


Test Execution Directives (TED) User's Guide

Owner	Andrei Kirshin
Approvers	Kenneth Nagin
Status	Approved
Date	05 February 2004

AGEDIS



Change History

Version	Date	Author	Summary of changes
1.0	2/12/2003	Andrei Kirshin	Draft
1.0	06/02/2004		Approved

Contents

1. OVERVIEW	4
2. GENERAL USAGE ELEMENTS.....	4
2.1. value.....	4
2.2. complexValue	6
2.3. altValue.....	7
2.4. class	7
2.5. exception.....	7
3. TESTEXECUTIONDIRECTIVES	8
3.1. description.....	8
3.2. constants & constant.....	8
3.3. objects & object.....	10
3.4. processes & process.....	11
3.4.1. directive.....	12
3.5. hosts & host.....	13
3.5.1. address.....	14
3.6. initSets & initSet	14
3.6.1. init	16
3.7. invariants & invariant	16
3.7.1. interaction.....	18
3.8. options.....	20
3.8.1. testSuiteRef	20
3.8.2. testCaseRef.....	21
3.8.3. traceOutput	21
3.8.4. repeatTestSuite	22
3.8.5. repeatTestCase.....	22
3.8.6. timeoutInteraction.....	22
3.8.7. delayTestSuite	22
3.8.8. delayTestCase.....	22
3.8.9. delayStep	23
3.8.10. pollingTime	23
3.8.11. onError	23
3.8.12. defaultVerdict.....	23
3.9. mappings.....	23
3.9.1. atsSignature	24
3.9.2. typeMapping.....	25
3.9.3. basic	25
3.9.4. array	25
3.9.5. valueMapping.....	26
3.9.6. memberMapping.....	26
3.9.7. sutSignature	28

1. Overview

ATS is an input to the Test Suite Driver telling it **what** to execute against the System Under Test. Test Execution Directives (TED) are another input to the Test Suite Driver providing the information on **how** to execute the ATS.

It includes:

- constants – global constants for substitute throughout the test execution directives
- definitions - description of SUT (host machines, processes, objects and classes) for distribution, synchronization and multiplication (cloning)
- initialization – initialization directives for all objects, processes, and hosts
- invariants – repetitive actions to extend the ATS (setup, cleanup, additional evaluations)
- options - various global execution options
- mappings - ATS to SUT mapping directives for types, constants, controls and observables.

You can run the TED Wizard to create a template of the Test Execution Directives based on the abstract test suit . Then you edit the generated file to fit the actual SUT. The edited test execution directives is input to the Test Suite Driver.

Test Execution Directives has two basic features:

- **XML based** – it is written in as a XML document.
- **reviewable** – it is simple to edit, review and archive

We provide a formal specification of the test execution directives in its XML schema.

In the following sections we describe each XML element.

2. General Usage Elements

The elements in this section are used throughout the test execution directives so we introduce here.

2.1. value

The value element describes input value used during the execution. It has the two forms: `<value ref="reference">` or `<value> input </value>`.

It does not contain any elements but its content text describes its value if any or it references another element.

It contains the following attributes:

- **ref**- the ref attribute is used to reference other elements in the test execution directives. Its meaning is the same as described in the testSuite definition but it also may reference test execution directive [constant](#), [object](#), [process](#), or [host](#). You can even use it to reference a specific field in an object, process or host. The ref

attribute form is: (<constant> | (<object> | <process> | <host>) [".." field]>)|"NULL"

These XML segments describe the value 32.112, reference the constant SERVER_IP, the o_client object, the o_client field state, and the NULL value:

```
<value>32.112</value>
<value ref="SERVER_IP"/>
<value ref="o_client"/>
<value ref="o_client.state"/>
<value ref="NULL"/>
```

Writing XML markup characters in a value element:

The special characters used for XML markup like <>,&," can not be directly used in a value. Rather a CDATA section or escape markup sequences are used.

An XML **CDATA section** may occur anywhere that character data may occur. It is used to escape blocks of text containing characters that would otherwise be interpreted as part of the XML markup, such as ampersand (&) and less than (<). CDATA sections begin with the string '<![CDATA[' and end with the string ']]>'.</p></div>

CDATA Section Syntax:

CDSect	CDStart CData CEnd
CDStart	'<![CDATA['
CData	(Char* - (Char* ']]>' Char*))
CEnd	']]>'

Within a CDATA section, only the CEnd string is recognized as markup, so that left angle brackets and ampersands may occur in their literal form.

The other approach is to use escape markups:

- & is &
- ' is '
- > is >
- < is <
- " is "

This is an XML value example that uses a CDATA section to input unparsed data:

```
<value>
  <![CDATA[
    Unparsed Data:
    <step 1> asdf
    <step 2> zxcv
```

```
-->>> OK <<<--  
  ]>  
</value>
```

This is an XML value example that inputs the text without CDATA:

```
<value>  
  Data Unparsed  
  &lt; 1step &gt;asdf  
  &lt;2step &gt;zxcv  
  --&lt;&lt;&lt; OK &lt;&lt;&lt;--  
</value>
```

2.2. complexValue

The **complexValue** element is a collection of values and other complexValues. It starts with **<complexValue ref="reference">** and ends with **</complexValue>**.

It contains the following intermixed elements. At least one element must be present:

1..∞ [value](#)

1..∞ **complexValue**

The **ref** attribute has an equivalent meaning to the [value](#) element's **ref** attribute.

This is an example of an interaction that uses XML complexValue to pass an array of addresses. Each a address has two fields:

```
<interaction object="test"  
  signature="printAddresses(addresses):int"  
  type="call_return">  
  <complexValue>  
    <complexValue>  
      <value>127.0.0.1</value>  
      <value>8080</value>  
    </complexValue>  
    <complexValue>  
      <value>127.0.0.1</value>  
      <value>8081</value>  
    </complexValue>  
    <complexValue>  
      <value>127.0.0.1</value>  
      <value>8082</value>  
    </complexValue>
```

```
</complexValue>
<value>3</value>
</interaction>
```

2.3. altValue

The altValue element is not implemented in the current release.

The **altValue** element is a collection of values and other complexValues. It starts and ends with **<altValue>** and ends with **</altValue>**.

It contains 1..∞ [value](#) or 1..∞ [complexValue](#) elements. It is not allowed to intermix the two kinds of elements. The contained elements represent the list of alternative data choices for an interaction's response or input parameters. When it maps to a response each value or complexValue represents an expected response for the interaction. When it maps to an input parameter the test may choose any one of value or complexValues. A value or complexValue may reference the abstract test suite's Global class's constants, or members.

This XML segment invokes the o_db.state() – valid return values are *not empty* and *empty*.

```
<interaction object="o_db" signature="state():string"/>
  <altValue>
    <value>not empty</value>
    <value>empty</value>
  </altValue>
</interaction>
```

2.4. class

The **class** element describes a real SUT class or procedure in a code library. It starts and ends with **<class name="class name" language="javalcplc" >** and ends with **</class >**.

Its contents contain no additional elements or text.

The element contains the following attributes:

- **name** – real name of class (required)
- **language** - java | cpp | c. (optional, but defaults to java)

This is an XML segment describing a Java class:

```
<class name="ibm.hrl.dolphine.server" language="java"/>
```

2.5. exception

The exception element is not implemented in the current release.

The **exception** element describes an exception thrown by an operation or an attribute access method. It starts with **<exception name="exception class">** and ends with **</exception>**.

The exception contents may contain additional text describing the exception.

The **name** attribute describes the exception name, i.e. class.

```
<exception name="java.io.EOFException"/>
```

3. testExecutionDirectives

The **testExecutionDirectives** element is the root element of the Test Execution Directives XML specification. It starts with **<testExecutionDirectives version="version number">** and ends with **</testExecutionDirectives>**.

It contains the following elements:

- 0..1 [description](#)
- 0..1 [constants](#)
- 0..1 [objects](#)
- 0..1 [processes](#)
- 0..1 [hosts](#)
- 0..1 [initSets](#)
- 0..1 [invariants](#)
- 0..1 [options](#)
- 0..∞ [mappings](#)

It contains the following attributes:

- **version** –describes the version of the test execution directives, e.g. 1.0.

3.1. description

The **description** element describes the intent of the Test Execution Directives XML. It starts and ends with **<description>** the description text **</description>**.

It contains no additional elements or attributes but its content text describes the test execution directives.

This is a **description** XML example:

```
<description>Sample Test Execution Directives</description>
```

3.2. constants & constant

The **constants** element defines the set of constants necessary to describe the testing interface of the class. **<constants>** indicates the start of the section and **</constants>** indicates its end.

It contains the following elements:

- 1..∞ **constant**

The **constant** element defines a constant used in the testing interface of the class. It starts with **<constant name="name" type=" string|char|int|float|bool| byte|short|long|double|defined" ref="type reference">** and ends with **</constant>**.

It contains one the following elements:

- 0..1 [value](#)
- 0..1 [complexValue](#)

The value or complexValue elements describe the constant's actual value. A missing value will have to be supplied in the test execution directives.

It contains the following attributes:

- **name** - the name of the constant (required).
- **type** – valid values are string|char|int|float|bool|byte|short|long|double|defined. When type is “defined”, the **ref** attribute references another type element in the abstract test suite's Global class (optional).
- **ref** – reference to another type name in the abstract test suite's Global class (optional, only valid when type is defined).

This is an XML example:

```
<constant name="SERVER_IP" type="string">
  <value>9.148.32.112</value>
</constant>
```

This is a **constants** XML example that describes 5 constants:

```
<constants>
  <constant name="SERVER_IP" type="string">
    <value>9.148.32.112</value>
  </constant>
  <constant name="SERVER_PORT" type="int">
    <value>7777</value>
  </constant>
  <constant name="USER1" type="string">
    <value>Harry Potter</value>
  </constant>
  <constant name="USER2" type="string">
    <value>Ron Weasley</value>
  </constant>
  <constant name="ALIVE" type="bool">
    <value>true</value>
  </constant>
</constants>
```

This is its tabular representation:

name	type	value
SERVER_IP	string	9.148.32.112
SERVER_PORT	int	7777
USER1	string	Harry Potter
USER2	string	Ron Weasley
ALIVE	bool	true

In the other XML examples we reference these constants.

Note: Current release does not support constants.

3.3. objects & object

The **objects** element is a collection of **object** elements and it contains no attributes. It starts with **<objects>** and ends with **</objects>**.

It contains following elements:

- 1..∞ **object**

The **object** element describes a SUT object used during the execution. It begins with **<object name="objects name" initSet="initialization set" clones="number of clones" synchronization="concurrent|sequential">** and ends with **</object>**.

It describes a real SUT object upon which to run the test. By default the test driver uses the ATS class name to instantiate its objects as Java objects, however when you require other concrete names, languages, customized initialization or cloning use the object element. You can specify a number of objects to clone and customize their initialization. SUT object cloning is described in section [Abstract Tests Extended with Object Partitioning](#).

It contains the following element:

- 1 [class](#)

It contains the following attributes.

- **name** - alias used as reference in other TED tables. (Required and unique)
- **initSet** – reference to an initialization set. Only one initialization set allowed for an object entry. Initialization sets are defined in the [initSets](#) element. (Optional)
- **clones** – number of object clones. (Optional but defaults to 1)
- **synchronization** – mode of synchronization: concurrent | sequential. (Optional but defaults to concurrent)
- **environment** - describes the object's relationship to the system under test (SUT). When it is false, then the object describes an SUT object or proxy object that communicates with the SUT. When its environment attribute is true, then the object describes the environment at the border of the SUT, e.g. UML actor. A true setting may also be used in a component test to indicate that the tester (test execution engine) should impersonate the SUT object (optional but defaults to false).

This is an XML segment describing the o_server object:

```
<object name="o_server" initSet="i_o_server" clones="1" synchronization="concurrent">  
  <class name="ibm.hrl.dolphine.server" language="java"/>  
</object>
```

This is an XML segment that describes 4 objects:

```
<objects>  
  <object name="o_server" initSet="i_o_server" clones="1" synchronization="concurrent">  
    <class name="ibm.hrl.dolphine.server" language="java"/>  
  </object>  
  <object name="o_client" initSet="i_o_client" clones="1" synchronization="concurrent">  
    <class name="ibm.hrl.dolphine.client" language="java"/>  
</objects>
```

```

</object>
<object name="o_client_proxy" initSet="i_o_client" clones="1" synchronization="concurrent">
  <class name="client_ats_expected_results" language="java"/>
</object>
<object name="o_db" clones="1" synchronization="concurrent">
  <class name="db" language="cpp"/>
</object>
</objects>

```

This is its tabular representation:

name	initSet	clones	synch.	class name	language
o_server	i_o_server	1	concurrent	ibm.hrl.dolphine.server	java
o_client	i_o_client	1	concurrent	ibm.hrl.dolphine	java
o_client_proxy	i_o_client	1	concurrent	client_ats_expected	java
o_db		1	concurrent	db	cpp

3.4. processes & process

The **processes** element is a collection of **process** elements and it contains no attributes. It starts with **<processes>** and ends with **</processes>**.

It contains following elements:

- 1..∞ **process**

The **process** element describes a process in which SUT objects are created. It begins with **<process name="process name" objects="object list" language=" javalcppl" initSet="initialization set" clones="number of clones" synchronization="concurrent|sequential">** and ends with **</process>**.

By default the test driver runs the test in its process space, however when you want to distribute the test to run in other processes use the processes element. You can specify a number of processes to clone and customize their initialization. Process cloning is described in section [Abstract Tests Extended with Object Partitioning](#).

A **process** element contains:

- 0..∞ [directive](#)

A object element contains the following attributes.

- **name** - alias used as reference in other TED tables. (Required and unique)
- **objects** - list of objects this process consists of.
- **language** - java | cpp | c. (Optional but defaults to java)
- **initSet** - reference to an initialization set. Only one initialization set is allowed. Initialization sets are defined in the [initSets](#) element. (Optional)
- **clones** - number of object clones. (Optional but defaults to 1)
- **synchronization** - mode of synchronization: concurrent | sequential. (Optional but defaults to concurrent)

This is an XML segment describing the o_server object:

```
<process name="p_server" objects="o_server" language="java" initSet="i_p_server">
  <directive>-classpath "}.${dolphine}/lib/dolphine.jar:${dolphine}:${CLASSPATH}"</directive>
</process>
```

This is an XML segment that describes 3 processes:

```
<processes>
  <process name="p_server" objects="o_server" language="java" initSet="i_p_server" clones="1"
    synchronization="concurrent">
    <directive>-classpath "}.${dolphine}/lib/dolphine.jar:${dolphine}:${CLASSPATH}" </directive>
  </process>
  <process name="p_client" objects="o_client o_client_proxy" language="java" initSet="i_p_client"
    clones="1" synchronization="concurrent">
    <directive>-classpath ".;%dolphine%\lib\dolphine.jar;%dolphine%;%CLASSPATH%"</directive>

  </process>
  <process name="p_db" objects="o_db" language="cpp" clones="1" synchronization="concurrent">
    <directive>lib_dolphin.sl </directive>
  </process>
</processes>
```

This is its tabular representation:

name	objects	language	initSet	clones	synch.	directives
p_server	o_server	java	i_p_client	1	concurrent	-classpath "}.\${dolphine}/lib/dolphine.jar:\${dolphine}:\${CLASSPATH}"
p_client	o_client o_client_proxy	java		1	concurrent	-classpath ".;%dolphine%\lib\dolphine.jar;%dolphine%;%CLASSPATH%"
p_db	o_db	cpp		1	concurrent	lib_dolphin.sl

3.4.1.directive

The **directive** element describes instructions about a process's configuration characteristics. It starts and ends with **<directive>** the directive **</directive>**.

These directives are specific to the process's language type. For instance java directives may be any Java command line options. Directives for c++ and c processes are to be determined.

A directive contains no elements, but their content text describes the directive.

The **format** attribute describes the text format (optional).

This is an XML segment directive describing a Java class path in Unix command line:

```
<directive>-classpath "}.${dolphine}/lib/dolphine.jar:${dolphine}:${CLASSPATH}"</directive>
```

3.5. hosts & host

The **hosts** element is a collection of **host** elements and it contains no attributes. It starts with **<hosts>** and ends with **</hosts>**.

It contains following elements:

- 1..∞ **host**

The **host** element describes a host machine used to create processes. It begins with **<host name="host name" processes="process list" initSet="initialization set" clones="number of clones" synchronization="concurrent|sequential">** and ends with **</host>**.

By default the test driver runs the test on its host machine, however when you want to distribute the test to run on other host use this element. You can specify a number of addresses for each host (host cloning). Host cloning is described in section [Abstract Tests Extended with Object Partitioning](#).

A host element contains the following elements:

- 1..∞ [address](#) (each address represents the address or IP of a host clone)

An object element contains the following attributes.

- **name** - alias used as reference in other TED tables. (Required and unique)
- **processes** - list of processes running on the host
- **initSet** - reference to an initialization set. Only one initialization set allowed for a host element. Initialization sets are defined in the [initSet](#) element. (Optional)
- **synchronization** - mode of synchronization: concurrent | sequential. (Optional but defaults to concurrent)

This is an XML segment describing the clientHost:

```
<host name="clientHost" processes="p_client" initSet="i_clientHost" synchronization="concurrent">
  <address>9.148.34.1</address>
  <address>9.148.34.2</address>
</host>
```

This is an XML segment that describes 2 host aliases:

```
<hosts>
  <host name="serverHost" processes="p_server p_db" synchronization="concurrent">
    <address>9.148.32.112:7777</address>
  </host>
  <host name="clientHost" processes="p_client" initSet="i_clientHost" synchronization="concurrent">
    <address>9.148.34.1</address>
    <address>9.148.34.2</address>
  </host>
</hosts>
```

This is its tabular representation:

name	processes	initSet	synch.	address
serverHost	p_server p_db		concurrent	9.148.32.112:7777
clientHosts	p_client	i_clientHosts	concurrent	9.148.34.1 9.148.34.2

3.5.1.address

The **address** element describes a host address. It starts and ends with **<address>** the host address **</address>**.

It contains no additional elements but its content text or **ref** attribute describes the address.

It contains the following attribute:

- **ref**- reference to another [constant](#) element. (optional)

These are two examples of XML addresses:

```
<address>9.148.34.1</address>
<address ref="SERVER_IP"/>
```

3.6. initSets & initSet

The **initSets** element is a collection of **initSet** elements and it contains no attributes. It starts with **<initSets>** and ends with **</initSets>**.

It contains following elements:

- 1..∞ **initSet**

The **initSet** element lists a set of entries used to initialize SUT objects, processes, and hosts. It starts with **<initSet name="init set name">** and ends with **</initSets>**.

It has the following elements:

- 0..∞ [init](#)

It has the following attributes:

- **name** - alias used as reference in other TED tables. (Required and unique)

This is a XML segment describing the initialization set *i_o_server*:

```
<initSet name="i_o_server">
  <init name="title" type="string">
    <value>Server Under Test</value>
  </init>
  <init name="version" type="float">
    <value>3.1</value>
  </init>
  <init name="mode" type="bool">
    <value ref="ALIVE"/>
  </init>
</initSet>
```

In the example below we describe the initialization sets used to initialize the objects in previous XML examples namely [objects](#), [processes](#), and [hosts](#). We also reference constants in the [constants](#) example.

i_o_server has three entries *title*, *version*, and *mode*. *Title* and *mode* are assigned explicit values, but *mode* reference the constant *ALIVE*. In our example *o_server* is initialized with these fields.

i_clientHosts has one entry *partition*, but it has two values *Partition A* and *Partition B*. In our example *i_clientHosts* is used to initialize the host controller, *clientHost*. But *clientHost* has two clones so the first clone gets the value *Partition A* and the second *Partition B*.

i_p_client has only one entry *partition* but it gets its value by referencing its host. In our example *i_p_client* is used to initialize *p_client*. Since *p_client* is associated with the *clientHost* clones it also represents two instances. One *p_client* is initialized with *Partition A* and the other with *Partition B*.

i_o_client has only one entry *partition* but it gets its value by referencing its process. In our example *i_o_client* is used to initialize *o_client*. Since *o_client* is associated with the *p_client* clones it also represents two instances. One *o_client* is initialized with *Partition A* and the other with *Partition B*.

```
<initSets>
  <initSet name="i_o_server" type="string">
    <init name="title">
      <value>Server Under Test</value>
    </init>
    <init name="version" type="float">
      <value>3.1</value>
    </init>
    <init name="mode" type="bool">
      <value ref="ALIVE"/>
    </init>
  </initSet>
  <initSet name="i_clientHost">
    <init name="partition" type="string">
      <value>Partition A</value>
      <value>Partition B</value>
    </init>
  </initSet>
  <initSet name="i_p_client">
    <init name="partition" type="string">
      <value ref="HOST.partition"/>
    </init>
  </initSet>
  <initSet name="i_o_client">
    <init name="partition" type="string">
      <value ref="PROCESS.partition"/>
    </init>
  </initSet>
  <initSet name="i_p_server"/>
</initSets>
```

This is its tabular representation:

Set Name	name	type	value or ref
i_o_server	title	string	Server Under Test
	version	float	3.1
	mode		ALIVE
i_clientHosts	partition	string	Partition A Partition B
i_p_client	partition		HOST.partition
i_o_client	partition		CLIENT.partition
i_p_server			

3.6.1.init

The **init** element describes an initialization field and its value or a reference to another field in the test execution directives. It starts with `<init name="name" type="string|char|int|float|bool|byte|short|long|double|defined" ref="type reference">` and ends with `</init>`.

Multiple values and/or references are used when there are a number of clones initialized differently or when the test suite is repeated for variation testing (see the [Input variation testing](#) use cases).

It has the following elements:

- 0..∞ [value](#)
- 0..∞ [complexValue](#)

It has the following attribute:

- **name** - alias used as reference in other TED tables. (Required)
- **type** – valid values are string|char|int|float|bool|byte|short|long|double|defined. When type is “defined”, the **ref** attribute references another type element (optional).
- **ref** – reference to a type defined in the abstract test suites Global class (optional).

Values may reference other [constant](#) elements or instances of other initialized fields. The values used to initialize [processes](#) may reference host init elements in which the process resides (see [hosts](#) element). Set values used to initialize SUT [objects](#) may reference process and host init elements in which the object resides (see [processes](#) element). These type of reference use the following form:

```
(<HOST> | <PROCESS>) "." <init name>
```

This means that the system initializes a process controller or SUT object with a value derived from a host controller or process controller.

This is a XML segment describing the initialization field for `client_id` with 3 initialization values:

```
<init name="client_id" type="string">  
  <value>Client under test</value>  
  <value ref="USER1"/>  
  <value ref="PROCESS.user_id"/>  
</init>
```

3.7. invariants & invariant

The **invariants** element is a collection of `inv` (invariant) elements. It starts with `<invariants>` and ends with `</invariants>`.

It contains the following elements:

- 1..∞ **invariant**

The **invariant** element describes an invariant that is repeated throughout the test suite. It starts with `<invariant where="testSuite|testCase|step|error" when="before|after">` and ends with `</invariant>`.

It contains the following elements:

- 1..∞ [interaction](#)

It contains the following attributes:

- **where** – testSuite | testCase | step | error (required).
- **when** – before | after (required).

Use **testSuite** and **testCase** for test setup and cleanup operations.

Use **step** to extend the abstract test suite's expected results with your own validation function to extend the ATS's expected results. The invariant's interaction response describes your expected results. The extended validation functions may want to know the ATS expected results. You can use the [SUT Initiated Interaction API](#) to get the expected results or map the expected results to your proxy by setting the [mapping](#) element's **echo** attribute.

The interactions are executed in the specified order within the specified context as specified in the where and when attributes. This means you should create of an object before invoking one of its methods or accessing one of its fields. Warning messages will be issued during execution if you send messages to objects that do not exist. By default all objects are created at the beginning of the test suite and destroyed at the conclusion.

An invariant with the combination “after error” is called when the Execution Engine detects an error executing a Test Suite.

This is a XML segment that specifies three actions to perform when starting the test suite:

1. Create the object o_db.
2. Send start() to the object o_db.
3. Create the object o_server with the DB ip address as input.

```
<invariant where="testSuite" when="before">
  <interaction object="o_db" signature="DB()" kind="create"/>
  <interaction object="o_db" signature="start()" kind="send"/>
  <interaction object="o_server" signature="Server(string)" kind="create"/>
    <value ref="DB_IP"/>
  </interaction/>
</invariant>
```

This is its tabular representation:

kind	SUT object	SUT control
create	o_server	
create	o_db	
send	o_db	start()

This XML segment invokes the two methods after each test case step:

- o_db.state() – valid return values are *not empty* and *empty* or the exception *recovery_in-progress*
- o_client_proxy.checkResults(o_client) – the valid return value is *true*. Notice that the parameter input is an object reference and the o_client_proxy and o_client must reside in the same process.

```
<invariant where="step" when="after">
  <interaction object="o_db" signature="state():string"/>
    <value>not empty</value>
    <value>empty</value>
    <exception name="recovery_in-progress"/>
</invariant>
```

```

</interaction>
<interaction object="o_client_proxy" signature="checkClient(object):bool">
  <value to="o_client"/>
  <value type="bool">true</value>
</interaction>
</invariant>

```

This is its tabular representation:

SUT object	SUT signature	Parameter Input value, ref	Expected value, exception
o_db	state():string		not empty empty recovery_in-progress
o_client_proxy	checkClient(object):bool	o_client	true

3.7.1.interaction

The **interaction** element describes a test action or observation used to extend the abstract test. Its meaning is the same as described in the AGEDIS test suite.

It starts with **<interaction initiator="initiating object" object="owning object" signature="interaction foot print" suffix="[i] | [j..k] | field name" type="call|return|call_return|send|receive|check|change|wait|for|create|destroy" timeout="time" delay="time" pollingTime="time interval" wait="time">** and ends with **</interaction>**.

An interaction contains the following elements:

- 0..∞ [value](#) and [complexValue](#) intermixed
- 0..1 [altValue](#)
- 0..∞ [exception](#)

The **value**, **complexValue** and **altValue** elements can be intermixed. They describe the interaction input, response and attribute value setting. The **exception** element is used when an interaction responds by throwing an exception. If more than one exception is present then any one of the exceptions is an expected responses.

An interaction element contains the following attributes:

- **object** – owning object (required). It references a specific [object](#) element to which this interaction belongs.
- **signature** – owning member signature or test case parameter (required). It references a specific member in the object's class associated with this interaction.
- **suffix** – signature suffix (optional). The suffix is useful for further qualifying the signature to describe array element(s) or a field name, using the following forms:
 - array element – It is expressed in the form [index]. The index references an array element. The index may a positive integer or enumeration [element](#) name.
 - array interval – It is expressed in the form [start_index..end_index]. The interval references the array elements from start_index through end_index. The indexes may be positive integers or enumeration [element](#) names.
 - field name – The field name references a [structure's field](#).
- **type** - describes the interaction type (required). The possible type enumerations are listed below:

- **call** – invoke a synchronous operation.
 - **return** – return from a previously issued synchronous operation call.
 - **call_return** – call followed by the return.
 - **send** – send an asynchronous signal.
 - **receive** - receive an asynchronous signal.
 - **send_receive** – send and receive a signal.
 - **check** – check an attribute value.
 - **change** – set an attribute value.
 - **waitFor** – wait to check an asynchronous attribute value.
 - **create** – create the object
 - **destroy** – destroy or de-reference the object
- **initiator** – initiating object. It describes who is initiating the interaction. If no initiator is provided, then the initiator is the environment, otherwise the attribute points to an [object](#) element (optional).
 - **timeout** – A SUT timeout describes the maximum amount of time in milliseconds to wait for an [interaction](#) to return before failing a test (optional) Valid values are $0 - 2^{31}-1$, $0=\infty$. The value 0 means to wait forever. It defaults to 0 or whatever value is specified in the [timeout](#) element.
 - **delay** – A SUT delay describes how long to wait in milliseconds after performing a an [interaction](#) (optional). Valid values are $0 - 2^{31}-1$. It defaults to 0 or whatever value is specified in the [delay](#) option.
 - **pollingTime** - It contains the polling time interval in milliseconds to evaluate any **waitFor** type interaction. Valid values are $1 - 2^{15}-1$, but the default is 100 or whatever value is specified in the [pollingTime](#) option.
 - **wait** – wait time before evaluating a **waitFor** type interaction (optional). It describes how long to wait in milliseconds until beginning to evaluate (polling) the interaction's value. Valid values are $0 - 2^{31}-1$. but the default is 0 or whatever value is specified in [pollingTime](#)'s wait attribute option.

The timeout, delay, pollingTime and wait attributes are not supported in the current release.

This XML segment invokes the `o_db.state()` – valid return values are *not empty* and *empty* or the exception *recovery_in-progress*

```
<interaction object="o_db" signature="state():string"/>
  <altValue>
    <value>not empty</value>
    <value>empty</value>
  </altValue>
  <exception name=" recovery_in-progress "/>
</interaction>
```

3.8. options

The **options** element describes a collection of test suite execution options for controlling the global behavior of the test suite execution. It starts with **<options>** and ends with **</options>**.

It contains the following elements intermixed:

- [testSuiteRef](#)
- [traceOutput](#)
- [repeatTestSuite](#)
- [repeaTestCase](#)
- [timeoutTestSuite](#)
- [timeoutTestCase](#)
- [timeoutStep](#)
- [delayTestSuite](#)
- [delayTestCase](#)
- [delayStep](#)
- [pollingTime](#)
- [onError](#)
- [defaultVerdict](#)

This is an **options** XML example:

```
<options>
  <testSuiteRef externalRef="options1.ats"/>
  <repeatTestSuite>2</repeat>
  <timeoutTestSuite>10000</timeout>
</options>
```

This is its tabular representation:

Option Name	OptionValue
testSuiteRef	options.ats
repeatTestSuite	2
timeoutTestSuite	10000

3.8.1.testSuiteRef

The **testSuiteRef** element references a test suite to execute. It starts with **<testSuiteRef externalRef="testSuite file reference">** and ends with **</testSuiteRef>**.

It contains the following elements:

- 0..∞ [testCaseRef](#)

The **externalRef** attribute is a reference to the test suite to execute (optional). It may contain complete or partial path to the test suite. If it only specifies a directory location then the default

name is used but it uses the specified directory. The test suite name defaults to the test execution directive's file name, and its file type defaults to **ats**. You change the default by specifying its name explicitly here.

This is a **testSuiteRef** XML example:

```
<testSuiteRef externalRef="../options/options1.ats">
  <testCaseRef ref="tc1"/>
  <testCaseRef ref="tc2"/>
</testSuiteRef>
```

3.8.2.testCaseRef

The **testCaseRef** element is a list of test cases to include or exclude from the referenced test suite. It starts and ends with **< testCaseRef ref="test case reference" type="include|exclude">** test case list **</testCaseRef>**.

It does not contain any additional elements.

It contains the following attributes:

- **ref** – test case reference (required)
- **type** - include or exclude (optional, but it defaults to include).

This is a **testCaseRef** XML example:

```
<testCaseRef ref="tc1" type="exclude">
```

3.8.3.traceOutput

The **traceOutput** element describes how and where to write the test trace. It starts and ends with **<traceOutput type="overWrite|append|insert|new">** trace path **</traceOutput>**.

It may contain complete or partial path to write the test trace. If it only specifies a directory location then the default name is used but it writes the trace to the specified directory. The trace file name defaults to the testSuite's file name, and its file type defaults to **set**. If you want to embed the trace in the testSuite then specify its name explicitly here.

The type attributes are not supported in the current release but you can specify an alternative trace path to write the trace.

Its **type** attribute is optional, but it defaults to overwrite. The following values are valid:

- **overwrite** – overwrite the most recent trace (default). If no trace is present create a new trace.
- **append** – append the trace after the last trace in the trace file. If no trace is present create a new trace.
- **insert** – insert the trace to be first trace in the trace file. If no trace is present create a new trace.
- **new** - create a new trace file. Generate a suffix to the file name to make it unique based on the date and time.

This is a traceOutput XML example:

```
<traceOutput>./traces/modelname.ted</traceOutput>
```

3.8.4.repeatTestSuite

The **repeatTestSuite** element describes how many times to repeat the test suite. It starts and ends with **<repeatTestSuite>** number of repetitions **</repeatTestSuite>**.

It contains a number of repetitions. Valid values are $1 - 2^{15} - 1$, but the default is 1.

This is a **repeatTestSuite** XML example:

```
<repeatTestSuite>2</repeatTestSuite>
```

3.8.5.repeatTestCase

The **repeatTestCase** element describes how many times to repeat each test case. It starts and ends with **<repeatTestCase>** number of repetitions **</repeatTestCase>**.

It contains a number of repetitions. Valid values are $1 - 2^{15} - 1$, but the default is 1.

This is a **repeatTestCase** XML example:

```
<repeatTestCase>2</repeatTestCase>
```

3.8.6.timeoutInteraction

The **timeoutInteraction** element describes how long to wait for an interaction to complete until reporting a timeout error. It starts and ends with **<timeoutInteraction>** timeout time **</timeoutInteraction>**.

It contains the time in milliseconds to wait until reporting a test error. Valid values are $0 - 2^{31} - 1$, $0 = \infty$, but the default is 0. The value 0 means to wait forever.

This is a **timeoutInteraction** XML example:

```
<timeoutInteraction>100</timeoutInteraction>
```

3.8.7.delayTestSuite

The **delayTestSuite** element describes how long to delay after executing the test suite. It starts and ends with **<delayTestSuite>** delay time **</delayTestSuite>**.

It contains the delay time in milliseconds. Valid values are $0 - 2^{31} - 1$, but the default is 0.

This is a **delayTestSuite** XML example:

```
<delayTestSuite>10</delayTestSuite>
```

3.8.8.delayTestCase

The **delayTestCase** element describes how long to delay after executing a test case. It starts and ends with **<delayTestCase>** delay time **</delayTestCase>**.

It contains the delay time in milliseconds. Valid values are $0 - 2^{31} - 1$, but the default is 0.

This is a **delayTestCase** XML example:

```
<delayTestCase>10</delayTestCase>
```

3.8.9. delayStep

The **delayStep** element describes how long to delay after executing a test case step. It starts and ends with **<delayStep>** delay time **</delayStep>**.

It contains the delay time in milliseconds. Valid values are $0 - 2^{31}-1$, but the default is 0.

This is a **delayStep** XML example:

```
<delayStep>10</delayStep>
```

3.8.10. pollingTime

The **pollingTime** element describes when to evaluate any **waitFor** type interaction. It starts and ends with **<pollingTime wait="wait time">** polling time **</pollingTime>**.

It contains the polling time interval in milliseconds to evaluate any **waitFor** type interaction. Valid values are $1 - 2^{15}-1$, but the default is 100.

The **wait** attribute describes how long to wait in milliseconds until beginning to evaluate (polling) the interaction's value (optional, but defaults to 0). Valid values are $0 - 2^{31}-1$.

This is a **pollingTime** XML example:

```
<pollingTime wait="2">100</pollingTime>
```

3.8.11. onError

The **onError** element describes the execution engine's behavior in case of a test error. It starts and ends with **<onError>**skip|stop|pause|continue**</onError>**.

It may contain the following values:

- **continue** – continue the execution (default)
- **stop** – stop the execution
- **pause** – pause the execution
- **skip** – skip to the next Test Case

This is a **onError** XML example:

```
<onError>stop</onError>
```

3.8.12. defaultVerdict

The **defaultVerdict** element describes the verdict for test error that is not specifically defined in the test suite (for instance a test execution directive's timeout). It starts and ends with **<defaultVerdict>** pass|fail|inconclusive **</defaultVerdict>**.

It may contain pass, fail, or inconclusive (optional but defaults to fail).

This is a **defaultVerdict** XML example:

```
<defaultVerdict>inconclusive</defaultVerdict>
```

3.9. mappings

The **mappings** element describes a collection of mapping elements and it contains no attributes. It starts with **<mappings>** and ends with **</mappings>**.

It contains the following mapping elements:

- 0..∞ [memberMapping](#)
- 0..∞ [typeMapping](#)
- 0..∞ [valueMapping](#)

3.9.1.atsSignature

The **atsSignature** references the signature of an ATS type, member or constant. It starts with **<atsSignature object="class or object" signature="member|constant|type foot print" suffix="signature suffix">** and ends with **</atsSignature>**. It is used in all of the mapping elements to identify the ATS part that is to be translated.

It may contain another **atsSignature** when it describes a complex data structure like an enumeration or record.

It contains the following attribute:

- **name** – ATS class or object name. (required)
- **signature** – unique signature. (optional)
- **suffix** – signature suffix. (optional)

The ATS signature has the following form:

```
<name>  
  [“(” [<parameter type> (<parameter type> “,”) *] “)”]  
  [ “:” <member type>]
```

The **<name>** part is required. The parameter () part describes the parameters in synchronous operations, signals, or attribute access methods. It may contain a list of 0..∞ parameters separated by “,”. The **<member type>** part describes the resulting type of synchronous operations or an attributes type. Signals do not have a **<member type>** part. Parameter and member types may be a basic type or a reference to a type element. The referenced type may be a type defined in the object’s class or in the abstract test suite’s global class. Scoping rules apply, so when the same name is used in this class and the global class it refers to this class’s type (for more details see the testSuite Users Guide).

This is an XML segment that describes the ATS Client class:

```
<atsSignature name="Client" />
```

This is an XML segment that describes the ATS Client classes' enumeration element OK:

```
<atsSignature name="Client" signature="CONNECT_RC" suffix="OK"/>
```

This is an XML segment that describes the ATS client object's connect() member:

```
<atsSignature name="client" signature="connect()"/>
```

3.9.2.typeMapping

The **typeMapping** element describes how to map an ATS type to a SUT type. It starts with `<typeMapping>` and ends with `</typeMapping>`

It contains the following elements:

- 1 [atsSignature](#) (A type exists in the context of a class. Therefore ATS class name should be used in the atsSignature.)
- 1 ([basic](#) or [class](#) or [array](#))

It contains no attributes.

This is an example of an XML segment that describes how to map the ATS Server class.

```
<typeMapping>
  <atsSignature name="Server"/>
  <class name="ibm.hrl.dolphine.server.Server" language="java"/>
</typeMapping>
```

By default all of the ATS structures defined within the Server class are mapped to static inner classes of `ibm.hrl.dolphine.server.Server`. Thus if the ATS Server contains an Address structure it is assumed that `ibm.hrl.dolphine.server.Server` also contains an Address class. However you can override the default with another type mapping as illustrated below:

```
<typeMapping>
  <atsSignature name="Server" signature="Address"/>
  <class name="ibm.hrl.dolphine.server.Address" language="java"/>
</typeMapping>
```

ATS enumerations default to a String type you can override the default with another type mapping as illustrated below:

```
<typeMapping>
  <atsSignature name="Client" signature="CONNECT_RC"/>
  <basic type="int"/>
</typeMapping>
```

3.9.3.basic

The **basic** element describes a basic data type. It starts with `<basic type="string|char|int|float|bool|byte|short|long|double">` and ends with `</basic>`.

It contains not element or text.

It contains the following attribute to describe the data type:

- **type** - string | int | bool | char | float | byte | short | long | double (optional but defaults to string).

This is an example of an XML segment:

```
<basic type="float"/>
```

3.9.4.array

The **array** element describes an array data type. It starts with `<array size="array size">` and ends with `</array>`.

It contains the follow element that describes the array type:

- 1 ([basic](#) or [class](#) or [array](#))

It contains the following attribute to describe the array size:

- **size** – array size (it may reference a constant) (required).

This is an example of an XML segment that describes an array of type `ibm.hrl.dolphine.server.Address`:

```
<typeMapping>
  <atsSignature name="Server" signature="addresses[]"/>
  <array size="addressSize">
    <class name="ibm.hrl.dolphine.server.Address" language="java"/>
  </array>
</typeMapping>
```

3.9.5.valueMapping

The **valueMapping** element describes how to map an ATS enumeration or constant to a SUT signature (method, signal, variable constant etc.). It starts with `<valueMapping >` and ends with `</memberMapping>`.

By default ATS enumerations and constants map to the string equivalent to their name. However, the element is used when a different value is assigned to the enumeration or constant.

It contains the following elements:

- 1 [atsSignature](#) (An enumeration or constant exists in the context of a class. Therefore ATS class name should be used in the `atsSignature`.)
- 1 ([value](#) | [complexValue](#))

This XML segment maps the ATS Client's RC enumeration elements:

```
<mappings>
  <valueMapping>
    <atsSignature name="Client" signature="RC" suffix="OK" />
    <value>0</value>
  </valueMapping>
  <valueMapping>
    <atsSignature name="Client" signature="RC" suffix="ERROR" />
    <value>-1</value>
  </valueMapping>
</mappings>
```

This XML segment maps the ATS client's `capacity` constant to the value 3:

```
<valueMapping>
  <atsSignature name="Client" signature="capacity"/>
  <value>3</value>
</valueMapping>
```

3.9.6.memberMapping

The **memberMapping** element describes how to map an ATS member to a SUT member (method, signal, variable etc.). It starts with `<memberMapping>` and ends with `</memberMapping>`.

By default ATS class members and constants map directly to their SUT counterparts and there is no requirement to use the mapping element. When all the ATS signatures have identical SUT counterparts then there is no need to specify the mapping. However, the element is used when special translation is required.

It contains the following elements:

- 1 [atsSignature](#) (A member exists in the context of a particular ATS object. Therefore ATS object name should be used in the atsSignature.)
- 1 [sutSignature](#)

It contains the no attributes:

This is a XML segment that maps the ATS client object's member start() to the SUT o_client method startDB().

```
< mappings >
  < memberMapping >
    < atsSignature name="client" signature="start()" />
    < sutSignature object="o_client" signature="startDB()" />
  < /memberMapping >
< / mappings >
```

This XML segment maps the ATS client's member initialized to the SUT o_client's init variable:

```
< mappings >
  < memberMapping >
    < atsSignature name="client" signature="initialized" />
    < sutSignature object="o_client" signature="init" />
  < /memberMapping >
< / mappings >
```

This XML segment maps the ATS client's member state to the SUT o_client's getState() access method:

```
< mappings >
  < memberMapping >
    < atsSignature name="client" signature="state" />
    < sutSignature object="o_client" signature="getState()" />
  < /memberMapping >
< / mappings >
```

Use an asterisk sign, "*", as a signature when all an ATS object's signatures exactly map to the ones of the SUT. However, you can re-map signatures in the subsequent TED elements. By re-mapping the signatures all the ATS and SUT signatures names don't have to be identical. Use a minus sign, "-", as a signature when an ATS signature does no map to any SUT signature. In this example, we map all the ATS signature to SUT signature with the same names except foo1() and bar:

```
< mappings >
  < memberMapping >
    < atsSignature name="client" signature="*" />
    < sutSignature object="o_client" signature="*" />
  < /memberMapping >
  < memberMapping >
    < atsSignature name="client" signature="foo1()" />
    < sutSignature object="o_client" signature="foo_you()" />
  < /memberMapping >
  < memberMapping >
    < atsSignature name="client" signature="bar" />
    < sutSignature object="o_client" signature="-" />
  < /memberMapping >
< / mappings >
```

ATS signatures can map to more than one SUT object and particular signatures can have more than one SUT counterpart. Conversely, ATS signatures from more than one ATS object can map to one SUT object and particular signatures can have more than one ATS counterpart. This is a

XML segment that maps all of the ATS client's signatures to the SUT o_client1 and o_client2 and all of the ATS server and DB signatures to the SUT o_server:

```
<mappings>
  <memberMapping>
    <atsSignature name="client" signature="**"/>
    <sutSignature object="o_client1" signature="**"/>
  </memberMapping>
  <memberMapping>
    <atsSignature name="client" signature="**"/>
    <sutSignature object="o_client2" signature="**"/>
  </memberMapping>
  <memberMapping>
    <atsSignature name="server" signature="**"/>
    <sutSignature object="o_server" signature="**"/>
  </memberMapping>
  <memberMapping>
    <atsSignature name="DB" signature="**"/>
    <sutSignature object="o_server" signature="**"/>
  </memberMapping>
</mappings>
```

3.9.7.sutSignature

The **sutSignature** references the signature of a SUT member. It starts with **<sutSignature object="owning object" signature="member foot print" timeout="time" delay="time" pollingTime="time interval" wait="time">** and ends with **</sutSignature>**.

It contains the following elements:

- 0..∞ ([value](#) | [complexValue](#)) – parameter input to the SUT member.

It contains the following attributes:

- **object** – SUT object name. (required)
- **signature** – SUT member signature. (required)
- **timeout** – A SUT timeout describes the maximum amount of time in milliseconds to wait for an [interaction](#) to return before failing a test (optional) Valid values are $0 - 2^{31} - 1$, $0 = \infty$. The value 0 means to wait forever. It defaults to 0 or whatever value is specified in the [timeout](#) element.
- **delay** – A SUT delay describes how long to wait in milliseconds after performing a an [interaction](#) (optional). Valid values are $0 - 2^{31} - 1$. It defaults to 0 or whatever value is specified in the [delay](#) option.
- **pollingTime** - It contains the polling time interval in milliseconds to evaluate any **waitFor** type interaction. Valid values are $1 - 2^{15} - 1$, but the default is 100 or whatever value is specified in the [pollingTime](#) option.
- **wait** – wait time before evaluating a **waitFor** type interaction (optional). It describes how long to wait in milliseconds until beginning to evaluate (polling) the interaction's value. Valid values are $0 - 2^{31} - 1$. but the default is 0 or whatever value is specified in [pollingTime](#)'s wait attribute option.

The timeout, delay, pollingTime and wait attributes are not supported in the current release.

The form of the SUT signature is the same as the ATS signature:

<name>

[“(” [<parameter type> (<parameter type> “,”)*] “(”]
[“:” <member type>]

- <name> - method, signal or variable name. (required)
- <paramType> - method parameter type or class (optional)
- <member type> - variable type (or class) or method return type (or class) (optional)

These are two XML signatures one describing a variable and the other describing a function:

```
<sutSignature object="o_client" signature="state:string"/>  
<sutSignature object="o_client" signature="connect(int,int):bool"/>
```

You can instruct the driver to supply predefined input to the method parameters. You describe the input in [value](#) elements contained in the sutSignature. The input is listed in the position corresponding to the subject parameter. This XML segment describes connect(string,int), but it supplies the value “lion.haifa.ibm.com” explicitly and references the SERVER_PORT constant. In the constants example above SERVER_PORT was assigned the value 7777 so the driver actually invokes `connect("lion.haifa.ibm.com", 7777)`

```
<sutSignature object="o_client" signature="connect(string,int)">  
  <value>lion.haifa.ibm.com</value>  
  <value ref="SERVER_PORT"/>  
</sutSignature>
```

Parameter input can reference other [object](#) element or its field with the value element’s **ref** attribute using the form:

The field is referenced in the ref element using the following form:

```
<SUT object> [“.” <SUT field>]
```

You can only reference objects that are created in the same process as the control’s SUT object. This XML segment maps describes SUT startDB(ibm.hrl.dolphine.db,int), but the 1st parameter is assigned a reference to the object o_db and the 2nd parameter is assigned the value in o_server.capacity:

```
<sutSignature object="o_server" signature="startDB(ibm.hrl.dolphine.db,int)">  
  <value ref="o_db"/>  
  <value ref="o_server.capacity"/>  
</sutSignature>
```

The signatures can have parameters and the mapping defines the parameter types. By default ATS parameters are mapped to SUT parameters in the order that they are printed; the 1st ATS parameter maps to 1st SUT parameter, the 2nd ATS to the 2nd SUT parameter, etc. However, the SUT/ATS parameters don’t have to have the same order or number. The parameter order is altered with the value element’s **ref** attribute. A number in the **ref** attribute is used to reference the subject ATS parameter and its position following the sutSignature describes the subject SUT parameter. This is a XML segment that maps the ATS client connect(host,port) to the SUT o_client connect(int,int), ATS register(int,int) maps to SUT register(int,int) (but the ATS 1st parameter maps to the 2nd SUT parameter and the ATS 2nd parameter maps to the 1st SUT

parameter) and disconnect(host,port) maps to the SUT o_client control disconnect(int) (only the ATS 2nd parameter is used):

```
<mappings>
  <memberMapping>
    <atsSignature name="client" signature="connect(host,port)"/>
    <sutSignature object="o_client" signature="connect(int,int)"/>
  </memberMapping>
  <memberMapping>
    <atsSignature name="client" signature="register(host,user)"/>
    <sutSignature object="o_client" signature="register(int,string)"/>
    <value ref="2"/>
    <value ref="1"/>
  </sutSignature/>
</memberMapping>
  <memberMapping>
    <atsSignature name="client" signature="disconnect(host,port)"/>
    <sutSignature object="o_client" signature="disconnect(int)"/>
    <value ref="2"/>
  </sutSignature/>
</memberMapping>
</mappings>
```

Parameter values within structures can be referenced with the value **ref** attribute. The form of the reference is:

```
<parameter position> [ ("." <ATS field>)* ]
```

The number after period references the field in the parameter's structure. The referencing scheme is recursive so it can reference a field's field.

This XML segment maps the ATS *host* structure with two fields to the 1st and 2nd parameters of the 1st and 2nd parameters in the SUT connect(string,int):

```
<mappings>
  <memberMapping>
    <atsSignature name="client" signature="connect(host)"/>
    <sutSignature object="o_client" signature="connect(string,int)">
      <value ref="1.1"/>
      <value ref="1.2"/>
    </sutSignature>
  </memberMapping>
</mappings>
```

Input to a SUT method can be generated by both ATS and the driver. Describe ATS input with [value](#)'s **ref** attribute, the driver input with [value](#) elements and list them after the [sutSignature](#) in the position corresponding the subject parameter. The value element can describe an explicit value or reference another value. This XML segment maps the ATS connect(host) to SUT connect(string,string,int), but the 1st SUT parameter is supplied as an explicit value, the 2nd SUT parameter is supplied by ATS and the 3rd parameter references the SERVER_PORT constant.:

```
<mappings>
  <memberMapping>
    <atsSignature name="client" signature="connect(host)"/>
    <sutSignature object="o_client" signature="connect(string,string,int)">
      <value type="string" >Connecting Test</value>
      <value ref="1"/>
      <value ref="SERVER_PORT">
    </sutSignature>
  </memberMapping>
</mappings>
```

You can specify special time related handling when executing a member with the timeout, delay, pollingTime and wait attributes. In this XML segment the invocation of connect() timeouts after 99 milliseconds and the driver delays another 10 milliseconds before doing its next external action.

```
< mappings >  
  < memberMapping >  
    < atsSignature name="client" signature="connect(host)" />  
    < sutSignature object="o_client" signature="*" timeout="99" delay="10" />  
  < /memberMapping >  
< / mappings >
```