Oracle® Tuxedo

Service Component Architecture

12*c* Release 1 (12.1.1)

June 2012

ORACLE

Oracle Tuxedo Programming an Oracle Tuxedo Application Using Java, 12*c* Release 1 (12.1.1)

Contents

1. Administering Oracle Tuxedo SCA Components

2. Oracle Tuxedo SCA Programming

3. SCA Command Reference

4. Oracle Tuxedo SCA Sample Applications

A. Oracle Tuxedo SCA ATMI Binding Reference

B. Appendix B: Oracle Tuxedo SCA Schemas

CHAPTER 1

Administering Oracle Tuxedo SCA Components

This chapter contains the following sections:

- Oracle SALT SCA Deployment Model
- Configuring Oracle Tuxedo SCA Components
- Administering Oracle Tuxedo SCA Components

Oracle Tuxedo SCA Deployment Model

An SCA composite is typically described in an associated configuration file, the file name ends with ".composite". This file uses an XML-based format call the Service Component Definition Language (SCDL) to describe the components this composite contains and specify how they related to one another. Deploying Oracle Tuxedo SCA requires at least one root composite file that is located in `$APPDIR`.

There are two configuration file types:

- SCA Composite Configuration File (`.composite`)
- SCA Component Configuration File (`.componentType`)

There can be one or more components configured in the root composite file, and each of these components has its own `.composite` and `.componentType` file residing in its own subdirectory.

SCA Composite Configuration File

There can be zero or more component elements within a composite. The root composite files must be stored in `$APPDIR` in a server environment.

Listing 1-1shows an example of a root composite which contains two components:

Listing 1-1 Root Composite with Two Components

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="ECHO.app">
       <component name="ECHO">
              <implementation.composite name="ECHO" />
       </component>
       <component name="TOUPPER">
              <implementation.composite name="TOUPPER" />
       </component>
</composite>
```

Based on the configuration in Listing 1-1, Listing 1-2 shows the implied the directory hierarchy.

Listing 1-2 SCA Composite Directory Hierarchy

```
$APPDIR/ECHO.app.composite
$APPDIR/ECHO
$APPDIR/ECHO/ECHO.composite
$APPDIR/ECHO/ECHO.componentType
$APPDIR/TOUPPER
$APPDIR/TOUPPER/TOUPPER.composite
$APPDIR/TOUPPER/TOUPPER.componentType
```

This example is a typical server configuration. The Oracle Tuxedo SCA client also has a similar application topology meaning that the client application is located in a subdirectory of the root

composite file. Listing 1-3 lists the directory structure for a client named `EchoClient` that uses the `ECHO1` service provided by `ECHO`.

Listing 1-3 Directory Structure

```
$APPDIR/root.composite
$APPDIR/EchoClient/EchoClient.composite
$APPDIR/EchoClient.composite
$APPDIR/EchoClient/EchoClient.dll
$APPDIR/EchoClient/EchoClient.exe
```

Note: One slight difference between an SCA server environment and an SCA client environment is that there is no need to have a component configuration file in the client environment.

SCA Component Configuration File

Components are the basic elements of business function in an SCA assembly, which are combined into complete business solutions by SCA composites. Components are configured instances of implementations. Components provide and consume services. More than one component can use and configure the same implementation, where each component configures the implementation differently.

Components are declared as sub-elements of a composite in an `xxx.composite` file. A component is represented by a component element that is a child of the composite element. Using the composite from Listing 1-1, the 2 components (`ECHO` and `TOUPPER`), contains specific information. For the `ECHO` service (`$APPDIR/ECHO/ECHO.composite`), the `ECHO.composite` information is shown in Listing 1-4.

Listing 1-4 ECHO.composite

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
       name="ECHO">
       <service name="ECHO">
              <interface.cpp header="ECHO.h" />
```

```
                <binding.atmi requires="legacy">
                        <map target="EchoString1">ECHO1</map>
                        <map target="EchoString2">ECHO2</map>
                </binding.atmi>
                <reference>EchoServiceComponent</reference>
        </service>
        <component name="EchoServiceComponent">
                <implementation.cpp library="ECHO" header="ECHOImpl.h" />
        </component>
</composite>
```

The `ECHO` service provides two Oracle Tuxedo services: `ECHO1` and `ECHO2`. `ECHO1` executes CPP function "`EchoString1`". `ECHO2` executes CPP function `"EchoString2"`. The existence of `$APPDIR/ECHO/ECHOImpl.componentType` and `$APPDIR/ECHO/ECHO.so`. are implied. Listing 1-5 shows information that may be contained in `ECHOImpl.componentType`.

Note: On some Unix systems the suffix is .so.71 or .sl.

`ECHO.so` (or `ECHO.dll` Windows), is the shared library that contains the actual implementation of `EchoString1` and `EchoString2` and is loaded into memory when the service is initialized. `ECHO1` and `ECHO2` are dynamically advertised at server initialization. For example, if `EchoServer` is the Oracle Tuxedo server that provides these two services, the Oracle Tuxedo UBBCONFIG file should contain information as shown in Listing 1-6.

Listing 1-5 ECHOImpl.componentType

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <service name="ECHO">
                <interface.cpp header="ECHO.h"/>
        </service>
</componentType>
```

Listing 1-6 UBBCONFIG File Example

```
...
*SERVERS
DEFAULT:
       CLOPT="-A"
EchoServer  SRVGRP=GROUP1 SRVID=1001
...
```

For the TOUPPER service, the existence of $APPDIR/TOUPPER/TOUPPER.composite is also implied by the ECHO.app.composite file. Listing 1-7 shows information that may be contained in TOUPPER.composite file.

Listing 1-7 TOUPPER.composite file Example

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
       name="TOUPPER">
       <service name="TOUPPER">
              <interface.cpp header="TOUPPER.h" />
                     <binding.atmi requires="legacy">
                            <map target="UpperString1">TOUPPER1</map>
                            <map target="UpperString2">TOUPPER2</map>
                     </binding.atmi>
              <reference>ToupperServiceComponent</reference>
       </service>

       <component name="ToupperServiceComponent">
              <implementation.cpp library="TOUPPER" header="TOUPPERImpl.h"
/>
       </component>
</composite>
```

This composite file also implies the existence of `$APPDIR/TOUPPER/TOUPPERImpl.componentType` and `$APPDIR/TOUPPER/TOUPPER.so`.

Note: Oracle Tuxedo SCA only supports `"cpp"` implementation types.

Configuring Oracle Tuxedo SCA Components

Configuring Oracle Tuxedo SCA components comprises the following:

- Configuring an SCA ATMI Client
- Configuring an SCA Workstation Client
- Configuring an SCA Workstation Client
- The above SCA component are hosted in an Oracle Tuxedo server built using `buildscaserver` with the -w option (for Web services) and named `WSServer`
- The above SCA component are hosted in an Oracle Tuxedo server built using `buildscaserver` with the -w option (for Web services) and named `WSServer`
- Configuring an SCA Web Service Server
- Configuring SCA Client Security

Configuring an SCA ATMI Client

The SCA ATMI client is a native Oracle Tuxedo client that is written using the SCA paradigm and built using the `buildscaclient` utility. The client executable must be in a subdirectory of a directory that contains the `root.composite` file.

Note: The `APPDIR` environment variable must point to the `root.composite` file directory.

Listing 1-8 shows the client application root composite file `$APPDIR/root.composite`.

Listing 1-8 Client Application Root Composite File

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="testApp">
       <component name="testStringClientComp">
              <implementation.composite name="ECHO"/>
       </component>
</composite>
```

The $APPDIR/ECHO directory contains the ECHO application. The directory name, "ECHO", must match the name specified in <implementation.composite name="ECHO"/>. Listing 1-9 shows the client application composite file.

Listing 1-9 Client Application Composite File

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="ECHO">
  <reference name="ECHO">
    <interface.cpp header="ECHO.h"/>
      <binding.atmi requires="legacy">
        <tuxconfig>/tux/application/ECHOServer/tuxconfig</tuxconfig>
          <inputBufferType target="TestString">STRING</inputBufferType>
          <outputBufferType target="TestString">STRING</outputBufferType>
          <errorBufferType target="TestString">STRING</errorBufferType>
       <binding.atmi>
  </reference>
</composite>
```

The client dynamic link library for this client application is also contained in this directory. For example, using the example in Listing 1-9, the $APPDIR/ECHO/ECHO.so shared object exists in the ECHO directory. The target "TestStr" is used to group buffer types together.

The client executable also exists in this directory. There is no naming convention associated with a client application. This client ECHO application could very well contain "doEchoClient" in the ECHO application directory. For example: $APPDIR/ECHO/doEchoClient.

Note: You must set SCA_COMPONENT. See Listing 1-9, SCA_COMPONENT=testStringClientComp.

Configuring an SCA JATMI Client

The JATMI client application configuration composite file is part of the Java .jar file. The JATMI client composite file is not part of any package and is located in the base of the .jar file.

When client application is invoked, SCA Java runtime loads the composite file. No special setup is required.

Note: The client application `.jar` file *must* be included in the `CLASSPATH`. The following `.jar` files should also be part of `CLASSPATH`:

- `binding-jatmi-extension.jar`
- `com.oracle.jatmi.dataxfm_1.0.0.0.jar`
- `com.bea.core.jatmi_1.2.0.3.jar`
- `com.bea.core.i18n_1.4.0.0.jar`
- `tuscany-sca-manifest.jar`

Listing 1-10 shows an SCA JATMI client composite file example.

Listing 1-10 SCA JATMI Client Composite File Example

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
xmlns:f="binding-atmi.xsd"
name="EchoComposite">
       <reference name="ECHO" promote="EchoComponent/ECHO">
              <interface.java class="com.abc.sca.java.Echo" />
              <f:binding.atmi requires="legacy">
                     <f:serviceType>RequestResponse</f:serviceType>
                     <f:inputBufferType>FML</f:inputBufferType>
                     <f:outputBufferType>FML</f:outputBufferType>
                     <f:fieldTables>com.abc.sca.java.fml.FMLTABLE
                     </f:fieldTables>
                     <f:workStationParameters>
                            <f:networkAddress>//STRIATUM:15011
                            </f:networkAddress>
                     </f:workStationParameters>
              </f:binding.atmi>
       </reference>
       <component name="EchoComponent">
              <implementation.java
              class="com.abc.sca.java.EchoComponentImpl />
```

```
       </component>
</composite>
```

Configuring an SCA Workstation Client

Configuring an SCA workstation clients is similar to configuring SCA native clients. One difference is that an SCA workstation client requires using the `<workStationParameters>` element and its sub-elements in the composite. The SCA runtime automatically detects whether the client is built as an SCA native client or SCA workstation client and loads the correct reference binding library accordingly.

An SCA Oracle Tuxedo Workstation client has a similar directory hierarchy to an SCA native client. Both rely on the environment variable `$APPDIR`, which points to where the client application is located.

Listing 1-11 and Listing 1-12 show SCA Oracle Tuxedo workstation client configuration examples.

Listing 1-11 $APPDIR/root.composite

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="testApp">
       <component name="testStringClientComp">
              <implementation.composite name="ECHO"/>
       </component>
</composite>
```

Listing 1-12 $APPDIR/ECHO/ECHO.composite

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="ECHO">
  <reference name="ECHO">
    <interface.cpp header="ECHO.h"/>
      <binding.atmi requires="legacy">
        <inputBufferType target="TestString">STRING</inputBufferType>
```

```
        <outputBufferType target="TestString">STRING</outputBufferType>
        <errorBufferType target="TestString">STRING</errorBufferType>
      <workStationParameters>
        <networkAddress>//STRIATUM:4890</networkAddress>
        <encryptBits>128/128</encryptBits>
      </workStationParameters>
      <remoteAccess>WorkStation</remoteAccess>
      </binding.atmi>
  <reference>
</composite>
```

Configuring an SCA Web Service Client

The SCA Web service client is basically the same as SCA native client except that is uses the `<binding.ws>` element instead of `<binding.atmi>`. The SCA runtime automatically detects which binding the client is using, and loads the correct reference binding accordingly.

The SCA Web service client has a similar directory hierarchy as native client. They both rely on the `$APPDIR` environment variable to point to where the client application is located.

Listing 1-13 and Listing 1-14 show SCA Web service client configuration examples.

Listing 1-13 $APPDIR/root.composite

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="testApp">
       <component name="calcClient">
              <implementation.composite name="calcClient"/>
       </component>
</composite>
```

Listing 1-14 $APPDIR/calcClient/calcClient.composite

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"name="calcClient">
       <reference name="Calculator">
```

```
            <interface.cpp header="CalcService.h"/>
            <binding.ws
            endpoint="http://calc.sample#wsdl.endpoint
            (Calculator/CalculatorSOAP11port)"/>
        </reference>

</composite>
```

The <interface.cpp> element is required to build the appropriate proxy stub. Also, the client directory should contain the WSDL file where the endpoint specified in <binding.ws> is located. In addition, the configuration of the Oracle Tuxedo Web services gateway (GWWS) is necessary and requires the following steps:

1. Make sure the TMMETADATA and GWWS servers are shut down.
2. Run wsdlcvt on the WSDL of the service(s) used. This produces a WSDF file, an Oracle Tuxedo Metadata Repository interface definitions file, fml32 field tables and XML schemas.
3. Optionally, modify the generated WSDF file to override the actual endpoint address used at runtime. For more information, see WSDF documentation.
4. Load the Oracle Tuxedo Metadata Repository interface definitions into the TMMETADATA server repository (e.g.: `$ tmloadrepos -I calc.mif metadata.repos -y).` For more information, see tmloadrepos documentation.
5. Add a reference to the WSDF in the GWWS configuration input file (named gwws.dep for example). Listing 1-15 shows the added elements highlighted in blue.
6. Reload the GWWS binary configuration file to take into account the changes performed in the previous five (e.g.: `$ wsloadcf -y gwws.dep)`.
7. Reboot GWWS and TMMETADATA.

Listing 1-15 GWWS Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<saltdep:Deployment
xmlns:saltdep="http://www.bea.com/Tuxedo/SALTDEPLOY/2007"
xmlns="http://www.bea.com/Tuxedo/SALTDEPLOY/2007"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
       <saltdep:WSDF>
              <saltdep:Import location="calc.wsdf"/>
       </saltdep:WSDF>
       <saltdep:WSGateway>
              <saltdep:GWInstance id="GWWS1">
                     <saltdep:Outbound>
                     <saltdep:Binding ref="calc:CalculatorSOAP11Binding">
                     <saltdep:Endpoint use="CalculatorSOAP11port"/>
                     </saltdep:Binding>
                     </saltdep:Outbound>
              </saltdep:GWInstance>
       </saltdep:WSGateway>
        <saltdep:System/>
</saltdep:Deployment>
```

Configuring an SCA ATMI Server

For an SCA ATMI server, the SCA ROOT is the same as `$APPDIR`. There should be at least one composite file that describes the SCA application. The SCA runtime searches for this composite file and from there it loads all the `composite` and `componentType` files for SCA server applications that are hosted in an Oracle Tuxedo environment.

Listing 1-16 shows a root composite file, named `root.composite` contains two SCA applications hosted in an Oracle Tuxedo application domain. The two applications are called Purchase and Finance. There are at least two subdirectories for these two SCA applications. One is called `Purchase.component` and the other is called `Finance.component`.

Listing 1-16 $APPDIR/root.composite

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="root">
       <component name="Purchase.component">
              <implementation.composite name="Purchase" />
       </component>
       <component name="Finance.component">
```

```
            <implementation.composite name="Finance" />
      </component>
</composite>
```

Listing 1-17 shows the `Purchase.component` directory contains a composite file for the Purchase application named `Purchase.composite`. Similarly, the `Finance.component` directory contains a composite file for the Finance application named `Finance.composite`.

Listing 1-17 $APPDIR/Purchase.component/Purchase.composite

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
      name="Purchase">
      <service name="purchase">
            <interface.cpp header="Purchase.h" />
      <binding.atmi requires="legacy">
            <map target="Order">ORDER</map>
            <map target="TrackOrder">TRACKORDER</map>
      </binding.atmi>
      <reference>PurchaseServiceComponent</reference>
      </service>
      <component name="PurchaseServiceComponent">
            <implementation.cpp library="Purchase"
header="PurchaseImpl.h" />
      </component>
</composite>
```

Listing 1-18 shows `Purchase.composite` contains the `PurchaseImpl.componentType file` in the `$APPDIR/Purchase.component` directory and uses CPP as its application implementation. When an SCA server using this configuration is built using the `buildscaserver` utility, it advertises two SCA services automatically at runtime (`ORDER` and `TRACKORDER`). The actual CPP implementation of the services is called `Order` and `TrackOrder`.

Listing 1-18 $APPDIR/Purchase.component/PurchaseImpl.componentType

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0"
               xmlns:xs="http://www.w3.org/2001/XMLSchema">
       <service name="purchase">
              <interface.cpp header="Purchase.h"/>
       </service>
</componentType>
```

Assume these two SCA applications hosted in Oracle Tuxedo and built using `buildscaserver` are called `PurchaseSvr` and `FinanceSvr`. You must add the following lines to the *SERVERS section in the UBBCONFIG file:

```
PurchaseSvr SRVGRP=PURCHASEGRP SRVID=500
FinanceSvr SRVGRP=FINANCEGRP SRVID=600
```

There is no need to add a service in the *SERVICES section. SCA services hosted by Oracle Tuxedo are dynamically advertised.

Configuring an SCA Web Service Server

Configuring Web services binding for components (server side) is similar to configuring ATMI binding for hosting SCA components.

Listing 1-19 shows a root composite file named `root.composite`. It contains one SCA component hosted in an Oracle Tuxedo application domain. The two applications are called Purchase and Finance. There are at least two subdirectories for these two SCA applications, one is called `Purchase.component`, and the other is called `Finance.component`.

Listing 1-20 shows the actual component subdirectory. Listing 1-21 shows the `componentType` side file

Listing 1-19 $APPDIR/root.composite

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="root">
       <component name="account">
```

```
              <implementation.composite name="account" />
       </component>
</composite>
```

Listing 1-20 $APPDIR/account/account.composite

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
       name="account">
              <service name="AccountService">
                     <interface.wsdl
interface="http://www.bigbank.com/AccountService#wsdl.interface(AccountSer
vice)"/>
                     <binding.ws/>
                            <reference>AccountServiceComponent</reference>
              </service>

       <component name="AccountServiceComponent">
              <implementation.cpp library="Account"
header="AccountServiceImpl.h"/>
       </component>
</composite>
```

Listing 1-21 $APPDIR/account/AccountServiceImpl.componentType

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0"
       xmlns:xs="http://www.w3.org/2001/XMLSchema">
       <service name="AccountService">
              <interface.cpp header="AccountService.h"/>
       </service>
</componentType>
```

The above SCA component are hosted in an Oracle Tuxedo server built using `buildscaserver` with the -w option (for Web services) and named `WSServer`

Then in the Oracle Tuxedo UBBCONFIG file you need to add the following line in the *SERVERS section: `WSServer SRVGRP=ACCTGRP SRVID=500`.

There is no need add a service in the *SERVICES section. SCA services hosted by Oracle Tuxedo are dynamically advertised.

In addition, configuration of the Oracle Tuxedo Web services gateway (GWWS) is necessary. Do the following steps:

1. Make sure the `TMMETADATA` and GWWS servers are shut down
2. Run `wsdlcvt` on the WSDL of the service(s) used. This produces a WSDF file, an Oracle Tuxedo Metadata Repository interface definitions file, fml32 field tables and XML schemas.
3. Modify the generated WSDF file to specify the actual endpoint address used at runtime to accept requests. For more information, see WSDF documentation.
4. Load the Oracle Tuxedo Metadata Repository interface definitions into the `TMMETADATA` server repository (for example, `$ tmloadrepos -I AccountService.mif metadata.repos -y`). For more information, see `tmloadrepos` documentation.
5. Add a reference to the WSDF in the GWWS configuration input file (named `gwws.dep` for example). Listing 1-22 shows the elements added highlighted in blue.
6. Reload the GWWS binary configuration file to take into account the changes performed in the step five (e.g.: `$ wsloadcf -y gwws.dep`).
7. Reboot GWWS and `TMMETADATA`.

Listing 1-22 gwws.dep File

```
<?xml version="1.0" encoding="UTF-8"?>
<saltdep:Deployment
xmlns:saltdep="http://www.bea.com/Tuxedo/SALTDEPLOY/2007"
xmlns="http://www.bea.com/Tuxedo/SALTDEPLOY/2007"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
       <saltdep:WSDF>
              <saltdep:Import location="AccountService.wsdf"/>
       </saltdep:WSDF>
       <saltdep:WSGateway>
```

```
            <saltdep:GWInstance id="GWWS1">
                  <saltdep:Inbound>
                        <saltdep:Binding
ref="AccountService:AccountServiceSOAP">
                        <saltdep:Endpoint use="AccountServiceSOAP"/>
                            </saltdep:Binding>
                  </saltdep:Inbound>
            </saltdep:GWInstance>
      </saltdep:WSGateway>
      <saltdep:System/>
</saltdep:Deployment>
```

Configuring SCA Client Security

Oracle Tuxedo SCA components support two types of security:

- Oracle Tuxedo Application Domain Security
- Oracle Tuxedo Link-Level Security

Oracle Tuxedo Application Domain Security

Oracle Tuxedo Application Domain Security is set when the `TUXCONFIG` file for the Oracle Tuxedo Application Domain contains the `SECURITY` keyword in the `*RESOURCES` section. There are five levels of application security: `NONE`, `APP_PW`, `USER_PW`, `ACL`, and `MANDATORY_ACL`. All security levels except `NONE` require at least an application password from user to gain access to the Oracle Tuxedo application. At the `USER_PW` level and above there is an additional user password to authenticate the user and establish user credentials. In total there are potentially two passwords that need to be configured.

All SCA clients require this password information in order to gain access to Oracle Tuxedo application servers. There are two ways for an SCA client to retrieve password information:

- The client application may provide password information to ATMI/JATMI reference binding extensions through a callback mechanism.
- The client application may configure the identification of the password to be retrieved by the ATMI/JATMI reference binding extensions in the appropriate composite file.

Note: For more information, see Password callback methods in Oracle Tuxedo SCA Programming.

In order for the Oracle Tuxedo administrator to configure password retrieval, the administrator must:

- Maintain the `password.store` file and set this file up correctly for the client application. The administrator must duplicate the `password.store` file across different machines if necessary.
- Add or delete password and identification pairs when necessary.
- Configure the client application composite file with correct information.

Listing 1-23 and Listing 1-24 contain SCA ATMI client application examples.

Listing 1-23 $APPDIR/password.store $APPDIR/simple.app.composite

```
<?xml version="1.0" encoding="UTF-8"?>

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
       name="simple.app">

       <component name="simpapp.TuxClient">
              <implementation.composite name="simpapp.client"/>
              <reference name="TOUPPER">tuxToupper</reference>
       </component>

</composite>
```

Listing 1-24 $APPDIR/simpapp.client/simpapp.client.composite

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
name="simpapp.client">

  <reference name="TOUPPER">
    <interface.cpp header="ToupperTuxService.h"/>
    <binding.atmi requires="legacy">
```

```
      <tuxconfig>d:\tests\tuxedo\sca\tests\TUXCONFIG</tuxconfig>
      <inputBufferType target="charToup">STRING</inputBufferType>
      <outputBufferType target="charToup">STRING</outputBufferType>
        <authentication
          <passwordIdentifier>aaa</passwordIdentifier>
        </authentication>
    </binding.atmi>
  </reference>
</composite>
```

The above composite defines an Oracle Tuxedo application domain password identification "aaa" which causes the ATMI reference binding to retrieve the password with identification "aaa" from the `password.store` file at the runtime. If you increased Oracle Tuxedo application domain security by requiring user authentication. (`SECURITY=USER_PW` or above) you would use the following command: `scapasswordtool -i crusoe -a`.

Then use a text editor or any other tool that can edit the `simpapp.client.composite` file and add the following entry in the `<binding.atmi/authentication>` element: `<userPasswordIdentifier>crusoe</userPasswordIdentifier>`

Anyone using the password "crusoe" can access Oracle Tuxedo applications.

Oracle Tuxedo Link-Level Security

Oracle Tuxedo Link-Level Security has two variations. One is the easily configured Link-Level Encryption (LLE) and the other one is the more commonly used Transport Layer Security (TLS) also known as Secured Socket Layer (SSL). An SCA ATMI client using the native ATMI reference binding does not need link-level security configured at the SCA level since its transport method is native message queues and the Oracle Tuxedo BRIDGE.

The SCA JATMI client reference binding does not support link-level security. The only type of SCA client that allows configuration of link-level security is SCA Workstation ATMI client.

The SCA Workstation ATMI client contains a `<workStationParameters>` element configured in the composite file. The SCA runtime automatically loads the correct reference binding for this type of client.

Configuring Link-Level Encryption

Link-level encryption can be configured by adding an <encryptBits> element in the composite file. The following elements *should not* be configured for LLE, since they are specific to SSL encryption and imply that SSL encryption is used:

- secPrincipalName
- secPrincipalLocation
- secPrincipalPassId

The <encryptBits> element specifies the encryption strength that this client attempts to negotiate. The syntax for the <encryptBits> element is <minimum encryption strength>/<maximum encryption strength>. To configure minimum 56-bit encryption you must add the following to the composite file:

```
<networkAddress>//STRIATUM:8741</networkAddress>
<encryptBits>56/128</encryptBits>
```

Note: encryptBits specifies the encryption strength that the client connection attempts to negotiate. The format is <minencryptbits>/<maxencprytbits> (for example, 128/128). Values can be 0 (no encryption is used), 40, 56, 128, or 256. Invalid values result in a configuration exception.

This tells SCA Workstation Reference binding to require 56 to 128 bits encryption strength when negotiating with WSH. You must also add the following line to the *SERVERS section in the UBBCONFIG file:

```
WSL   SRVGRP=GROUP1 SRVID=1001 CLOPT="-A -- -n //STRIATUM:8741 -a -z 56 -Z
256
```

Configuring Transport Layer Security

In addition to <encryptBits>, to enable Link-Level Security over TLS/SSL you must configure secPrincipalName, secPrincipalLocation, and secPrincipalPassId.

- secPrincipalName - the name of the security principal. It is used for searching the client X.509 certification from the LDAP server.
- secPrincipalLocation - the client private key file.
- secPrincipalPassId - the password identifier that is used to retrieve client password used to encrypt the private key file.

Note: The "cn" attribute of a distinguished name is used as key for certificate lookup. Wildcards used in a name are not supported. Empty subject fields are not allowed. This limitation is also found in Oracle Tuxedo.

These three parameters specify the parameters needed when a TLS/SSL connection needs to be established by a SCA Workstation ATMI client.

Listing 1-25 contains the lines you must add to the client composite file in `/binding.atmi/workStationParameters` to configure TLS/SSL.

Listing 1-25 Client Composite File

```
<networkAddress>//STRITUM:8742</networkAddress>
<secPrincipalName>crusoe</secPrincipalName>
<secPrincipalLocation>/tux/udataobj/security/keys/crusoe.pem</secPrincipal
Location>
<secPrincipalPassId>crusoe</secPrincipalPassId>
```

In Oracle Tuxedo, you must add `-S 8742` to WSL to indicate that TLS/SSL is used if the client connects through port 8742.

```
WSL  SRVGRP=GROUP1 SRVID=1001
       CLOPT="-A -- -n //STRIATUM:8741 -S 8742 -z 56 -Z 128"
```

Administering Oracle Tuxedo SCA Components

This section contains the following topics:

- Tracing the SCA ATMI Server and Client
- Log File Contents
- Log File Contents

Tracing the SCA ATMI Server and Client

Both The SCA ATMI server and client can utilized the existing tracing capability provided by Oracle Tuxedo and SCA. The following sections describe how to use them in detail:

- Oracle Tuxedo TMTRACE

- SCA Runtime, ATMI Service, and Reference Binding Tracing

Oracle Tuxedo TMTRACE

SCA ATMI servers and clients support the Oracle Tuxedo `tmtrace(5)` function. All traces generated from `TMTRACE` are logged in the ULOG file. Checking the ULOG file trace information helps to determine the cause of a failure. The Oracle Tuxedo `TMTRACE` facility is enabled by setting `TMTRACE` environmental variable, or by using the `tmadmin chtr` sub-command.

Note: To trace Oracle Tuxedo ATMI messages enter: `export TMTRACE=atmi:ulog` at the command line.

SCA Runtime, ATMI Service, and Reference Binding Tracing

There are two environment variables used for tracing:

- `SCACPP_LOGGING`: Set to a numeric value and controls the number of trace messages produced.
- `SCACPP_ULOG`: Set to "yes" to send trace messages to the ULOG. If this environment variable is not set or is set to "no", then trace messages are written to standard output.

Note: These tracing facilities are only available for Oracle Tuxedo server builds using `buildscaserver` and SCA client builds using `buildscaclient`.

Listing 1-26 shows a ULOG example containing SCA runtime tracing:

Note: Lines starting with ">>" or with "<<" is not printed when the code is compiled

Listing 1-26 SCA Runtime Tracing Information ULOG File

```
142059.STRIATUM!?proc.1108.3000.-2:
osoa::sca::CompositeContext::getCurrent
142059.STRIATUM!?proc.1108.3000.-2:     >>
Tuscany::sca::SCARuntime::getCurrent Runtime
142059.STRIATUM!?proc.1108.3000.-2:     >>
tuscany::sca::util::ThreadLocal::getValu e
142059.STRIATUM!?proc.1108.3000.-2:     <<
tuscany::sca::util::ThreadLocal::getValu e
142059.STRIATUM!?proc.1108.3000.-2:     >>
tuscany::sca::SCARuntime::getShared Runtime
```

```
142059.STRIATUM!?proc.1108.3000.-2:     SCARuntime::getSharedRuntime()
142059.STRIATUM!?proc.1108.3000.-2:     >> tuscany::sca::util::Mutex::lock
142059.STRIATUM!?proc.1108.3000.-2:     << tuscany::sca::util::Mutex::lock
142059.STRIATUM!?proc.1108.3000.-2:     >>
tuscany::sca::util::Mutex::unlock
142059.STRIATUM!?proc.1108.3000.-2:     <<
tuscany::sca::util::Mutex::unlock
142059.STRIATUM!?proc.1108.3000.-2:     <<
tuscany::sca::SCARuntime::getSharedR untime
142059.STRIATUM!?proc.1108.3000.-2:     >>
tuscany::sca::util::ThreadLocal::Thread Local
142059.STRIATUM!?proc.1108.3000.-2:     <<
tuscany::sca::util::ThreadLocal::Thread Local
142059.STRIATUM!?proc.1108.3000.-2:     >>
tuscany::sca::SCARuntime::SCARuntime
142059.STRIATUM!?proc.1108.3000.-2:     SCA runtime install root
f:\tuxedo\tux101rp _wsc\udataobj\salt\sca
142059.STRIATUM!?proc.1108.3000.-2:     Default component:
testStringClientComp
142059.STRIATUM!?proc.1108.3000.-2:     >>
tuscany::sca::util::ThreadLocal::getValu e
142059.STRIATUM!?proc.1108.3000.-2:     <<
tuscany::sca::util::ThreadLocal::getValu e
```

Monitoring SCA ATMI Servers

An Oracle Tuxedo SCA server built with the `buildscaserver` utility can be monitored using the `scaadmin` utility. This utility shows service statistics information and helps perform maintenance through dynamic shared library loading and unloading.

To reload all components hosted by the `uBikeServer` Oracle Tuxedo server previously built using the `buildscaserver` command, do the following:

1. `prompt> scaadmin`
2. `prompt> reload -s uBikeServer`

Enter the following at the command line to display statistics on the services offered by the `uBikeServer` Oracle Tuxedo server (Table 1-1 shows the results):

1. `prompt> scaadmin`
2. `prompt> pstats -s uBikeServer`

Table 1-1 pstats Output Service Statics

Service	Method	Status	Requests Processed
SEARCHINVENTORY	searchInventory	A	37

Before `scaadmin` is executed, you must set the `TUXCONFIG` environment variable. Table 1-2 lists `scaadmin` sub-commands.

Table 1-2 scaadmin Sub-Commands

Sub-Command	Abbrev.	Description
default	d	Sets the corresponding argument to default, and it can be machine name, group name, server id, or server name. If the default command is entered without an argument, the current default values is printed.
reload	r	Dynamically reloads the SCA components hosted in a Oracle Tuxedo server.
printstats	pstats	Displays the list of services hosted by an Oracle Tuxedo server, and the associated method, number of queries, and status (active, idle)
verbose	v	Produces output in verbose mode.
echo	e	Switches echo input on/off echo.
quit	q	Terminates the session.

Note: Both Windows and HP systems have a limitation using the "reload" sub-command.

When multiple servers share the same component library on Windows and HP systems, the shared component library cannot be reloaded. To reload a component library common to multiple servers, the "`scaadmin`" reload sub-command must be performed on all affected servers simultaneously.

Tracing SCA JATMI Clients

The Oracle Tuxedo SCA Java reference binding and data transformation support output to the console and to a log file. By default there are at most 10 log files, the maximum size of each file is 100000 bytes, and are located in `$APPDIR` with name `jatmi<number>.log` file. The log file names are cycled with the latest one using the number 0, and the one just before latest one uses 1 (for example. `jatmi0.log` is the latest log file, and `jatmi9.log` is the oldest log file). If the `APPDIR` environment variable is not set and `com.oracle.jatmi.APPDIR` java property is not specified, the log is placed in the current working directory.

By default, the log files are overwritten each time the application starts. Many logger parameters can be fine tuned. Table 1-3 lists tunable Java properties related to logging.

Table 1-3 Logger Tuning Property Table

Function	Properties	Value Range	Default Value
Log File Location	`com.oracle.jatmi.APPDIR`	valid path name	`APPDIR` environmental variable, if `APPDIR` is not set uses current working directory
Log File Size	`com.oracle.jatmi.LogFileSize`	`0` ... maximum file size supported by the system	`100,000 bytes`
Append File	`com.oracle.jatmi.LogFileAppend`	`{true,false}`	`false`
Number of Log Files	`com.oracle.jatmi.LogFileCount`	1 ... maximum number of files in a directory	10
Log Output	`com.oracle.jatmi.LogDestination`	`{file,console,both}`	`both`
Log Format	`com.oracle.jatmi.LogFileFormat`	`{xml,plain}`	`plain`

To have the Oracle Tuxedo SCA Java reference binding log in a different language, first check the supported languages that are installed. The default is `English`. To switch to a different language, add: "`-Duser.language=<your preferred language>`" to your Java command line when starting the Oracle Tuxedo SCA Java client. For example:

```
java -classpath .:/apps/classes:$CLASSPATH -Duser.langueage=ES
-Dcom.oracle.jatmi.LogDestination=console myApplication.
```

This generates an English log in plain text format to the console only.

Table 1-3 shows an example of the log file contents.

Listing 1-27 Log File Contents

```
9/3/08:3:19:14 PM:10:TRACE[TuxedoConversion,processSendBuf]< (10) return
1st args
9/3/08:3:19:14 PM:10:DBG[AtmiBindingInvoker,invoke]ServiceType:
requestresponse
9/3/08:3:19:14 PM:10:DBG[AtmiBindingInvoker,invoke]Return Type Class:
simpapp.View7Rep
9/3/08:3:19:14 PM:10:DBG[AtmiBindingInvoker,invoke]target service name:
RULE7
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]TPURCODE: 0
9/3/08:3:19:15 PM:10:TRACE[TuxedoConversion,processReplyBuffer]> (reply
simpapp.View7Rep@191777e:0:null)
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processReplyBuffer]returnType:
simpapp.View7Rep
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processReplyBuffer]Reply Buffer
Class: simpapp.View7Rep
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processReplyBuffer]Reply Buffer
Type: X_COMMON
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processReplyBuffer]Reply Buffer
Subtype: View7Rep
9/3/08:3:19:15 PM:10:TRACE[TuxedoConversion,processReplyBuffer]< (30)
return buffer directly
9/3/08:3:19:15 PM:10:DBG[Accessors,getConventionProperty]Convention
Property: CONVENTIONS_TUX
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]networkAddress: host =
STRIATUM, port = 8080
9/3/08:3:19:15
PM:10:TRACE[AtmiBindingInvoker,determineServiceCallParameters]> ()
9/3/08:3:19:15 PM:10:DBG[AtmiBindingImpl,isLegacy]> ()
9/3/08:3:19:15 PM:10:DBG[AtmiBindingImpl,isLegacy]< (10) return true
9/3/08:3:19:15 PM:10:DBG[AtmiBindingImpl,isMap]> ()
```

```
9/3/08:3:19:15 PM:10:DBG[AtmiBindingImpl,isMap]< (10) return false
9/3/08:3:19:15
PM:10:DBG[AtmiBindingInvoker,determineServiceCallParameters]Operation name
= rule7_OVVO
9/3/08:3:19:15 PM:10:TRACE[AtmiBindingImpl,getServiceType]> (rule7_OVVO)
9/3/08:3:19:15 PM:10:TRACE[AtmiBindingImpl,getServiceType]< (10) return
null
9/3/08:3:19:15 PM:10:TRACE[AtmiBindingImpl,getInputBufferType]>
(rule7_OVVO)
9/3/08:3:19:15 PM:10:TRACE[AtmiBindingImpl,getInputBufferType]< (10)
return null
9/3/08:3:19:15 PM:10:TRACE[AtmiBindingImpl,getOutputBufferType]>
(rule7_OVVO)
9/3/08:3:19:15 PM:10:TRACE[AtmiBindingImpl,getOutputBufferType]< (10)
return null
9/3/08:3:19:15 PM:10:DBG[AtmiBindingImpl,getErrorBufferType]> (rule7_OVVO)
9/3/08:3:19:15 PM:10:DBG[AtmiBindingImpl,getErrorBufferType]< (10) return
null
9/3/08:3:19:15
PM:10:DBG[AtmiBindingInvoker,determineServiceCallParameters]svcName =
RULE7
9/3/08:3:19:15
PM:10:DBG[AtmiBindingInvoker,determineServiceCallParameters]svcType =
requestresponse
9/3/08:3:19:15
PM:10:DBG[AtmiBindingInvoker,determineServiceCallParameters]inbuf =
X_COMMON
9/3/08:3:19:15
PM:10:DBG[AtmiBindingInvoker,determineServiceCallParameters]outbuf =
X_COMMON
9/3/08:3:19:15
PM:10:DBG[AtmiBindingInvoker,determineServiceCallParameters]errbuf = null
9/3/08:3:19:15
PM:10:TRACE[AtmiBindingInvoker,determineServiceCallParameters]< (10)
return
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]Input Buffer Type:
X_COMMON
```

```
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]Output Buffer Type:
X_COMMON
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]Error Buffer Type: null
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]inBufType:X_COMMON,
count: 1
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]outBufType:X_COMMON,
count: 1
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]View Classes:
simpapp.View7Req,simpapp.View7Rep
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,getClassList]getClassList:
Getting class for simpapp.View7Req
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,getClassList]getClassList:
Getting class for simpapp.View7Rep
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,setFieldClasses]setFldClasses:
null
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]Passing thro invoker...
9/3/08:3:19:15 PM:10:TRACE[TuxedoConversion,processSendBuf]> (args
[Ljava.lang.Object;@ab1b4)
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processSendBuf]args[0] class
simpapp.Rule7Req
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,needConversion]buftype: X_COMMON
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processSendBuf]Argument Class
Name: simpapp.Rule7Req
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processSendBuf]Input Buffer Id
: XCOMMON
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processSendBuf]Type code       :
10
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processSendBuf]InputBufferType:
XCOMMON
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,getClassList]getClassList:
Getting class for simpapp.View7Req
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,getClassList]getClassList:
Getting class for simpapp.View7Rep
9/3/08:3:19:15 PM:10:TRACE[Accessors,determineConvention]>
(simpapp.Rule7Req)
9/3/08:3:19:15 PM:10:DBG[Accessors,determineConvention]Method name: getId
```

```
9/3/08:3:19:15 PM:10:DBG[Accessors,determineConvention]Method name: setCmd
9/3/08:3:19:15 PM:10:DBG[Accessors,determineConvention]Method name: setId
9/3/08:3:19:15 PM:10:DBG[Accessors,determineConvention]Method name: getCmd
9/3/08:3:19:15 PM:10:TRACE[Accessors,determineConvention]< (30) return BEAN
```

CHAPTER 2

Oracle Tuxedo SCA Programming

This chapter contains the following topics:

Overview

One important aspect of Service Component Architecture (SCA) is the introduction of a new programming model. As part of the Oracle Tuxedo architecture, SCA allows you to better blend high-output, high-availability and scalable applications in an SOA environment.

SCA components run on top of the Oracle Tuxedo infrastructure using ATMI binding. The ATMI binding implementation provides native Oracle Tuxedo communications between SCA components, as well as SCA components and Oracle Tuxedo programs (clients and servers).

In addition to the programming model, the Service Component Definition Language (SCDL) describes what components can perform in terms of interactions between each other, and instructs the framework to set-up necessary links (wires).

SCA Utilities

The following utilities are used in conjunction with Oracle Tuxedo SCA programming:

- `buildscaclient`: Builds client processes that call SCA components.
- `buildscacomponent`: Builds SCA components.
- `buildscaserver`: Builds an Oracle Tuxedo server containing SCA components.
- `SCAHOST`: Generic server for Python, Ruby or PHP SCA components.
- `scatuxgen`: Generates Oracle Tuxedo Service Metadata Repository interface information from an SCA interface.
- `scastructc32`, `scastructc`: Structure description file compiler.
- `scastructdis32`, `scastructdis`: Binary structure and view files disassembler.
- `tuxscagen`: Generates SCA, SCDL, and server-side interface files for Oracle Tuxedo services.

For more information, see the SCA Command Reference.

SCA Client Programming

The runtime reference binding extension is the implementation of the client-side aspect of the SCA container. It encapsulates the necessary code used to call other services, SCA components, Oracle Tuxedo servers or even Web services, transparently from an SCA-based component.

SCA Client Programming Steps

Developing SCA client programs requires the following steps:

1. Setting Up the Client Directory Structure
2. Developing the Client Application
3. Composing the SCDL Descriptor
4. Building the Client Application
5. Running the Client Application
6. Handling TPFAIL Data

Setting Up the Client Directory Structure

You must define the applications physical representation. Strict SCA client applications are SCA component types. Listing 2-1shows the directory structure used to place SCA components in an application.

Listing 2-1 SCA Component Directory Structure

```
myApplication/ (top-level directory, designated by the APPDIR environment
variable)
    root.composite (SCDL top-level composite, contains the list of
components in this application)
    myClient/ (directory containing actual client component described in
this section)
        myClient.composite (SCDL for the client component)
        myClient.cpp (client program source file)
        TuxService.h (interface of component called by client program)
```

Listing 2-2 shows an example of typical `root.composite` content.

Listing 2-2 root.composite Content

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
       name="simple.app">
       <component name="myClientComponent">
              <implementation.composite name="myClient"/>
       </component>
</composite>
```

The `implementation.composite@name` parameter references the directory that contains the component named '`myClientComponent`'. This value is required at runtime. For more information, see Running the Client Application.

Developing the Client Application

Client programs are required to implement a call to a single API. This following call is required in order to set up the SCA runtime:

```
...
    CompositeContext theContext = CompositeContext::getCurrent();
...
```

Actual calls are based on an interface. This interface is usually developed along with the component being called. In the case of existing Oracle Tuxedo ATMI services, this interface can be generated by accessing the Oracle Tuxedo METADATA repository.

In the case of calling external Web services, an interface matching the service WSDL must be provided. For more information, see SCA Data Type Mapping for the correspondence between WSDL types and C++ types.

Listing 2-3 shows an interface example.

Listing 2-3 Interface Example

```
#include <string>
/**
       *  Tuxedo service business interface
       */
       class TuxService
```

```
{
public:
virtual std::string TOUPPER(const std::string inputString) = 0;
};
```

In the interface shown in Listing 2-3, a single method `TOUPPER` is defined. It takes a single parameter of type `std::string`, and returns a value of type `std::string`. This interface needs to be located in its own `.h` file, and is referenced by the client program by including the `.h` file.

Listing 2-4 shows an example of a succession of calls required to perform an invocation.

Listing 2-4 Invocation Call Example

```
// SCA definitions
// These also include a Tuxedo-specific exception definition:
ATMIBindingException
#include "tuxsca.h"
// Include interface
#include "TuxService.h"
...
       // A client program uses the CompositeContext class
       CompositeContext theContext = CompositeContext::getCurrent();
...
       // Locate Service
       TuxService* toupperService =
              (TuxService *)theContext.locateService("TOUPPER");
...
       // Perform invocation
       const std::string result = toupperService->TOUPPER("somestring");
...
```

Notes: The invocation itself is equivalent to making a local call (as if the class were in another file linked in the program itself).

For detailed code examples, see the SCA samples located in following directories:

- UNIX samples: `$TUXDIR/samples/salt/sca`
- Windows samples: `%TUXDIR%\samples\salt\sca`

Composing the SCDL Descriptor

The link between the local call and the actual component is made by defining a binding in the SCDL side-file. For example, Listing 2-4 shows a call to an existing Oracle Tuxedo ATMI service, the SCDL descriptor shown in Listing 2-5 should be used. This SCDL is contained in a file called `<componentname>.composite`.

Listing 2-5 SCDL Descriptor

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
          name="simpapp.client">
     <reference name="TOUPPER">
        <interface.cpp header="TuxService.h"/>
        <binding.atmi requires="legacy">
           <inputBufferType target="TOUPPER">STRING</inputBufferType>
           <outputBufferType target="TOUPPER">STRING</outputBufferType>
        </binding.atmi>
    </reference>
</composite>
```

This composite file indicates that a client component may perform a call to the `TOUPPER` reference, and that this call is performed using the ATMI binding. In effect, this results in a `tpcall()` to the "`TOUPPER`" Oracle Tuxedo service. This Oracle Tuxedo service may be an actual existing Oracle Tuxedo ATMI service, or another SCA component exposed using the ATMI binding. For more information, see SCA Component Programming.

The `inputBufferType` and `outputBufferType` elements are used to determine the type of Oracle Tuxedo buffer used to exchange data. For more information, see SCA Data Type Mapping and the ATMI Binding Element Reference for a description of all possible values that can be used in the `binding.atmi` element.

Building the Client Application

Once all the elements are in place, the client program is built using the `buildscaclient` command. You must do the following steps:

1. Navigate to the directory containing the client source and SCDL composite files
2. Execute the following command:

   ```
   $ buildscaclient -c myClientComponent -s . -f myClient.cpp
   ```

   This command verifies the SCDL code, and builds the following required elements:

   - A shared library (or DLL on Windows) containing generated proxy code
   - The client program itself

If no syntax or compilation error is found, the client program is ready to use.

Running the Client Application

To execute the client program, the following environment variables are required:

- `APPDIR` - designates the application directory; in the case of SCA this typically contains the top-level SCDL composite.
- `SCA_COMPONENT` - the default SCA component (the value '`myClientComponent`' in the example shown in Listing 2-2). It tells the SCA runtime where to start when looking for services in the `locateService()` call.

Invoking Existing Oracle Tuxedo Services

Access to existing Oracle Tuxedo ATMI services from an SCA client program can be simplified using the examples shown in Listing 2-6, Listing 2-7, and Listing 2-8.

Note: These examples can also be used for server-side SCA components.

Starting from a Oracle Tuxedo `METADATA` repository entry as shown in Listing 2-6, the `tuxscagen` command can be used to generate interface and SCDL.

Listing 2-6 SCA Components Calling an Existing Oracle Tuxedo Service

```
service=TestString
tuxservice=ECHO
servicetype=service
```

```
inbuf=STRING
outbuf=STRING

service=TestCarray
tuxservice=ECHO
servicetype=service
inbuf=CARRAY
outbuf=CARRAY
```

Listing 2-7 Generated Header

```
#ifndef ECHO_h
#define ECHO_h
#include <string>
#include <tuxsca.h>
class ECHO
{
public:
    virtual std::string TestString(const std::string arg) = 0;
    virtual std::string TestCarray(const struct carray_t * arg) = 0; };
#endif /* ECHO_h */
```

Listing 2-8 Generated SCDL Reference

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
name="ECHO">
  <reference name="ECHO">
    <interface.cpp header="ECHO.h"/>
    <binding.atmi requires="legacy">
      <serviceType target="TestString">RequestResponse</serviceType>
      <inputBufferType target="TestString">STRING</inputBufferType>
      <outputBufferType target="TestString">STRING</outputBufferType>
      <serviceType target="TestCarray">RequestResponse</serviceType>
      <inputBufferType target="TestCarray">CARRAY</inputBufferType>
      <outputBufferType target="TestCarray">CARRAY</outputBufferType>
```

```
    </binding.atmi>
  </reference>
</composite>
```

The steps used to invoke these services are identical to the examples shown in Listing 2-6 through Listing 2-8.

Handling TPFAIL Data

Invoking a non-SCA Oracle Tuxedo ATMI service may return an error, but still send back data by using `tpreturn(TPFAIL, ...)`. When this happens, an SCA client or component is interrupted by the `ATMIBindingException` type.

The data returned by the service, if present, can be obtained by using the `ATMIBindingException.getData()`API. For more information see, TPFAIL Return Data.

The example in Listing 2-9 corresponds to a `binding.atmi` definition as shown in Listing 2-10.

Listing 2-9 Invocation Interruption Example

```
...
        try {
          const char* result = toupperService->charToup("someInput");
        } catch (tuscany::sca::atmi::ATMIBindingException& abe) {
          // Returns a pointer to data corresponding to
          // mapping defined in <errorBufferType> element
          // in SCDL
          const char* *result = (const char **)abe.getData();
          if (abe.getData() == NULL) {
              // No data was returned
          } else {
              // Process data returned
              ...
          }
        } catch (tuscany::sca::ServiceInvocationException& sie) {
          // Other type of exception is returned
```

```
        }
...
```

Listing 2-10 /binding.atmi Definition

```
...
        <binding.atmi requires="legacy">
              <inputBufferType target="charToup">STRING</inputBufferType>
              <outputBufferType
target="charToup">STRING</outputBufferType>
              <errorBufferType target="charToup">STRING</errorBufferType>
<binding.atmi/>
...
```

Other returned data types must be cast to the corresponding type. For example, an invocation returning a `commonj::sdo::DataObjectPtr` as shown in Listing 2-11.

Listing 2-11 SCDL Invocation Example

```
...
       <errorBufferType target="myMethod">FML32/myType</errorBufferType>
...
```

The `ATMIBindingException.getData()` result is shown in Listing 2-12.

Listing 2-12 ATMIBindingException.getData() Results

```
...
              catch (tuscany::sca::atmi::ATMIBindingException& abe) {
              const commonj::sdo::DataObjectPtr *result =
              (const commonj::sdo::DataObjectPtr *)abe.getData();
...
```

The rules for returning `TPFAIL` data to the calling application are as follows:

- For each `<errorBufferType>`, a canonical type is defined, where `<errorBufferType>` is converted. When the `<errorBufferType>` is equal to the `<outputBufferType>`, the canonical type is the same C++ type that is returned in a successful service implementation.
- When the `<errorBufferType>` is different from the `<outputBufferType>`, the canonical type is as follows:
  - For `STRING` buffers, a C++ `char*` or `char[ ]`data type.
  - For `MBSTRING` buffers, a C++ `wchar_t*` or `wchar_t[ ]`.
  - For `CARRAY` buffers, a C++ CARRAY_PTR.
  - For `X_OCTET` buffers, a C++ X_OCTET_PTR.
  - For XML buffers, a C++ XML_PTR.
  - For FML, FML32, VIEW, VIEW32, X_COMMON, and X_C_TYPE buffers, a C++ `commonj::sdo::DataObjectPtr`.
- In each case, the value returned by `getData()` is a pointer to one of the types listed above.

For more conversion rules between Oracle Tuxedo buffer types and C++ data information, see SCA Data Type Mapping.

SCA Component Programming

The SCA Component terminology designates SCA runtime artifacts that can be invoked by other SCA or non-SCA runtime components. In turn, these SCA Components can perform calls to other SCA or non-SCA components. This is different from strict SCA clients which can only make calls to other SCA or non-SCA components, but cannot be invoked.

The Oracle Tuxedo SCA container provides the capability of hosting SCA components in an Oracle Tuxedo server environment. This allows you to take full advantage of proven Oracle Tuxedo qualities: *reliability*, *scalability* and *performance*.

Figure 2-1 summarizes SCA components and Oracle Tuxedo server mapping rules.

Figure 2-1 SCA Component and Oracle Tuxedo Server Mapping Rules

While SCA components using Oracle Tuxedo references do not require special processing, SCA components offering services must still be handled in an Oracle Tuxedo environment.

The mapping is as follows:

- An SCA composite declaring one or more services with a `<binding.atmi>` definition maps to a single Oracle Tuxedo server advertising the same number of services as the SCA composite.
- There can be more than one composite.
- Composites can be nested.
- Promotion handling:
  - A composite promoting a service contained in a nested component results in the promoted service being advertised as an Oracle Tuxedo service.
  - A service declared in a component, but not promoted, is not advertised.
- The resulting Oracle Tuxedo server advertises as many services as there are `binding.atmi` sections in the SCDL definition

- Interfaces may declare multiple methods. Each method is linked to an Oracle Tuxedo native service using the `/binding.atmi/@map` attribute. A method not declared via the `/binding.atmi/@map` attribute is not accessible through Oracle Tuxedo. The use of duplicate service names are detected at server generation time, so that Oracle Tuxedo service names-to-interface method mapping in a single Oracle Tuxedo server instance is 1:1.
- A generated Oracle Tuxedo server acts as a proxy for SCA components. An instance of this generated server corresponds to an SCA composite as defined in the SCDL configuration. Such servers are deployed as necessary by the Oracle Tuxedo administrator.

SCA composites are deployed in an Oracle Tuxedo application by configuring instances of generated SCA servers in the `UBBCONFIG` file. Multiple instances are allowed. Multi-threading capabilities are also allowed and controllable using already-existing Oracle Tuxedo features.

SCA Component Programming Steps

The steps required for developing SCA component programs are as follows:

1. Setting Up the Component Directory
2. Developing the Component Implementation
3. Composing the SCDL Descriptor
4. Compiling and Linking the Components
5. Building the Oracle Tuxedo Server Host

Setting Up the Component Directory

You must first define the applications physical representation. Listing 2-13 shows the directory structure used to place SCA components in an application:

Listing 2-13 SCA Component Directory Structure

```
myApplication/ (top-level directory, designated by the APPDIR environment
variable)
    root.composite (SCDL top-level composite, contains the list of
components in this application)
    myComponent/ (directory containing actual component described in this
section)
```

```
        myComponent.composite (SCDL for the component)
        myComponent.componentType
        myComponentImpl.cpp (component implementation source file)
        TuxService.h (interface of component being exposed)
        TuxServiceImpl.h (component implementation definitions)
```

Listing 2-14 shows typical `root.composite` content.

Listing 2-14 root.composite Content

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
       name="simple.app">
       <component name="myComponent">
              <implementation.composite name="myComponent"/>
       </component>
</composite>
```

The `implementation.composite@name` parameter references the directory that contains the `'myComponent'` component.

Developing the Component Implementation

Components designed to be called by other components do not need to be aware of the SCA runtime. There are, however, limitations in terms of interface capabilities, such as:

- C++ classes (other than `std::string` and `commonj::sdo::DataObjectPtr`) cannot be used as parameters or return values
- Parameter arrays are not supported

  For more information, see SCA Data Type Mapping.

Listing 2-15 shows an example of an interface implemented for a client program.

Listing 2-15 Component Implementation Interface

```
#include <string>
/**
  *  Tuxedo service business interface
  */
  class TuxService
  {
  public:
       virtual std::string TOUPPER(const std::string inputString) = 0;
  };
```

The component implementation then generally consists of two source files (as shown Listing 2-16 and Listing 2-17 respectively):

- component implementation definitions, contained in a `<servicename>Impl.h file`, and
- component implementation, contained in a `<servicename>Impl.cpp file`

Listing 2-16 Example (TuxServiceImpl.h):

```
#include "TuxService.h"

      /**
       * TuxServiceImpl component implementation class
       */
      class TuxServiceImpl:  public TuxService
      {
      public:
       virtual std::string toupper(const std::string inputString);
      };
```

Listing 2-17 Example (TuxServiceImpl.cpp):

```
#include "TuxServiceImpl.h"
 #include "tuxsca.h"

using namespace std;
using namespace osoa::sca;

/**
* TuxServiceImpl component implementation
*/
std::string TuxServiceImpl::toupper(const string inputString)
{
    string result = inputString;

    int len = inputString.size();

    for (int i = 0; i < len; i++) {
       result[i] = std::toupper(inputString[i]);
    }

    return result;
}
```

Additionally, a side-file (`componentType`), is required. It contains the necessary information for the SCA wrapper generation and possibly proxy code (if this component calls another component).

This `componentType` file (`<componentname>Impl.componentType`)is an SCDL file type. Listing 2-18 shows an example of a `componentType` file (`TuxServiceImpl.componentType`).

Listing 2-18 componentType File Example

```
<?xml version="1.0" encoding="UTF-8"?>
 <componentType xmlns="http://www.osoa.org/xmlns/sca/1.0" >
       <service name="TuxService">
```

```
            <interface.cpp header="TuxService.h"/>
      </service>
</componentType>
```

Composing the SCDL Descriptor

The link between the local implementation and the actual component is made by defining a binding in the SCDL side-file. For example, for the file type in Listing 2-18 to be exposed as an Oracle Tuxedo ATMI service, the SCDL in Listing 2-19 should be used. This SCDL is contained in a file called `<componentname>.composite` (for example, `myComponent.composite`).

Listing 2-19 Example SCDL Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="myComponent">
  <service name="TuxService">
   <interface.cpp header="TuxService.h"/>
        <binding.atmi requires="legacy"/>
   <map target="toupper">TUXSVC</map>
   <inputBufferType target="toupper">STRING</inputBufferType>
   <outputBufferType target="toupper">STRING</outputBufferType>
          <reference>MYComponent</reference>
    </service>
    <component name="MYComponent">
      <implementation.cpp library="TuxService" header="TuxServiceImpl.h"/>
    </component>
</composite>
```

This composite file indicates that the service, `mySVC`, can be invoked via the Oracle Tuxedo infrastructure. It further indicates that the `toupper()` method is advertised as the `TUXSVC` service in the Oracle Tuxedo system. Once initialized, another SCA component may now call this service, as well as a non-SCA Oracle Tuxedo ATMI client.

The `inputBufferType` and `outputBufferType` elements are used to determine the type of Oracle Tuxedo buffer used to exchange data. For more information, see SCA Data Type Mapping and the ATMI Binding Element Reference for a description of all possible values that can be used in the `binding.atmi` element.

Note: The `mycomponent.componentType` service name should be same as the composite file, otherwise an exception is thrown.

Compiling and Linking the Components

Once all the elements are in place, the component is built using the `buildscacomponent` command. The steps are as follows:

1. Navigate to the `APPDIR` directory. The component and side files should be in its own directory one level down
2. Execute the following command:

   ```
   $ buildscacomponent -c myComponent -s . -f TuxServiceImpl.cpp
   ```

   This command verifies the SCDL code, and builds the following required elements:

   - A shared library (or DLL on Windows) containing generated proxy code

Building the Oracle Tuxedo Server Host

In order for components to be supported in an Oracle Tuxedo environment, a host Oracle Tuxedo server must be built. This is achieved using the `buildscaserver` command.

For example: `$ buildscaserver -c myComponent -s . -o mySCAServer`

When the command is executed, `mySCAServer` is ready to be used. It automatically locates the component(s) to be deployed according to the SCDL, and performs the appropriate Oracle Tuxedo/SCA associations.

SCA Python, Ruby, and PHP Programming

This section contains the following topics:

- SCA Python, Ruby, and PHP Programming Overview

- Python, Ruby, and PHP Client Programming
- Python, Ruby, and PHP Component Programming
- Python, Ruby, and PHP Data Type Mapping
- Python, Ruby, and PHP Binding

SCA Python, Ruby, and PHP Programming Overview

Integration of Python, Ruby or PHP scripts in an environment such as Oracle Tuxedo via SALT, is intended for providing additional flexibility in terms of program development.

Python, Ruby, and PHP are comparable object-oriented scripting languages that offer many advantages over C/C++:

- No compilation
- Dynamic data typing
- Garbage collection
- Existing libraries of utility functions and objects

SALT SCA Python, Ruby, and PHP support provides a set of APIs to perform SCA calls from Python, Ruby or PHP client programs, and language extensions to call Python, Ruby or PHP components. For more information, see Python, Ruby, and PHP Client Programming and Python, Ruby, and PHP Component Programming.

The `buildscaclient`, `buildscaserver` and `buildscacomponent` commands do not need adapting for use with Python, Ruby or PHP programs, as they are not be required to produce executables or component libraries.

Note: A system server, `SCAHOST`, is provided to correctly marshal requests and responses to and from Python, Ruby or PHP scripts. It contains Python, Ruby, and PHP scripts exposed as SCA services (via the Oracle Tuxedo Metadata Repository). The definitions describe the parameters and return types of the corresponding exposed Python, Ruby or PHP functions.

For more information, see Python, Ruby, and PHP Data Type Mappingfor Service Metadata Repository entry examples.

Available bindings are used from Python, Ruby or PHP programs, or are used to invoke Python, Ruby or PHP components. Like C++, the Python, Ruby, and PHP language extension is binding-independent.

Figure 2-2 provides an overview of the SALT SCA environment Python, Ruby, and PHP support architecture.

Figure 2-2 SALT SCA Python, Ruby, and PHP Programming Support Architecture

Python, Ruby, and PHP Client Programming

SCDL Clients

From a client component perspective, the SCDL code only has to mention the referenced component and possibly the binding used (that is, no interface element is required).

For example, the following snippet allows a Python, Ruby or PHP client to make an invocation to an SCA component via ATMI binding, and using the default buffer type (STRING input, STRING output):

```
<reference name="CalculatorComponent">
```

```
    <binding.atmi/>
</reference>
```

Python Clients

To invoke an SCA component from a Python program, you must do the following:

1. Import the SCA library using the following command:

   ```
   import sca
   ```

2. Use the following API to locate the service:

   ```
   calc = sca.locateservice("CalculatorComponent")
   ```

   The `calc` object is used to invoke the “add” operation (for example, `result = calc.add(val1, val2)`).

Ruby Clients

To invoke an SCA component from a Ruby program, you must do the following:

1. Load the Ruby proxy extension:

   ```
   require("sca_ruby")
   ```

2. Use the following API to locate the service:

   ```
   calculator = SCA::locateService("CalculatorComponent")
   ```

   The calculator object is used to invoke the “add” operation (for example, `x = calculator.add(3, 2)`).

PHP Clients

To invoke an SCA component from a PHP program, you must do the following:

1. users will have to first load the SCA library as follows:

   ```
   <?php
   dl('sca.so');
   ```

2. Use the following API to locate the service:

   ```
   $svc = Sca::locateService("uBikeService");
   ```

   At this point the svc object can be used to invoke the searchBike operation, for instance:

```
$ret = $svc->searchBike('YELLOW');
```

Python, Ruby, and PHP Component Programming

SCDL Components

In order to use Python, Ruby or PHP scripts in SCA as components, you must use the `implementation.python`, `implementation.ruby` and `implementation.php` parameters.

Note: `implementation.python` `implementation.ruby` and `implementation.php` usage is similar to the `implementation.cpp` element (see Listing 2-19 and Listing 2-31); the difference is that the `interface.python` and `interface.ruby` elements, or `.componentType` are not required.

Their syntax and attributes are as follows:

- `implementation.python`

  ```
  <implementation.python
       module="string"
       scope="scope"? >
  <implementation.python/>
  ```

  The `implementation.python` element has the following attributes:

  – module: string (1..1)

    Name of the Python module (`.py` file) containing the operation(s) that this component offers in the form of module-level function(s).

  – scope: PythonImplementationScope(0..1)

    Identifies the scope of the component implementation. The default is stateless, indicating that there is no correlation between implementation instances used to dispatch service requests. A composite value indicates that all service requests are dispatched to the same implementation instance for the lifetime of the containing composite.

- `implementation.ruby`

```
<implementation.ruby
     script="string"
     class="string"
     scope="scope"? >
<implementation.ruby/>
```

The implementation.ruby element has the following attributes:

– script: string(1..1)

  Name of the Ruby script (`.rb` file) containing the operation(s) that the component offers in the form of methods of a class contained in the script file. The name of the script is its full name (that is, it also includes the `.rb` extension).

– class: string(1..1)

  Name of the Ruby class (`.rb` file) containing the operation(s) that the component offers.

– scope: RubyImplementationScope(0..1)

  Identifies the scope of the component implementation. The default is stateless, indicating that there is no correlation between implementation instances used to dispatch service requests. A composite value indicates that all service requests are dispatched to the same implementation instance for the lifetime of the containing composite.

- `implementation.php`

```
<implementation.php
     script="string"
     class="string"
     scope="scope"? >
<implementation.php/>
```

The implementation.php element has the following attributes:

– script: string(1..1)

  Name of the PHP script (`.php` file) containing the operation(s) that this component will offer, in the form of methods of a class contained in the script file. The name of the script is its full name, i.e. it also includes the `.php` extension.

– class: string(1..1)

  Name of the PHP class (`.php` file) containing the operation(s) that this component will offer.

– scope: PHPImplementationScope(0..1)

Identifies the scope of the component implementation. The default is stateless, indicating that there is no correlation between implementation instances used to dispatch service requests. A value of composite indicates that all service requests are dispatched to the same implementation instance for the lifetime of the containing composite.

Listing 2-20 shows an example of a Python component in an SCA composite accessible using the ATMI binding. In this example, runtime looks for a Python component located in a file named `ToupperService.py` in the same location as the composite file.

Similarly, a Ruby component is required in a file named `ToupperService.rb`, in the same location as the composite file.

Listing 2-20 Python Component in an SCA Composite

```
<?xml version="1.0" encoding="UTF-8"?>

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
              name="simpapp.server">

       <service name="SCASVC">
              <binding.atmi/>
              <reference>ToupperServiceComponent</reference>
</service>

<component name="ToupperServiceComponent">
       <implementation.python module="ToupperService"
                                    scope="composite"/>
</component>

</composite>
```

Listing 2-21 shows an example of a PHPcomponent in an SCA composite accessible using the ATMI binding

Listing 2-21 PHP Component in an SCA Composite

```
<?xml version="1.0" encoding="UTF-8"?>

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
              name="simpapp.PHP">

       <service name="TESTPHP">
         <!-- No interface, it is contained in TMMETADATA -->
         <binding.atmi>
           <map target="charToup">TOUPPHP</map>
           <inputBufferType target="charToup">STRING</inputBufferType>
           <outputBufferType target="charToup">STRING</outputBufferType>
         </binding.atmi>
         <reference>ToupperServiceComponent</reference>
</service>

<component name="ToupperServiceComponent">
       <implementation.php script="toupper.php"
                                  class="Toupper"/>
                                  scope="composite"/>
</component>

</composite>
```

Python Components

Python operations are exposed as module-level functions contained in a Python module file. For example, a `ToupperService.py` file would contain the code shown in Listing 2-22.

Listing 2-22 Python Module File

```
def charToup(val1):
    print "input: " + val1
    result = "result"
    print "Python - toupper"
    return result
```

Parameter and return values types are dynamically determined at runtime. Application exceptions are caught by the extension runtime and re-thrown as `tuscany::sca::ServiceInvocationException`.

During input, unsupported types or an error processing an input DataObject results in the following exception:
a `tuscany::sca::ServiceDataException`.

During output, simple return types are always processed. An error generating a DataObject (from XML data) results in the following exception:
`tuscany::sca::ServiceDataException`.

For more information, see Python, Ruby, and PHP Data Type Mapping.

Ruby Components

Ruby operations are exposed as methods of an implementation class contained in a Ruby script file (`.rb` extension). For example, a `ToupperService.rb` file would contain the code shown in Listing 2-23.

Listing 2-23 Ruby Script File

```
class ToupperService

  def initialize()
```

```
    print "Ruby - ToupperService.initialize\n"
  end

  def charToup(arg1)
    print "Ruby - ToupperService.div\n"
    arg1.ToUpper()
  end

end
```

Parameter and return values types are dynamically determined at runtime. Application exceptions are caught by the extension runtime and re-thrown as `tuscany::sca::ServiceInvocationException`.

During input, unsupported types or an error processing an input DataObject results in the following exception:
a `tuscany::sca::ServiceDataException`.

During output, simple return types are always processed. An error generating a DataObject (from XML data) results in the following exception: `tuscany::sca::ServiceDataException`.

For more information, see Python, Ruby, and PHP Data Type Mapping.

PHP Components

PHP operations are exposed as functions contained in a PHP class. For example, a `toupper.php` file would contain the code shown in Listing 2-24

Listing 2-24 PHP Class

```
<?php
class MyClass {
    public static function toupper(val) {
    print "PHP - toupper";
    return val.toupper();
  }
```

```
}
?>
```

Parameter and return values types are dynamically determined at runtime. Application exceptions are caught by the extension runtime and re-thrown as `tuscany::sca::ServiceInvocationException`.

During input, unsupported types or an error processing an input DataObject results in the following exception:
a `tuscany::sca::ServiceDataException`.

During output, simple return types are always processed. An error generating a DataObject (from XML data) results in the following exception: `tuscany::sca::ServiceDataException`.

For more information, see Python, Ruby, and PHP Data Type Mapping.

SCA Structure Support

This section contains the following topics:

- SCA Structure Support Overview
- Using SCA Structure Description Files
- Using tuxscagen to Generate Structures

Note: This section applies to application defined structures only. For information on Oracle Tuxedo SCA defined structures, see SCA Data Type Mapping.

SCA Structure Support Overview

SCA Structure support provides:

- Additional C++ structure functionality
- Improved performance for applications processing data that can be placed in a structure without significant wasted space

You must use the `struct` data type specified in the SCA method parameter definition or in the definition of a return value from an SCA method as follows:

- `struct` *`structurename`* `*`

- `struct` *`structurename &`*

Elements within the structure can be any of the following simple data types/arrays that are supported as an SCA parameter:

- `bool`
- `char, unsigned char, signed char`
- `wchar_t`
- `short, unsigned short`
- `int, unsigned int`
- `long, unsigned long`
- `long long, unsigned long long`
- `float`
- `double`
- `long double`
- `struct` *`nestedstructurename`*
- `typedef`

  Note: The `scagen` utility parses `typedef` and `struct` keywords. For more information, see the SCA Command Reference.

SCA Structure Limitations

- The following cannot be specified as part of a structure”
  - `DataObjectPtr`
  - Point data types
  - `std::string` or a `std::wstring`
  - A union
  - `struct carray_t`, `struct_x_octet_t`, or `struct xml_t`
- CARRAY data is supported in the same way that it is supported for views
- `.h` and `.cpp` files referencing the use of structures are required to include a definition for the structure being used and for any nested structures contained within that structure.

Using SCA Structure Description Files

A structure description file may be used to describe the format of an SCA structure parameter. Structure description files are very similar to Oracle Tuxedo viewfiles, with additional capabilities added for SCA.

Note: The use of structure description files is optional, and is needed only when FML field names corresponding to structure elements are different from the names of the structure elements, or when some other non-default structure related feature is required. If an application wants to make use of an Associated Length Member, an Associated Count Member, or an application-specified default value for a structure element, it may choose to make use of a structure description file.

If no structure description file is provided for a particular structure, then the structure definition used in application code is used, and FML field names in SCA-ATMI mode are the same as structure element names. Since field numbers are generated automatically for SCA-SCA applications, these applications do not need to specify a structure description file.

The structure description file format is identical to the Oracle Tuxedo viewfile format, with the following exceptions:

- The type parameter in column 1 allows the additional values `bool`, `unsignedchar`, `signedchar`, `wchar_t`, `unsignedint`, `unsignedlong`, `longlong`, `unsignedlonglong`, `longdouble`, and `struct`.
- If the value in column1 is `struct`, then the `cname` value in column 2 is the name of a previously defined VIEW that describes a nested structure. In this case, the count value in column 4 may optionally be specified to specify the number of occurrences of the nested structure.

If a structure described in a structure description file is converted to (or from) an FML32 or FML buffer at runtime in an SCA-ATMI application, then the name of the corresponding FML field is the `fbname` value specified in column 3, if any, and is the `cname` value specified in column 2 (if no value is specified in column 3). When compiled, the structure description file produces a binary structure description file as shown in Listing 2-25. The binary structure header file is shown in shown in Listing 2-26.

Note: In an SCA-SCA application, FML32 field numbers are generated automatically.

Listing 2-25 SCA Structure Description File

```
VIEW empname
#TYPE       CNAME     FBNAME         COUNT      FLAG      SIZE      NULL
string      fname     EMP_FNAME        1          -         25        -
char        minit     EMP_MINI         1          -          -        -
string      lname     EMP LNAME        1          -         25        -
END

VIEW emp
struct         empname      ename     1          -          -        -
unsignedlong   id           EMP_ID    1          -          -        -
long           ssn          EMP_SSN   1          -          -        -
double         salaryhist   EMP_SAL   10         -          -        -
END
```

Listing 2-26 Binary Structure Header File

```
struct empname {
  char fname[25];
  char minit;
  char lname[25];
};

struct emp {
  struct empname ename;
  unsigned long id;
  long ssn;
  double salaryhist[10];
}
```

The `scastructc32` and `scastructc` commands are used to convert a source structure description file into a binary structure description file and to generate a header file describing the structure(s) in the structure description file. The `scastructdis32` and `scastructdis`

commands accept the same arguments as `viewdis32` and `viewdis`. For more information, see the SCA Command Reference.

Notes: `scastructc32` and `scastructc` generate a binary file with suffix .V on Unix and suffix .VV on Windows.

If the structure description file contains no SCA extensions that are not available in Oracle Tuxedo views, then the magic value for the binary structure description file shall be the same as the magic value used by viewc32. If any SCA specific extensions are used, then a different magic value shall be used for the binary structure description file.

Using tuxscagen to Generate Structures

When invoked with the option `-S`, `tuxscagen` generates a structure for any function parameter or return value that would otherwise have been passed using `DataObjectPtr`.

Note: If `tuxscagen -S` is run, then simple data types are generated just as they would have been if `tuxscagen` were run without the `-S` option. It is possible to mix simple data types, structures, and other complex data types within a single metadata repository. In order to use simple data types in an application that also uses structures, it is not necessary to run tuxscagen without `-S`.

SCA Remote Protocol Support

SCA Oracle Tuxedo invocation supports the following remote protocols:

- /WS
- /Domains

/WS

SCA invocations made using the SCA container have the capability of being performed using the Oracle Tuxedo WorkStation protocol (/WS). This is accomplished by specifying the value `WorkStation` (not abbreviated so as not to confuse it with `WebServices`) in the `<remoteAccess>` element of the `<binding.atmi>` element.

Only reference-type invocations are be available in this mode. Service-type invocations may be performed using the /WS transparently (there is no difference in behavior or configuration, and setting the `<remoteAccess>` element to `WorkStation` for an SCA service has no effect).

Since native and `WorkStation` libraries cannot be mixed within the same process, client processes must be built differently depending on the type of remote access chosen.

Note: When using the value `propagatesTransaction` in `/binding.atmi/@requires`, the behavior of the ATMI binding does not actually perform any transaction propagation. It actually starts a transaction, since the use of this protocol is reserved for client-side access to Oracle Tuxedo (SCA or non-SCA) applications only. For more information, see ATMI Binding.

/Domains

SCA invocations made using the SCA container have the capability of being performed using the Oracle Tuxedo /Domains protocol. No additional configurations are necessary on `<binding.atmi>` declarations in SCDL files.

Note: /Domains interoperability configuration is controlled by the Oracle Tuxedo administrator.

The SCA service name configured for Oracle Tuxedo /Domains is as follows:

- SCA -> SCA mode - `/binding.atmi/service/@name` attribute followed by a '`/`' and method name
- Legacy mode (SCA -> Tux interop mode) - `/binding.atmi/service/@name` attribute.

For more information, see SCA and Oracle Tuxedo Interoperability.

SCA Binding

Oracle Tuxedo supports

- ATMI Binding
- Java ATMI (JATMI) Binding
- Python, Ruby, and PHP Binding
- Web Services Binding

ATMI Binding

Oracle Tuxedo communications are configured in SCDL using a `<binding.atmi>` element. This allows you to specify configuration elements specific to the ATMI transport, such as the location of the TUXCONFIG file, the native Oracle Tuxedo buffer types used, Oracle Tuxedo-specific authentication or /WS (WorkStation) configuration elements, etc.

Listing 2-27 shows a summary of the <binding.atmi> element.

Note: ? refers to a parameter that can be specified 0 or 1 times.

* refers to a parameter that can be specified 0 or more times.

For more information, see Oracle Tuxedo SCA ATMI Binding Reference.

Listing 2-27 ATMI Binding Pseudoschema

```
<binding.atmi requires="transactionalintent legacyintent"?>
       <tuxconfig>...</tuxconfig>?
       <map target="name">...</map>*
       <serviceType target="name">...</serviceType>*
       <inputBufferType target="name">...</inputBufferType>*
       <outputBufferType target="name">...</outputBufferType>*
       <errorBufferType target="name">...</errorBufferType>*
       <workStationParameters>?
              <networkAddress>...</networkAddress>?
              <secPrincipalName>...</secPrincipalName>?
              <secPrincipalLocation>...</secPrincipalLocation>?
              <secPrincipalPassId>...</secPrincipalPassId>?
              <encryptBits>...</encryptBits>?
       </workStationParameters>
       <authentication>?
              <userName>...</userName>?
              <clientName>...</clientName>?
              <groupName>...</groupName>?
              <passwordIdentifier>...</passwordIdentifier>?
              <userPasswordIdentifier>...
                                          </userPasswordIdentifier>?
       </authentication>
       <fieldTablesLocation>...</fieldTablesLocation>?
       <fieldTables>...</fieldTables>?
       <fieldTablesLocation32>...</fieldTablesLocation32>?
       <fieldTables32>...</fieldTables32>?
       <viewFilesLocation>...</viewFilesLocation>?
       <viewFiles>...</viewFiles>?
       <viewFilesLocation32>...</viewFilesLocation32>?
```

```
        <viewFiles32>...</viewFiles32>?
        <remoteAccess>...</remoteAccess>?
        <transaction timeout="xsd:long"/>?
</binding.atmi>
```

Java ATMI (JATMI) Binding

Java ATMI (JATMI) binding allows SCA clients written in Java to call Oracle Tuxedo services or SCA components. It provides one-way invocation of Oracle Tuxedo services based on the Oracle Tuxedo WorkStation protocol (/WS). The invocation is for outbound communication only from a Java environment to Oracle Tuxedo application acting as a server. Apart from a composite file for SCDL binding declarations, no external configuration is necessary. The service name, workstation address and authentication data are provided in the binding declaration.

Note: SSL is supported through the Oracle 11gR1 JCA Adapter. LLE is not currently supported.

Most of the Oracle Tuxedo CPP ATMI binding elements support JATMI binding and have the same usage. However, due to different underlying technology and running environment differences, some elements are not supported and some that are supported but have different element names.

The following Oracle Tuxedo CPP ATMI binding elements are not supported:

- `binding.atmi/tuxconfig`
- `binding.atmi/fieldTablesLocation`
- `binding.atmi/fieldTablesLocation32`
- `binding.atmi/viewFilesLocation`
- `binding.atmi/viewFilesLocation32`
- `binding.atmi/transaction`

The following Oracle Tuxedo CPP ATMI binding `workStationParameters` elements are not supported:

- `binding.atmi/workStationParameters/secPrincipalName`
- `binding.atmi/workStationParameters/secPrincipalLocation`
- `binding.atmi/workStationParameters/secPrincipalPassId`
- `binding.atmi/workStationParameters/encryptBits`

The following Oracle Tuxedo CPP ATMI binding element is supported in a limited fashion.

- `binding.atmi/remoteAccess`

  Note: Only the value "`WorkStation`" is allowed. If not specified, "`WorkStation`" is assumed.

All the classes in the elements mentioned below must be specified in Java CLASSPATH:

- `binding.atmi/fieldTables` - Specifies a comma-separated list of Java classes that are extended from the `weblogic.wtc.jatmi.TypedFML` base class.
- `binding.atmi/fieldTables32` - Specifies a comma-separated list of Java classes that are extended from the `weblogic.wtc.jatmi.TypedFML32` base class.
- `binding.atmi/viewFiles` - Specifies a comma-separated list of Java classes that are extended from the `weblogic.wtc.jatmi.TypedView` base class. These derived classes usually are generated from an Oracle Tuxedo VIEW file using the `weblogic.wtc.jatmi.viewj` compiler. These also includes derived from `weblogic.wtc.jatmi.TypedXCType` and `weblogic.wtc.jatmi.TypedXCommon`.

  For more information, see How to Use the viewj Compiler in the Oracle Tuxedo WebLogic Tuxedo Connector Programmer's Guide.

- `binding.atmi/viewFiles32` - Specifies a comma-separated list of Java classes that are extended from the `webogic.wtc.jatmi.TypedView32` base class. These derived classes usually are aslo generated from an Oracle Tuxedo VIEW file using the `weblogic.wtc.jatmi.viewj32` compiler.

Listing 2-28 shows an example of composite file for binding declaration of an Oracle Tuxedo service named "`ECHO`".

Listing 2-28 ECHO Composite File

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
xmlns:f="binding-atmi.xsd"
name="ECHO">
    <reference name="ECHO" promote="EchoComponent/ECHO">
       <interface.java interface="com.abc.sca.jclient.Echo" />
       <f:binding.atmi requires="legacy">
       <f:inputBufferType target="echoStr">STRING</f:inputBufferType>
         <f:outputBufferType target="echoStr">STRING</f:outputBufferType>
```

```
          <f:errorBufferType target="echoStr">STRING</f:errorBufferType>
           <f:workStationParameters>

<f:networkAddress>//STRIATUM:9999,//STRIATUM:1881</f:networkAddr
ess>
           </f:workStationParameters>
           <f:remoteAccess>WorkStation</f:remoteAccess>
        </f:binding.atmi>
    </reference>
    <component name="EchoComponent">
      <implementation.java class="com.abc.sca.jclient.EchoComponentImpl"
/>
    </component>
</component>
```

Listing 2-29 shows the interface for the example mentioned in Listing 2-28.

Listing 2-29 ECHO Interface

```
package com.abc.sca.jclient;

import com.oracle.jatmi.AtmiBindingException;

public interface Echo {
        String echoStr(String requestString) throws AtmiBindingException;
}
```

Listing 2-30 shows an example of an SCA client implementation.

Listing 2-30 SCA Client Implementation

```
package com.abc.sca.jclient;
```

```
import org.osoa.sca.annotations.Constructor;
import org.osoa.sca.annotations.Reference;
import com.oracle.jatmi.AtmiBindingException;

/**
* A simple client component that uses a reference with a JATMI binding.
*/
public class EchoComponentImpl implements Echo {

       private Echo echoReference;

        @Constructor
       public EchoComponentImpl(@Reference(name = "ECHO", required = true)
Echo
 echoReference) {
                this.echoReference = echoReference;
       }

        public String echoStr(String requestString) throws
AtmiBindingException {
                return echoReference.echoStr(requestString);
        }
}
```

Python, Ruby, and PHP Binding

The Python, Ruby, and PHP language extensions are binding-independent, meaning that binding extensions are not aware of the language of clients or components. Language extensions are not aware of the binding used.

Binding extensions are not modified to comply with Python, Ruby, and PHP program support. Note the following:

- If the data types defined in Python, Ruby or PHP binding do not match the Python, Ruby or PHP source files, then Oracle Tuxedo will throw an exception.
- If a binding is configured with a data type that does not match what the Python, Ruby or PHP component is designed to handle, an exception is thrown by the Python, Ruby or PHP

runtime (for example, `binding.atmi` configured with STRING Oracle Tuxedo buffers and a Python function handling numerical data).

- For a Python, Ruby or PHP client code mismatch with what binding is configured with, an exception occurs originating from the binding code.
- Since Python, Ruby, and PHP code is not compiled, any configuration mismatch between binding and component/client can only be detected at runtime.
- Python, Ruby or PHP programs with a *composite* scope require an Oracle Tuxedo server reload when the script is modified. A *stateless* scope allows dynamic reloading of modified scripts.
- In order to expose Python, Ruby or PHP scripts as Web services, the `SCAHOST` command must use the `-w` option in order to load the correct service binding during initialization.

  Note: `SCAHOST` does not allow mixing both ATMI and Web services binding types in one `SCAHOST` instance.

  For more information, see the SCA Command Reference.
- `TMMETADATA` server is required in order to expose Python, Ruby, and PHP components.

Python, Ruby, and PHP Binding Limitations

Using Python, Ruby, and PHP bindings have the following limitations:

- When using the ATMI binding for interoperability calls (that is, when `requires="legacy"` is set), mixing named and non-named parameters is not allowed (for example, Python: `def func(a, *b, **c)`, Ruby: `def func(a, *b, hash)`), since there is no mechanism to restore the parameter names.

  The names of the parameters must be configured in FML32 tables (ATMI binding), or by way of WSDL (Web services binding). It is not possible to interoperate with lists of non-named parameters because such calls cannot be accurately mapped to C++ or WSDL interfaces due to the lack of guaranteed ordering of `FML/FML32` Oracle Tuxedo buffers.

  The supported modes are:

  – Multiple parameters: `def func(a, b, c)` (same syntax for Python, Ruby, and PHP)

  – Multiple parameters and list of parameters: `def func(a, *b)` (same syntax for Python and Ruby)

  – Named parameters: PHP - `$svc->searchBike(array('COLOR' => 'RED', 'TYPE' => 'MTB'))`. For more information, see PHP Data Type Mapping.

  - Dictionary or hash: Python: `def func(**a)`, Ruby: `def func(hash)`

    Note: Python parameters defined with `**` are considered named parameters. Ruby parameters defined with `hash` are considered named parameters. For more information, see Python Parameters and Ruby Parameters.

- In SCA to SCA mode, the above limitation still concerns named parameters since the order of elements in a Python dictionary or Ruby hash is not guaranteed. To transmit a Python dictionary or Ruby hash, you must work in "legacy" mode.
- In SCA to SCA mode, using lists of parameters (excluding dictionaries or hashes) are supported since Oracle Tuxedo Service Metadata interfaces describe service-side lists of parameters/types (on the reference side parameters/types are self-described at runtime).
- Unicode strings are not supported; accordingly `MBSTRING` buffers or `FLD_MBSTRING` fields are not supported.

Web Services Binding

The Web services binding (`binding.ws`) leverages previously existing Oracle Tuxedo capabilities by funneling Web service traffic through the GWWS gateway. SCA components are hosted in Oracle Tuxedo servers, and communications to and from those servers are performed using the GWWS gateway.

SCA clients using a Web services binding remain unchanged whether the server is running in an Oracle Tuxedo environment or a native Tuscany environment (for example, exposing the component using the Axis2 Web services binding).

Note: HTTPS is not currently supported.

When SCA components are exposed using the Web services binding (`binding.ws`), tooling performs the generation of WSDF information, metadata entries and FML32 field definitions.

When SCDL code of SCA components to be hosted in an Oracle Tuxedo domain (for example, service elements) contains <`binding.ws`> elements, the `buildscaserver` command generates an WSDF entry in a file named `service.wsdf` where '`service`' is the name of the service exposed. An accompanying `service.mif` and service.`fml32` field table files are also generated, based on the contents of the WSDL interface associated with the Web service. You must compose a WSDL interface. If no WSDL interface is found, an error message is generated.

Web services accessed from an Oracle Tuxedo domain using a Web services binding (for example, reference elements found in SCDL) require the following manual configuration steps:

1. Convert the WSDL file into a WSDF entry by using the `wsdlcvt` tool. Simultaneously, a Service Metadata Entry file (`.mif`), and fml32 mapping file are generated.
2. Make sure that the UBB source has the `TMMETADATA` and GWWS servers configured
3. Import the WSDF file into the `SALTDEPLOY` file
4. Convert the `SALTDEPLOY` file into binary using `wsloadcf`.
5. Load the Service Metadata Entry file (`.mif`) into the Service Metadata Repository using the `tmloadrepos` command.
6. Boot (or re-boot) the GWWS process to initiate the new deployment.

The Web services binding reference extension initiates the Web services call.

Listing 2-31 shows an SCA component service exposed as a Web service.

Listing 2-31 Example SCA Component Service Exposed as a Web Service

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
       name="bigbank.account">
...
   <service name="AccountService">
       <interface.wsdl interface="http://www.bigbank.com/AccountService
              #wsdl.interface(AccountService)"/>
       <binding.ws/>
       <reference>AccountServiceComponent</reference>
   </service>

   <component name="AccountServiceComponent">
       <implementation.cpp
                     library="Account" header="AccountServiceImpl.h"/>
       <reference name="accountDataService">
              AccountDataServiceComponent
       </reference>
   </component>
...
</composite>
```

The steps required to expose the corresponding service are as follows:

1. Compose a WSDL interface matching the component interface.
2. Use `buildscacomponent` to build the application component runtime, similar to building a regular SCA component.
3. `buildscaserver -w` is used to convert SCDL code into a WSDF entry, and produce a deployable server (Oracle Tuxedo server + library + SCDL).

   The service from the above SCDL creates a WSDF entry as shown in Listing 2-32.

Listing 2-32 WSDF Entry

```
<Definition>
       <WSBinding id="AccountService_binding">
              <ServiceGroup id="AccountService">
                     <Service name="TuxAccountService"/>
              </ServiceGroup>
       </WSBinding>
</Definition>
```

4. `buildscaserver -w` also constructs a Service Metadata Repository entry based by parsing the SCDL and interface. The interface needs to be in WSDL form, and manually-composed in this release.
5. Make sure that the UBB source has the `TMMETADATA` and GWWS servers configured.
6. The Service Metadata Repository entry is loaded into the Service Metadata Repository using the `tmloadrepos` command.
7. The WSDF file must be imported into the `SALTDEPLOY` file and `SALTDEPLOY` converted into binary using `wsloadcf`.
8. The Service Metadata Entry file (`.mif`) is loaded into the Service Metadata Repository.
9. The Oracle Tuxedo server hosting the Web service is booted and made available.
10. The GWWS is rebooted to take into account the new deployment.

These steps are required, in addition to the SALTDEPLOY configuration, in order to set up the GWWS gateway for Web services processing (for example, configuration of GWInstance, Server Level Properties, etc.). When completed, Web service clients (SCA or other) have access to the Web service.

Listing 2-33 shows a reference accessing a Web service.

Listing 2-33 Example Reference Accessing a Web Service

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
       name="bigbank.account">
...
       <reference name="StockQuoteWebService">
              <interface.wsdl interface="http://www.webserviceX.NET/#
                            wsdl.interface(StockQuoteSoap)"/>
              <binding.ws endpoint="http://www.webserviceX.NET/#
                            wsdl.endpoint(StockQuote/StockQuoteSoap)"/>
       </reference>
...
</composite>
```

The steps required to access the Web service are as follows:

1. A WSDL file is necessary. This is usually published by the Web Service provider.
2. The WSDL file must be converted into a WSDF entry using the wsdlcvt tool. At the same time a Service Metadata Entry file (.mif), and fml32 mapping file is generated.
3. The WSDF file must be imported into the SALTDEPLOY file and SALTDEPLOY converted into binary using wsloadcf.
4. The Service Metadata Entry file (.mif) is loaded into the Service Metadata Repository using the tmloadrepos command.
5. The GWWS process is rebooted to take into account the new deployment.

These steps are required, in addition to the SALTDEPLOY configuration, in order to set up the GWWS gateway for Web services processing (for example, configuration of GWInstance, Server Level Properties, etc.). When completed, the SCA client has access to the Web service.

The process is the same, whether the client is stand-alone SCA program or an SCA component (already a server) referencing another SCA component via the Web service binding.

SCA Data Type Mapping

Using ATMI binding leverages the Oracle Tuxedo infrastructure. Data exchanged between SCA components, or Oracle Tuxedo clients/services and SCA clients/components is performed using Oracle Tuxedo typed buffers. Table 2-1 through Table 2-10 summarize the correspondence between native types and Oracle Tuxedo buffers/types, as well as SOAP types when applicable.

In the example shown in Listing 2-34, implementations send and receive an Oracle Tuxedo `STRING` buffer. To the software (binding and reference extension implementations), the determination of the actual Oracle Tuxedo buffer to be used is provided by the contents of the `/binding.atmi/inputBufferType`, `/binding.atmi/outputBufferType`, or `/binding.atmi/errorBufferType` elements in the SCDL configuration, and the type of buffer returned (or sent) by a server (or client). It does not matter whether client or server is an ATMI program or an SCA component.

Notice that the Oracle Tuxedo `simpapp` service has its own `namespace` within `namespace services`. A C++ method `toupper` is associated with this service.

Listing 2-34 C++ Interface Example

```
#include <string>
namespace services
{
    namespace simpapp
    {
        /**
         *  business interface
         */
           class ToupperService
        {
        public:

        virtual std::string
           toupper(const std::string inputString) = 0;
        };
```

```
    } // End simpapp
} // End services
```

The following data type mapping rules apply:

- Run-Time Data Type Mapping
- SCA Utility Data Type Mapping

Run-Time Data Type Mapping

- Simple Oracle Tuxedo Buffer Data Mapping
- Complex Return Type Mapping
- Complex Oracle Tuxedo Buffer Data Mapping

Simple Oracle Tuxedo Buffer Data Mapping

The following are considered to be simple Oracle Tuxedo buffers:

- `STRING`
- `CARRAY (and X_OCTET)`
- `MBSTRING`
- `XML`

Table 2-1 lists simple Oracle Tuxedo buffer types that are mapped to SCA binding.

Table 2-1 Simple Oracle Tuxedo Buffer Type Data Mapping

C++ or STL Type	Java Type	Oracle Tuxedo Buffer Type	Notes
`char*`, `char array` or `std::string`	`java.lang.String`	`STRING`	
`CARRAY_T`	`byte[]` or `java.lang.Byte[]`	`CARRAY`	

Table 2-1 Simple Oracle Tuxedo Buffer Type Data Mapping

C++ or STL Type	Java Type	Oracle Tuxedo Buffer Type	Notes
`X_OCTET_T`	`byte[]` or `java.lang.Byte[]`	`X_OCTET`	
`XML_T`	`byte[]` or `java.lang.Byte[]`	`XML`	This type is passed as a C++ array within the data element of struct XML or as an array of java bytes. It is transformed to SDO.
`wchar_t *` or `wchar_t array`	`N/A`	`MBSTRING`	See Multibyte String Data Mapping
`std::wstring`	`java.lang.String`	`MBSTRING`	See Multibyte String Data Mapping

When a service called by an SCA client returns successfully, a pointer to the service return data is passed back to the Proxy stub generated by `buildscaclient`. The Proxy stub then de-references this pointer and returns the data to the application.

Table 2-1 can be interpreted as follows:

- When the reference or service binding extension runtime sees an Oracle Tuxedo `STRING` buffer, it looks for either a `char*`, `char array`, `std::string` parameter or `return` type (depending on the direction). If a different type is found, an exception is thrown with a message explaining what happened.
- When the reference or service binding extension runtime sees a `char*` (for example) as a single parameter or return type, it looks for STRING as the buffer type in the `binding.atmi` element. If a different Oracle Tuxedo buffer type is found, an exception is thrown with a message explaining what happened.

Multibyte String Data Mapping

Oracle Tuxedo uses multibyte strings to represent multibyte character data with encoding names based on `iconv` (as defined by Oracle Tuxedo). C++ uses a `wstring`, `wchar_t*`, or `wchar_t[]` data type to represent multibyte character data with encoding names (as defined by the C++ library).

Oracle Tuxedo and C++ sometimes use different names to represent a particular multibyte encoding. Mapping between Oracle Tuxedo encoding names and C++ encoding names is as follows:

- Receiving a Multibyte String Buffer

  When an SCA client or server receives an `MBSTRING` buffer or an `FML32` buffer with a `FLD_MBSTRING` field, it considers the encoding for that multibyte string to be the first locale from the following cases:

  a. Locale associated with the `FLD_MBSTRING` field, if present.

     Note: For more information, see Table 2-2.

  b. Locale associated with the `MBSTRING` or `FML32` buffer.

  c. Locale set in the environment of the SCA client or server.

  If case a or b is matched, Oracle Tuxedo invokes the `setlocale()` function for locale type `LC_CTYPE` with the locale for the received buffer. If `setlocale()` fails (indicating there is no such locale) and an alternate name has been associated with this locale in the optional `$TUXDIR/locale/setlocale_alias` file, Oracle Tuxedo attempts to set the `LC_CTYPE` locale to the alternate locale.

  The `$TUXDIR/locale/setlocale_alias` file may be optionally created by the Oracle Tuxedo administrator. If present, it contains a mapping of Oracle Tuxedo `MBSTRING` codeset names to an equivalent operating system locale accepted by the `setlocale()` function.

  Lines consist of an Oracle Tuxedo MBSTRING codeset name followed by whitespace and an OS locale name. Only the first line in the file corresponding to a particular `MBSTRING` codeset name are considered. Comment lines begin with #.

  The `$TUXDIR/locale/setlocale_alias` file is used by the SALT SCA software when converting `MBSTRING` data into C++ `wstring` or `wchar_t[]` data. If `setlocale()` fails when using the Oracle Tuxedo `MBSTRING` codeset name, then the SALT SCA software attempts to use the alias name, if present. For example, if the file contains a line '`GB2312 zh_CN.GB2312`' then if `setlocale(LC_CTYPE, 'GB2312')` fails, the SALT SCA software attempts `setlocale(LC_CTYPE, 'zh_CN.GB2312')`.

- Sending a Multibyte String Buffer

  When an SCA client or server converts a `wstring`, `wchar_t[]`, or `wchar_t*` to an `MBSTRING` buffer or a `FLD_MBSTRING` field, it uses the `TPMBENC` environment variable value as the locale to set when converting from C++ wide characters to a multibyte string.

If the operating system does not recognize this locale, Oracle Tuxedo uses the alternate locale from the `$TUXDIR/locale/setlocale_alias` file, if any.

Note: It is possible to transmit multibyte data retrieved from an `MBSTRING` buffer, an `FML32 FLD_MBSTRING` field, or a `VIEW32` mbstring field. It is also possible to transmit multibyte data entered using the SDO `setString()` method.

However, it is not possible to enter multibyte characters directly into an XML document and transmit this data via SALT. This is because multibyte characters entered in XML documents are transcoded into multibyte strings, and SDO uses `wchar_t` arrays to represent multibyte characters.

Complex Return Type Mapping

The following C++ built-in types (used as return types) are considered complex and automatically encapsulated in an `FML/FML32` buffer as a single generic field following the complex buffer mapping rules described in Complex Oracle Tuxedo Buffer Data Mapping. This mechanism addresses the need for returning types where a corresponding Oracle Tuxedo buffer cannot be used.

Note: Interfaces returning any of the built-in types assume that `FML/FML32` is the output buffer type. The name of this generic field is `TUX_RTN`*`datatype`* based on the type of data being returned. `TUX_RTN`*`datatype`* fields are defined in the `Usysflds.h/Usysfl32.h` and `Usysflds/Usysfl32` shipped with Oracle Tuxedo.

- `bool` : maps to `TUX_RTNCHAR` field
- `char`: maps to `TUX_RTNCHAR` field
- `signed char`: maps to `TUX_RTNCHAR` field
- `unsigned char`: maps to `TUX_RTNCHAR` field
- `short`: maps to `TUX_RTNSHORT` field
- `unsigned short`: maps to `TUX_RTNSHORT` field
- `int`: maps to `TUX_RTNLONG` field
- `unsigned int`: maps to `TUX_RTNLONG` field
- `long`: maps to `TUX_RTNLONG` field
- `unsigned long`: maps to `TUX_RTNLONG` field
- `long long`: (maps to `TUX_RTNLONG` field

- `unsigned long long`: maps to `TUX_RTNLONG` field
- `float`: maps to `TUX_RTNFLOAT` field
- `double`: maps to `TUX_RTNDOUBLE` field
- long double: maps to `TUX_RTNDOUBLE` field

Complex Oracle Tuxedo Buffer Data Mapping

The following are considered to be complex Oracle Tuxedo buffers:

- `FML`
- `FML32`
- `VIEW (and X_* equivalents)`
- `VIEW32`

Table 2-2 lists the complex Oracle Tuxedo buffer types that are mapped to SCA binding.

For FML and FML32 buffers, parameter names in interfaces must correspond to field names, and follow the restrictions that apply to Oracle Tuxedo fields (length, characters allowed). When these interfaces are generated from metadata using `tuxscagen(1)`, the generated code contains the properly formatted parameter names.

If an application manually develops interfaces without parameter names, manually develops interfaces that are otherwise incorrect, or makes incompatible changes to SALT generated interfaces, then incorrect results are likely to occur.

VIEW (and `X_*` equivalents) and VIEW32 buffers require the use of SDO `DataObject` wrappers.

Listing 2-35 shows an interface example. The associated field definitions (following the interface) must be present in the process environment.

Table 2-2 Complex Oracle Tuxedo Buffer Type Data Mapping

C++, STL, or SDO type	Java Type	Oracle Tuxedo field type	Oracle Tuxedo view type	Notes
`bool`	`boolean` or `java.lang.Boolean`	`FLD_CHAR`	`char`	Maps to 'T' or 'F'. (This matches the mapping used elsewhere in SALT.)
`char`, `signed char`, or `unsigned char`	`byte` or `java.lang.Byte`	`FLD_CHAR`	`char`	
`short` or `unsigned short`	`short` or `java.lang.Short`	`FLD_SHORT`	`short`	An unsigned short is cast to a short before being converted to `FLD_SHORT` or `short`.
`int` or `unsigned int`	`int` or `java.lang.Integer`	`FLD_LONG`	`int`	An unsigned `int` being converted to FML or FML32 is cast to a long before being converted to `FLD_LONG` or `long`. An unsigned `int` being converted to a VIEW or VIEW32 member is cast to an `int`.
`long` or `unsigned long`	`long` or `java.lang.Long`	`FLD_LONG`	`long`	An exception is thrown if the value of a `64-bit long` does not fit into a `FLD_LONG` or `long` on a 32-bit platform. An unsigned long is cast to long before being converted to `FLD_LONG` or `long`.

Table 2-2 Complex Oracle Tuxedo Buffer Type Data Mapping

C++, STL, or SDO type	Java Type	Oracle Tuxedo field type	Oracle Tuxedo view type	Notes
long long or unsigned long long	N/A	FLD_LONG	long	An exception is thrown if the data value does not fit within a FLD_LONG or long. An unsigned long long is cast to long long before being converted to FLD_LONG or long.
float	float or java.lang.Float	FLD_FLOAT	float	
double	double or java.lang.Double	FLD_DOUBLE	double	
long double	N/A	FLD_DOUBLE	double	
char* or char array	N/A	FLD_STRING	string	
std::string	java.lang.String	FLD_STRING	string	
CARRAY_T or X_OCTET_T	class CARRAY	FLD_CARRAY	carray	Will map externally following GWWS rules. This departs from the OSOA spec. (which does not support them), and should be considered an improvement.
Bytes	N/A	FLD_CARRAY	Carray	This mapping is used when part of a DataObject
wchar_t* or wchar_t array	N/A	FLD_MBSTRING (FML32 only)	mbstring (VIEW32 only)	(Java char is Unicode and can range from -32768 to +32767.) See also Multibyte String Data Mapping

Table 2-2 Complex Oracle Tuxedo Buffer Type Data Mapping

C++, STL, or SDO type	Java Type	Oracle Tuxedo field type	Oracle Tuxedo view type	Notes
`std::wstring`	`java.lang.String`	`FLD_MBSTRING (FML32 only)`	`mbstring (VIEW32 only)`	See also Multibyte String Data Mapping
`commonj::sdo::DataObjectPtr`	`TypedFML32`	`FLD_FML32 (FML32 only)`	`N/A`	Generate a data transformation exception, which is translated to an `ATMIBindingException` before being returned to the application, when: • Attempting to add such a field in an Oracle Tuxedo buffer other than FML32 • The data object is not typed (i.e., there is no corresponding schema describing it). See also Multibyte String Data Mapping
`commonj::sdo::DataObjectPtr`	`TypedView32`	`FLD_VIEW32 (FML32 only)`	`N/A`	See also Multibyte String Data Mapping
`struct` *`structurename`*	`N/A`	`FLD_FML32 (FML32 only)`	*`structurename`*	See also SCA Structure Support

Listing 2-35 Interface Example

```
...
int myService(int param1, float param2); ...
Field table definitions
#name         number type          flag comment
#-------------------------------------------------------
param1 20     int           -     Parameter 1
param2 30     float         -     Parameter 2
...
```

SDO Mapping

C++ method prototypes that use `commonj::sdo::DataObjectPtr` objects as parameter or return types are mapped to an `FML, FML32, VIEW, or VIEW32` buffer.

You must provide an XML schema that describes the SDO object. The schema is made available to the service or reference extension runtime by placing the schema file (`.xsd` file) in the same location as the SCDL composite file that contains the reference or service definition affected. The schema is used internally to associate element names and field names.

Note: When using view or view32, a schema type (for example, `complexType`) which name matches the view or view32 used is required.

For more information, see `mkfldfromschema` and `mkfld32fromschema` in the *SALT* 11*g* Release 1 (11.1.1.0) *Command Reference*.

For example, a C++ method prototype defined in a header such as:

```
long myMethod(commonj::sdo::DataObjectPtr data);
```

Listing 2-36 shows the associated schema.

Listing 2-36 Schema

```
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
     xmlns="http://www.example.com/myExample"
     targetNamespace="http://www.example.com/myExample">

     <xsd:element name="bike" type="BikeType"/>
     <xsd:element name="comment" type="xsd:string"/>

     <xsd:complexType name="BikeType">
         <xsd:sequence>
             <xsd:element name="serialNO" type="xsd:string"/>
             <xsd:element name="name" type="xsd:string"/>
             <xsd:element name="type"  type="xsd:string"/>
             <xsd:element name="price"  type="xsd:float"/>
         </xsd:sequence>
```

```
    </xsd:complexType>
</xsd:schema>
```

Table 2-3 shows the generated field table.

Table 2-3 Generated Field Tables

NAME	NUMBER	TYPE	FLAG	Comment
bike	20	fml32	-	
comment	30	string	-	
serialNO	40	string	-	
name	50	string	-	
type	60	string	-	
price	70	float	-	

The following restrictions in XML schemas apply:

- Attributes cannot be specified and are ignored if specified
- Values in restrictions are ignored (their meaning is application-related), only the field name and type are generated
- When using XML schema types, only signed integral types are supported.
  See "SDO C++ Specification" for a list of available SDO primitive types.

SCA Utility Data Type Mapping

The `scatuxgen` and `tuxscagen` utilities are used to generate manual SCA data type mapping. The `scatuxgen` mapping rules are as follows:

- C++ Parameter/Return Type and Oracle Tuxedo Buffer Type Mapping
- C++ Parameter Type and Oracle Tuxedo Parameter Type Mapping
- C++ Parameter Type and Oracle Tuxedo Complex Type Mapping
- Parameter and Return Types to Parameter-Level Keyword Restrictions

Note: The mapping rules for `tuxscagen` are executed in the reverse direction (Oracle Tuxedo Buffer Type -> C++ Parameter Type).

C++ Parameter/Return Type and Oracle Tuxedo Buffer Type Mapping

Table 2-4 shows the correspondence between parameter/return types and Oracle Tuxedo buffer types (inbuf service-level keyword).

Table 2-4 'inbuf' Keyword Buffer Type Mapping Table

C++ Parameter Type	Oracle Tuxedo Buffer Type
`std::string or char*`	`STRING`
`struct carray_t`	`CARRAY`
`char`	`FML32`
`short`	`FML32`
`int`	`FML32`
`long`	`FML32`
`float`	`FML32`
`double`	`FML32`
`wchar_t[ ]`	`MBSTRING`
`struct xml_t`	`XML`
`struct x_octet_t`	`X_OCTET`
`commonj::sdo::DataObjectPtr`	`X_COMMON, X_C_TYPE, VIEW, VIEW32, FML, or FML32` depending on intputBufferType setting

Table 2-4 'inbuf' Keyword Buffer Type Mapping Table

C++ Parameter Type	Oracle Tuxedo Buffer Type
`struct structurename`	`X_COMMON, X_C_TYPE, VIEW, VIEW32, FML, or FML32` depending on intputBufferType setting
multiple parameters, or one commonj::sdo::DataObjectPtr or struct structurename and no binding.atmi or no corresponding inputBufferType and the input buffer is not specified using a command line option	`FML32`

Table 2-5shows the correspondence between parameter/return types and Oracle Tuxedo buffer types (outbuf or err buf service-level keywords).

Table 2-5 outbuf' or 'errbuf' Keyword Buffer Type Mapping Table

C++ Return Type	Oracle Tuxedo Buffer Type
`std::string or char*`	`STRING`
`struct carray_t`	`CARRAY`
`char`	`FML32`
`short`	`FML32`
`int`	`FML32`
`long`	`FML32`
`float`	`FML33`
`double`	`FML32`
`wchar_t[], wstring`	`MBSTRING`
`struct xml_t`	`XML`
`struct x_octet_t`	`X_OCTET`

Table 2-5 outbuf' or 'errbuf' Keyword Buffer Type Mapping Table

C++ Return Type	Oracle Tuxedo Buffer Type
`commonj::sdo::DataObjectPtr`	`X_COMMON`, `X_C_TYPE`, `VIEW`, `VIEW32`, `FML` or `FML32` depending on the `binding.atmi/outputBufferType` or `binding.atmi/errorBufferType` setting.
`commonj::sdo::DataObjectPtr`	`FML32` if no binding.atmi is set, or binding.atmi is set and binding.atmi/outputBufferType or binding.atmi/errorBufferType aren't specified.
`struct structurename`	`X_COMMON`, `X_C_TYPE`, `VIEW`, `VIEW32`, `FML` or `FML32` depending on the binding.atmi/outputBufferType or binding.atmi/errorBufferType setting.
`struct` *`structurename`*	`FML32` if no binding.atmi is set, or binding.atmi is set and binding.atmi/outputBufferType or binding.atmi/errorBufferType are not specified.

C++ Parameter Type and Oracle Tuxedo Parameter Type Mapping

Table 2-7 shows how `scatuxgen` handles interface parameter types and converts them to an Oracle Tuxedo Service Metadata Repository parameter-level keyword value when more than one parameter is used in the method signature.

Table 2-6 Parameter-Level/Field Type Mapping Table

C++ Parameter Data Type	Oracle Tuxedo Parameter-Level Keyword (FML FIELD Type)
`char`	`byte(FLD_CHAR)`
`short`	`short(FLD_SHORT)`
`int`	`integer(FLD_LONG)`
`long`	`integer(FLD_LONG)`
`float`	`float(FLD_FLOAT)`
`double`	`double(FLD_DOUBLE)`
`std::string or char *`	`string(FLD_STRING)`
`struct carray_t`	`carray(FLD_CARRAY)`
`std::wstring`	`mbstring(FLD_MBSTRING)`

Table 2-6 Parameter-Level/Field Type Mapping Table

C++ Parameter Data Type	Oracle Tuxedo Parameter-Level Keyword (FML FIELD Type)
`commonj::sdo::DataObjectPtr`	`fml32(FLD_FML32)`
`struct structurename`	`fml32(FLD_FML32)`

C++ Parameter Type and Oracle Tuxedo Complex Type Mapping

This section contains the following topics:

- SDO Mapping
- C Struct Mapping

SDO Mapping

When a method takes an SDO object as an argument, or returns an SDO object, for example as follows: `commonj::sdo::DataObjectPtr myMethod(commonj::sdo::DataObjectPtr input)`.

The corresponding runtime type may be described by an XML schema as shown in Listing 2-37 and then referenced in the binding as shown in Listing 2-38.

Listing 2-37 XML Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="tuxedo"
targetNamespace="tuxedo">

    <xsd:complexType name="BikeInventory">
        <xsd:sequence>
           <xsd:element name="BIKES" type="Bike"
                        minOccurs="0" maxOccurs="unbounded"/>
           <xsd:element name="STATUS" type="xsd:string" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="Bike">
        <xsd:sequence>
```

```
            <xsd:element name="SERIALNO" type="xsd:string"/>
            <xsd:element name="SKU" type="xsd:string"/>
            <xsd:element name="NAME" type="xsd:string"/>
            <xsd:element name="TYPE" type="xsd:string"/>
            <xsd:element name="PRICE" type="xsd:float"/>
            <xsd:element name="SIZE" type="xsd:int"/>
            <xsd:element name="INSTOCK" type="xsd:string"/>
            <xsd:element name="ORDERDATE" type="xsd:string"/>
            <xsd:element name="COLOR" type="xsd:string"/>
            <xsd:element name="CURSERIALNO" type="xsd:string"/>
         </xsd:sequence>
    </xsd:complexType>

</xsd:schema>
```

Listing 2-38 Binding

```
...
  <reference name="UBIK">
    <interface.cpp header="uBikeService.h"/>
    <binding.atmi>
       <inputBufferType>FML32/Bike</inputBufferType>
       <outputBufferType>FML32/BikeInventory</outputBufferType>
    </binding.atmi>
  </reference>
  ...
```

When such a schema is present, `scatuxgen` parses it and generates the corresponding parameter-level mapping entries as listed in Table 2-7.

Table 2-7 Parameter-level/Field Type Mapping

XML Schema element type	Oracle Tuxedo Parameter-Level Keyword (FML FIELD Type)
xsd:byte	byte(FLD_CHAR)
xsd:short	short(FLD_SHORT)
xsd:int	integer(FLD_LONG)
xsd:long	integer(FLD_LONG)
xsd:float	float(FLD_FLOAT)
xsd:double	double(FLD_DOUBLE)
xsd:string	string(FLD_STRING)
xsd:string	mbstring(FLD_MBSTRING) when -t option is specified
xsd:base64binary	carray(FLD_CARRAY)
xsd:complexType	fml32(FLD_FML32)
xsd:minOccurs	requiredcount
xsd:maxOccurs	count

C Struct Mapping

When a method takes a C struct as an argument, or returns a C struct (for example, as shown in Listing 2-39), scatuxgen parses it and generates the corresponding parameter-level mapping entries listed in Table 2-8.

Listing 2-39 C Struct

```
struct customer {
    char firstname[80];
    char lastname[80];
    char address[240];
};

struct id {
```

```
    int SSN;
    int zipCode;
};

struct customer* myMethod(struct *id input);
```

Table 2-8 Parameter-Level/Field Type Mapping

Struct Member Type	Oracle Tuxedo Parameter-Level Keyword (FML FIELD Type)
char, unsigned char, signed char	byte(FLD_CHAR)
char [ ]	string(FLD_STRING)
wchar_t [ ]	mbstring(FLD_MBSTRING)
short, unsigned short	short(FLD_SHORT)
int, unsigned int	integer(FLD_LONG)
long, unsigned long, long long, unsigned long long	integer(FLD_LONG)
float	float(FLD_FLOAT)
double, long double	double(FLD_DOUBLE)
struct *nestedstructname* (for more information, see SCA Structure Support)	fml32 (FLD_FML32)
array type	count=requiredcount=array specifier

Parameter and Return Types to Parameter-Level Keyword Restrictions

For parameter-level keywords, the Oracle Tuxedo buffer type/parameter type restrictions are consistent with the contents expected by `tmloadrepos`. An error message is returned when an attempt to match any combinations that are not listed in Table 2-9 and Table 2-10.

Table 2-9 Oracle Tuxedo Buffer Type/Parameter Type Restrictions (Part 1)

Parameter Type / Oracle Tuxedo Buffer	byte(char)	short	integer	float	double	String
CARRAY						
FML	X	X	X	X	X	X
FML32	X	X	X	X	X	X
VIEW	X	X	X	X	X	X
VIEW32	X	X	X	X	X	X
X_COMMON		X		X		X
X_C_TYPE	X	X	X	X	X	X
X_OCTET						
STRING						X
XML						X
MBSTRING						

Table 2-10 Oracle Tuxedo Buffer Type/Parameter Type Restrictions (Part 2)

Parameter Type / Oracle Tuxedo Buffer	carray	xml	view32	fml32	mbstring
CARRAY	X				
FML	X				
FML32	X	X	X	X	X

Table 2-10 Oracle Tuxedo Buffer Type/Parameter Type Restrictions (Part 2)

Parameter Type / Oracle Tuxedo Buffer	carray	xml	view32	fml32	mbstring
VIEW	X				
VIEW32	X				X
X_COMMON					
X_C_TYPE					
X_OCTET	X				
STRING					
XML		X			
MBSTRING	X				X

Python, Ruby, and PHP Data Type Mapping

The following sections describe the supported data types in Python, Ruby, and PHP clients or components with respect to the native, C/C++ based environment, and in order to give the correspondence for writing the Oracle Tuxedo Service Metadata Repository interface required by the ATMI binding. Corresponding Oracle Tuxedo buffer and field type are also indicated for uses with the ATMI or Web Services binding.

- Python Data Type Mapping
- Ruby Data Type Mapping
- PHP Data Type Mapping

Python Data Type Mapping

In Python, clients or components only use parameters and return values which types are listed in Table 2-11. Multiple parameters are supported (in the same way that multiple parameters are supported in C++), using FML32 Oracle Tuxedo buffers.

Note: Arrays are not supported as they are not supported by bindings or the C++ language extension.

Table 2-11 Supported Python, C++ and Oracle Tuxedo Buffer Types

Python parameter(s) or Return Type	C/C++ Native Type	ATMI Binding Type Buffer type/Field Type
`int`	`short, unsigned short`	`FML32/FLD_SHORT`
`long`	`short, unsigned short`	`FML32/FLD_SHORT`
`int`	`long, unsigned long`	`FML32/FLD_LONG`
`long`	`long, unsigned long`	`FML32/FLD_LONG`
`bool`	`bool`	`FML32/FLD_CHAR`
`float`	`float`	`FML32/FLD_FLOAT`
`float`	`double, long double`	`FML32/FLD_DOUBLE`
`string of length 1`	`char`	`FML32/FLD_CHAR`
`string`	`char *, std::string`	`STRING`
`xml`	`commonj::sdo::DataObject Ptr`	`FML32, VIEW, VIEW32`

Notes: int (short), long, int (long), float (float) are allowed in the C++ to Python direction only. The Python runtime catches any overflow situation (e.g.: when copying a C++ long into a Python int).

In order to map a string of length 1 to a `char*/std::string/STRING`, the originating Python variable will have to have 2 ending zeroes (for example, `'t = "a\x00"`).

Supported XML objects in Python must be `xml.etree.ElementTree objects`, (that is, the language extension converts `xml.etree.ElementTree` objects into `commonj::sdo::DataObjectPtr` objects, and `commonj::sdo::DataObjectPtr` objects into `xml.etree.ElementTree` objects.

Using lists and dictionaries are also supported, as detailed in Python Parameters and Dictionaries.

Note: Lists and dictionaries are allowed as parameters, but are not allowed to be returned.

Some limitations concerning multiple parameters and lists will stand with respect to using bindings. For more information, see Python, Ruby, and PHP Binding.

Python Parameters

You can use the list notation (*) to pass an undetermined number of parameters to/from a Python program. For example:

```
def test(*params)
    for p in params:
        print "parameter:", p
```

and an example of call: `test(1, 2, 3, 4, 5)`

This notation is equivalent to having an actual list of parameters, such as:

```
def test(parm1, parm2, parm3, parm4, parm5)
    ...
```

Individual supported types are limited to the types listed in Table 2-11.

Exposing a Python function as an SCA service with ATMI or Web services binding requires an interface. This interface is stored in the Oracle Tuxedo Service Metadata Repository as outlined in Python, Ruby, and PHP Component Programming.

When called, the Python function receives a list of parameters corresponding exactly to what the interface specifies. Any extra parameters passed by the client are ignored, and any type mismatch results in a data mapping exception.

Note: Using this notation is limited to local calls (no binding), or using ATMI binding between SCA components (that is, the `<binding.atmi>` element with no `requires="legacy"` attribute).

For local calls (no binding specified), or references, no interface is required.

Dictionaries

You can use the named parameters notation (**) to pass name/value pairs, also known as dictionaries, to/from Python programs. For example:

```
def test(**params):
    for p in params.keys():
        print "key:", p, " parameter:", params[p]
```

and an example of call: `test(a=1, b=2)`

Individual supported types are limited to the types listed in Table 2-11.

Exposing a Python function as an SCA service with the ATMI or Web Services binding requires an interface. This interface is stored in the Oracle Tuxedo Service Metadata Repository as outlined in Python, Ruby, and PHP Component Programming.

For example, consider the Oracle Tuxedo Service Metadata Repository entry shown in Listing 2-40

Listing 2-40 Oracle Tuxedo Service Metadata Repository Entry for Python

```
##
service=testPython2
tuxservice=TESTPT
inbuf=FML32
outbuf=FML32

param=NUMBER
type=long
access=in

param=TEXT
type=string
access=in

param=FNUMBER
type=double
access=in
##
```

When called, the Python function receives a list of parameters corresponding exactly to what the interface specifies. Any extra parameters passed by the client are ignored, and any type mismatch results in a data mapping exception.

The names of the parameters match the key names passed to the Python function. The interface is obtained by making an internal call to the `TMMETADATA` server. The `TMMETADATA` server must be running in order to make calls to Python, Ruby or PHP functions.

A Python function called with the interface is equivalent to the following Python call:

```
test(a=1, b=2)
```

Ruby Data Type Mapping

Table 2-12 lists supported Ruby, C/C++ and Oracle Tuxedo buffer types. Multiple parameters are supported (in the same way that multiple parameters are supported in C++), using FML32 Oracle Tuxedo buffers.

Arrays are not supported as they are not supported by bindings or the C++ language extension.

Table 2-12 Supported Ruby, C++ and Oracle Tuxedo Buffer Types

Ruby parameter or return type	C/C++ native type	ATMI binding type Buffer type/Field type
Fixnum	short, unsigned short	FML32/FLD_SHORT
Fixnum	long, unsigned long	FML32/FLD_LONG
Bignum	double, long double	FML32/FLD_DOUBLE
True/false	bool	FML32/FLD_CHAR
Float	float	FML32/FLD_FLOAT
Float	double, long double	FML32/FLD_DOUBLE
String	char *, std::string	STRING
REXML Object	commonj::sdo::DataObject Ptr	FML32, VIEW, VIEW32

Notes: Ruby runtime may catch an overflow exception.

Possible loss of precision when the Ruby Bignum is bigger than a C++ double.

Float (float) is allowed in C++ to Ruby direction only.

There is no mapping to single character (char/FLD_CHAR) possible in Ruby.

Supported XML objects in Ruby must be REXML (that is, the language extension converts `REXML::Document` objects into `commonj::sdo::DataObect` objects, and `commonj::sdo::DataObjectPtr` objects into `REXML::Document` objects.

Using variable argument lists and hashes are also be supported, as detailed in the following paragraphs.

Note: Variable argument lists and hashes are allowed as parameters, but are not allowed to be returned.

Some limitations concerning multiple parameters and lists will stand with respect to using bindings. For more information, see Python, Ruby, and PHP Binding.

Ruby Parameters

You can use the list notation (*) to pass an undetermined number of parameters to/from a Ruby script. For example:

```
def func(a, b, *otherargs)
    puts a
    puts b
    otherargs.each { |arg| puts arg }
end
```

which can be called like this: `func(1, 2, 3, 4, 5)`

Individual supported types are limited to the types listed in Table 2-12.

Exposing a Ruby function as an SCA service with the ATMI or Web Services binding requires an interface. This interface is stored in the Oracle Tuxedo Service Metadata Repository as outlined in Python, Ruby, and PHP Component Programming.

For example, consider the Oracle Tuxedo Service Metadata Repository entry shown in Listing 2-41

Listing 2-41 Oracle Tuxedo Service Metadata Repository Entry for Ruby

```
##
service=testRuby
tuxservice=TESTRU
inbuf=FML32
outbuf=FML32

param=first
type=char
access=in

param=next
```

```
type=long
access=in

param=last
type=string
access=in

##
```

When called, the Ruby function receives a list of parameters corresponding exactly to what the interface specifies. Any extra parameters passed by the client are ignored, and any type mismatch results in a data mapping exception.

Notes: Using this notation is limited to local calls (no binding), or with using the ATMI binding between SCA components (that is, the `<binding.atmi>` element with no `requires="legacy"` attribute).

Local calls (no binding specified), or references, do not require an interface.

Hash

You can use named parameters in the form of hash type parameters to pass name/value pairs to/from Ruby scripts. For example:

```
def func2(hash)
    hash.each_pair do |key, val|
        puts "#{key} -> #{val}"
    end
end
```

which can be called like this: `func2("first" => true, "next" => 5, "last" => "hi")`

Individual supported types are limited to the types listed inTable 2-12.

When exposing a Ruby function as an SCA service with the ATMI or Web Services binding, an interface is required. This interface is stored in the Oracle Tuxedo Service Metadata Repository as outlined in Python, Ruby, and PHP Component Programming.

When called, the Ruby function receives a list of parameters corresponding exactly to what the interface specifies. Any extra parameters passed by the client are ignored, and any type mismatch results in a data mapping exception.

The names of the parameters match the key names passed to the Ruby function (that is, a Ruby function called with the above interface is equivalent to the following Ruby client call:

```
func2("first" => true, "next" => 5, "last" => "hi")
```

where the values 'true', 5 and 'hi' are arbitrary, not the keys.

PHP Data Type Mapping

Table 2-13 lists supported Ruby, C/C++ and Oracle Tuxedo buffer types. Multiple parameters are supported (in the same way that multiple parameters are supported in C++), using FML32 Oracle Tuxedo buffers.

Arrays are not supported as they are not supported by bindings or the C++ language extension.

Table 2-13 Supported PHP, C++ and Oracle Tuxedo Buffer Types

PHP parameter(s) or return type	C/C++ native type	ATMI binding type Buffer type/Field type
integer	short, unsigned short	FML32/FLD_SHORT
integer	long, unsigned long	FML32/FLD_LONG
boolean	bool	FML32/FLD_CHAR
float1		
float	FML32/FLD_FLOAT	
float	double, long double	FML32/FLD_DOUBLE
string of length 1	char	FML32/FLD_CHAR
string	char *, std::string	STRING
string (return type, see below)	commonj::sdo::DataObject Ptr	FML32, VIEW, VIEW32
object of type SimpleXMLElement (parameter, see below)	commonj::sdo::DataObject Ptr	FML32, VIEW, VIEW32

Returning XML data in PHP is done by returning a STRING object which is then converted into a `SimpleXMLElement` as follows:

```
$ret = $svc->searchBike('YELLOW');
$xml = new SimpleXMLElement($ret, LIBXML_NOWARNING);
```

Once the XML object constructed, it will be accessed as follows:

```
echo "First serialno:".$xml->BIKES[0]->SERIALNO."\n";
echo "Second serialno:".$xml->BIKES[1]->SERIALNO."\n";
```

List of Parameters

You are permitted to pass an undetermined number of parameters when making an SCA reference using the PHP extension. For example:

```
test(1, 2, 3, 4, 5);
```

Individual supported types are limited to the types listed in Listing 2-13, with the exception of types originating from or becoming `commonj::sdo::DataObjectPtr objects`.

Note: Using this notation is limited to:

- local calls (no binding), or
- using the ATMI binding between SCA components (i.e., <binding.atmi> element with no requires="legacy" attribute). For local calls (no binding specified), or
- references

No interface is required.

Named Parameters

You can use named parameters to pass name/value pairs using the PHP SCA extension. For example:

```
$svc->searchBike(array('COLOR' => 'RED', 'TYPE' => 'MTB'));
```

Individual supported types are limited to the types listed in Table 2-13.

SCA Structure Data Type Mapping

In SCA-ATMI applications, an SCA structure parameter can be mapped to an `ATMI FML32, FML, VIEW32, VIEW, X_COMMON, or X_C_TYPE` data type, and this is the data type that is specified in the SCA composite file.

If a `VIEW32, VIEW, X_COMMON, or X_C_TYPE` data type is specified, then this view must exactly match the structure used as an SCA parameter or return type.

Note: In order for the view to exactly match the structure, the compilation of the view needs to produce the same structure with the same fields and same offsets as the structure used in the application.

SCA Structure and FML32 or FML Mapping

If the SCA structure parameter is mapped to `FML32` or `FML`, then the field type of the associated FML32 or FML field is a type that can be converted to and from the SCA structure data type For more informations, see SCA Data Type Mapping.

FML Field Naming Requirements

In SCA-SCA applications, fields are identified by field number, and FML32 field numbers are automatically generated. In the case of nested structures, field numbers are assigned as if the fields in the inner structure had occurred as flat fields in the outer structure in the place where the inner structure is defined in the outer structure.

In SCA-ATMI applications, the FML32 or FML field name associated with a structure element shall be obtained from the structure description file. For more information, see Using SCA Structure Description Files.

Long Element Truncation

When converting an FML32 or FML string, carray, or mbstring field to a structure element, any data that does not fit in the structure element is truncated (without warning) to the provided length.

For example, if a structure element is char `COMPANY_NAME[20]`; and FML field `COMPANY_NAME` with value "International Business Machines" is mapped to this structure element, then "International Busine" is copied to the structure element with no trailing null character.

SCA Structure and VIEW32, VIEW, X_OCTET, or X_C_TYPE Mapping

If an SCA structure is mapped to a `VIEW32, VIEW, X_OCTET, or X_C_TYPE` data type, then the structure used for the Oracle Tuxedo view-based type must exactly match the SCA structure, and is copied byte-by-byte. In other words, no marshalling of data is done when converting between an SCA structure, and a view. FML32 or FML should be used if data marshalling is required.

When an SCA structure is mapped to a view-based Oracle Tuxedo type, you cannot specify `bool`, `wchar_t`, `long long`, `unsigned long long`, `long double`, or nested structure data types within the SCA structure since corresponding data types do not exist within Oracle Tuxedo

views. Elements corresponding to any Oracle Tuxedo Associated Count Member or Associated Length Member fields must be provided. Appropriate values for any such elements must also be provided by the application if converting an SCA structure to an Oracle Tuxedo view.

SCA Structure and Mbstring Mapping

An mbstring field type currently exists in VIEW32 (for more information, see `tpconvvmb32`). SCA structures treat the mbstring field type in the same way as in VIEW32. The encoding information is part of an mbstring structure element, and `Fmbunpack32()` and `Fmbpack32()` must be used in application programs using mbstring data in structures.

TPFAIL Return Data

You can specify a structure pointer as data returned on TPFAIL if the same structure pointer is also returned on successful output. Since SCA must store internal information describing the returned structure along with the application data, `<tuxsca.h>` is used to define the structure and `typedef` as shown in Listing 2-42.

Listing 2-42 <tuxsca.h> SCA Structure and Typedef Definition

```
struct scastruct_t {
  void *data;
  void *internalinfo;
};
typedef struct scastruct_t *SCASTRUCT_PTR;
```

If an application normally returns "`struct mystruct *`" data, it accesses TPFAIL data as shown in Listing 2-42.

Listing 2-43 TPFAIL Example

```
… catch (Tuscany::sca::atmi::ATMIBindingException& abe) {
  SCASTRUCT_PTR *scap = (SCASTRUCT_PTR *)abe.getData();
  struct mystruct *result = (struct mystruct *)scap->data;
}
```

SCA and Oracle Tuxedo Interoperability

Existing Oracle Tuxedo service interoperability is performed by using the `/binding.atmi/@requires` attribute with the legacy value. When a legacy value is specified, invocations are performed using the following behavior:

- If a `<map>` element is present in either a `<reference>` or a `<service>`, that value is used to determine which Oracle Tuxedo service is associated with the specified method name to call or advertise.

Otherwise:

- In a `<reference>` element: the value specified in the `/reference/@name` element is used to perform the Oracle Tuxedo call, with semantics according to the interface method used.
- In a `<service>` element: the Oracle Tuxedo service specified in the `/binding.atmi/map` element is advertised, and mapped to the method specified in the `/binding.atmi/map/@target` attribute.

Additionally, the `/binding.atmi/@requires` attribute is used to internally control data mapping, such that FML32 or FML field tables are not required.

Note: When *not* specified, communications are assumed to have SCA -> SCA semantics where the actual Oracle Tuxedo service name is constructed from `/service/@name` or `/reference/@name` and actual method name (see the pseudo schema shown Listing 2-27).

SCA Transactions

The ATMI binding schema supports SCA transaction policies by using the `/binding.atmi/@requires` attribute and three transaction values. These transaction values specify the transactional behavior that the binding extension follows when ATMI binding is used (see the pseudo schema shown Listing 2-27).

The transaction values are as follows:

- Not specified (no value)

  All transactional behavior is left up to the Oracle Tuxedo configuration. If the Oracle Tuxedo configuration supports transactions, then a transaction can be propagated if it exists.

  If the Oracle Tuxedo configuration does not support transactions and a transaction exists, then an error occurs.

Note: A transaction is not started if a transaction does not already exist.

- `suspendsTransaction`

  When specified, the transaction context is not propagated to the service called. For a `<service>`, the transaction (if present), is automatically suspended before invoking the application code, and resumed afterwards, regardless of the outcome of the invocation. For a `<reference>`, equivalent to making a `tpcall()` with the `TPNOTRAN` flag.

- `propagatesTransaction`

  Only applicable to `<reference>` elements, ignored for `<service>` elements. Starts a new transaction if one does not already exist, otherwise participate in existing transaction. Such a behavior can be obtained in a component or composite `<service>` by configuring it `AUTOTRAN` in the UBBCONFIG. An error is generated if an Oracle Tuxedo server hosts the SCA component implementation and is not configured in a transactional group in the UBBCONFIG.

SCA Security

SCA references pass credentials using the `<authentication>` element of the `binding.atmi` SCDL element.

SCA services can be ACL protected by referencing their internal name: `/binding.atmi/service/@name` attribute followed by a '`/`' and method name in SCA -> SCA mode, `/binding.atmi/service/@name` attribute in `legacy` mode (SCA -> Tux interop mode).

For more information, see SCA and Oracle Tuxedo Interoperability.

CHAPTER 3

SCA Command Reference

Table 1 lists SCA commands and functions.

Table 1 Oracle Tuxedo Commands and Functions

Name	Description
`buildscaclient`	Builds processes that call SCA components.
`buildscacomponent`	Builds SCA components.
`buildscaserver`	Parses SCDL definitions and interfaces and produces a Tuxedo-deployable server and elements.
`mkfldfromschema`, `mkfld32fromschema`	The `mkfldfromschema` and `mkfld32fromschema` commands take an XML schema as input and produce a field table.
`mkviewfromschema`, `mkview32fromschema`	The `mkviewfromschema` and `mkview32fromschema` commands take an XML schema as input and produce a view file.
`scaadmin`	SCA server management command interpreter.
`SCAHOST (5)`	Generic server for Python, Ruby, or PHP components.
`scapasswordtool`	Manages passwords for Oracle Tuxedo authentication in SCA clients.

Table 1 Oracle Tuxedo Commands and Functions

Name	Description
scastructc32, scastructc(1)	Oracle Tuxedo structure description file compiler
scastructdis32, scastructdis	Binary structure and view files disassembler.
scatuxgen(1)	Generates Oracle Tuxedo Service Metadata Repository interface information from an SCA interface.
setSCAPasswordCallback(3c)	Sets the callback for retrieving a password associated with an identifier in a <binding.atmi> element.
tuxscagen(1)	Generates SCA, SCDL, and server-side interface files for Tuxedo services.

buildscaclient

Name

buildscaclient – Builds processes that call SCA components.

Synopsis

```
buildscaclient -c default_component [-v] [-h] [-k] [-o name] [-s SCAroot]
[-f firstfiles] [-l lastfiles] [-S structurefiles]
```

Description

This command is used to build client processes that can call SCA components hosted in Tuxedo environments. The command combines files, specified using the -f and -l options, with the SCA and standard Tuxedo ATMI libraries to form a client application. The client application is built using the default C++ language compile command defined for the operating system in use, unless overridden using the CC environment variable.

All specified .c and .cpp files are compiled in one invocation of the compilation system based on the operating system. Users may specify the compiler to invoke by setting the CC environment variable to the name of the compiler. If the CC environment variable is not defined when buildscaclient is invoked, the default C++ language compile command for the operating system is invoked to compile all .c and .cpp files.

You may specify additional options to be passed to the compiler by setting the CFLAGS or the CPPFLAGS environment variables. If CFLAGS is not defined when buildobjclient is invoked, then buildscaclient uses the value of CPPFLAGS, if that variable is defined.

Parameters and Options

buildscaclient supports the following parameters and options:

-c defaultcomponent
Required parameter. Indicates which component should be used for this application.

[-v]
Specifies that the buildscaclient command should work in verbose mode. In particular, it writes the compile command to its standard output.

[-k]
Maintains the generated stubs. buildscaclient generates proxy files that allow dynamic interfacing of clients and references. This is normally compiled and then removed when the proxy is built. This option indicates that the source file should be retained.

Caution: The generated contents of this file may change from release to release. It is advised that you *do not* depend on the data structures and interfaces exposed in this file. This option is provided to aid in debugging of build problems.

[-o name]
Specifies the name of the client application generated by this command. If the name is not supplied, the application file is named client<.type>, where type is an extension that is dependent on the operating system. For example, on a UNIX system, there would not be a type, but on a Windows system, the type would be .EXE.

[-s scaroot]
Specifies the location of SCA root, where the SCDL files for the required components are located. If not set, the APPDIR environment value is used.

[-f firstfiles]
Specifies the file to be included first in the compile and link phases of the buildscaclient command. The specified file is included before the SCA libraries are included. There are two ways of specifying a file or files:

Filename Specification	Description
`-f firstfile`	One file is specified
`-f "file1.cpp file2.cpp file3.cpp …"`	Multiple files may be specified if their names are enclosed in quotation marks and are separated using white spaces.

Note: Filenames that include spaces are not supported. The `-f` option may be specified multiple times.

`[-l lastfiles]`

Specifies a file to be included last in the compile and link phases of the `buildscaclient` command. The specified file is included after the SCA libraries are included. There are two ways of specifying the file, as shown in the following table.

Filename Specification	Description
`-l lastfile`	One file is specified
`-l "file1.cpp file2.cpp file3.cpp …"`	Multiple files may be specified if their names are enclosed in quotation marks and are separated using white spaces.

Note: Filenames that include spaces are not supported. The `-l` option may be specified multiple times.

`[-S structurefiles]`

Specifies an SCA structure description file. The structure description file may be either a source file or a binary structure description file. If more than one file is specified, file names must be separated by white space and the entire list must be enclosed in quotation marks. The `-S` option may be specified multiple times on the same command line.

The use of structure description files is optional. If a structure description is not provided for a particular structure then the source code where the structure is defined is used describe the structure; in SCA-ATMI mode, the FML32 field name corresponding to each structure element is the same as the name of the structure element.

Note: Filenames that include spaces are not supported. The `-S` option may be specified multiple times.

Environment Variables

Following is a list of environment variables for `buildSCAclient`:

`TUXDIR`
Finds the SCA libraries and includes files to use when compiling the client applications.

`CC`
Indicates the compiler for all files with `.c` or `.cpp` file extensions. If not defined, the default C++ language compile command is invoked to compile all `.c` and `.cpp` files, based on the operating system.

`CFLAGS`
Indicates any arguments that are passed as part of the compiler command line for any files with `.c` or `.cpp` file extensions. If `CFLAGS` does not exist in the `buildscaclient` command environment, the command checks for the `CPPFLAGS` environment variable.

Note: Arguments passed by the `CFLAGS` environment variable take priority over the `CPPFLAGS` variable.

`CPPFLAGS`
Contains a set of arguments that are passed as part of the compiler command line for any files with `.c` or `.cpp` file extensions.

This is in addition to the command line option `"-I$(TUXDIR)/include"` for UNIX systems or the command line option `/I%TUXDIR%\include` for Windows systems, which is passed automatically by the `buildscaclient` command. If `CPPFLAGS` does not exist in the `buildscaclient` command environment, no compiler commands are added.

`LD_LIBRARY_PATH (UNIX systems)`
Indicates the directories that contain shared objects to be used by the compiler, in addition to the objects shared by the CORBA software. A colon (:) is used to separate the list of directories. Some UNIX systems require different environment variables:

- HP-UX systems use the SHLIB_PATH environment variable
- AIX systems use LIBPATH

`LIB (Windows systems)`
Indicates a list of directories that contain the library files. A semicolon (;) is used to separate the list of directories.

Portability

This utility can be used on any platform that supports the Oracle Tuxedo environment.

Example(s)

```
buildscaclient -s /myApplication/scaSrc/uBike -c uBike.client -f
uBikeClient.cpp -o uBikeClient
```

See Also

[-S],

buildscacomponent

Name

`buildscacomponent` - builds SCA components

Synopsis

```
buildscacomponent [-v] [-s scaroot] [-f firstfiles] [-l lastfiles] [-S
structurefiles] -c compositename[/componentname][,compositename,..]] [-y]
[-k] [-h]
```

Description

`buildscacomponent` is used to build individual SCA components from source code. The command reads SCDL source, finds the component(s) in the composite(s) file(s) specified, parses the corresponding .componentType file(s) and produces corresponding executable libraries, in the same location as the .componentType files.

The command automatically builds component implementations based on the contents of `<implementation.cpp>` elements as follows:

- The value of `/implementation.cpp/@header` is used to determine the name of the source and `componentType` files containing the implementation.

  For example, an element such as

  ```
  <implementation.cpp library="myLib" header="myComponentImpl.h"/>
  ```

  causes `buildscacomponent` to look for a myComponentImpl.cpp file and compile it, along with stubs generated from its interface located in a corresponding myComponentImpl.componentType file.

Composites may contain one or more components, and the `buildscacomponent` command may build one or more composites in one pass. If more than one component is built, the files specified using the `-f` and `-l` switches are included in each component. To build a single component, the

`-c composite/component` syntax should be used. This addresses the cases where individual components are made up of specific sets of source code or libraries.

All specified `.c` and `.cpp` files are compiled in one invocation of the compilation system for the operating system in use. Users may specify the compiler to be invoked by setting the `CC` environment variable to the name of the compiler. If the `CC` environment variable is not defined when `buildscacomponent` is invoked, the default C++ language compile command for the operating system in use is invoked to compile all `.c` and `.cpp` files.

Users may specify options to be passed to the compiler by setting the `CFLAGS` or the `CPPFLAGS` environment variable. `If CFLAGS` is not defined but `CPPFLAGS` is defined when `buildscacomponent` is invoked, the `CPPFLAGS` value is used.

Parameters and Options

`buildscacomponent`supports the following parameters and options:

`[-v]`
Specifies that `buildscacomponent` should work in verbose mode.

`[-s scaroot]`
Specifies the location of the SCA root, where the SCDL file(s) for the component(s) is (are) located, and where the source code of components is processed.

If not specified, the value of APPDIR is used.

`[-f firstfiles]`
Specifies a file to be included first in the compile and link phases of the `buildscacomponent` command. The specified file is included before the SCA libraries are included. There are two ways of specifying a file or files, as shown in the following table.

Table 2 File Specification Using [-f firstfiles]

Filename Specification	Definition
`-f firstfile`	One file is specified
`-f "file1.cpp file2.cpp file3.cpp …"`	Multiple files may be specified if their names are enclosed in quotation marks and are separated by white space.

Note: Filenames that include spaces are not supported.
The `-f` option may be specified multiple times.

[-l lastfiles]

Specifies a file to be included last in the compile and link phases of the `buildscacomponent` command. The specified file is included after the SCA libraries are included. There are two ways of specifying a file, as shown in the following table.

Table 3 File Specification Using [-l lastfiles]

Filename Specification	Definition
`-l lastfile`	One file is specified
`-l "file1.cpp file2.cpp file3.cpp …"`	Multiple files may be specified if their names are enclosed in quotation marks and are separated by white space.

Note: Filenames that include spaces are not supported.
The `-l` option may be specified multiple times.

`-c {composite[,composite]|composite/component}`

Specifies the name(s) of the composite(s) processed. The composite(s) is (are) searched in `APPDIR` or in the SCDL directory specified above with the -s switch. If it cannot be found, the component libraries are not built.

A list of composites may be specified, in which case all the components listed in the composites will be built. If any of the composites cannot be found or an error is detected (incorrect name, composite does not have any ATMI service binding), a warning message is displayed and the user is prompted to confirm whether the command should continue processing or abort.
If the composite/component notation is used, a single component contained in the specified composite is allowed. This notation covers the situation where specific source files specified with `-f` and `-l` need to be included in the build process of a component.

[-y]

Optionally forces processing of input files, automatically ignoring warnings, such as composites specified using the `-c` switch but not physically present from the root directory.

[-k]

Keeps the generated proxy and wrapper source. `buildscacomponent` generates proxy and wrapper code with data structures such as the method operation and parameter handling. This is normally compiled and then removed when the component is built. This option indicates that the source file should be kept (to see what the source filename is, use the `-v` option).

Note: The generated contents of this file may change from release to release. *Do Not* count on the data structures and interfaces exposed in this file. This option is provided to aid in debugging of build problems.

`[-S structurefiles]`

Specifies an SCA structure description file. The structure description file may be either a source file or a binary structure description file. If more than one file is specified, file names must be separated by white space and the entire list must be enclosed in quotation marks. The `-S` option may be specified multiple times on the same command line.

The use of structure description files is optional. If a structure description is not provided for a particular structure then the source code where the structure is defined is used describe the structure; in SCA-ATMI mode, the FML32 field name corresponding to each structure element is the same as the name of the structure element.

Note: Filenames that include spaces are not supported.
The `-S` option may be specified multiple times.

Environment Variables

`TUXDIR`

Finds the SCA libraries and include files to use when compiling the client applications.

`APPDIR`

Indicates the SCA application root location, where the top-level composite should reside.

`CC`

Indicates the compiler to use to compile all files with `.c` or `.cpp` file extensions. If not defined, the default C++ language compile command for the operating system in use will be invoked to compile all `.c` and `.cpp` files.

`CFLAGS`

Indicates any arguments that are passed as part of the compiler command line for any files with a `.c` or `.cpp` file extensions. If `CFLAGS` does not exist in the `buildscacomponent` command environment, the `buildscacomponent` command checks for the `CPPFLAGS` environment variable.

`CPPFLAGS`

Note: Arguments passed by the `CFLAGS` environment variable take priority over the CPPFLAGS variable.

Contains a set of arguments that are passed as part of the compiler command line for any files with a `.c` or `.cpp` file extensions.

This is in addition to the command line option `-I$(TUXDIR)/include` for UNIX systems or the command line option `/I%TUXDIR%\include` for Windows systems, which is

passed automatically by the `buildscacomponent` command. If `CPPFLAGS` does not exist in the `buildscacomponent` command environment, no compiler commands are added.

`LD_LIBRARY_PATH (UNIX systems)`
Indicates which directories contain shared objects to be used by the compiler, in addition to the objects shared by the CORBA software. A colon (:) is used to separate the list of directories. Some UNIX systems require different environment variables: for HP-UX systems, use the `SHLIB_PATH` environment variable; for AIX, use `LIBPATH`.

`LIB (Windows systems)`
Indicates a list of directories within which to find libraries. A semicolon (;) is used to separate the list of directories.

Portability

This utility can be used on any platform that supports the Oracle Tuxedo environment.

Example(s)

```
buildscacomponent -f utils.c -c searchInventory,updateItem
```

See also

`[-S],Filenames that include spaces are not supported. The -f option may be specified multiple times.`

buildscaserver

Name

`buildscaserver` – Builds an Oracle Tuxedo server containing SCA components.

Synopsis

```
-o servername -c composite[,composite][-v][-s scaroot]
[-w] [-r rmname][-y] [-k] [-t] [-S]
```

Description

`buildscaserver` is used to build a Tuxedo server that is used to route requests to SCA components previously built with the buildscacomponent command. The command generates a main routine that contains bootstrap routines to route Tuxedo or SCA requests to SCA components, and compiles it to form a server host application. The server host application is built using the default C++ compiler provided for the platform.

If the SCDL code contains references or services with <binding.ws> elements, these are automatically converted into WSDF files for use by the Web Services gateway (GWWS). All SCA servers built using buildscaserver are multi-threaded servers.

Parameters and Options

buildscaserver supports the following parameters and options:

-o servername
Required. Specifies the name of the server application generated by this command.

-c compositename[,compositename]
Required. Specifies the name of the composite hosted. The composite is searched for starting in APPDIR, or in the SCDL directory specified above with the -s switch. If it is not found, the server is not built. In case you specify a list of composites, then all the listed composites are hosted by the same Tuxedo server.

If any of the composites are not found or an error is detected such as incorrect name or composite does not have any atmi service binding, a warning message is displayed and the user is prompted to confirm whether the command should continue processing or abort.

[-v]
Specifies that buildscaserver should work in verbose mode.

[-s scaroot]
Specifies the target location of the SCA root, where the SCDL files for the components to be deployed are located.

This directory has a layout suitable to SCA composites and components. Each composite is represented as a directory and contains components in the run-time form, which includes SCDL code and libraries. At run time, the server application uses this directory to find the run-time SCA components.

If components are using the Web Services binding, the root location also receives a WSDF definition file.

[-w]
Specifies that the generated server will host Web services binding enabled components. By default, a server hosting ATMI binding enabled components is generated. Both types of servers can host the same actual components simultaneously (i.e. there can exist an ATMI and a WS servers, both hosting the same components previously built using the buildscacomponent command).

[-r rmname]

Specifies the resource manager associated with this server. The value `rmname` must appear in the resource manager table located in `$TUXDIR/udataobj/RM` on UNIX systems or `%TUXDIR%\udataobj\RM` on Windows systems. Each entry in this file is of the following form:

```
rmname:rmstructure_name:library_names
```

Using the `rmname` value, the entry in `$TUXDIR/udataobj/RM` or `%TUXDIR%\udataobj\RM` automatically includes the associated libraries for the resource manager and sets up the interface between the transaction manager and the resource manager. The value `TUXEDO/SQL` includes the libraries for the Oracle Tuxedo System/SQL resource manager. Other values can be specified once they are added to the resource manager table. If the `-r` option is not specified, the null resource manager is used, by default.

[-y]

Optionally forces processing of input files, automatically ignoring warnings.

[-k]

Keeps the server main stub. `buildscaserver` generates a main stub with data structures such as the service table and a `main()` function. This is normally compiled and then removed when the server is built. This option indicates that the source file should be retained.

Note: To see the source filename, use the `-v` option.

Caution: The generated contents of this file may change from release to release. It is advised that you *do not* depend on the data structures and interfaces exposed in this file. This option is provided to aid in debugging build problems.

[-t]

Not used in current release.

[-S]

Required when the server makes use of C structure input or output buffers and the `-w` option is specified.

Note: When the `-w` option is not specified, `buildscaserver` uses ATMI binding to determines if structures are used.The `-S` option is not required.

The `buildscaserver -S` option *does not* take an option argument.

Environment Variables

TUXDIR

Finds the SCA libraries and include files to use when compiling the client applications.

CC

Indicates the compiler to use to compile all files with `.c` or `.cpp` file extensions. If not defined, the default C++ language compile command is invoked to compile all `.c` and `.cpp` files.

CFLAGS

Indicates any arguments that are passed as part of the compiler command line for any files with a `.c` or `.cpp` file extensions. If `CFLAGS` does not exist in the `buildscaserver` command environment, the `buildscaserver` command checks for the `CPPFLAGS` environment variable.

Note: Arguments passed by the `CFLAGS` environment variable take priority over the `CPPFLAGS` variable.

CPPFLAGS

Contains a set of arguments that are passed as part of the compiler command line for any files with a `.c` or `.cpp` file extensions.

This is in addition to the command line option `"-I$(TUXDIR)/include"` for UNIX systems or the command line option `/I%TUXDIR%\include` for Windows systems, which is passed automatically by the `buildscaserver` command. If CPPFLAGS does not exist in the `buildscaserver` command environment, no compiler commands are added.

LD_LIBRARY_PATH (UNIX systems)

Indicates the directories that contain shared objects to be used by the compiler, in addition to the objects shared by the CORBA software. A colon `(:)` is used to separate the list of directories. Some UNIX systems require different environment variables:

- HP-UX systems use `SHLIB_PATH`
- AIX systems use `LIBPATH`

LIB (Windows only)

Indicates a list of directories where libraries are available. A semicolon `(;)` is used to separate the list of directories.

Portability

This utility can be used on any platform that supports the Oracle Tuxedo environment.

Example(s)

```
buildscaserver -c uBike.server -o uBikeSCASvr
```

Error Reporting

This command checks for the following inconsistencies in the SCDL code and reports error messages if:

- at least one syntax error in the SCDL files
- none of the composites contain any service with an ATMI binding
- at least one composite contains services defining ATMI bindings with incompatible `<remoteAccess>` elements. `<remoteAccess>` elements with a value of WorkStation are not supported by this command.
- `/binding.atmi/@requires` contains a legacy value and `/binding.atmi/map` elements contain values that conflict (for example, the same Tuxedo service name mapped to two or more different methods)

mkfldfromschema, mkfld32fromschema

The `mkfldfromschema` and `mkfld32fromschema` commands take an XML schema as input and produce a field table. This table can be processed by the `mkfldhdr` or `mkfldhdr32` command or is loaded by programs that need it. `mkfldfromschema` is used with 16-bit FML and `mlfld32fromschema` is used with 32-bit FML.

These commands have the following restrictions:

- Attributes cannot be specified
- Restrictions are ignored because their meaning is application-related

Name

`mkfldfromschema, mkfld32fromschema` – Generates field table from an XML schema

Synopsis

```
mkfldfromschema [{-i schema|-u schemaurl}] [-b basenumber] ]-o outputfile]
```

```
mkfld32fromschema [{-i schema|-u schemaurl}] [-b basenumber] ]-o
outputfile]
```

Description

These commands take an XML schema as input and generate a field table. The XML schema may be specified using either the `-i` option or the `-u` option. If neither option is specified, the schema is read from standard input.

Parameters and Options

`mkfldfromschema` and `mkfld32fromschema` supports the following options:

`-b basenumber`
Adds a `*base basenumber` line to the generated field table.

`-i schema`
Displays the name of a file containing an XML schema. The `-i` option cannot be specified in conjunction with the `-u` option.

`-u schemaurl`
A URL where the input schema is located. The URL must start with http://. The `-u` option cannot be specified in conjunction with the `-i` option.

`-o outputfile`
The name of a file that will contain the field table. If this option is not specified, the field table will be written to standard output.

Portability

This utility can be used on any platform that supports the Oracle Tuxedo server environment.

See Also

`SCAHOST (5)`

mkviewfromschema, mkview32fromschema

The `mkviewfromschema` and `mkview32fromschema` commands take an XML schema as input and produce a view file. This file can be processed by the `viewc` or `viewc32` command. `mkviewfromschema` is used with 16-bit views and `mkview32fromschema` is used with 32-bit views.

Name

`mkviewfromschema, mkview32fromschema` – Generates view table from an XML schema

Synopsis

```
mkviewfromschema [{-i schema|-u schemaurl}] [-o outputfile]
```

```
mkview32fromschema [{-i schema|-u schemaurl}] [-o outputfile]
```

Description

These commands take an XML schema as input and generate a view file. The XML schema may be specified using either the `-i` option or the `-u` option. If neither option is specified, the schema is read from standard input.

Options

`mkviewfromschema, mkview32fromschema` supports the following options:

`-i schema`
The name of a file containing an XML schema. The `-i` option cannot be specified in conjunction with the `-u` option.

`-u schemaurl`
A URL where the input schema is located. The URL must start with http://. The `-u` option cannot be specified in conjunction with the -i option.

`-o outputfile`
The name of a file that contains the output view file. If this option is not specified, the field table is written to standard output.

Portability

This utility can be used on any platform that supports the Oracle Tuxedo server environment.

See Also

`SCAHOST (5)`

SDO for C++ Specification V2.1published December, 2006

scaadmin

Name

`scaadmin` – SCA server management command interpreter

Synopsis

```
scaadmin [-v]
```

Description

Use the `scaadmin` command to dynamically redeploy SCA composites or display statistics and status of individual services. The `TUXCONFIG` environment variable is used to determine the location where the Tuxedo configuration file is loaded.

This command has no effect on servers that have not been built using the `buildscaserver(1)` command.

Options

The `scaadmin` command supports the following option:

`[-v ]`
: Causes `scaadmin` to display the Oracle Tuxedo version number, Tuxedo Patch Level. The command exits after print out.

  `scaadmin` must run on an active node.

Commands

`default [-m machine] [-g groupename] [-i srvid] ] [-s servername]`
: Sets the corresponding argument to be the default machine name, groupname, server id, or servername. If the default command is entered with no arguments, the current defaults are printed.

`reload [-m machine] [-g groupname] [-i srvid] ] [-s servername]`
: This command dynamically reloads the SCA components hosted on Tuxedo servers. The `-m`, `-g`, `-i` and `-s` options can be used to restrict the reloaded servers to any combination of machine, group, server id and server name.

`printstats [-m machine] [-g groupname] [-i srvid] [-s servername]`
: This command displays the list of services hosted by a server and the associated method, number of queries, and status (`active`, `idle`). The `-m`, `-g`, `-i` and `-s` options can be used to restrict the reloaded servers to any combination of machine, group, server id and server name.

`verbose (v) [{off | on}]`
: Produces output in verbose mode. If no option is given, the current setting is toggled and the new setting is printed. The initial setting is set to `off`.

`help (h) [{command | all}]`
: Prints help messages. If command is specified, the abbreviation, arguments, and description for that command are printed. all causes a description of all commands to be displayed. Omitting all arguments causes the syntax of all commands to be displayed.

`echo (e) [{off | on}]`
: Echoes input command lines when set to on. If no option is given, the current setting is toggled, and the new setting is printed. The initial setting is `off`.

`quit (q)`
: Terminates the session

Interoperability

The `scaadmin` command must run on an active node.

Environment Variables

`TUXCONFIG`

Used to determine the location where the Tuxedo configuration file is loaded.

Portability

This utility can be used on any platform that supports the Oracle Tuxedo environment.

Example(s)

The following command reloads all the composites hosted by the `uBikeServer` Tuxedo application server, which was built using the `buildscaserver(1)` command.

```
scaadmin
> reload -s uBikeServer
```

The following command displays statistics on the services offered by the `uBikerServer` Tuxedo application server, which was built using the `buildscaserver(1)` command.

```
scaadmin
> printstats -s uBikeServer
Service            Method             Status      Requests
                                                  Processed
SEARCHINVENTORY    searchInventory    A           37
```

SCAHOST (5)

Name

`SCAHOST` - Generic server for Python, Ruby, or PHP SCA components.

Synopsis

```
SCAHOST SRVGRP="identifier" SRVID="number"
  CLOPT="[-A] [servopts options]
         -- -w -c composite"
```

Description

SCAHOST is an Oracle Tuxedo system provided server that provides boot-strapping functionality for Python, Ruby, or PHP programs hosted as SCA components.

SCAHOST relies on Oracle Tuxedo Service Metadata Repository information, and therefore requires being defined after the TMMETADATA system process in the UBBCONFIG file.

Python, Ruby, and PHP components can be hosted by a single SCAHOST. It is preferable that the component(s) hosted contain only Python, Ruby, and PHP components (i.e., no C++ components).

Parameters and Options

-w

Specifies that an SCAHOST instance exposes Web services. By default, only ATMI binding services are exposed. Webs services and ATMI bindings cannot be hosted by the same SCAHOST server, if a composite has services exposed with both bindings, two SCAHOST instances must be configured in order to expose all ATMI and Web Services bindings.

-c composite

Specifies the name of the component that this server will host.

Portability

This command is available on any platform on which the Oracle Tuxedo server environment is supported.

Example(s)

Listing 1 provides an SCAHOST example.

Listing 1 SCAHOST Example

```
*SERVERS
SCAHOST         SRVGRP=GROUP1 SRVID=100
                CLOPT="-A -- -c Account"
SCAHOST         SRVGRP=GROUP2 SRVID=100
                CLOPT="-A -- -c Loan"
```

scapasswordtool

Name

`scapasswordtool` – Manages passwords for Tuxedo authentication in SCA clients.

Synopsis

```
scapasswordstore -i passwordidentifier -[a|d]
```

Description

This command manages the `password.store` file used by SCA components to refer to Tuxedo-based services.

Passwords are prompted and encrypted. The encrypted version is stored in this file, associated with a clear-text identifier. This command is also used to delete identifier/password pairs from the file.

The password is limited to 40 characters. If standard input is not a terminal, that is, if the user cannot be prompted for a password (as with a Here file, for example), then the `APP_PW` environment variable is accessed to set the password. If the `APP_PW` environment variable is not set and standard input is not a terminal, then `scapasswordtool` prints an error message and exits.

A `password.store` file is created in the current directory if it does not previously exist.

Parameters and Options

`-i passwordidentifier`
Required. The identifier specified in the `<binding>` element. SCA components search the password for this element.

`-[a|d]`
The `-a` option adds an identifier/password pair, whereas the `-d` option deletes it. An error message is printed out and the command processing is aborted in one of the following situations:

- If `-a` is used to add an already existing identifier
- If `-d` is used to delete a non-existing identifier

Portability

This utility can be used on any platform that supports the Oracle Tuxedo environment.

See Also

setSCAPasswordCallback(3c)

scastructc32, scastructc(1)

Name

scastructc32, scastructc - Structure description compiler for Oracle Tuxedo.

Synopsis

```
scastructc32 [-n] [-d viewdir] structfile [structfile . . . ]
scastructc [-n] [-d viewdir] structfile [structfile . . . ]
```

Description

scastructc32 and scastructc are a Oracle Tuxedo SCA structure description compiler programs. These commands take a source structure description file and produces:

- A binary file, which is interpreted at run time to effect the actual mapping of data between FML buffers and C++ structures.
- One or more header files.

Note: COBOL is not supported in the SCA environment, therefore scastructc32 and scastructc do not have options to generate COBOL copyfiles.

SCA structure description files are identical to Oracle Tuxedo viewfiles, with the exception that SCA structure description files allow the following extensions:

- Nested structures are supported. A nested structure may be specified by using the struct keyword in column 1. When this keyword is used, the "cname" value in column 2 must be the name of a previously defined view that describes a nested structure.

  The value in column 3 will be interpreted as the name of the element for the inner structure within the outer structure. If the value in column 3 is "-", then the name of the inner structure element will be the same as the name of the inner structure.

  As with other types, the value in column 4 can be used to specify a count of the number of times the inner structure is included in the outer structure. The "flag" and "size" values in columns 5 and 6 are not used for struct elements.

scastructc32 is used for 32-bit FML. It uses the FIELDTBLS32 and FLDTBLDIR32 environment variables. scastructc is used for 16-bit FML. It uses the FIELDTBLS and FLDTBLDIR environment variables.

If none of the SCA structure file extensions are used, then binary files produced by `scastructc32` are compatible with binary files produced by `viewc32` and binary files produced by `scastructc` are compatible with binary files produced by `viewc`.

The structfile is a file containing source structure descriptions. More than one structfile can be specified on the `scastructc32` or `scastructc` command line as long as the same VIEW name is not used in more than one structfile.

By default, all views in the structfile are compiled and two or more files are created: a view object file (with a `.V` suffix) and a C header file (with a `.h` suffix). The name of the object file is structfile.V in the current directory unless an alternate directory is specified through the `-d` option. C header files are created in the current directory.

Note: `scastructc32` and `scastructc` generate a binary file with suffix `.V` on Unix and suffix `.VV` on Windows.

At `scastructc32` or `scastructc` compile time, the compiler matches each field id and field name specified in the viewfile with information obtained from the field table file, and stores mapping information in an object file for later use. Therefore, it is essential to set and export the environment variables FIELDTBLS and FLDTBLDIR to point to the related field table file. For more information, see Programming an Oracle Tuxedo ATMI Application Using FML and Programming an Oracle Tuxedo ATMI Application Using C.

If the `scastructc32` or `scastructc` compiler cannot match a field name with its field id because either the environment variables are not set properly or the field table file does not contain the field name, a warning message, Field not found, is displayed.

With the `-n` option, it is possible to create a view description file for a C structure that is not mapped to an FML buffer. Programming an Oracle Tuxedo ATMI Application Using C discusses how to create and use such an independent view description file.

Parameters and Options

The following options are interpreted by `scastructc32` and `scastructc`:

`-n`
Used when compiling a structure description file for a C structure that does not map to an FML buffer. It informs the structure compiler not to look for FML information.

`-d viewdir`
Used to specify that the structure object file is to be created in a directory other than the current directory.

Note: On Windows, the following additional options are recognized:

-c { m | b }

Specifies the C compilation system to be used. The supported value for this option is m for the Microsoft C compiler. The Microsoft C compiler is the default for this option. The -c option is supported for Windows only.

-1 filename

Specifies that pass 1 should be run, and the resulting batch file called filename.bat should be created. After this file is created, it, should be executed before running

pass 2. Using pass 1 and pass 2 increases the size of the views that can be compiled. The -1 option is supported for Windows only.

-2 filename

Specifies that pass 2 should be run to complete processing, using the output from pass 1. The -2 option is supported for Windows only.

Portability

The output view file is a binary file that is machine and compiler-dependent. It is not possible to generate a view on one machine with a specific compiler and use that view file on another machine type or with a compiler that generates structure offsets differently (for example, with different padding or packing).

See Also

scastructdis32, scastructdis

Programming an Oracle Tuxedo ATMI Application Using FML

Introduction to FML Functions in Oracle Tuxedo ATMI FML Function Reference

Programming an Oracle Tuxedo ATMI Application Using C

scastructdis32, scastructdis

Name

`scastructdis32`, `scastructdis` - Disassembler for binary structure files and viewfiles.

Synopsis

```
scastructdis32 [-E envlabel] viewobjfile [viewobjfile...]
scastructdis [-E envlabel] viewobjfile [viewobjfile...]
```

Description

`scastructdis32` disassembles a view object file produced by `scastructc32` or `viewc32` and displays view information in viewfile format. In addition, it displays the offsets of structure members in the associated structure.

One or more viewobjfiles (with a `.V` suffix) can be specified on the command line. By default, the viewobjfile in the current directory is disassembled. If this is not found, an error message is displayed.

Because the information in the viewobjfile was obtained from a match of each field id and field name in the viewfile with information in the field table file, it is important to set and export the environment variables FIELDTBLS32 and FLDTBLDIR32.

The `scastructdis32` output looks the same as the original structure description(s), and is mainly used to verify the accuracy of the compiled object structure descriptions.

`scastructdis` is used for files originally compiled with scastructc or viewc. It uses the FIELDTBLS and FLDTBLDIR environment variables instead of FIELDTBLS32 and FLDTBLDIR32.

See Also

`pass 2. Using pass 1 and pass 2 increases the size of the views that can be compiled. The -1 option is supported for Windows only.`

Programming an Oracle Tuxedo ATMI Application Using FML

scatuxgen(1)

Name

`scatuxgen` - Generates Tuxedo Service Metadata Repository interface information from an SCA interface.

Synopsis

```
scatuxgen (-c <composite file name> | -i <interface file name> [-I <inbuf>]
[-O <outbuf>])-s <service name> [-t <string-type>][-w [-n <namespace> -a
<network address>]] [-v]
```

Description

Generates Tuxedo Service Metadata Repository interface information based on SCA abstract class definitions. Service Metadata generation is performed by parsing a composite file (in

SCDL) which allows locating the interface referenced by the <service name> value, or directly by specifying the interface to process at the command line.

The interface is an SCA-compliant abstract class definition contained in a C++ header file. Parsing the composite file allows you to take advantage of binding.atmi details (for example, buffer types and xsd schemas) when available.

When binding.atmi information is not available, scatuxgen can directly process a C++ interface directly by giving the name of the header file containing it as an argument to the command line.

The generated file name is composed using the service name, input using the command-line option, and the .mif file, and possibly the.wsdf extension.

Options

-c composite file name
Specifies the pathname of the composite file to be processed. This path is relative to where the command is run.

-i interface file name
Specifies the name of the interface file to be processed. This path is relative to where the command is run.

-I inbuf
Specifies the type of input Tuxedo buffer to generate in the service metadata entry. This option is only valid when used in conjunction with the -i and -w options . Acceptable values are STRING, CARRAY, X_OCTET, VIEW/<viewname>, X_C_TYPE /<viewname>, X_COMMON/<viewname>, VIEW32/<viewname>, FML, FML32, MBSTRING and XML.

-O outbuf
Specifies the type of output Tuxedo buffer to generate in the service metadata entry. This option is only valid when used in conjunction with option -i. Acceptable values are STRING, CARRAY, X_OCTET, VIEW/<viewname>, X_C_TYPE/<viewname>, X_COMMON/<viewname>, VIEW32/<viewname>, FML, FML32, MBSTRING and XML.

-E outbuf
Specifies the type of error Tuxedo buffer to generate in the service metadata entry. This option is only valid when used in conjunction with option -i. Acceptable values are STRING, CARRAY, X_OCTET, VIEW/<viewname>, X_C_TYPE/<viewname>, X_COMMON/<viewname>, VIEW32/<viewname>, FML, FML32, MBSTRING and XML.

-s service name
Specifies the name of the service to be generated when using an interface file. It also specifies the base of the output file(s).

-t string-type
: Specifies that `scatuxgen` should map `xsd:string` types in XML schemas to Tuxedo `mbstring (FLD_MBSTRING)`.

-w
: Specifies `scatuxgen` produces a WSDF document.

-n
: When producing a WSDF document, can be used to indicate the `Definition/@wsdlNameSpace` attribute value. If not specified, the `Definition/@wsdlNamespace` attribute contains the `'##NAMESPACE##'` placeholder.

-a
: When producing a WSDF document, can be used to indicate the `Definition/WSBinding/AccessingPoints/Endpoint/@address` attribute value. If not specified, the `Definition/WSBinding/AccessingPoints/Endpoint/@address` attribute will contain the `'##ADDRESS##'` placeholder.

-v
: Specifies `scatuxgen` in verbose mode.

Portability

This utility can be used on any platform that supports the Oracle Tuxedo environment.

Example

The following example results in a `TOUPPER.mif` file created in the same directory where `scatuxgen` is invoked:

```
$ scatuxgen -c simpapp.composite -s TOUPPER
```

See Also

`Flat File view. If this option is specified, then all the generated files are put in the target root directory. The default is Tree File view.`

setSCAPasswordCallback(3c)

Name

`setSCAPasswordCallback()` – Sets the callback for retrieving a password associated with an identifier in a `<binding.atmi>` element.

Synopsis

```
#include <tuxsca.h>
void setSCAPasswordCallback(char * (_TMDLLENTRY *)(*disp) (char
*identifier))
```

Description

`setSCAPasswordCallback()` allows an SCA component to identify the callback that returns the clear-text password that is passed to the appropriate authentication code.

The function pointer passed on the call to `setSCAPasswordCallback()` must conform to the specified parameter definition. The `_TMDLLENTRY` macro is required for Windows-based operating systems to obtain the proper calling conventions between the Tuxedo libraries and your code. On UNIX systems, the `_TMDLLENTRY` macro is not required because it expands to the null string.

The identifier points to the password identifier passed to the callback function. The callback function then returns a `char *` that points to the actual clear-text password.

Return Values

The `setSCAPasswordCallback()` function does not return any data.

Errors

On failure, `setSCAPasswordCallback()` sets `tperrno` to one of the following values:

`[TPEPROTO]`
`setSCAPasswordCallback()` has been called in an improper context.

`[TPESYSTEM]`
An Oracle Tuxedo system error has occurred. The exact nature of the error is written to a log file.

`[TPEOS]`
An operating system error has occurred.

See Also

`scapasswordtool`

tuxscagen(1)

Name

tuxscagen – Generates SCA, SCDL, and server-side interface files for Tuxedo services.

Synopsis

```
tuxscagen [-s <target-root-directory>] [-d <service-name>][-C
<TUXEDO_cltname>][-u <TUXEDO_username>][(-S | -j <java_package_name>)][-o
<output_SCDL_filename>][-i <output_interface_filename>[-m
<max-intf-arguments>][-y] [-v] [-F] [-c] [-h][-g<i|a|s>]
[-trepository=<filename> | -tinfile=<metarepos.infile> | -tmetadata]
```

Description

tuxscagen is used to generate interface and SCDL files. The interface files are used for developing the SCA component using ATMI binding, or wrap existing Tuxedo services in an SCA component. The SCDL files are assembly artifacts that help SCA run time to locate the module and services.

Parameters and Options

tuxscagen supports the following options:

-s target-root-directory
: Specifies the location of the root directory where the generated SCDL and interface files are located. The directory must exist and with write access permission; if it does not exist, the tool issues an error message and fails.

-d<service-name>
: Specifies the name of Tuxedo service in the Tuxedo Metadata Repository. If this option is not specified, all services in the repository or in the input file are selected.

  Abbreviation: there is no abbreviation for this option

-C <TUXEDO_cltname>
: The Tuxedo client name. Use cltname as the client name when joining the Tuxedo application.

-u <TUXEDO_username>
: The Tuxedo user name. Use username as the user name when joining the Tuxedo application. This is required when Tuxedo security level is higher than APP_PW and input method is to retrieve Tuxedo Service Metadata from TUXEDO.TMMETAREPOS Service.

-j <java_package_name>

This option generates JAVA interface files. By default, `tuxscagen` generates C++ header files. If `-g` is not specified but if `-j <java_package_name>` is specified then `-ga` is assumed. However, if `-g` sub-option `i` or `s` is specified, a warning message is displayed.

-o <output_SCDL_filename>

This option specifies the output SCDL filenames for single composite and single `componentType` file. If this option is not specified, then by default, one composite and one `componentType` are generated for each Tuxedo service. However, if this option is specified with the output filename, only one composite and one `componentType` file is generated for all the matching Tuxedo services. If the specified `<output_SCDL_filename>` already exists, an interactive prompt is displayed and requires user input (unless `-y` is specified). If this option is specified, `-F` is automatically implied.

-i <output_interface_filename>

This option specifies the output interface filenames for single abstract class header file and single class implementation header file. If this option is not specified, then by default, it generates one abstract interface class header file and one implementation class header file.

However, if this option is specified with output interface filename then only one abstract class header file and one implementation header file is generated for all matching Tuxedo services. If the specified `<output_interface_filename>` already exists, an interactive prompt is displayed and requires user input (unless `-y` is specified).

If this option is specified, `-F` is automatically implied.

-m <max-intf-arguments>

This option specifies the maximum number of arguments allowed in the interface method. If the number of arguments exceeds the specified threshold then a complex data type is used as the input argument for the interface method. The complex data type used is `commonj::sdo::DataObjectPtr`.

If **`-m`** is not specified, the default threshold is 10.

If `0` specified, it will always generate using `commonj::sdo::DataObjectPtr`.

If `-ga` is not specified, this option is ignored.

-y

This option suppresses `Really overwrite files:<filename> [y, q] ?` so that the script can run without user input. This question appears if either or both `-o` and `-i` are specified. If both these options are not specified, by default existing files are replaced.

-v

This option turns on the verbose mode.

-h

If this option is specified, online help is printed and all other options are ignored.

-F

`Flat File` view. If this option is specified, then all the generated files are put in the target root directory. The default is `Tree File` view.

-c

Generates client-side SCDL. By default `tuxscagen` generates server-side SCDL, specifying this option changes it to generate client-side SCDL.

-g a|i|s

This option is used to specify the files to generate. The sub-options can be combined. The a sub-option is used to generate abstract base class header files. The sub-option `i` is to generate implementation class header files. Sub-option `s` is used to generate SCDL files. To generate both header files, specify `-gai`. To generate all files, specify `-gais`.

If not specified, `-gais` is assumed.

[-trepository=<filename> | -tinfile=<metarepos.infile> | -tmetadata]

This option specifies the processing type.

If `-trepository=<filename>` is specified, `tuxscagen` retrieves service parameter information from the Service Metadata repository file `<filename>`. If `-tinfile=<metarepos.infile>` is specified, then `tuxscagen` retrieves service parameter information from `<metarepos.infile>`, where the `<metarepos.infile>` syntax is suitable for input to `tmloadrepos`. If `-tmetadata` is specified, `tuxscagen` retrieves service parameter information from the Tuxedo TMMETADATA server.

At most, one `-t` option can be specified; the default is `-tmetadata`.

[-S]

Specifies `tuxscagen` generate a structures for any function parameter or return value that would otherwise have been passed using `DataObjectPtr`.

When the `-S` option is used, a structure definition is generated as part of the generated abstract class header file `${TUXSERVICE}.h`. `tuxscagen -S` also generates a Tuxedo view file `${TUXSERVICE}.v` describing the generated view(s).

If `tuxscagen` input does not specify a maximum number of occurrences for a field, then `tuxscagen -S` generates 1 occurrence for that field. If `tuxscagen` input specifies an unlimited number of occurrences for a field, then `tuxscagen -S` generates an error.

If `tuxscagen` input does not specify a maximum length for a string, carray, or mbstring parameter, then `tuxscagen` generates a maximum length of 80 characters plus trailing NULL for that parameter and outputs a warning message to check if this is sufficient.

Note: The use of an 80 character default is different from viewc. An unspecified length in viewc causes a length of 1 character plus trailing null to be generated, which is insufficient for most applications.

The `tuxscagen -S` option will not change the underlying Tuxedo transport type specified for the `<inputBufferType>`, `<outputBufferType>`, and `<errorBufferType>` elements in the generated composite file. When data is passed via `DataObjectPtr` or via a structure, this will normally be FML32.

Note: Structures are not supported for the SCA Java interface. Using `tuxscagen` with both the `-j` and `-S` options results in an error.

Portability

This utility can be used on any platform that supports the Oracle Tuxedo environment.

Example

The following command is used to generate SCDL, interface, and implementation header files from a Tuxedo Metadata Repository file named `myrepository` in the current working directory. The number of interface method input arguments is limited to 8. If the limit is exceeded, the XSD schema file is still generated.

```
tuxscagen -s /home/tux/sca -Dname=TRANSFER -gais -m 8
-trepository=myrepository
```

See Also

`scatuxgen(1)`,`tmloadrepos(1)`, `tmunloadrepos(1)`

Managing The Tuxedo Service Metadata Repository in *Setting up an Oracle Tuxedo Application*

CHAPTER 4

Oracle Tuxedo SCA Sample Applications

Three bundled SCA sample applications demonstrate how to develop applications using the SCA programming model, as well as configure the Oracle Tuxedo SCA container.

- Basic Sample: simpappp
- Advanced Sample: uBike
- SCA Sample Using Web Services: calc client

Basic Sample: simpappp

The Basic Sample demonstrates how to write a simple SCA application made up of a client program calling an SCA component via the Tuxedo infrastructure. It contains all the needed files to configure and deploy an SCA component hosted on a Tuxedo server, as well as the needed files to compile and configure an SCA client program to invoke the component. It represents an end-to-end application of SCA technology.

Other Uses

The Basic Sample can invoke a regular Tuxedo ATMI service, or the SCA component may be invoked by a regular ATMI client. Also, the same SCA code can run without using `<binding.atmi>` in its SCDL configuration, demonstrating the flexibility of the setup.

Advanced Sample: uBike

The Advanced Sample contains all the needed files to configure and deploy an SCA component hosted on a Tuxedo server, as well as the needed files to compile and configure an SCA client program to invoke the component. Data exchanged between client and component is of type `commonj::sdo::DataObject`, with the underlying transport being Tuxedo ATMI using STRING and FML32 Tuxedo buffers. It represents an end-to-end application of SCA and SDO technology.

Other Uses

The Advanced Sample can invoke a regular Tuxedo ATMI service, or the SCA component can be invoked by a regular ATMI client. Also, the same SCA code may run without using `<binding.atmi>` in its SCDL configuration, demonstrating the flexibility of the setup.

SCA Sample Using Web Services: calc client

The Web Services Sample demonstrates how to develop an SCA client program that invokes an external Web service. It contains all the needed files to configure Oracle Tuxedo as needed by the runtime SCA configuration.

APPENDIX A

Oracle Tuxedo SCA ATMI Binding Reference

The following sections provide SCA ATMI Binding reference information:

- SCA ATMI Binding Schema
- SCA ATMI Binding Attributes Description

SCA ATMI Binding Schema

Listing A-1 shows how the ATMI binding element (`<binding.atmi>`) is defined. This is a pseudoschema that depicts how the grammar is used and what parameters are legal.

Notes: The parameters "`transactionalintent legacyintent`" are not literal values. `transactionalintent` can be substituted with "`suspendsTransaction`" or "`propagatesTransaction`" or omitted. "`legacyintent`" can be substituted with "legacy" or omitted.

Parameters with a `?` may be specified 0 or 1 times, and parameters with * may be specified 0 or more times.

When using the `<binding.atmi>`element, the total length of `/reference/@name (or/service/@name)` and method name must be equal to or less than the maximum length of a Tuxedo service name (this varies depending on the Tuxedo release). To overcome this limitation, see `</binding.atmi/map>`.

Listing A-1 SCA ATMI Binding Pseudoschema

```
<binding.atmi requires="transactionalintent legacyintent"?>
       <tuxconfig>...</tuxconfig>?

       <map target="name">...</map>*
       <serviceType target="name">...</serviceType>*
       <inputBufferType target="name">...</inputBufferType>*
       <outputBufferType target="name">...</outputBufferType>*
       <errorBufferType target="name">...</errorBufferType>*
       <workStationParameters>?
              <networkAddress>...</networkAddress>?
              <secPrincipalName>...</secPrincipalName>?
              <secPrincipalLocation>...</secPrincipalLocation>?
              <secPrincipalPassId>...</secPrincipalPassId>?
              <encryptBits>...</encryptBits>?
       </workStationParameters>
       <authentication>?
              <userName>...</userName>?
              <clientName>...</clientName>?
              <groupName>...</groupName>?
              <passwordIdentifier>...</passwordIdentifier>?
              <userPasswordIdentifier>...
                                          </userPasswordIdentifier>?
       </authentication>
       <fieldTablesLocation>...</fieldTablesLocation>?
       <fieldTables>...</fieldTables>?
       <fieldTablesLocation32>...</fieldTablesLocation32>?
       <fieldTables32>...</fieldTables32>?
       <viewFilesLocation>...</viewFilesLocation>?
       <viewFiles>...</viewFiles>?
       <viewFilesLocation32>...</viewFilesLocation32>?
       <viewFiles32>...</viewFiles32>?
       <remoteAccess>...</remoteAccess>?
       <transaction timeout="xsd:long"/>?
</binding.atmi>
```

SCA ATMI Binding Attributes Description

The `<binding.atmi>` element supports the following attributes

- </binding.atmi/@requires>
- </binding.atmi/tuxconfig>
- </binding.atmi/map>
- </binding.atmi/serviceType>
- </binding.atmi/inputBufferType>, </binding.atmi/outputBufferType>, </binding.atmi/errorBufferType>
- </binding.atmi/workStationParameters>
- </binding.atmi/authentication>
- </binding.atmi/fieldTablesLocation>
- </binding.atmi/fieldTablesLocation32>
- </binding.atmi/fieldTables>
- </binding.atmi/fieldTables32>
- </binding.atmi/viewFilesLocation>
- </binding.atmi/viewFilesLocation32>
- </binding.atmi/viewFiles>
- </binding.atmi/viewFiles32>
- </binding.atmi/remoteAccess>
- </binding.atmi/transaction/@timeout>

</binding.atmi/@requires>

- When this attribute contains the `legacy` value, it is used to perform interoperability with existing Tuxedo services. When not specified, communications are assumed to have SCA to SCA semantics where the actual Tuxedo service name is constructed from `/service/@name` or `/reference/@name` and actual method name (see Listing A-1), unless a `/binding.atmi/map` element is defined. When this attribute encounters a legacy

value, and no /binding.atmi/map element is defined for the method being called, it has the following run-time behavior:

  – In a <reference> element: the value specified in the /reference/@name is used to perform the Tuxedo call, with semantics used according to the interface method.

  – In a <service> element: the Tuxedo service specified in the /binding.atmi/map element is advertised, and mapped to the method specified in the /binding.atmi/map/@target attribute.

- When this attribute contains a transaction value, it specifies the transactional behavior that the binding extension follows when this binding is used. Possible values are as follows:

  – not specified (no value) - all transactional behavior is controlled by the Tuxedo configuration. If the Tuxedo configuration supports transactions, then one may be propagated if it exists. If the Tuxedo configuration does not support transactions and one exists then an error will occur. However, a transaction cannot start if one does not already exist.

  – suspendsTransaction - transaction context is propagated to the called service. For a <service> element when a transaction is present, it is automatically suspended before invoking the application code. It resumes afterwards, regardless of the outcome of the invocation. For a <reference> element, it is equivalent to making a tpcall() with the TPNOTRAN flag.

  – propagatesTransaction - only applicable to <reference> elements. It is ignored for <service> elements. This value starts a new transaction if one does not already exist, otherwise it participates in the existing transaction.

    Such behavior can be obtained in a component or composite <service> element by configuring AUTOTRAN in the UBBCONFIG file. An error is generated if a Tuxedo server hosts the SCA component implementation and it is not configured in a transactional group in the UBBCONFIG file.

</binding.atmi/tuxconfig>

Used in <reference> elements when /binding.atmi/workstationParameters is not set, and for client-only processes. It indicates the Tuxedo application that the process should join. One process can join multiple applications, or switch applications without having to restart.

If not set, the TUXCONFIG environment variable is used. If not set, but one is required, the process exits and returns an error.

</binding.atmi/map>

For <reference> elements, </binding.atmi/map>provides the Tuxedo service name that should be used when performing the invocation to the corresponding /binding.atmi/map/@target value, this value being the name of the method being called.

For <service> elements, </binding.atmi/map> provides the Tuxedo service name that should be advertised for the corresponding /binding.atmi/map/@target value.

The /binding.atmi/map/@target value *must* match the method name of the corresponding service interface.

If a /binding.atmi/map element is present, it takes precedence over any other form of service/method to Tuxedo service name mapping. See </binding.atmi/@requires> attribute.

</binding.atmi/serviceType>

Optional element that specifies the type of call being handled. The accepted values are:

- Oneway - the call will not expect a response.
- RequestResponse - regular call paradigm, default value.

</binding.atmi/inputBufferType>, </binding.atmi/outputBufferType>, </binding.atmi/errorBufferType>

Optional elements that specify the type of buffer that the processes exchange. The inputBufferType element is used by the binding extension to determine or check the type of the request.

The outputBufferType element is used by the binding extension to determine or check the type of the reply.

The errorBufferType element is used to determine the type of buffer specified in the data portion of the Exception thrown received by a client or thrown by a server.

Table A-1 lists supported values and corresponding Tuxedo buffer types. An incorrect value or syntax is detected at run time and causes the call to fail. If not specified, the default value used is STRING.

Table A-1 SCA Supported Tuxedo Buffer Types

/binding.atmi/bufferType value	Tuxedo buffer type	Note
STRING	STRING	
CARRAY	CARRAY	
X_OCTET	X_OCTET	
VIEW	VIEW	Format is VIEW/<subtype>
X_C_TYPE	X_C_TYPE	Format is X_C_TYPE/<subtype>
X_COMMON	X_COMMON	Format is: X_COMMON/<subtype>
VIEW32	VIEW32	Format is VIEW32/<subtype>
XML	XML	
FML	FML	Format is: FML/<subtype>,<subtype> is optional The <subtype> value allows to specify the SDO type to use for that message (request or response) when it is described in an XML schema **Note:** FML32 <subtype> is not available for JATMI binding.

Table A-1 SCA Supported Tuxedo Buffer Types

/binding.atmi/bufferType value	Tuxedo buffer type	Note
FML32	FML32	Format is: FML32/<subtype>, <subtype> is optional The <subtype> value allows to specify the SDO type to use for that message (request or response) when it is described in an XML schema **Note:** FML32 <subtype> is not available for JATMI binding.
MBSTRING	MBSTRING	

</binding.atmi/workStationParameters>

An optional element that specifies parameters specific to the Tuxedo WorksStation protocol. Only used in references.

- /binding.atmi/workStationParameters/networkAddress

  The address of the workstation listener to which this application will connect. Any address format accepted by the Tuxedo workstation software is allowed. The most common address format is:

  //<hostname or IP address>:<port>.

  For more information, see the SALT Programming Guide

More than one address can be specified (if required), by specifying a comma-separated list of pathnames for WSNADDR Addresses are tried in order until a connection is established. Any member of an address list can be specified as a parenthesized grouping of pipe-separated network addresses. For example:

```
<networkAddress>
 (//m1.acme.com:3050|//m2.acme.com:3050),//m3.acme.com:3050
</networkAddress>
```

Tuxedo randomly selects one of the parenthesized addresses. This strategy distributes the load randomly across a set of listener processes. Addresses are tried in order until a connection is established.

On versions of Tuxedo that support ipv6, the corresponding addressing format will also be supported, following the same format as used in `WSNADDR` for Tuxedo /WS clients.

- `secPrincipalName`, `secPrincipalLocation`, `secPrincipalPassId`

  These parameters specify the necessary parameters when an SSL connection is required by a workstation client. The password is stored in a separate file and accessed using a callback mechanism. The default callback uses the `password.store` file maintained using the `scapasswordtool` command. For more information, see the SALT Programming Guide

- `encryptBits`

  Specifies the encryption strength that this client connection will attempt to negotiate. The format is `<minencryptbits>/<maxencprytbits>` (for example, 128/128), those values being numerical. Invalid values will result in a configuration exception being thrown. Values can be 0 (if no encryption is used), or 40, 56, 128, or 256 (if the number specified is the number of significant bits in the encryption key).

</binding.atmi/authentication>

Specifies the security parameters used in reference-type calls to establish a connection with the Tuxedo application. The following values respectively correspond to the TPINFO structure elements `usrname`, `cltname`, `grpname` and `passwd` (for more information, see `tpinit(3c)` in the Oracle Tuxedo ATMI C Function Reference guide):

- `/binding.atmi/authentication/userName`
- `/binding.atmi/authentication/clientName`
- `/binding.atmi/authentication/groupName`
- `/binding.atmi/authentication/passwordIdentifier-(application password)`
- `/binding.atmi/authentication/userPasswordIdentifier-(user password in per-user authentication)`

  Passwords are not stored in clear text, but are looked up using an identifier. A callback function may be used to retrieve passwords. For more information, see `setSCAPasswordCallback()`in the Oracle Tuxedo Reference Guide.

  By default, passwords are maintained encrypted in a passwords store file located in the same directory as the composite file that contains the `/reference/binding.atmi/authentication/passwordIdentifier` or

`/reference/binding.atmi/authentication/userPasswordIdentifier` element. This identifier is read as necessary to perform authentication.

For more information, see `scapasswordtool` and setSCAPasswordCallback(3c) in the Oracle Tuxedo Reference Guide.

Note: This information should be handled with policy sets and intents when the SCA Kernel supports it.

</binding.atmi/fieldTablesLocation>

Optional element that specifies a directory in the local file system where field tables should be searched. If a relative path is specified, files are searched in that location relative to `$APPDIR`, otherwise the location is assumed to be absolute.

</binding.atmi/fieldTablesLocation32>

Same as `fieldTablesLocation`, but for FML32 buffers.

</binding.atmi/fieldTables>

Optional element that specifies the FML field tables available. Field tables are searched in the location specified by the `/binding.atmi/fieldTablesLocation element`.

If the `/binding.atmi/bufferType` value is FML and this element is not specified or invalid (that is, the tables indicated cannot be found or are not field tables), an error is displayed at initialization time for client processes, or boot time for server processes.

</binding.atmi/fieldTables32>

Same as `fieldTables`, but for FML32 buffers.

</binding.atmi/viewFilesLocation>

Optional element that specifies a directory in the local file system where view tables should be searched. If a relative path is specified, files are searched in that location relative to `$APPDIR`, otherwise the location is assumed to be relative.

</binding.atmi/viewFilesLocation32>

Same as `viewTablesLocation`, but for VIEW32 buffers.

</binding.atmi/viewFiles>

Optional element that specifies the VIEW files to be used by the affected component(s). If the `/binding.atmi/bufferType` value is VIEW and this element is not specified or invalid (that is, the files indicated cannot be found, or are not view files), an error is displayed at run time for client processes, or boot time for server processes.

</binding.atmi/viewFiles32>

Same as ViewFiles but for VIEW32 buffers.

Note: FML/FML32 and VIEW/VIEW32 parameters are optional and may be omitted, in which case the corresponding Tuxedo environment variables are required (FLDTBLDIR/32, FLDTBLS/32, VIEWDIR/32 and VIEWFILES/32). If neither are used, an error message is printed at run time when attempting to use a fielded buffer. If both are set, the parameters contained in the SCDL code take precedence.

</binding.atmi/remoteAccess>

Optional element that specifies the communication protocol with one of the values below. The default is Native.

- Native - indicates that components use standard Tuxedo native communications (IPC queues)
- WorkStation - indicates that components use the Tuxedo /WS communication protocol.

  If set to this value, the binding extension checks that the `/binding.atmi/workStationParameters` element is also populated and valid; if not, it reports a run-time error message.

</binding.atmi/transaction/@timeout>

Specifies the amount of time, in seconds, a transaction can execute before timing out. This attribute affects components or clients that effectively start a global transaction. It is mandatory for `<reference>` components and ignored if set on `<service>` components. Additionally, the value is ignored on components for which the transaction has already been started. If a transaction needs to be started and this attribute is not present (for example, "`requires=propagatesTransaction`" is set), a configuration error occurs.

-

APPENDIX B

Appendix B: Oracle Tuxedo SCA Schemas

This section contains the following information:

- ATMI and JTMI Binding Schema For C/C++
- Web Service Binding Schema

ATMI and JTMI Binding Schema For C/C++

Listing B-1 shows an ATMI and JTMI C/C++ binding schema.

Listing B-1 ATMI and JTMI Binding Schema For C/C++

```
<?xml version="1.0" encoding="UTF-8"?>

<schema xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
         xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
         elementFormDefault="qualified">

   <element name="binding.atmi" type="sca:AtmiBinding"
         substitutionGroup="sca:binding"/>

   <complexType name="AtmiBinding">
```

```
<complexContent>
   <extension base="sca:Binding">
      <sequence>
         <element name="tuxconfig" type="string"
            minOccurs="0"/>
         <element name="map" type="sca:TargetMapType" minOccurs="0"
            maxOccurs="unbounded"/>
         <element name="serviceType" type="sca:SvcType"
            minOccurs="0" maxOccurs="unbounded"/>
         <element name="inputBufferType" type="sca:BufferType"
            minOccurs="0" maxOccurs="unbounded"/>
         <element name="outputBufferType" type="sca:BufferType"
            minOccurs="0" maxOccurs="unbounded"/>
         <element name="errorBufferType" type="sca:BufferType"
            minOccurs="0" maxOccurs="unbounded"/>
         <element name="workStationParameters"
            type="sca:WorkStationParameters"
            minOccurs="0"/>
         <element name="authentication" type="sca:Authentication"
            minOccurs="0"/>
         <element name="fieldTablesLocation" type="string"
            minOccurs="0"/>
         <element name="fieldTables" type="string"
            minOccurs="0"/>
         <element name="fieldTablesLocation32" type="string"
            minOccurs="0"/>
         <element name="fieldTables32" type="string"
            minOccurs="0"/>
         <element name="viewFilesLocation" type="string"
            minOccurs="0"/>
         <element name="viewFiles" type="string" minOccurs="0"/>
         <element name="viewFilesLocation32" type="string"
            minOccurs="0"/>
         <element name="viewFiles32" type="string"
            minOccurs="0"/>
         <element name="remoteAccess" type="sca:RemoteAccess"
            minOccurs="0"/>
```

```
            <element name="transaction" type="sca:TransactionType"
               minOccurs="0"/>
         </sequence>
         <anyAttribute namespace="##any" processContents="lax" />
      </extension>
   </complexContent>
</complexType>

<complexType name="TargetMapType">
   <simpleContent>
      <extension base="TargetSimple">
         <attribute name="target" type="string" use="optional"/>
      </extension>
   </simpleContent>
</complexType>

<simpleType name="TargetSimple">
   <restriction base="string"/>
</simpleType>

<complexType name="SvcType">
   <simpleContent>
      <extension base="SvcTypeEnum">
         <attribute name="target" type="string" use="optional"/>
      </extension>
   </simpleContent>
</complexType>

<simpleType name="SvcTypeEnum">
   <restriction base="string">
      <enumeration value="oneway"/>
      <enumeration value="requestresponse"/>
   </restriction>
</simpleType>

<complexType name="BufferType">
   <simpleContent>
      <extension base="BufferTypeEnum">
```

```
            <attribute name="target" type="string" use="optional"/>
         </extension>
      </simpleContent>
   </complexType>

   <simpleType name="BufferTypeEnum">
      <restriction base="string">
         <enumeration value="string"/>
         <enumeration value="carray"/>
         <enumeration value="x_octet"/>
         <enumeration value="view"/>
         <enumeration value="x_c_type"/>
         <enumeration value="x_common"/>
         <enumeration value="view32"/>
         <enumeration value="xml"/>
         <enumeration value="fml"/>
         <enumeration value="fml32"/>
         <enumeration value="mbstring"/>
      </restriction>
   </simpleType>

   <complexType name="WorkStationParameters">
      <sequence>
         <element name="networkAddress" type="string" minOccurs="0"/>
         <element name="secPrincipalName" type="string" minOccurs="0"/>
         <element name="secPrincipalLocation" type="string"
               minOccurs="0"/>
         <element name="secPrincipalPassId" type="string"
               minOccurs="0"/>
         <element name="encryptbits" type="string" minOccurs="0"/>
      </sequence>
   </complexType>

   <complexType name="Authentication">
      <sequence>
         <element name="userName" type="string" minOccurs="0"/>
         <element name="clientName" type="string" minOccurs="0"/>
         <element name="groupName" type="string" minOccurs="0"/>
```

```
         <element name="passwordIdentifier" type="string"
               minOccurs="0"/>
         <element name="userPasswordIdentifier" type="string"
               minOccurs="0"/>
      </sequence>
   </complexType>

   <complexType name="RemoteAccess">
      <restriction base="string">
         <enumeration value="native"/>
         <enumeration value="workstation"/>
      </restriction>
   </complexType>

   <complexType name="TransactionType">
      <attribute name="timeout" type="int" use="optional"/>
   </complexType>
</schema
```

Web Service Binding Schema

Listing B-2 shows a Web service binding schema..

Listing B-2 Web Service Binding Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
   Licensed to the Apache Software Foundation (ASF) under one
   or more contributor license agreements.  See the NOTICE file
   distributed with this work for additional information
   regarding copyright ownership.  The ASF licenses this file
   to you under the Apache License, Version 2.0 (the
   "License"); you may not use this file except in compliance
   with the License.  You may obtain a copy of the License at

      http://www.apache.org/licenses/LICENSE-2.0
```

```
   Unless required by applicable law or agreed to in writing,
   software distributed under the License is distributed on an
   "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
   KIND, either express or implied.  See the License for the
   specific language governing permissions and limitations
   under the License.
-->

<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
   xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
   elementFormDefault="qualified">

   <element name="binding.ws" type="sca:WebServiceBinding"
substitutionGroup="sca:binding"/>
   <complexType name="WebServiceBinding">
      <complexContent>
         <extension base="sca:Binding">
            <sequence>
                        <element name="soapbinding" type="sca:SOAPBinding"
minOccurs="0" maxOccurs="unbounded"/>
                  <any namespace="##other" processContents="lax"
minOccurs="0" maxOccurs="unbounded" />
               </sequence>
               <attribute name="endpoint" type="anyURI" use="optional" />
               <attribute name="location" type="anyURI" use="optional" />
               <attribute name="conformanceURIs"
type="sca:ConformanceURIList" use="optional" />
               <attribute name="interfaceMapping" type="string"
use="optional" />
               <anyAttribute namespace="##any" processContents="lax" />
            </extension>
         </complexContent>
      </complexType>

   <complexType name="SOAPBinding">
```

```
         <sequence>
            <any namespace="##other" processContents="lax" minOccurs="0"
maxOccurs="unbounded" />
         </sequence>
      <attribute name="name" type="NCName" use="optional" />
      <attribute name="version" type="string" use="optional" />
              <anyAttribute namespace="##any" processContents="lax" />
   </complexType>

      <simpleType name="ConformanceURIList">
            <list itemType="anyURI"/>
      </simpleType>
</schema>
```