VMware Smart Assurance XML Adapter User Guide

VMware Smart Assurance 10.0.0

vmware®

You can find the most up-to-date technical documentation on the VMware website at:

https://docs.vmware.com/

If you have comments about this documentation, submit your feedback to

docfeedback@vmware.com

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Copyright © 2019 VMware, Inc. All rights reserved. Copyright and trademark information.

Contents

1 Preface 5

Purpose 5

Audience 6

VMware Smart Assurance Service Assurance Manager installation directory 6

VMware Smart Assurance Service Assurance Manager 6

Related documentation 7

Suggestions for searching PDF files 8

Conventions used in this document 8

Pathname conventions 9

Graphical conventions 9

Manager 9

Where to get help 9

Your comments 10

2 Introduction 11

Purpose of the XML Adapter 11

Framework description 13

Structure of the XML document 14

3 ICIM XML Elements 16

Element declarations 16

Attribute-list declarations 17

Object elements 18

Attribute elements 19

Relationship elements 21

ICIM/XML DTD specification 22

4 Building Topology in XML 23

Overview 23

Using XML for VMware Smart Assurance 23

Adding and updating topology 23

Removing topology 27

Datatypes 28

5 Exporting VMware Smart Assurance Information to XML 29

Overview 29

The ASL Export script 30

Building the ASL Export script 30

Parsing the XML 34
Running sm_xml for export 34
Sample ASL Export script 35

6 Importing Information into VMware Smart Assurance 38
XML import process 38

7 Understanding the sm_edit Utility 41
Example 1 41
Example 2 41

Preface

1

As part of an effort to improve its product lines, VMware Smart Assurance periodically releases revisions of its software and hardware. Therefore, some functions described in this document might not be supported by all versions of the software or hardware currently in use. The product release notes provide the most up-to-date information on product features.

Contact your VMware technical support professional if a product does not function properly or does not function as described in this document.

Note This document was accurate at publication time. Go to VMware Online Support docs.vmware.com to ensure that you are using the latest version of this document.

This chapter includes the following topics:

- Purpose
- Audience
- VMware Smart Assurance Service Assurance Manager installation directory
- VMware Smart Assurance Service Assurance Manager
- Related documentation
- Suggestions for searching PDF files
- Conventions used in this document
- Pathname conventions
- Graphical conventions
- Manager
- Where to get help
- Your comments

Purpose

This document describes the configuration and operation of the VMware Smart Assurance XML Adapter and is intended for integrators using the VMware Smart Assurance XML Adapter to exchange topological information between VMware Smart Assurance and other applications.

Audience

This guide is intended for administrators who are responsible for deploying, installing, and configuring the Global Manager. IT managers who seek to understand the role of the Global Manager in the context of an VMware Smart Assurance solution may also find this guide useful.

In addition to the configuration guides for specific components, administrators should also read the VMware Smart Assurance Service Assurance Manager Deployment Guide and the VMware Smart Assurance System Administration Guide Guide.

VMware Smart Assurance Service Assurance Manager installation directory

In this document, the term BASEDIR represents the location where VMware Smart Assurance software is installed:

- For UNIX, this location is: /opt/InCharge/<*productsuite*>.

  For example, on UNIX operating systems, VMware Smart Assurance Service Assurance Manager is, by default, installed to /opt/InCharge/SAM/smarts.

  Optionally, you can specify the root of BASEDIR to be something other than/opt/InCharge (on UNIX) but you cannot change the <*product*> location under the root directory.

  The VMware Smart Assurance System Administration Guide provides more information about the directory structure of VMware Smart Assurance software.

VMware Smart Assurance Service Assurance Manager

The VMware Smart Assurance Service Assurance Manager includes the following products:

- Service Assurance Manager (Global Manager), includes Business Impact Manager (BIM) and Failover System
- Global Console
- Business Dashboard
- Companion UI
- SAM Native Adapters:
  - Service Assurance Manager Notification Adapters (E-Mail Notifier Adapter, Script Notifier Adapter, SNMP Trap Notifier Adapter, Log File Notifier Adapter)
  - Adapter Platform (Adapter Platform server, Syslog Adapter, SNMP Trap Adapter, sm_ems command-line interface)
  - VMware Smart Assurance Adapter for UIM/O
  - XML Adapter

Related documentation

In addition to this document, VMware Smart Assurance Corporation provides a help system for command line programs as well as product documentation.

Help for command line programs

Descriptions of command line programs are available as HTML pages. The index.html file, which provides an index to the various commands, is located in the BASEDIR/smarts/doc/html/usage directory.

VMware Smart Assurance documentation

Readers of this guide may find the following related documentation helpful:

- Smarts Foundation EMC Data Access API (EDAA) Programmer Guide
- VMware Smart Assurance Common Information Model (ICIM) 1.11 Reference for Non-Service Assurance Manager Products
- VMware Smart Assurance Common Information Model (ICIM) Reference for Service Assurance Manager
- *VMware Smart Assurance Foundation Documentation Portfolio*
- VMware Smart Assurance System Administration Guide
- VMware Smart Assurance ASL Reference Guide
- VMware Smart Assurance Perl Reference Guide
- VMware Smart Assurance Dynamic Modeling Tutorial
- VMware Smart Assurance MODEL Reference Guide

VMware Smart Assurance Service Assurance Manager documentation

The following documents are relevant to users of the
VMware Smart Assurance Service Assurance Manager:

- VMware Smart Assurance Service Assurance Manager Release Notes
- VMware Smart Assurance Open Source License and Copyright Information
- VMware Smart Assurance Service Assurance Manager Introduction
- VMware Smart Assurance Installation Guide for SAM, IP, ESM, MPLS, and NPM Managers
- VMware Smart Assurance Service Assurance Manager Deployment Guide
- VMware Smart Assurance Service Assurance Manager Configuration Guide
- VMware Smart Assurance Service Assurance Manager Operator Guide

- VMware Smart Assurance Service Assurance Manager Dashboard Configuration Guide
- VMware Smart Assurance Business Impact Manager User Guide
- VMware Smart Assurance Failover System User Guide
- VMware Smart Assurance Service Assurance Manager Notification Adapters User Guide
- VMware Smart Assurance Service Assurance Manager Adapter Platform User Guide
- VMware Smart Assurance Adapter for Unified Infrastructure Manager/Operations Configuration Guide
- VMware Smart Assurance XML Adapter User Guide
- VMware Smart Assurance Notification Module User Guide
- VMware Smart Assurance Service Assurance Manager Troubleshooting Guide
- VMware Smart Assurance Companion UI Online Help
- VMware Smart Assurance Companion UI Open Source License and Copyright Information for GPLv3 as Included with a Distribution of SLES 11
- VMware Smart Assurance Service Assurance Manager Documentation Portfolio

  The VMware Smart Assurance Documentation Catalog provides documentation resources for other VMware Smart Assurance products.

Suggestions for searching PDF files

You can search across multiple PDF files by using the Adobe Acrobat Reader software:

1. If the documentation is not accessible to all users of the VMware Smart Assurance product, copy the contents of the BASEDIR/smarts/doc/pdf directory to a central location, such as a shared drive on your LAN, so that operators and others can view the documentation.
2. To search throughout the documentation library, open the Acrobat Reader software:
3. Select **Edit > Search**, and type a word or phrase.
4. Select **All PDF Documents** in, in the **Where would you like to search** option, and type the pathname of the location where the PDF documents reside.

   If you have more than one VMware Smart Assurance product installed, you can set up cross-product document searches by copying files from the BASEDIR/smarts/doc/pdf directory for each product into this common documentation directory path.

Conventions used in this document

VMware Smart Assurance uses the following convention for special notices:

Note NOTICE is used to address practices not related to personal injury.

Typographical conventions

VMware Smart Assurance uses the following type style conventions in this document:

Bold	Use for names of interface elements.
Italic	Use for full titles of publications referenced in text
`Monospace`	Use for: ■ System output, such as an error message or script ■ System code ■ Pathnames, filenames, prompts, and syntax ■ Commands and options
`Monospace italic`	Use for variables.
`Monospace bold`	Use for user input.
[ ]	Square brackets enclose optional values
|	Vertical bar indicates alternate selections — the bar means “or”
{ }	Braces enclose content that the user must specify, such as x or y or z
...	Ellipses indicate nonessential information omitted from the example

Pathname conventions

Directory pathnames are shown with forward slashes (/).

Graphical conventions

Under UNIX, the consoles appear with slight differences. For example, in views that display items in a tree hierarchy such as the Topology Browser, an open circle appears for UNIX.

Manager

Unless otherwise specified, the term Manager is used to refer to VMware Smart Assurance programs such as Domain Managers, Global Managers, and adapters.

Where to get help

VMware Smart Assurance support, product, and licensing information can be obtained as follows:

Product information

For documentation, release notes, software updates, or information about VMware products, go to VMware Online Support at:

docs.vmware.com

Technical support

Go to VMware Online Support. You will see several options for contacting VMware Technical Support. Note that to open a service request, you must have a valid support agreement. Contact your VMware sales representative for details about obtaining a valid support agreement or with questions about your account.

Your comments

Your suggestions will help us continue to improve the accuracy, organization, and overall quality of the user publications. Provide your feedback of this document to:docs.vmware.com.

Introduction

2

This chapter includes the following topics:

- Purpose of the XML Adapter
- Framework description
- Structure of the XML document

Purpose of the XML Adapter

An adapter controls the communication of information between devices or applications and a Manager. An adapter normalizes topology and event information from sources other than VMware Smart Assurance. Typical sources include SNMP traps or system log files. Once normalized to the ICIM data model, the information is transferred to a Manager (including either Service Assurance Global Manager or Domain Manager).

Adapters work with third-party applications in a variety of ways to import topology and event information and prepare it for use by the Global Manager. The XML Adapter is ideal for exchanging topology information with a system for which there is no native VMware Smart Assurance adapter. The XML Adapter can import topology into a Global Manager or underlying domain as well as export from a Global Manager or Domain Manager. Using the XML Adapter, topology information can be exchanged between Global Managers or underlying domains.

XML Adapter process

When importing topology into VMware Smart Assurance, start with an XML document that complies with the ICIM/XML standard and represents any topological elements managed by VMware Smart Assurance. The XML document is imported into the Manager using the XML Adapter and the objects are migrated into the repository of the Manager. VMware Smart Assurance XML Adapter import process illustrates the XML Adapter import process.

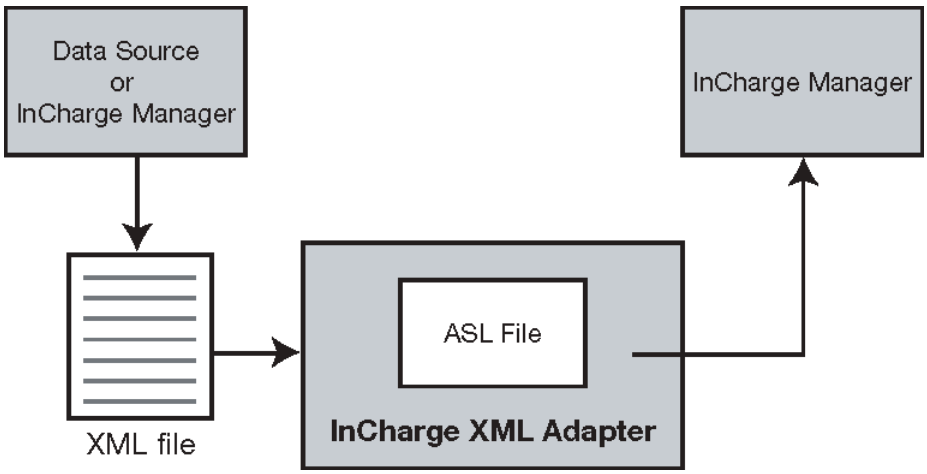


Figure 2-1. VMware Smart Assurance XML Adapter import process

The XML Adapter can import a large set of topology elements in one process or it can be used to make incremental updates to existing objects in the repository.

When exporting topology from VMware Smart Assurance, the XML Adapter extracts topological information from the Manager and produces an ICIM/XML-compliant document. VMware Smart Assurance XML Adapter export process illustrates the export process for the XML Adapter.

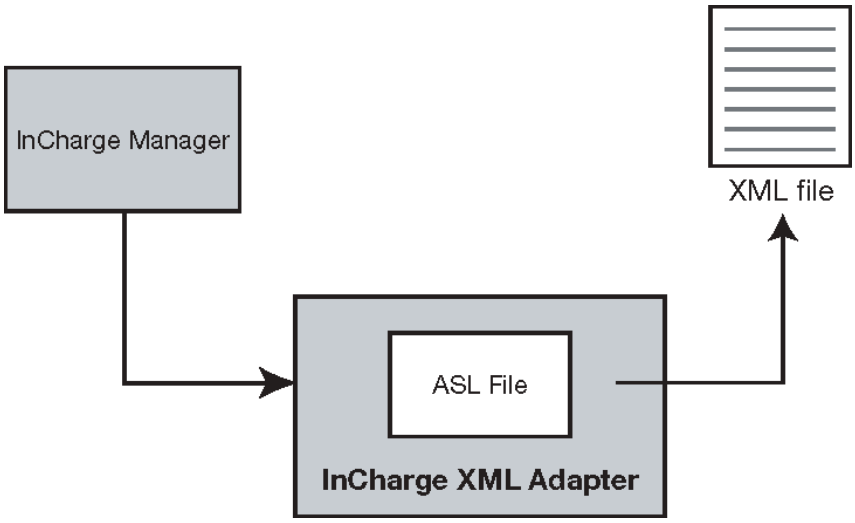


Figure 2-2. VMware Smart Assurance XML Adapter export process

The XML Adapter provides a structured method for transferring information in and out of a running Manager. Use it to perform these tasks, including:

- Adding or deleting objects
- Setting the value of attributes
- Inserting objects in or removing objects from relationships

Framework description

This section includes the following information:

- ICIM/XML DTD specification
- ICIM/XML document
- Command-line interface (**sm_xml**)
- ASL script

ICIM/XML DTD specification

The Document Type Definition (DTD) defines the building blocks of the XML document structure. It provides a list of allowable elements that can be used in the XML file. XML tags are formally described in this document. The DTD is declared as an external document.

Attributes are used to associate name-value pairs within elements. Attributes may appear only within open tags of the element. Attribute-list declarations in the DTD are used to define the set of attributes pertaining to a given element type or to provide default values for attributes.

The type of information associated with the source objects must be identical to the information associated with objects in the repository. For example, to import objects from an existing data source, ensure that the data source and the target repository database contain the same information in the same format, prior to importing. If the information models do not correspond exactly, you will lose data. The DTD specification is a guide to correctly map your data to objects to the repository.

ICIM/XML DTD specification on page 30 contains detailed ICIM/SML DTD information.

ICIM/XML document

Each XML document has both a physical and a logical structure. Physically, the document is composed of units called entities. An entity may refer to other entities to cause their inclusion in the document. A document begins in a "root" or document entity and is structured with objects, attributes and relationships. Logically, the document is composed of declarations, elements, comments, character references, and processing instructions, all of which are indicated in the document by explicit markup.

The XML DTD contains markup declarations that provide a grammar for the XML documents.

Each XML document contains one or more elements, the boundaries of which are delimited by open and close tags. Each element has a type (object, attribute, or relationship) and may have a set of attribute specifications. Each attribute specification has a name and a value.

Chapter 3 ICIM XML Elements contains detailed information on the structure of the ICIM/XML document.

Command-line interface (sm_xml)

The **sm_xml** command line utility is used to perform the import and export functions to the Manager. The sm_xml file is found in BASEDIR/smarts/bin. This interface can receive information from the Manager and send the output to a file or to standard output. It can also import a file of defined objects into the repository.

XML export ASL script

When exporting repository objects to XML, this adapter script extracts the VMware Smart Assurance objects to be represented in the XML output file. This Adapter Scripting Language (ASL) file utilizes the Adapter Scripting Language in addition to some unique commands for extracting the objects into an XML output file. A sample file, xml-export-sample.asl, is located in BASEDIR/smarts/rules/xml-if.

Chapter 5 Exporting VMware Smart AssuranceInformation to XML contains detailed information on exporting VMware Smart Assurance information to XML.

Structure of the XML document

The basic building blocks of an XML document are elements, attributes, and values. "Tag" is a general term that can be used for elements and attributes. A relevant sample of the ICIM/XML DTD is provided with each of the following descriptions.

Element

An element is a tag that is a container for subelements and attributes. For example, the object element contains defining information for the instance you are adding into the repository.

The root element, <icim>, is shown as the opening and closing tags for the entire file. For example, the ICIM/XML DTD allows you to import and export object, relationship, and attribute elements:

```
<!ELEMENT object ( attribute | relationship )* >
<!ELEMENT relationship ( object )* >
<!ELEMENT attribute (
        char | string | int | unsignedint | short |
   unsignedshort | long | unsignedlong | boolean | struct    | float   | double )* >
```

Attribute declaration (attribute list)

Attribute declarations, as defined in the ATTList of the DTD, expand the details of an element. All elements utilize a name declaration to identify it as well as a method declaration that instructs the adapter on whether to insert, update, or delete information about the element.

An attribute, as defined by the Attribute List in the DTD help identify the object instance. This should not be confused with an ICIM object attribute that describes the particular instance. For example:

```
<!ATTLIST object
        class     CDATA #IMPLIED
        name      CDATA #IMPLIED
        method    ( delete | update ) "update" >
<!ATTLIST relationship
        name      CDATA #REQUIRED
        method  ( insert | remove | put ) "put" >
```

Value

The value setting of the attribute declaration is paired with the declaration in name=value groupings. The values define the properties of the element you are adding, modifying, or deleting in the repository.

In the following example, the Object element is made up of an element start tag (<object>), attributes (Class, Name, and Method), values ("Application," "Banking," and "update"), and a close tag for the element (</Object>):

```
<object class="Application" name="Banking" method="update" />
```

Rules for a well-formed XML document

Follow these syntax rules when creating an XML document:

- Root element contains all other elements
- Elements and subelements are properly nested:
- <A><B></B></A> not <A><B></A></B>
- All elements have start and close tags:
- <object>Router</object> as opposed to <object>Router
- Start and close tags should match cases:
- <B>word</B> not <B>word
- Attribute declaration values should be in single or double quotes:
- <attribute name="Customer 1"> not
- <attribute name=Customer 1>

ICIM XML Elements

3

This chapter describes the elements used in ICIM XML and the syntax required to use them properly. *"ICIM/XML DTD specification" on page 30*includes a complete version of the ICIM/XML DTD.

This chapter includes the following topics:

- Element declarations
- Attribute-list declarations
- Object elements
- Attribute elements
- Relationship elements
- ICIM/XML DTD specification

Element declarations

The XML document type declaration contains or points to markup declarations that provide a grammar for a class of documents. This grammar is known as a document type definition or DTD.

Four elements are defined within the ICIM/XML document:

- ICIM Root
- Objects
- Attributes
- Relationships

  For each element declaration, you may have a set of attribute declarations that can include the class, name, or method of the element.

  Note The ICIM Root element is created when the XML document is generated. Create the remaining elements.

Attribute-list declarations

For each ICIM/XML element, there is an attribute-list declaration that defines values for that element. Attribute-list declarations specify the name, datatype, and default value (if any) of each attribute associated with a given element type. The attribute-list also defines whether the value is required or implied. If the value is required, define it in the document. If the value is implied, there is a default value for that attribute.

The syntax of the ATTLIST is defined as:

```
<!ATTLIST <element-name> <attribute-name> <attribute-type> default-value>
```

Class

The class attribute defines the ICIM objects with common properties, common behaviors, and similar relationships. Examples of classes are Switches, Routers, Applications, Service Subscribers, and Service Offerings. Any class already modelled and existing in your current repository is allowable in the XML document.

Name

This attribute identifies a unique name of the element or subelement displayed in the repository.

Method

The method attribute is used to declare the specific topology change you want to make. In the following examples, the Service Offering named GE is being updated and deleted:

```
<object class="ServiceOffering" name="GE" method="update" />
<object class="ServiceOffering" name="GE" method="delete" />
```

Method descriptions for ICIM XML references identifies the methods associated with each element type.

Table 3-1. Method descriptions for ICIM XML references

Element	Method	Description
<object>	update	Modifies the properties of an existing object or creates the object if it does not exist.
	delete	Removes an object from the repository.
<attribute>	insert	Adds a value to a set of attributes.
	remove	Removes an attribute from an object.
	put	Assigns a value to an attribute.
<relationship>	insert	Adds an object to a relationship.

Table 3-1. Method descriptions for ICIM XML references (Continued)

Element	Method	Description
	remove	Removes an object from a relationship.
	put	Assigns an object or set of objects to a relationship.

Note A "put" associates a value or structure of values with an object. If a value already exists, it will be overwritten. An “insert” adds a value to a set of attributes if the attribute allows for multiple values. If the attribute can only have one value, then using the insert method will replace the existing single value. Propagated attributes, instrumented attributes, and computed attributes cannot be modified with the insert or put methods.

Object elements

An <object> element identifies a repository object.

In the ICIM/XML DTD specification, the object element is declared as:

```
<!ELEMENT object (attribute | relationship)>
```

This declares the VMware Smart Assurance object element to be modified. Nested in this element will be subelements that define the attribute information associated with it. The subelement of the object element can be one or more ICIM attributes or ICIM relationships.

<Object> element attribute list declarations

An example of an <object> element attribute list declaration is:

```
<!ATTLIST object
       class     CDATA #IMPLIED
  name      CDATA #IMPLIED
  method    ( update | delete ) "update" >
```

Attribute-list declarations for the object element identifies the attribute declarations for the <object> element. If the object you specify does not already exist in the repository, define the class of the instance to be created.

Table 3-2. Attribute-list declarations for the object element

XML attribute	Description	Possible value	Value required/implied
class	The ICIM class of the object being declared. This can be any ICIM class name.	CDATA	Implied Default is MR_Object
name	Name of the object instance.	CDATA	Implied No default value
method	Specifies the desired topology change. You can delete or update an existing object. An update will create the object if it does not exist in the repository.	delete update	Implied Default is "update"

An object will have zero or more attributes or relationships within one declaration.

Attribute elements

An attribute element specifies a property present for a particular object element. Attributes expand the details of an object. For instance, DisplayName is an attribute of the element Application and sets how the instance will be identified in the Global Console. An attribute declaration includes the attribute's type and value. The value describes the state of the instance.

The attribute element must be nested within the object element.

<Attribute> element attribute-list declarations

For the <attribute> element, the ATTLIST is defined as:

```
<!ELEMENT attribute (
  char    | string        | int     | unsignedint
  short   | unsignedshort | long    | unsignedlong  |
  boolean | struct        | float   | double        )* >
<!ATTLIST attribute
 name      CDATA #REQUIRED
 method ( insert | remove | put ) "put" >
```

Attribute-list declarations for the attribute element identifies the attribute list declarations for the <attribute> element.

Table 3-3. Attribute-list declarations for the attribute element

XML attribute	Description	Possible value	Value required/implied
name	Name of the object instance.	CDATA	Required
method	Specifies the desired topology change. You can create a new attribute, remove an existing attribute, or update (put) an existing attribute.	insert remove put	Implied Default is "put"

For example, the following sample identifies an Application named App-BankingDataBase having a display name of Banking:

```
<object class="Application" name="APP-BankingDataBase">
 <attribute name="DisplayName">
  <string>Banking</string>
 </attribute>
</object>
```

In this example, the UtilizationThreshold attribute is set with an integer datatype:

```
<attribute name="UtilizationThreshold">
 <int>75</int>
</attribute>
```

Applying multiple attributes to an object

Since an attribute element can have one or more values within one declaration, you must be able to apply values in multiples. If the target to be modified is a table, the <attribute> element may be modified using a <struct> tag to modify multiple values. A struct is a collection of different datatypes. It allows you to apply a set of values to an element (such as string, integer, boolean). For example, to add two properties of different types (string and unsignedshort) to the AgentInfoSet attribute, use the following syntax:

```
<attribute name="AgentInfoSet" method="insert">
 <struct>
  <string>public</string>
  <unsignedshort>162</unsignedshort>
 </struct>
</attribute>
```

If an object contains an attribute called IntegerList, which contains a list of three integers, this attribute can be set in the following way:

```
<attribute name="IntegerList">
 <struct>
  <int>4</int>
  <int>8</int>
  <int>9</int>
 </struct>
</attribute>
```

If the attribute represents an IntegerList2, which has an unspecified number of integers, it can be imported in the following manner:

```
<attribute name="IntegerList2">
 <struct><int>4</int></struct>
 <struct><int>8</int></struct>
 <struct><int>9</int></struct>
</attribute>
```

Relationship elements

A relationship maps instances of one class to instances of the same or another class. For each relationship, an inverse relationship may also exist that maps from the related class back to the originating class.

The <relationship> element modifies a relationship. Like the <attribute> element, <relationship> may be used to put, insert, or remove an object from a relationship.

<Relationship> element attribute list declarations

Am example of the <relationship> element is:

```
<!ELEMENT relationship ( object )* >
<!ATTLIST relationship
 name      CDATA #REQUIRED
 method  ( insert | remove | put ) "put" >
```

Attribute-list declarations for the relationship element identifies the attribute declarations available for the <relationship> element.

Table 3-4. Attribute-list declarations for the relationship element

XML attribute	Description	Possible value	Value required/implied
name	Name of the object instance.	CDATA	Required
method	Specifies the desired topology change. You can create a new relationship, delete an existing relationship, or update an existing relationship.	insert remove put	IMPLIED Default is "put"

For example, to add the HostedBy relationship between the application Banking and the host BankingHost, use the following syntax:

```
<object class="Application" name="Banking">
 <relationship name="HostedBy" method="put" />
  <object class="Host" name="BankingHost" method="update" />
 </relationship>
</object>
```

If multiple objects are specified when the method is "insert" (or "remove") then all are inserted (or removed). The methods "put" and "insert" are equivalent if the relationship is a singleton. The "remove" method may be used to clear a singleton relationship:

```
<relationship name="HostedBy" method="remove" />
```

A relationship can have zero or more objects associated with it within one declaration, meaning you may want to add a relationship even if you do not have a specific object to assign to it. To declare zero objects, put or insert the NULL object declaration:

```
<relationship name="HostedBy">
 <object/>
</relationship>
```

ICIM/XML DTD specification

The ICIM/XML DTD file is icim_xml.dtd and is in BASEDIR/smarts/rules/xml-if.The contents of this file cannot be modified. The icim_xml.dtd file contains the following:

```
<!ELEMENT icim ( object )* >
<!ATTLIST icim source CDATA #IMPLIED >
<!ELEMENT object ( attribute | relationship )* >
<!ATTLIST object
        class     CDATA #IMPLIED
        name      CDATA #IMPLIED
        method    ( delete | update ) "update" >
<!ELEMENT relationship ( object )* >
<!ATTLIST relationship
        name      CDATA #REQUIRED
        method  ( insert | remove | put ) "put" >
<!ELEMENT attribute (
        char | string | int | unsignedint | short |
   unsignedshort | long | unsignedlong | boolean | struct    | float   | double )* >
<!ATTLIST attribute
        name      CDATA #REQUIRED
        method ( insert | remove | put ) "put" >
<!ELEMENT string          ( #PCDATA ) >
<!ELEMENT char            ( #PCDATA ) >
<!ELEMENT int             ( #PCDATA ) >
<!ELEMENT unsignedint     ( #PCDATA ) >
<!ELEMENT short           ( #PCDATA ) >
<!ELEMENT unsignedshort   ( #PCDATA ) >
<!ELEMENT long            ( #PCDATA ) >
<!ELEMENT unsignedlong    ( #PCDATA ) >
<!ELEMENT boolean         ( #PCDATA ) >
<!ELEMENT float           ( #PCDATA ) >
<!ELEMENT double          ( #PCDATA ) >
<!ELEMENT struct          (
  char | string | int | unsignedint | short |
  unsignedshort | long  | unsignedlong | boolean |
  struct| float | double )* >
```

Building Topology in XML

4

This chapter includes the following topics:

- Overview
- Using XML for VMware Smart Assurance
- Adding and updating topology
- Removing topology
- Datatypes

Overview

The XML document is a database. The following actions can be controlled using XML:

- Add topology
- Update topology
- Delete topology
- Assign values to an attribute, relationship, or relationship set
- Insert and remove information in a table attribute or relationship set

Using XML for VMware Smart Assurance

Working with objects in XML means you are creating, updating, and removing instances of classes in the Manager. Classes cannot be added to VMware Smart Assurance using the XML Adapter; only add instances and relationships can be added to existing classes.

This section contains XML excerpts to introduce the basic XML declarations to use to import topology changes. This set of examples is importing Business Impact Manager topology into the Service Assurance Manager. For some examples, a Services Map in the Global Console illustrates the results of the topology import.

Adding and updating topology

This section shows examples to represent new objects, relationships, and attributes in XML.

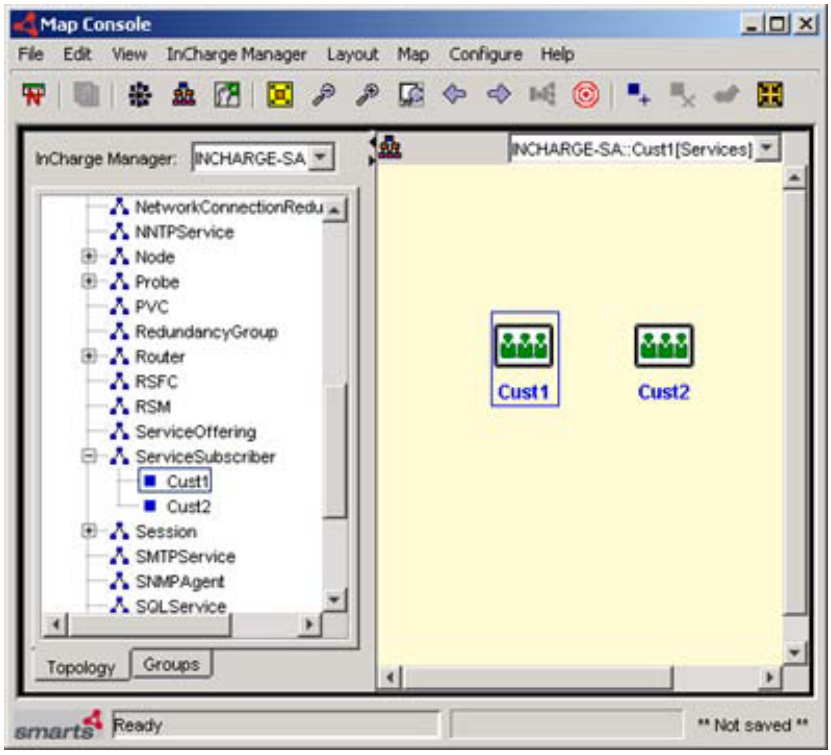
Adding objects

Importing instances requires creating objects in the XML file. Two ServiceSubscriber instances, called Cust1 and Cust2, are created in the XML file, for example:

```
<icim>
 <object class="ServiceSubscriber" name="Cust1" />
 <object class="ServiceSubscriber" name="Cust2" />
</icim>
```

By default, the method for an object element is "update," so it is unnecessary to specify it. If the object does not exist, then the update method automatically adds it to the repository. Map of imported Service Subscribers illustrates a Services Map and topology browser after importing these Service Subscribers.

Figure 4-1. Map of imported Service Subscribers



Adding attributes

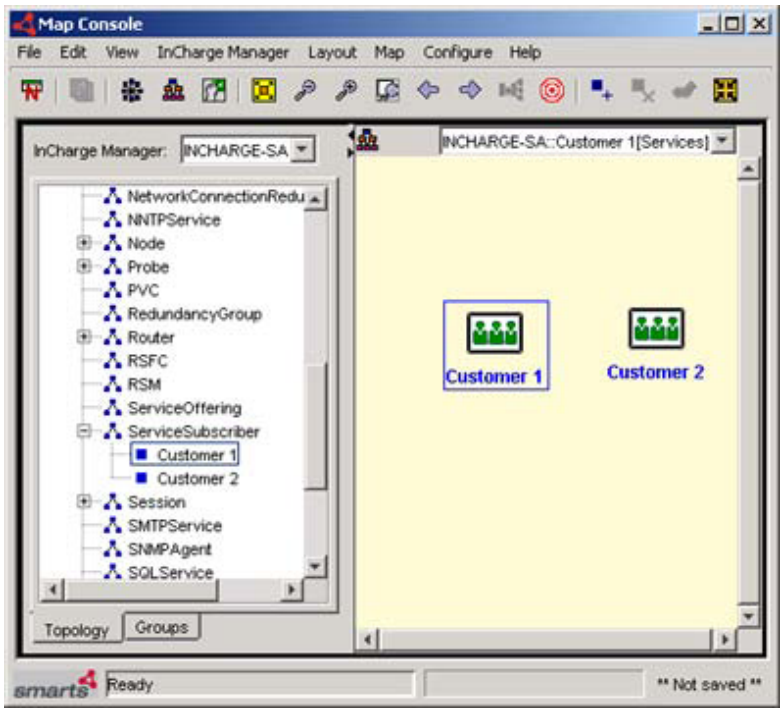
Adding attribute elements requires nesting the attribute declaration within the appropriate object declaration. This example adds the Display Name attribute to the Service Subscribers you created. This changes the Display Name of "Cust1" to "Customer 1" and "Cust2" to "Customer 2". For example:

```
<icim>
<object class="ServiceSubscriber" name="Cust1" method="update">
 <attribute name="DisplayName" method="put">
     <string>Customer 1</string>
 </attribute>
</object>
```

```
<object class="ServiceSubscriber" name="Cust2" method="update">
 <attribute name="DisplayName" method="put">
     <string>Customer 2</string>
 </attribute>
</object>
</icim>
```

Map of imported Service Subscribers with new display names illustrates the Services Subscribers with their new display names.

Figure 4-2. Map of imported Service Subscribers with new display names



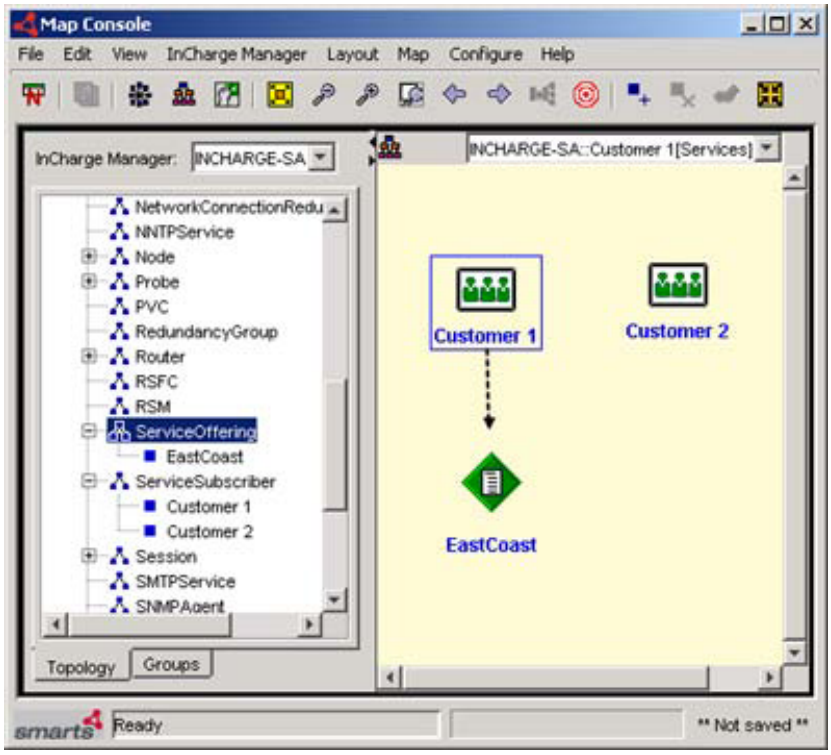
Adding relationships using the put method

Adding relationships requires nesting the relationship declarations within the object declarations. Using the "put" method means you are assigning the only attribute intended for that object. The following example shows how to add a Service Offering called "EastCoast" and use the "Subscriptions" relationship to associate that offering with the Customer 1 subscriber created previously. For example:

```
<icim>
    <object class="ServiceSubscriber" name="Cust 1" method="update">
 <relationship name="Subscriptions" method="put">
      <object class="ServiceOffering" name="EastCoast" />
 </relationship>
    </object>
</icim>
```

Map of imported Service Subscribers and Service Offering illustrates the new Service Offering, EastCoast and Services Subscriber, Customer 1 with a Subscriptions relationship.

Figure 4-3. Map of imported Service Subscribers and Service Offering



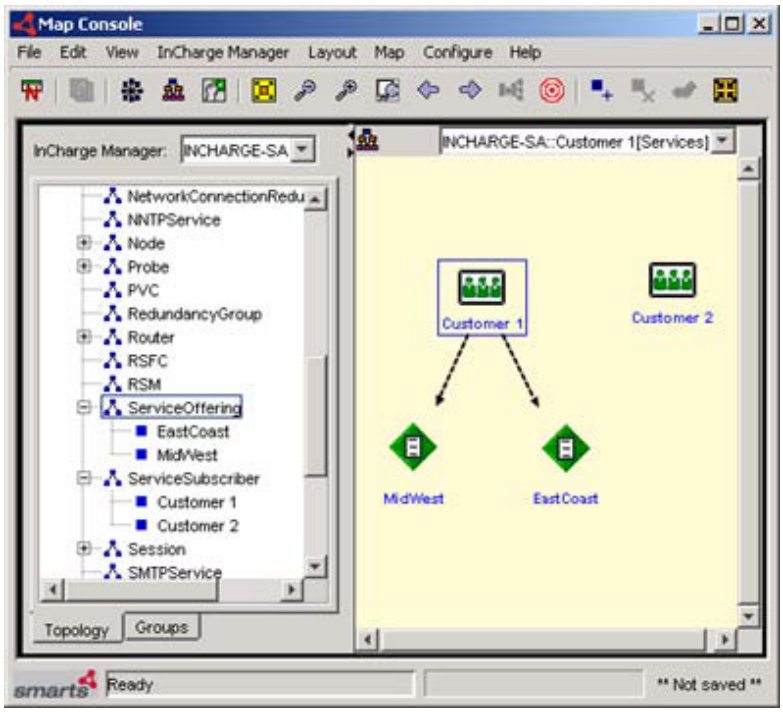
Adding relationships using the insert method

Using the "insert" method allows you to add to the list of attributes intended for that object. The following example shows how to insert another Service Offering called MidWest into the Subscriptions relationship for Customer 1. For example:

```
<icim>
<object class="ServiceSubscriber" name="Cust 1" method="update">
  <relationship name="Subscriptions" method="insert">
     <object class="ServiceOffering" name="MidWest" />
 </relationship>
    </object>
</icim>
```

Map of imported Service Subscribers and Service Offering illustrates the new Service Offering, MidWest, and Services Subscriber, Customer 1 with a Subscriptions relationship.

Figure 4-4. Map of imported Service Subscribers and Service Offering



Removing topology

This section describes how to remove topology from a Manager.

Attempting to perform the following tasks causes a runtime error when you import the XML file:

- Deleting a nonexistent object
- Deleting a nonexistent attribute or relationship

Removing objects

To remove an object from the repository, use the "delete" method for an object element that already exists. For example, to delete the ServiceSubscriber called Cust 2, use the following syntax:

```
<object class="ServiceSubscriber" name="Cust 2" method="delete" />
```

Removing attributes and relationships

To remove a particular attribute or relationship for an object, adhere to the syntax of the ICIM/XML DTD. This means that the relationship and attribute updates must be nested within the appropriate object declaration. For example, to remove the Subscriptions relationship from the EastCoast service offering, use the following syntax:

```
<icim>
    <object class="ServiceSubscriber" name="Cust 1" method="update">
  <relationship name="Subscriptions" method="remove">
        <object class="ServiceOffering" name="EastCoast" />
  </relationship>
    </object>
</icim>
```

The topology returns to look as it did in Removing attributes and relationships.

Datatypes

Supported datatypes lists the datatypes supported in the ICIM/XML DTD.

Table 4-1. Supported datatypes

Value	Description
boolean	Datatype defined as two possible values (such as 0 or 1, on or off, true or false)
char	Any single ASCII character
double	Any double-precision floating point number (similar to C++)
float	Any single-precision floating point number (similar to C++)
int	A 32-bit integer ranging from -2^31 to +(2^31 -1)
long	A 64-bit integer ranging from -2^63 to +(2^63 -1)
short	A 16-bit integer ranging from -32,768 to 32,767
string	An ordered list of characters
struct	A collection of fields, each field specifies its own type
unsignedint	Unsigned 32-bit integer 232-1
unsignedlong	Unsigned 64-bit integer ranging from 264-1 to +263 -1
unsignedshort	Unsigned 16-bit integer 216-1

Exporting VMware Smart Assurance Information to XML

5

This chapter includes the following topics:

- Overview
- The ASL Export script
- Building the ASL Export script
- Parsing the XML
- Running sm_xml for export
- Sample ASL Export script

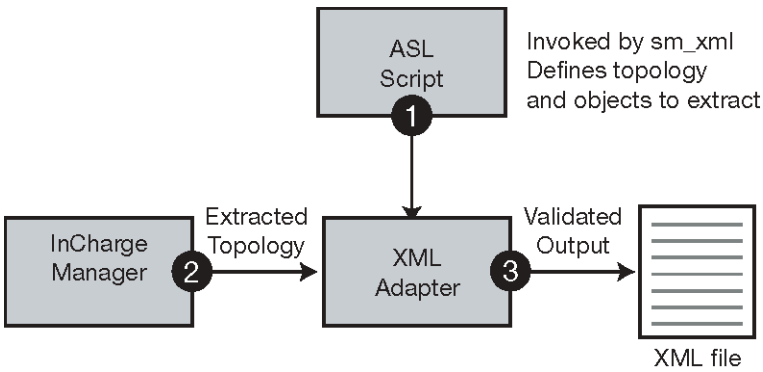
Overview

Exporting object data to an XML file is a convenient way to move and store repository information in a format that can be used across the web or by another Manager.

To export data to XML, you must define an ASL script that queries the repository. In the ASL script, you specify the information and matching criteria for the objects to be gathered and exported. A sample ASL file, xml-export-sample.asl, is provided in BASEDIR/smarts/rules/xml-if.

The XML Adapter invokes the specified ASL script against the specified Manager and produces the specified XML output file. Process diagram: Exporting VMware Smart Assurance data in XML illustrates the process for exporting VMware Smart Assurance data in XML.

Figure 5-1. Process diagram: Exporting VMware Smart Assurance data in XML



The ASL Export script

A template for the ASL Export script, called xml-export-sample.asl, is installed in BASEDIR/smarts/rules/xml-if. Use ASL to modify the functionality of the local copy of the script. Use the **sm_edit** utility to modify the ASL script so that a copy is made in the local directory and the proper permissions are associated to it. Chapter 7 Understanding the sm_edit Utility includes information on using the **sm_edit** utility.

To open a file called xml-export-sample.asl using **sm_edit**:

```
t# BASEDIR/smarts/bin>sm_edit BASEDIR/smarts/rules/xml-if/xml-export-sample.asl s
```

Save this file with a new name, replacing "sample" with an appropriate grouping name (for example, "Services" for an ASL file intended to gather information on Service Subscribers and Service Offerings).

Understanding the ASL script

Note A thorough understanding of ASL is required to work in the export script. The VMware Smart Assurance ASL Reference Guide provides detailed information on working with the Adapter Scripting Language.

The first rule executed in the ASL script is the START rule. The START rule runs repeatedly until all the data input is processed. This rule invokes other rules or can specify which objects, attributes, and relationships to export.

Data input gets processed as it is matched with all the patterns of a rule. The START rule has components in which the request details are defined.

The XML headers and the ICIM root element are added automatically to the XML document by the sm_XML Adapter. The ASL script is responsible for gathering and adding the object, attribute, and relationship elements to the XML file.

Any modifications to the action blocks are made after the following comment:

```
"NOTE:  Customize the script from this point on"
```

These modifications define the specific information to retrieve from the repository.

After the ASL script is executed, the closing tag for the ICIM root element is added to the document automatically.

Building the ASL Export script

This section describes how to retrieve specific instances of objects, build foreach statements and use the builder statements to export the exact information you need from the repository.

The ASL file is constructed from these functions and statements: getInstance, foreach statements and builder statements.

getInstance function

The getInstances function returns a list of object names for the given class. The syntax of the getInstances function is:

```
listname = getInstances(<classname>);
```

For example, to retrieve all instances of the ServiceSubscriber class and add them to your XML document, begin with the following line:

```
EXPORT_SERVICESUBSCRIBER {
} do {
  svcSubList = getInstances("ServiceSubscriber");
...
}
```

Once the objects you want to export are defined, then define how to handle them. Use the builder statements described in builder statements and the foreach statements that follow to complete this task.

foreach statement

The foreach statement iterates through all the members of a list or a table. The syntax of the foreach statement is:

```
foreach variable (listortablename) {statements}
```

When the foreach statement is used with a loop, the variable stores the value of each member of the list every time the foreach statement loops. The members returned are in numerical order by their list indices.

Building on our ServiceSubscriber example above, add the following statements:

```
EXPORT_SERVICESUBSCRIBER {
} do {
 svcSubList = getInstances("ServiceSubscriber")? LOG, NEXT;
 foreach svcSub (svcSubList) {
  builder->addObject(svcSub, "ServiceSubscriber");
  builder->addAttribute(svcSub, "DisplayName");
  builder->addObjectClosing();
    }
}
```

builder statements

Each element is created and formatted in the ICIM/XML document using builder statements that define the objects, attributes, and relationships. The syntax for the builder statements is as follows:

```
builder-> <BuilderStatement>(<Argument>,"<Value>");
```

Object element type builder statements

An object element requires two builder statements. Builder statements for the object element type describes the builder statements and arguments for an object element.

Table 5-1. Builder statements for the object element type

Builder statement	Description	Arguments	Required / optional
addObject()	Adds the instance to the XML document. It creates the open tag for the object declaration.	Instance Name	Optional
		Class Name	Required
		Method	Optional
addObjectClosing()	Adds the closing tag for the object declaration in the XML file.	No Arguments	N/A
addNullObject	Adds a null object "<object/>" into the output XML document.	No Argument	N/A

For example, to export an instance of the ServiceSubscriber class to the XML document, the builder statement resembles this:

```
foreach svcSub (svcSubList) {
 builder->addObject(svcSub, "ServiceSubscriber");
 builder->addAttribute(svcSub, "DisplayName");
 builder->addObjectClosing();
```

Relationship element type builder statements

A relationship element has two required builder statements. Builder statements for the relationship element type describes the builder statements and arguments for a relationship element.

Table 5-2. Builder statements for the relationship element type

Builder statement	Description	Arguments	Required / optional
addRelationship( )	This adds the relationship to the XML document. It creates the open tag for the relationship declaration.	Relationship Name	Required
		Method	Optional
addRelationshipClosing ()	Adds the closing tag for the relationship declaration in the XML file.	No Arguments	N/A

For example, to export all the objects with a Subscriptions relationship to the XML document, the builder statement resembles this:

```
builder->addRelationship("Subscriptions");
 foreach...
builder->addRelationshipClosing();
```

Attribute element type builder statements

An attribute element has only one required builder statement. Builder statements for the attribute element type describes the builder statement and arguments for an attribute element.

Table 5-3. Builder statements for the attribute element type

Builder statement	Description	Arguments	Required / optional
addAttribute( )	This adds the attributes of the instance to the XML document.	Instance Name	Required
		Attribute Name	Required
		Method	Optional

Note The closing builder statement is unnecessary for this element because there is no recursive part so we can predict the end of the statement.

Since the ICIM/XML DTD requires that attribute definitions are nested within an object definition, this builder statement is defined with an object element. For example, to export instances of the ServiceSubscriber class, with their display names, to the XML document the builder statement resembles the following:

```
builder–>addObject(svcSub, "ServiceSubscriber");
builder–>addAttribute (svcSub, "DisplayName");
builder–>addObjectClosing();
```

String builder statement

Optionally, you can add text into the XML document along with any of the declarations. After any of the builder statements defined above, you can use the addText statement. The argument may be any string that complies with the ICIM/XML DTD. Builder statements for the string describes the builder statement and arguments for this element.

Table 5-4. Builder statements for the string

Builder statement	Description	Arguments	Required / optional
addText( )	Adds optional text to the XML declaration.	String	Required

For example, to export instances of the ServiceSubscriber class, with their display names, and additional text that says Sample Offeringto the XML document, use the following syntax:

```
builder–>addObject(svcSub, "ServiceSubscriber");
builder–>addAttribute (svcSub, "DisplayName");
builder–>addObjectClosing();
builder–>addText("<object class=\"ServiceOffering"\) name=\"Sample Offering"\ method=\"update"\ />;
```

Parsing the XML

The resulting XML generated by the ASL script and the XML Adapter is passed through a validating parser. This is to ensure compliance with the ICIM/XML DTD. This parser, enables the XML Adapter to read and write XML data.

VMware, Inc. provides the Xerces XML Parser with the installation of the XML Adapter.

Running sm_xml for export

XML exporter command parameters describes four export-specific parameters associated with the **sm_xml** utility.

Table 5-5. XML exporter command parameters

Parameter	Description	Optional flag
Options:		
--server=<name>	Name of the Manager from which you are retrieving repository information. By default, this Manager name is INCHARGE.	-s
--broker=<location>	Alternate location of the VMware Smart Assurance broker. Must be typed as **host:port.** The default is the broker specified for the Manager at installation.	-b
Commands:		
export --xmlfile=<output-xmlfile> <asl-script>	Export an XML document using the XML export ASL script named <asl-script>. By default, the XML is written to standard output. Optionally, the XML can be written to a file if you specify: --xmlfile=<output-xmlfile>.	-x <output-xmlfile>
Command arguments:		
output=<xmlfile>	Name of the resulting XML file (and qualified path to save the file). Optionally, you can choose not to define a directory and filename and the resulting XML will be displayed in standard output.	-x
<asl-script>	The ASL rule script used to export the VMware Smart Assurance topology. The adapter will look for the ASL script in BASEDIR/local/rules/xml-if first. If it is not found there, it will locate it in BASEDIR/rules/xml-if.	N/A

The resulting XML is passed through a validating parser. If the document is noncompliant with the ICIM/XML DTD, an error will appear and the program will exit with a nonzero exit status. When an error occurs, the XML is still written to standard output.

For example, to generate an XML file called InChargeApplications.xml containing information from a Manager called INCHARGE, with an ASL Export script called xml-export-Applications.asl, use the following command:

```
tsm_xml --server=<
         INCHARGE
         > export --xmlfile=InChargeApplications.xml xml-export-Applications.asls
```

Sample ASL Export script

In this example ASL script, you are requesting XML output for all instances of the ServiceSubscriber class, the ServiceOfferings to which they are subscribed, and the Applications that compose the ServiceOfferings:

```
/* xml-export-sample.asl
 *
 * Copyright (C) 2003, System Management ARTS (SMARTS)
 * All Rights Reserved
 *
 * Sample Rule Set to export the topology of the ServiceSubscribers, the
 * ServiceOfferings they are subscribed to, the Applications that these
 * ServiceOfferings are composedOf, and the Hosts that Host these
*  Applications from the server into an XML document that will confirm
* to the ICIM/XML DTD specifications.
 *
 * This is a sample xml-export-*.asl file. Users should make copies of this
 * file, and edit them according to their needs. But they should always name
 * these asl files according to the Naming convention, xml-export-<name>.asl
 */
/*
 * the XML_Builder object
 */
default builderName = "";
aslname = " ".getRuleFileName().": ";
builder = self->object("XML_Builder", builderName)? LOG, STOP;
if (builder->isNull()) {
    print(aslname."ERROR: The Wrapper object has not been created.");
    print(aslname."ERROR: Please make sure the CREATE_BUILDER is called ".
                        "before calling the INITIALIZE rule in the script.");
    stop();
}
/*
 * NOTE: Customize the script from this point on.
 */
START() {
    .. eol
    EXPORT_BIM_INSTANCES
}
EXPORT_BIM_INSTANCES {
    EXPORT_SERVICESUBSCRIBER
    EXPORT_SERVICEOFFERRING
    EXPORT_APPLICATION
}
EXPORT_SERVICESUBSCRIBER {
} do {
    /*
     * Get all the instances of ServiceSubscriber class and start adding to the
     * XML Document.
     */
    svcSubList = getInstances("ServiceSubscriber")? LOG, NEXT;
    foreach svcSub (svcSubList) {
```

```
        svcSubObj = object("ServiceSubscriber", svcSub)? LOG, IGNORE;
        builder->addObject(svcSub, "ServiceSubscriber");
        builder->addAttribute(svcSub, "DisplayName");
        builder->addRelationship("Subscriptions");
        /*
         * Get all the ServiceOffering objects that each of these
         * ServiceSubscribers subscribe to, and add them to the document.
         */
        foreach svcOffObj (svcSubObj->Subscriptions) {
            builder->addObject(svcOffObj->Name, svcOffObj->CreationClassName);
            builder->addObjectClosing();
        }
        builder->addRelationshipClosing();
        builder->addObjectClosing();
    }
}
EXPORT_SERVICEOFFERRING {
} do {
    /*
     * Get all the instances of ServiceOffering class and start adding to the
     * XML Document.
     */
    svcOffList = getInstances("ServiceOffering")? LOG, NEXT;
    foreach svcOff (svcOffList) {
        svcOffObj = object("ServiceOffering", svcOff)? LOG, IGNORE;
        builder->addObject(svcOff, "ServiceOffering");
        builder->addAttribute(svcOff, "DisplayName");
        builder->addRelationship("ConsistsOf");
        /*
         * Get all the <infrastructure> objects that each of these
         * ServiceOfferings consists of, and add them to the document.
         */
        foreach infObj (svcOffObj->ConsistsOf) {
            builder->addObject(infObj->Name, infObj->CreationClassName);
            builder->addObjectClosing();
        }
        builder->addRelationshipClosing();
        builder->addObjectClosing();
    }
}
EXPORT_APPLICATION {
} do {
    /*
     * Get all the instances of Application class and start adding to the
     * XML Document.
     */
    appList = getInstances("Application")? LOG, NEXT;
    foreach app (appList) {
        appObj = object("Application", app)? LOG, IGNORE;
        builder->addObject(app, "Application");
        builder->addAttribute(app, "DisplayName");
/*
         * Get the Host object that this Application is HostedBy,
         * and add it to the document.
         */
```

```
        hostObj = appObj->HostedBy;
        builder->addRelationship("HostedBy");
        builder->addObject(hostObj->Name, hostObj->CreationClassName);
        builder->addObjectClosing();
        builder->addRelationshipClosing();
        /*
         * Get all the Transaction objects that these Applications Produce and
         * Consume, and export them to the XML Document also.
         */
        builder->addRelationship("Produces");
        foreach trans (appObj->Produces) {
            builder->addObjectClosing();
        }
builder->addRelationshipClosing();
        builder->addRelationship("Consumes");
        foreach trans (appObj->Consumes) {
            builder->addObject(trans->Name, trans->CreationClassName);
            builder->addObjectClosing();
        }
        builder->addRelationshipClosing();
        builder->addObjectClosing();
    }
}
```

Importing Information into VMware Smart Assurance

6

This chapter describes how to import XML data into the repository and contains the following information:

- XML import process

XML import process

To import an XML document containing topology, events, or other objects, your XML document must comply with the ICIM/XML DTD. Process diagram: Importing XML data into VMware Smart Assurance illustrates the XML import process. You may have a utility that exports the desired topology data as XML using a unique DTD as in Option A. If this is the case, use an XSL Transformation (XSLT), which is a language for transforming XML documents into other XML documents.

Alternatively, you can manually create an XML file that complies with the ICIM/XML DTD as in Option B.

When your XML document is ready, use the sm_XML Adapter to import the new objects into the repository. Process diagram: Importing XML data into VMware Smart Assurancedisplays the import process.

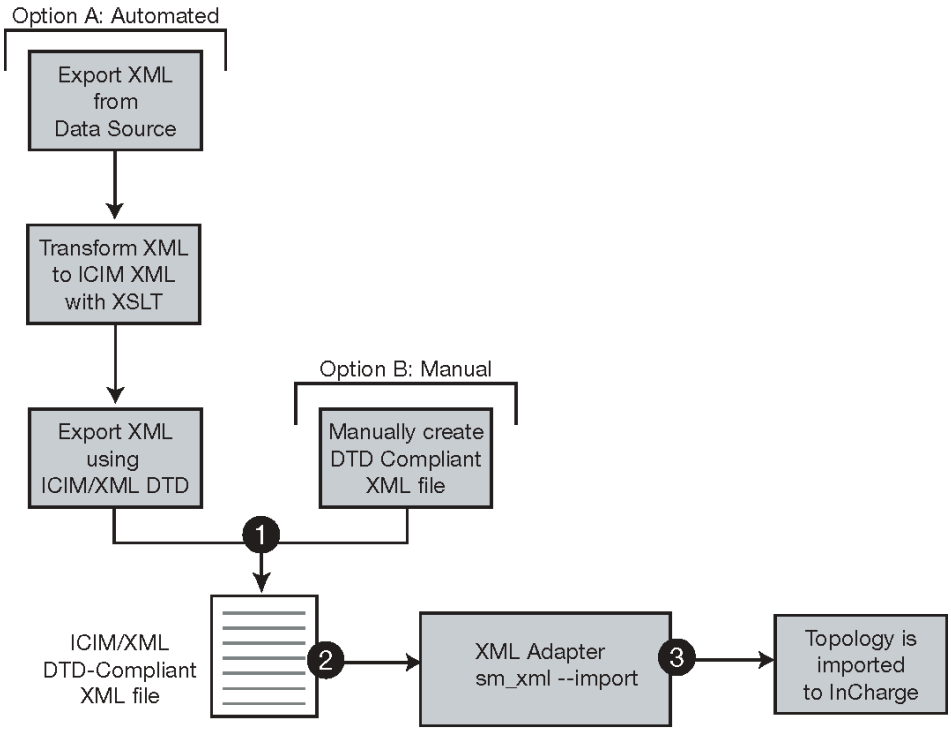


Figure 6-1. Process diagram: Importing XML data into VMware Smart Assurance

Before importing XML

It is unnecessary to shut down the Manager before you import the XML file. However, when importing, verify your XML is valid according to the ICIM/XML DTD. The DTD file, called icim_xml.dtd is located in BASEDIR/smarts/conf/xml-if.

A runtime error is generated if the XML importer cannot execute the instructions encoded in the XML document. By default, the importer skips any element (or child element) containing an error. It continues with the next element and generates a report of errors at the end.

To determine if the XML is valid and complies with the ICIM/XML DTD, run the following command:

```
tsm_xml --server=<
         INCHARGE
         > import <
         Applications.xml
         >
         --verifys
```

If there are any errors within your XML document, they are listed so that you can address them.

Note If you create an object with read-only attributes (such as ICS_Notification), those attributes may cause a runtime error.

Running sm_xml to import XML

XML exporter command parameters describes the command parameters to use with **sm_xml**.

Table 6-1. XML exporter command parameters

Parameter	Description	Optional flag
Options:		
--server=<name>	Name of the target manager. By default, this manager name is INCHARGE.	-s
--broker=<location>	Alternate location of the VMware Smart Assurance broker. Must be typed as host:port. The default is the broker specified for the manager at installation.	-b
Commands:		
import [--exit-on-error] [--verify] <input-xmlfile>	Import an XML document called <input-xmlfile>. Optionally, the XML can be verified without importing it to the manager using --verify. Additionally, using --exit-on-error, the XML can be imported, but the import process will stop when an error is detected.	
Command arguments:		
<input-xmlfile>	The XML file used to import the VMware Smart Assurance topology.	N/A

Table 6-1. XML exporter command parameters (Continued)

Parameter	Description	Optional flag
--exit-on-error	This option allows you to import the XML, but stops the import process when a runtime error is encountered. By default, without this command option, a runtime error is reported, but the import process resumes at the next element. **Note** Anything imported prior to the runtime error remains in the repository.	-e
--verify	This option validates the XML file against the ICIM/XML DTD specification, but does not import the XML to the Manager.	N/A

For example, to import a file called Applications.xml into a Manager called INCHARGE, use the following command:

```
sm_xml import --exit-on-error Applications.xml
```

Handling runtime errors

A runtime error is generated if the XML importer cannot execute the instructions encoded in the XML document. By default, the importer skips any element (or subelement) containing an error. It continues with the next element and generates an error report at the end.

There are two ways to avoid the runtime error:

- --verify: Without importing the ICIM/XML document, run the XML to see if it will be validated by the parser.
- --exit-on-error: By default, objects will be imported as they are interpreted from the XML document and any invalid objects will be identified in a summary file. Use this command option to quit the import process when the adapter detects an error.

  Note Any objects imported before the error was detected will remain in the repository.

Understanding the sm_edit Utility

7

As part of the VMware Smart Assurancedeployment and configuration process, you need to modify certain files. User modifiable files include configuration files, rule set files, templates, and files (such as seed files, and security configuration files) that contain encrypted passwords. Original versions of these files are installed into appropriate subdirectories under the BASEDIR/smarts/ directory.

Original versions of files should not be altered. If a file requires modification, it must be stored as a local copy of the file in BASEDIR/smarts/local or one of its subdirectories.

This chapter includes the following topics:

- Example 1
- Example 2

Example 1

Modify serverConnect.conf

VMware Smart Assurance software is designed to first search for user modifiable files in BASEDIR/smarts/local or one of its subdirectories. If a modified version of a file is not found in the local area, VMware Smart Assurance software then searches appropriate nonlocal directories.

To facilitate proper file editing, VMware, Inc. provides the sm_edit utility with every VMware Smart Assurance products. When used to modify an original version of a file, this utility automatically creates a local copy of the file and places it in the appropriate location under BASEDIR/smarts/local. This ensures that the original version of the file remains unchanged. You can invoke sm_edit from the command line.

To invoke the sm_edit utility from the command line, specify the path and the name of the file to edit under BASEDIR/smarts. If multiple VMware Smart Assurance products are running on the same host, invoke sm_edit from the bin directory of the product with the files to edit.

Example 2

Edit a configuration file

To edit the configuration file for connecting to a server (serverConnect.conf), invoke the sm_edit utility as follows:

```
/opt/InCharge/SAM/smarts/bin>sm_edit conf/serverConnect.conf
```

In this example, the sm_edit utility automatically creates a local copy of the serverConnect.conf file in the /opt/InCharge/SAM/smarts/local/conf directory and opens the file in a text editor. If a local version of the file already exists, the sm_edit utility opens the local version in a text editor. In addition, sm_edit creates any necessary subdirectories.

The VMware Smart Assurance System Administration Guide provides additional information about the sm_edit utility.