>
        </plan>
</blueprint>
```

Session

A session element is contained within a blueprint and implements the control logic for a test. The session element is the component which is typically used to control the execution of dialogs and other test-specific components. A session element **must** include a plan element. A Basic session element would appear as:

```
…
<session name="Basic UAC">
        …
        <plan>
                <execute type="dialog" id="Basic Outgoing Telephony Call" />
        </plan>
</session>
```

A session element is typically where a repetitive test would put its control logic for the iterations as follows:

```
…
<session name="Basic UAC">
        …
        <plan>
                <repeat iterations="[input.frequency]">
                        <execute type="dialog" id="Basic Outgoing Telephony Call" />
                </repeat>
        </plan>
</session>
```

Dialog

A dialog element is contained within a blueprint or session depending upon scope, and implements the send/receive or receive/send control logic for a call. The dialog element is loosely modeled after a SIP dialog. Dialogs are a structural, reusable element and are not necessarily required by a session. A basic dialog element would appear as:

```
…
<session name="Basic UAC">
        <dialog name=" Basic Outgoing Telephony Call">
                …
        </dialog>
        …
        <plan>
                <repeat iterations="[input.frequency]">
                        <execute type="dialog" id="Basic Outgoing Telephony Call" />
                </repeat>
        </plan>
</session>
```

Transaction

A transaction element is typically contained within a dialog or session depending upon scope, and implements the send/receive logic for a transaction (e.g. BYE transaction). The transaction element is loosely modeled after a SIP transaction. Transactions are structural, reusable elements and are not necessarily required. A basic transaction element would appear as:

```
…
<transaction name="Basic UAC">

        <message name="bye">
                …
        </message>
        …
        <send message="bye" ip="1.1.1.1" port="5060" id="bye " retransmit="500" />
        <waitevents>
                …
        </waitevents>
</transaction>
```

Message

A message element is contained within a blueprint, session, or dialog depending upon scope, and implements the template for reusable, outgoing messages. A basic message element would appear as:

```
…
<message name="bye">
  <![CDATA[
     BYE sip:[remote.id]@[remote.address]:[remote.port] SIP/2.0
     [dialog.routeset]
     Via: SIP/2.0/UDP 1.1.1.1:5060;branch=[new.branch]
     From: [local.name] <sip:[local.id]@[local.address]:[local.port]>;tag=[local.tag]
     To: [remote.name] <sip:[remote.id]@[remote.address]:[remote.port]>;tag=[remote.tag]
     Call-ID: [callid]
     CSeq: [local.cseq] BYE
     Max-Forwards: 70

  ]]>
</message>
```

Note the extra line break between the last line and the close of the CDATA XML section. A message is used in conjunction with a send element.

Send

A send element is contained within a blueprint, session, and implements the transmission functionality of the TestML engine. There are two ways to implement a send element. You may send a message element as above, or send raw or dictionary-based text. A basic send element would appear as:

```
…
<send ip="[remote.address]" port="[remote.port]" id="bye.request" retransmit="500" />
  <![CDATA[
     BYE sip:[remote.id]@[remote.address]:[remote.port] SIP/2.0
     [dialog.routeset]
     Via: SIP/2.0/UDP 1.1.1.1:5060;branch=[new.branch]
     From: [local.name] <sip:[local.id]@[local.address]:[local.port]>;tag=[local.tag]
     To: [remote.name] <sip:[remote.id]@[remote.address]:[remote.port]>;tag=[remote.tag]
     Call-ID: [callid]
     CSeq: [local.cseq] BYE
     Max-Forwards: 70

  ]]>
</send>
```

As mentioned before, you may also send a pre-defined message template by including the message parameter:

```
<send message="bye" ip="[remote.address]" port="[remote.port]" id="bye " retransmit="500" />
```

WaitEvents

The waitevents section **must** be included in order to process user events, receive incoming messages, and respond to timer events or retransmission timeouts. The WaitEvents block is typically located within a dialog or transaction element. Each of the subcomponents of the waitevents element can either “complete” the waitevents (i.e. exit the block) or not (i.e. continue to wait for incoming messages or events). The following is an example of a waitevents block:

```
…
<waitevents>
     <receive protocol="sip" method="BYE" type="response" code="100..199" complete="false" />
     <receive protocol="sip" method="BYE" type="response" code="200..299" complete="true" />
</waitevents>
```

Receive

The receive element **must** be included in a WaitEvents section in order to process incoming messages.

```
…
<receive protocol="sip" method="BYE" type="response" code="100..199" complete="false">
        …
</receive>
```

Event

The event element **must** be included in a WaitEvents section in order to process user events such as pressing the stop button, timer events, and retransmission timeouts:

```
…
<event id="call.duration.timer" complete="true">
        …
</event>
```