Sun N1 Service Provisioning System 5.2 XML Schema Reference Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 819–4451–10
April 2006

060331@14558

Contents

Examples

Preface

The *Sun N1 Service Provisioning System 5.2 XML Schema Reference Guide* provides detailed information about the XML schemas used to define components, component types, plans, plug-ins, and plug-in user interfaces.

Who Should Use This Book

Anyone who develops components, plans, or plug-ins for the Sun N1™ Service Provisioning System environment might need to use this book.

Before You Read This Book

You should already be familiar with the general concepts and tasks in the Sun N1 Service Provisioning System environment, as explained in these documents:

- *Sun N1 Service Provisioning System 5.2 System Administration Guide*
- *Sun N1 Service Provisioning System 5.2 Operation and Provisioning Guide*
- *Sun N1 Service Provisioning System 5.2 Plan and Component Developer's Guide*
- *Sun N1 Service Provisioning System 5.2 Plug-In Developer's Guide*

How This Book Is Organized

Chapter 1 provides an overview of the XML schemas in the Sun N1 Service Provisioning System product.

Chapter 2 provides detailed information about common elements.

Chapter 3 provides detailed information about the elements and attributes that are used to define components and component types.

Chapter 5 provides detailed information about the elements and attributes that are used to define resource descriptor files.

Chapter 4 provides detailed information about the elements and attributes that are used to define execution plans.

Chapter 6 provides detailed information about the elements and attributes that are used to describe a plug-in.

Chapter 7 provides detailed information about the elements and attributes that are used to define an interface to a plug-in.

Appendix A provides detailed information about the compatibility of changes among components.

Documentation, Support, and Training

Sun Function	URL	Description
Documentation	`http://www.sun.com/documentation/`	Download PDF and HTML documents, and order printed documents
Support and Training	`http://www.sun.com/supportraining/`	Obtain technical support, download patches, and learn about Sun courses

Typographic Conventions

The following table describes the typographic changes that are used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.

TABLE P–1 Typographic Conventions *(Continued)*

Typeface or Symbol	Meaning	Example
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. Perform a *patch analysis*. Do *not* save the file. [Note that some emphasized items appear bold online.]

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

◆ ◆ ◆ CHAPTER 1

XML Schema Overview

This chapter provides an overview of the XML schemas that are used by the Sun N1 Service Provisioning System (N1 SPS).

The following topics are covered in this chapter:

Note – In this book, the terms "derived component," "child component," and "parent component" refer to component inheritance relationships, not to component composition relationships.

Service Provisioning Languages and Schemas

The N1 SPS software uses XML to implement plans, components, resource descriptors, and plug-in definitions. Each of these N1 SPS constructs use a specific kind of XML schema.

The N1 SPS product includes these XML schemas:

- `component.xsd` – Schema used to define components and component types. See Chapter 3.
- `plan.xsd` – Schema used to define execution plans. See Chapter 4.
- `planCompShared.xsd` – Schema that contains elements common to plans and components. See Chapter 2.
- `resourceDescriptor.xsd` – Schema used to define resource descriptors. See Chapter 5.
- `plugin.xsd` – Schema used to define a plug-in in a plug-in descriptor file. See Chapter 6.

- `pluginUI.xsd` – Schema used to define a user interface to a plug-in in the N1 SPS browser interface. See Chapter 7.

Each schema is described in detail in the referenced chapter of this book. General information that relates to all of the schemas is contained in the remainder of this chapter.

Requirements for Locales and Character Sets

Plans and components can include multibyte data. If a plan or component is authored in XML, the input files must be in UTF-8 format or use a byte order mark at the start of the file to signify their Unicode encoding. When plans and components are downloaded, they are always written in UTF-8 format. If a simple component refers to a configurable resource, that resource should be encoded in the native encoding of the master server or should use a byte order mark to note its encoding.

Pattern Matching

A number of element attributes can contain regular expression patterns. Unless otherwise specified, these patterns are glob-style patterns rather than fully generic regular expressions. Thus, an asterisk (`*`) is used to match zero or more characters, and a question mark (`?`) is used to match exactly one character. Characters listed inside of square brackets (`[]`) match any one of the enclosed characters. Characters that are separated by a hyphen (`-`) match any character that is lexically between the range of characters and includes the characters specified.

For example, `[ab]` matches either `a` or `b`. `[a-z]` matches any lowercase character. In this case, only the strict ASCII characters are matched, but not extended variants of the characters such as those with accents.

To match all Unicode letters, the pattern would have to include a POSIX character class like the Perl 5 `regex [[:lower]]`. You can also include non-ASCII characters directly, for example, `[eé]` matches either `e` or `é`.

Variables and Parameter Passing

Both plans and components can declare variables that are used by their steps.

Component variables are evaluated and bound when the component is installed. Thus, if a step in a component control block refers to a component-scoped variable, the value used is the same as when the component was installed.

Plan variables, however, are evaluated and bound each time the plan is run. If a step in a plan refers to a plan variable, the value used is the value defined at the time the plan was run. Thus, the value might vary from one run to another.

A plan can declare both parameters and variables. The value of a *variable* is defined at the point of declaration based on the values of other variables and constants. A *parameter* is a special kind of

variable whose value is defined by the caller. In the case of a top-level plan, the caller is the user who initiates the plan run. For each parameter that is declared by the plan, the user specifies the value for that parameter before running the plan. When a plan is invoked as the result of an `<execSubplan>` call, the plan that contains the call must explicitly pass values for each parameter that is declared by the called plan.

The `<install>`, `<uninstall>`, `<snapshot>`, and `<control>` blocks can declare parameters and local variables. As with plans, local variable values are locally defined, while parameter values are defined based on the values passed by the caller of the block. Both types of values can vary each time a plan is run.

You cannot reassign the value of a parameter.

The `<assign>` step and `<assign*>` elements provide a means for assigning new values to local variables. If an enclosing block is not allowed to declare new variables, then only those variables found in an enclosing local scope, if any, may be changed.

Component Compatibility

Compatibility is a concern when you modify or create new versions of a component that has already been deployed. Each time you modify a component and check it in, you create a new version of that component. When you modify a component, you must ensure that other objects that use or reference that component are not broken as a result of the changes. The ways in which you can use or reference components are by using dependencies, component targeters, component containment, and inheritance.

The N1 SPS product supports the following types of component compatibility:

- *Call compatibility* describes the set of changes that you can make to a component and still ensure that relationships that exist through dependencies and component targeters are not violated.
- *Install compatibility* describes the set of changes that you can make to a component and still ensure that relationships that exist through inheritance and component containment relationships are not violated.

Different versions of components can be deployed to different parts of the data center at different points in time. Thus, you should be aware of compatibility requirements and understand the way in which changes to one component might affect other existing components. In certain cases, the N1 SPS product enforces compatibility requirements, while in other cases you must ensure that the new component is compatible.

For a list of the types of changes that can be made to a component, see Appendix A.

Call Compatibility

A component *B* is said to be *call compatible* with component *A* if uses of *A* can be safely replaced with uses of *B* in the following cases:

- A call to a control block of *A*
- A call to an uninstall block of *A*
- A call to a snapshot block of *A* (using `addSnapshot`)
- A check for dependency of *A* (using `<checkDependency>` or `<createDependency>`)
- A reference to a variable of *A* (using `config gen :[component:A:var]`)

Call compatibility is also known as API compatibility or interface compatiblity.

Note – Usually, two call-compatible components are different versions of a component that are in the same version tree. However, the second component can also be in a distinct version tree if it is an instance of the first component.

The N1 SPS product enforces call compatibility for components that provide system services. When a system service is updated to refer to a new component, the new component must be call compatible with the original component. This policy ensures that clients of the system service can continue to function properly when the system service is upgraded.

The N1 SPS product also optionally verifies call compatibility when it resolves components that are referenced by certain installed component targeters. See "Installed Component Targeters" on page 47.

Though not required, ensure that a component is call compatible with earlier versions of itself.

Install Compatibility

A component can be *install compatible* with another component. The first component must be call compatible with the other component. Uses of the other component must also be able to be replaced safely with uses of the first component in these cases:

- When making a call to an install block of the other component
- When a component extends from the other component
- When a component contains a reference to the other component

Install compatibility is also known as structural compatibility.

Any existing installed component can be safely replaced by another install-compatible component. You do not have to modify the data structures that describe how the original was installed. Call compatibility is a much weaker statement, because the call-compatible component might need to be reinstalled to properly update the data structures.

Note – For install compatibility to hold, both components *must* belong to the same version tree. They cannot be components from two distinct version trees.

The N1 SPS product only enforces install compatibility for components that serve as types, which are called *component types*. When a component type is updated to refer to a new version of a component, the new version is install compatible with the original version. Thus, you can make install-compatible updates to component types without rebuilding and reinstalling all of the existing components that have been derived from that type.

If you make a change to a component type that is not install compatible, you must create a new component type in a new version tree with a new name. In such cases, the new component type can maintain call compatibility with the original by extending from the original component type. To easily identify the relationship between types, use a versioning system to encode component type names. For example, the component type names `EJB-1.0` and `EJB-1.1` are an easy way to indicate that `EJB-1.1` is a later version of the `EJB-1.0` component type.

Install compatible implies call compatible, but the reverse is not true. Also, if a component is not call compatible, it cannot be install compatible.

Targetable Components

The presence of a `<targetRef>` element indicates that the component, once installed, can be used as a deployment target for other components. This targetability is achieved by associating a unique virtual host or physical host with each installed instance of the component.

By using targetable components, plans can logically target an installed component by targeting its associated host. Usually, a targetable component creates a virtual host. However, you can have the targetable component create a physical host, as well. A physical host is useful for models in which the associated host has its own remote agent. Such is the case for a component that models a Solaris™ 10 Zone.

A component that defines a `<targetRef>` element is called a *targetable component*. When a targetable component is installed, it creates a host that serves as a deployment target for other components. When a targetable component is uninstalled, the host it created is automatically deleted. Such hosts still appear in the list of hosts on the Hosts page, but cannot be deleted and are restricted in the types of edits that can be made to them.

Common Attribute Types

An attribute type serves as a constraint on the value of a plan or component attribute. If an attribute does not list a specific type, its value is unconstrained.

The following sections describe the format of the attribute types that are used by the schema. `\p{N}` represents all Unicode numbers, while `\p{L}` represents all Unicode letters.

`entityName` Attribute Type

Attributes of type `entityName` have a maximum length of 512 characters and match the following pattern:

```
[\p{N}\p{L}-_. ]+
```

An example of an `entityName` would be `_Continent_Region.database server`.

As a special case, dot (`.`) and dot-dot (`..`) are not permitted to be entity names.

`systemName` Attribute Type

Attributes of type `systemName` consist of a *simpleSystemName* that has a maximum length of 64 characters, and optionally, a *pluginName* that also has a maximum length of 64 characters, as follows:

simpleSystemName
pluginName#simpleSystemName

Where *simpleSystemName* matches the following pattern:

```
[\p{L}_][\p{N}\p{L}-_. +]*
```

An example of a `systemName` would be *simpleSystemName* of `CENTRAL_AMERICA.TEXAS_systemusers.500k+` and *pluginName* of `com.sun.sjsas81` for a `pluginName#simpleSystemName` of `com.sun.sjsas81#CENTRAL_AMERICA.TEXAS_systemusers.500k+`.

`identifier` Attribute Type

Attributes of type `identifier` have a maximum length of 512 characters and match the following pattern:

```
[\p{L}_][\p{N}\p{L}_]*
```

An example of an `identifier` would be `NORTH_AMERICA_800AAA_555ABCD_`.

`pathName` Attribute Type

Attributes of type `pathName` have a maximum length of 512 characters and match one of the following patterns:

/
/pathPart

where *pathPart* is `[\p{N}\p{L}-_. ]+`

You can use the / separator to string together *pathPart*s, such as */pathPart/pathPart/pathPart*.

As a special case, dot (.) and dot-dot (. .) are not permitted to be included in *pathPart*.

`pathReference` Attribute Type

Attributes of type `pathReference` have the following syntax:

`pathReference`:
    absolutePath
    relativePath

absolutePath:
    /
    /*relativePath*

relativePath:
    relativePathStart
    relativePathStart/*relativePath*

relativePathStart:
    .
    ..
    `pathPart`

`modifierEnum` Attribute Type

Attributes of type `modifierEnum` have either `ABSTRACT` or `FINAL` as their value. In general, a value of `ABSTRACT` indicates that the associated entity must be overridden by a derived component. A value of `FINAL` indicates that the associated entity cannot be overridden.

`accessEnum` Attribute Type

Attributes of type `accessEnum` have one of these values:

- `PUBLIC` – Indicates that the associated entity can be accessed by any object
- `PROTECTED` – Indicates that the associated entity can be accessed by derived components and other entities that are in the same path
- `PATH` – Indicates that the associated entity can be accessed by other entities that are in the same path
- `PRIVATE` – Indicates that the associated entity can be accessed by the declaring component

`version` Attribute Type

Attributes of type `version` match the following pattern:

```
[0-9]+\.[0-9]+
```

Examples of `version` are lowest, `0.0`, to highest, `9.9` value.

`schemaVersion` Attribute Type

Attributes of type `schemaVersion` can have only one of these values, `5.0` , `5.1`, or `5.2`.

`HostEntityName` Attribute Type

Attributes of type `HostEntityName` have a maximum length of 64, and can include any combination of Unicode letters and numbers.

`pluginName` Attribute Type

Attributes of type `pluginName` have a maximum length of 64, and can include any combination of Unicode letters and numbers.

`pluginHostEntityName` Attribute Type

Attributes of type `pluginHostEntityName` have a maximum length of 64, and can include any combination of Unicode letters and numbers.

◆ ◆ ◆ CHAPTER 2

Shared Schema Used by Components and Simple Plans

This chapter describes the steps, targeters, and operators that can be used by both simple plans and components:

Unless indicated, the attributes that are described in this chapter cannot reference component-scoped substitution variables.

Shared Steps

This section lists the steps that can be used in a component or a simple plan.

<assign> Step

Use the <assign> step to reassign a value to a previously declared variable. It is a shared step that may be used in either a component or a simple execution plan. It is a check-in time error if the variable being assigned to is not an already defined variable in a plan or component local scope.

Attributes for the <assign> Step

The <assign> element has two required attributes:

- *varName* — A value of type Identifier that identifies the name of the local variable to which the value is assigned.
- *value* — A value of type String that contains the value to be assigned to the named variable.

<call> Step

Use the <call> step to execute a control block that is associated with a component that is already installed on the target host.

The <call> step has the following child elements:

- <argList> – An optional element that is a list of arguments to pass to the control block. If you specify this element, it can only appear one time.
- Installed component targeter – An optional element that identifies the component that contains the control block to execute. This element is optional if the <call> step appears in a component, but not if the step appears in a plan. If this element is omitted, the <thisComponent> targeter is used. If you specify this element, it can only appear one time. See "Installed Component Targeters" on page 47.
- <assign> — An optional element that specifies the name of the local variable to assign the value returned by the called block. The syntax and behavior of the <assign> child element is the same as that of the "<assign> Step" on page 24.

Attributes for the <call> Step

The <call> element has one required attribute of type entityName, *blockName*, which is the name of the control block to execute on the installed component.

<argList> Element

The <argList> element is a child of the <call>, <install>, <uninstall>, <execSubplan>, and <addSnapshot> steps. This element specifies a list of variables to be passed as arguments to the called service.

The called service declares the variables that it expects by using a <paramList> element. The collection of variables listed in the <argList> and in the called <paramList> need not be the same.

For each variable declared in the <paramList> that does not have a default value, a corresponding variable of the same name must be included in the <argList>. If this condition is not met, a preflight error is raised at plan runtime. The provisioning system runs through each variable in the called <paramList> for which there is a corresponding variable in the <argList>. The value of the variable in <paramList> obtains the value of the corresponding variable in <argList> for the duration of the execution of the called service.

Variables in <argList> that do not correspond to a variable in the called <paramList> are silently ignored. Thus, you can redefine a service by adding parameters while still ensuring backward compatibility. So, one plan could call both the old and new versions of the same service.

The arguments of the <argList> element are expressed as attributes. The order in which the attributes appear is not significant. The following <argList> declares two arguments, *password* and *path*:

```
<argList password=":[password]" path="/tmp"/>
```

Attributes for the <argList> Element

The <argList> element must have at least one attribute. Each attribute is treated as a named variable to be passed to the called service. The name of each attribute must be an identifier without substitution variable references. The attribute name should correspond to the name of a parameter in the called service. The value of each attribute is an arbitrary string that can include references to variables in the enclosing scope, but not other arguments within the <argList>.

<checkDependency> Step

Use the <checkDependency> step to verify that a particular component has been installed on a target host. If an appropriate component has not been installed, the step fails and execution stops.

The dependency is checked by using the contained component targeter. If the targeter successfully resolves a component, the dependency is satisfied. Otherwise, the dependency failed.

The <checkDependency> step has one required child element, an installed component targeter, which identifies the component to check for dependency. See "Installed Component Targeters" on page 47.

<execJava> Step

The <execJava> step executes a Java™ executor instance on the target host. If the executor instance raises an exception, the step fails and execution stops.

The <execJava> step has the following optional child elements:

- <argList> – A list of arguments to pass to the executor instance. If you specify this element, it can only appear one time.

- <assignOutput> – The name of the local variable to assign the value of the InfoStream written by the called class. If no content is written to the stream, the empty string is assigned.

  The <assignOutput> element has one attribute, *varName*, whose value is the name of the local variable to which to assign the standard output of the called command.

- <assignError> – The name of the local variable to assign the value of the ErrorStream written by the called class. If no content is written to the stream, the empty string is assigned.

  The <assignError> element has one attribute, *varName*, whose value is the name of the local variable to which to assign the standard error of the called command.

Attributes for the <execJava> Step

The <execJava> element has the following attributes:

- *className* – A required attribute that is the full name of a public class with a public no-arg constructor that implements the ExecutorFactory interface. For more information, see "execJava API" in *Sun N1 Service Provisioning System 5.2 Plug-in Development Guide*. This attribute can reference simple substitution variables.
- *classPath* – An optional attribute that is the class path that contains the class named by the *className* attribute. If this attribute is omitted, the system class path of the remote agent is used. The class path format is a semicolon-separated list of absolute paths to Java Archive (JAR) files on the agent. This attribute can reference simple substitution variables.
- *timeout* – An optional attribute of type `positiveInteger`, which specifies the number of seconds to wait for the command to complete before timing out. If this attribute is omitted, the plan's <execNative> timeout period applies. The value should be greater than 0.
- *label* – An optional attribute of type `String` that defines a persisted label assigned to this <execJava> step used to more easily retrieve the output later. The `label` is not required to be unique.

When an <execJava> step is contained within a component, the step typically calls classes that are contained within one or more resources that are deployed by that component. When the step is contained within a plan, the classes called are already resident on the agent. The classes can be a system class of the agent itself or a resource that was deployed with an existing component.

execJava assign* Child Elements

The semantics of the <assign*> elements are similar to those of the <assign> step. A check-in error is triggered if the same variable name is specified for the <assignOutput> and <assignError> elements. The output and error streams are truncated in exactly the same way as the database output is currently truncated, that is, by specifying a value for `pe.maxOutputSnapshotBytes` in the configuration file.

If the variable indicated by `varName` for the <assign*> elements does not exist in a plan or component local scope or a plan local scope, a check-in error will result.

<execNative> Step

The <execNative> step executes a command that is native to the operating system on the target host. If the command produces an unexpected result, the step fails and execution stops.

The <execNative> step can have the following nested child elements:

- <env> – An optional element that specifies environment variables for the child process. For each environment variable, specify one <env> element.
- <background> – An optional element that specifies that the command is to run as a background process. If you specify this element, it can only appear one time.

Note – If you specify the <background> element, you must also specify the <outputFile> and <errorFile> elements. If you use the <background> element, you cannot use the <assignOutput>, <assignError>, or <assignStatus> elements.

- <outputFile> – An optional element that is the name of the file in which to store standard output from the command. If you specify this element, it can only appear one time. This element is required if the <background> element is specified.
- <errorFile> – An optional element that is the name of the file in which to store standard error output from the command. If you specify this element, it can only appear one time. This element is required if the <background> element is specified.
- <assignOutput> – The name of the local variable to assign the value of the InfoStream written by the called class. If no content is written to the stream, the empty string is assigned.

  The <assignOutput> element has one attribute, *varName*, whose value is the name of the local variable to which to assign the standard output of the called command.
- <assignError> – The name of the local variable to assign the value of the ErrorStream written by the called class. If no content is written to the stream, the empty string is assigned.

  The <assignError> element has one attribute, *varName*, whose value is the name of the local variable to which to assign the standard error of the called command.
- <assignStatus> – An optional element that specifies the name of the local variable to assign the value of the status code returned by the called command.

  The <assignStatus> element has one attribute, *varName*, whose value is the name of the local variable to which to assign the status value (completion code) of the called command.
- <inputText> – An optional element that specifies the text to be used as standard input to the command. If you specify this element, it can only appear one time. This element is mutually exclusive with the <inputFile> element.
- <inputFile> – An optional element that is the name of the file to be used as standard input to the command. If you specify this element, it can only appear one time. This element is mutually exclusive with the <inputText> element.
- <exec> – A required element that specifies the name of the executable to run. This element is mutually exclusive with the <shell> element.

- <shell> – A required element that specifies a shell command to run. This element can only appear one time. This element is mutually exclusive with the <exec> element.
- <successCriteria> – An optional element that specifies the criteria used to determine whether this step succeeded or failed. If you specify this element, it can only appear one time.

Attributes for the <execNative> Step

The <execNative> step has the following attributes:

- *userToRunAs* – An optional attribute that is the name of the user as which to run this command. If this attribute is omitted, the command is run as the value of `defaultUserToRunAs` in the configuration file. The value can be either a string user name or a numeric user ID. The value should be a nonempty string. This attribute can reference simple substitution variables.
- *dir* – An optional attribute that is the absolute path to the working directory for the command. If this attribute is omitted, the value defaults to the agent-specific configurable directory. This value should be a nonempty string. This attribute can reference simple substitution variables.
- *timeout* – An optional attribute of type `positiveInteger`, which specifies the number of seconds to wait for the command to complete before timing out. If this attribute is omitted, the plan's <execNative> timeout period applies. The value should be greater than 0.
- *label* – An optional attribute of type `String` that defines a persisted label assigned to this <execNative> step used to more easily retrieve the output later. The `label` is not required to be unique.

<env> Element

The <execNative> element can optionally have <env> elements to specify environment variables for the command. You can use <env> to supply new variables for the command's environment or to override existing variables.

The set of the command's environment variables is a union of the set of the remote agent's environment variables and the variables supplied by using the <env> elements.

Attributes for the <env> Element

The <env> element has the following attributes. They can reference simple substitution variables.

- *name* – A required attribute that is the name of the environment variable. The value should be a nonempty string.
- *value* – A required attribute that is the value of an environment variable. If the value is a `${`*var-name*`}` string it refers to the value of the remote agent's environment variable. If *var-name* refers to a variable that is overridden by an <env> element, the substituted value is the one that is defined in the remote agent's environment, not the overridden one. If the string `${` must appear in the value, escape this string by using `${{`. All instances of `${{` are replaced by `${`.

<background> Element

The <background> element is a child of the <execNative> element. When present, this element specifies that the command should be executed as a background process. This element has no attributes or child elements.

An <execNative> with a <background> element has the following constraints:

- <successCriteria> need not be specified. The step succeeds if no issues arise when starting the background process. If specified, the <successCriteria> is tested against the script that is used to run the command as a background process.
- No standard output or standard error output is captured for the executed command.

  By viewing the details in the browser interface, you see an exit status of 0 and the empty standard output and standard error output. However, if problems occurred when starting the native command in the background, a different exit status is shown. The value depends on the nature of the error and diagnostic output.
- The <outputFile> and <errorFile> elements must be specified. If the specified files already exist, they are overwritten.

<outputFile> Element

The <outputFile> element is a child of the <execNative> element. The <outputFile> element specifies the path to a local file on the agent in which to store the standard output of the command that is being executed. Specify the path to the local file by using the <outputFile> *name* attribute. Relative paths are interpreted as being relative to the command's working directory.

You must specify the <outputFile> element if you specify the <background> element.

If <outputFile> is not specified and the <successCriteria> element does not specify *outputMatches*, the command output is not stored and will be lost. If *outputMatches* is specified, the standard output is stored in a temporary file that is deleted after the command finishes executing.

Attributes for the <outputFile> Element

The <outputFile> element has one required attribute, *name*, which is the name of the file to which the standard output of the command is written. This value should be a nonempty string. This attribute can reference simple substitution variables.

<errorFile> Element

The <errorFile> element is a child of the <execNative> element. The <errorFile> element specifies the path to a local file on the agent in which to store the error output of the command that is being executed. Specify the path to the local file by using the <errorFile> *name* attribute. Relative paths are interpreted as being relative to the command's working directory.

You must specify the <errorFile> element if you specify the <background> element.

If <errorFile> is not specified and the <successCriteria> element does not specify *errorMatches*, the command output is not stored and will be lost. If *errorMatches* is specified, the error output is stored in a temporary file that is deleted after the command finishes executing.

Attributes for the <errorFile> Element

The <errorFile> element has one required attribute, *name*, which is the name of the file to which the standard error output of the command is written. This value should be a nonempty string. This attribute can reference simple substitution variables.

<inputText> Element

The <inputText> element is a child element of the <execNative> element. This child element specifies the arbitrary text that should be used as standard input to the command. The text is specified as the contents of this element.

```
<execNative>
    <inputText>
        ls -l | fgrep '*test*' | sort -u > file.out
    </inputText>
    <command exec="sh"/>
</execNative>
```

The body of the <inputText> element can be enclosed in a CDATA section to preserve the formatting of the input and to avoid parsing errors that might be caused by input that contains the & and < characters.

If the <inputText> is specified, it is always enclosed within a CDATA section when generating XML for the <execNative> step. All of the characters within <inputText> are passed as standard input to the command that is being executed. If <inputText> contains only white spaces, those white spaces are passed, exactly as specified, to the command that is being executed.

The contents of <inputText> are config-generated.

The <inputText> element has no attributes.

<inputFile> Element

The <inputFile> element is a child element of <execNative>. This child element specifies the path to a local file on the remote agent whose contents are used as standard input to the command being executed. The path to the local file is specified by using the <inputFile> *name* attribute.

Note – The <inputFile> element cannot be used with the <inputText> element in an <execNative> command.

Attributes for the <inputFile> Element

The <inputFile> element as one required attribute, *name*, which is the file to act as standard input to the command. This value should be a nonempty string. If a relative path is specified, it is relative to the command's working directory. This attribute can reference simple substitution variables.

<exec> Element

An <execNative> step can contain only one <exec> element. The <exec> element specifies the details of the native command to be executed.

The <exec> element contains the following attributes:

- A *cmd* attribute that specifies the name of the command to execute
- A set of nested <arg> elements for each of the arguments of the *cmd* command

<execNative> executes the command specified by *cmd* with the arguments in the order in which they are specified.

For example, the following <execNative> step executes the `ps -fu sps` command:

```
<execNative>
    <exec cmd="ps">
        <arg value="-fu"/>
        <arg value="sps"/>
    </exec>
</execNative>
```

The <exec> element has an optional child element, <arg>. Specify one <arg> element for each argument that you want to pass to the command.

The <arg> element has one required attribute, *value*, which is the argument value. This argument is supplied as the *n*th argument to the command, where <arg> is the *n*th child of the command element.This attribute can reference simple substitution variables.

Attributes for the <exec> Element

The <exec> element has one required attribute, *cmd*, which is the path of the command to execute. If the specified path is not absolute, the command is found by using the platform-specified `PATH` environment variable set for the remote agent. This value should be a nonempty string. This attribute can reference simple substitution variables.

<shell> Element

The <shell> element is a child element of <execNative>. The contents of the <shell> element specifies the command to be executed. The command to be executed is interpreted using an interpreter, which is specified by the *cmd* attribute. The command is executed by using `sh -c "command"` syntax for the platform. In this form, the *cmd* attribute must be specified to indicate the shell command to use to execute the command.

The following <execNative> example executes the /usr/bin/bash -c ‘ls -l | fgrep ‘*test*’ | sort -u > file.out’ command.

```
<execNative>
    <shell cmd=”/usr/bin/bash -c”>
        ls -l | fgrep ‘*test*’ | sort -u > file.out
    </shell>
</execNative>
```

To preserve formatting and to avoid XML parsing problems, the text contents of the command is always enclosed within a CDATA element when the XML representation is generated from the <execNative> step.

A command string cannot be empty or contain only white space characters. The command string is supplied exactly as it is specified, including surrounding white space, to the shell.

The contents of the <shell> element are config-generated.

Attributes for the <shell> Element

The <shell> element has one required attribute, *cmd*, which is the shell command in the sh -c syntax. The string should not contain any embedded quote characters. The string is parsed to retrieve the shell name and the arguments by using white space as delimiters. For example, /usr/bin/bash -c. This value should be a nonempty string. This attribute can reference simple substitution variables.

<successCriteria> Element

The <successCriteria> element is a child element of <execNative>. This element specifies the criteria to be used to evaluate whether an <execNative> step executed successfully. If this element is not specified, the default value is <successCriteria status=”0”/>.

If the specified <successCriteria> is empty, it is ignored.

If you specify <successCriteria/>, the step always succeeds no matter what output or exit code the command generated.

Attributes for the <successCriteria> Element

The <successCriteria> element has the following optional attributes. If more than one of the *status*, *outputMatches*, and *errorMatches* attributes are specified, they are ANDed together.

- *status* – An optional attribute of type integer, which is the desired exit status of the command. This value should be a positive integer.
- *outputMatches* – An optional attribute that is a regular expression to match the standard output that is generated by the command. This value should be a nonempty string. This attribute can reference simple substitution variables.

- *errorMatches* – An optional attribute that is a regular expression to match the standard error output that is generated by the command. This value should be a nonempty string. This attribute can reference simple substitution variables.
- *inverse* – An optional attribute of type `Boolean`, which, if set to `true`, negates each of the conditionals specified in `<successCriteria>`. The default value is `false`. The step succeeds only if the conditions that are specified through each attribute of `<successCriteria>` are not met.

  For example, an `<execNative>` step with the following `<successCriteria>` succeeds if *status* is not 1, the standard output does not match `bin`, and standard error does not match `none`.

  ```
  <successCriteria status="1" outputMatches="bin"
  errorMatches="none" inverse="true"/>
  ```

  A `<successCriteria>` element that contains only the *inverse* attribute is saved without the *inverse* attribute. So, if you specify the following statement, the element is stored as `<successCriteria/>`, and the `<successCriteria>` element is ignored.

  ```
  <successCriteria inverse="true"/>
  ```

<if> Step

This step is used to conditionally execute a block of steps. This step has no attributes. Its child elements are `<condition>`, `<then>`, and `<else>`. The `<condition>` and `<then>` elements must appear one time. If you specify the `<else>` element, it can only appear one time.

If the contents of the `<condition>` element evaluate to true, the steps of the `<then>` block are executed. Otherwise, the steps of the `<else>` block are executed, if present.

EXAMPLE 2–1 Using the `<if>` Step

The following example shows an `<if>` step that is used to conditionally restart.

```
<if>
    <condition><istrue value=":[restart]"/></condition>
    <then>
        <call blockName="restart"/>
    </then>
</if>
```

<condition> Element

The `<condition>` element is a child of the `<if>` step and specifies a Boolean expression. This element has no attributes and must contain exactly one Boolean operator child element. See "Boolean Operators" on page 55.

<then> Element

The `<then>` element is a child element of the `<if>` step. This element specifies the steps to execute if the associated condition is true. The `<then>` element can contain any number of steps that are permitted within the scope of the block that contains the `<if>` step.

<else> Element

The <else> element is a child element of the <if> step. This element specifies the steps to execute if the associated condition is not true. The <else> element can contain any number of steps that are permitted within the scope of the block that contains the <if> step.

<pause> Step

The <pause> step pauses the execution of a plan for a specified amount of time. For example, you might use <pause> to make a plan wait for required services to come online after being started.

The <pause> element has one required attribute of type `positiveInteger`, *delaySecs*, which is the number of seconds to wait.

<processTest> Step

The <processTest> step is used to verify that a particular process is running on the target host. If the specified process does not exist, the step fails and execution stops.

Note – The <processTest> step only applies to UNIX® systems.

Attributes for the <processTest> Step

The <processTest> step has the following attributes:

- *delaySecs* – A required attribute of type `positiveInteger`, which is the number of seconds to wait before testing to see whether the process exists.
- *timeoutSecs* – A required attribute of type `positiveInteger`, which is the number of seconds to wait for the process to come online before failing. Time starts after the delay has completed.
- *processNamePattern* – A required attribute, which is a glob-style pattern to use to match the specified process name. This attribute can reference simple substitution variables.
- *user* – An optional attribute that is a glob-style pattern to use to match the name of the process owner. If this attribute is omitted, the process owner is not considered as part of the test. This attribute can reference simple substitution variables.

<raise> Step

The <raise> step is a step that always fails, though it can be caught and handled by a <try> step. See "<try> Step" on page 43.

The <raise> step is used to indicate a failure condition without having to construct an artificial step to do so. This step most often appears within a <catch> block to propagate an error condition after cleaning up. See "<catch> Element" on page 44.

Attributes for the <raise> Step

The <raise> element has one optional attribute, *message*, which is a message that describes the error condition. By default, the message is a generic system-specified message. This attribute can reference simple substitution variables.

EXAMPLE 2–2 Using the <raise> Step

The following example shows how the <raise> step is used to repropagate an error condition from within a <catch> block, after noting the error in a log.

```
<control blockName="default">
    <try>
        <block>
            <!-- some arbitrary processing here -->
        </block>
        <catch>
            <!-- note error in log -->
            <execNative>
                <exec cmd="appendLog">
                    <arg value="an error occurred"/>
                </exec>
            </execNative>
            <!-- rethrow error -->
            <raise/>
        </catch>
    </try>
</control>
```

MS Windows: <reboot> Step

This step causes the agent to reboot before running the rest of the plan. You can only use this step on Microsoft Windows (MS Windows) based systems. If encountered on a system other than MS Windows, an error is issued. If you reboot an agent that resides on the same host as the master server, an error is issued. These errors are preflight errors.

The <reboot> step has one optional attribute of type `positiveInteger`, *timeout*, which is the maximum number of seconds to wait for the server to reboot. If this attribute is omitted, the timeout period specified by the plan's <execNative> applies.

<retarget> Step

This step changes the execution target for a set of steps. Retarget steps can be nested.

The <retarget> step has the following child elements:

- `<varList>` – An optional element that lists the local variables available to the steps in the `<retarget>` block. The variables are evaluated within the scope of the new target host. If you specify this element, it can only appear one time.
- steps – Any number of steps to execute on the new target host. The steps can be any that are permitted within the scope of the block that contains the `<retarget>` step. You can specify more than one step. For an example, see Example 2–3.

Attributes for the `<retarget>` Step

The `<retarget>` step has a required attribute, *host*, which is the target host on which the contained steps should be executed. This attribute can reference simple substitution variables.

This attribute is used by the `<retarget>` step, as well as by various component targeter elements. Its value is the name of a host, which can include substitution variable references. The value can also include the symbolic name `/` to reference the root physical host of the current execution target, or `..(/..)*` to reference a parent host of the current execution target.

Note – When a component targeter specifies a *host* attribute, it is semantically equivalent to enclosing the containing step in a `<retarget>` step.

`<retarget>` Step Execution Semantics

When a `<retarget>` step is encountered, the *host* attribute is first evaluated in the context of the current host of the caller.

If the value of the *host* attribute is different from the name of the current host, the provisioning system takes the following steps:

1. Resolves the host name to an actual host. If no such host exists, an error is issued.
2. Verifies that the current user has "execute" permission on the given host for the plan's folder or for the component that contains this step. If not, an error is issued.
3. Verifies that the following conditions are met for the root physical host of the host:

   - It contains a remote agent.
   - You can connect to it.
   - It is updated and prepared.

   If these conditions are not met, an error is issued.
4. Obtains a lock on the host, while retaining locks on all previously visited hosts. If the current execution thread already locks the host, this operation is effectively a no-op. If the host is already locked by another execution thread, this operation blocks until the host is unlocked. If the request for a lock would result in a deadlock, an error is issued.
5. Resets the host to become the new "current" host. The "physical" host is reset based on the new "current" host. The "initial" host does not change.

After the previous steps complete, variables specified in the <varList> element are evaluated in the context of the new current host. Local variables can hide variables of enclosing scopes. After the variables are evaluated, each of the steps is executed in the context of the new current host.

Finally, if the retarget operation changed the current host, it is unlocked, as appropriate, and the current host is reset to the current host of the caller. If the host was locked previously in the current execution thread, it remains locked until the block that first acquired the lock completes.

You can use an empty <retarget> block to verify that the current user has the appropriate permissions and that the host is properly prepared.

EXAMPLE 2–3 Using the <retarget> Step

This example shows a "restart" control service that you might find on a WebLogic managed server. This control service is implemented by calling a control on the administrative server to stop the managed server. Then it makes a call on the local server to start the server.

The adminHostName variable is evaluated on the current host of the caller, which is assumed to be the virtual host that contains the managed server. The domainName variable is evaluated on the retargeted host, which is assumed to be the virtual host that contains the administrative server. The ADMIN_SERVER component is also resolved on the retargeted host.

```
<control name="restart">
    <varList>
        <var name="adminHostName" default=":[target:adminHostName]"/>
    </varList>
    <retarget host=":[adminHostName]">
        <varList>
            <var name="domainName" default=":[target:domainName]"/>
        </varList>
        <call blockName="stopServer">
            <argList serverName=":[serverName]"
              domainName=":[domainName]"/>
            <installedComponent name="ADMIN_SERVER"/>
        </call>
    </retarget>
    <call blockName="start"/>
</control>
```

<return> Step

Use the <return> step element to stop execution of the current block and optionally to return a *returns* value to the calling step. The <return> step element may appear only within the following elements:

- installSteps
- uninstallSteps
- control

- retarget
- then
- else
- block
- catch
- finally
- simpleSteps
- dependantCleanup

The presence of the *returns*=`true` attribute in an `install`, `uninstall`, or `call` block declaration mandates a corresponding uncaught `<raise>` or a `<return>` step in each execution path of the block definition.

The `<return>` step may appear anywhere in the block definition and generally follows the semantics for Java return statements. Depending on the complexity of the block, more than one `<return>` step may be required to ensure that all execution paths return a value. If additional steps are indicated subsequent to a return, a check-in error will be thrown, unless the `<return>` step appears in a `<try>` block and the additional steps are enclosed in the corresponding `<finally>` block. An execution path may also be terminated by a `<raise>` step, only if the `<raise>` step is not caught and handled in the current block scope. A `<return>` step may be used in a block even if the block does not specify a *returns* attribute, but in this case must not specify a value to return. If the `<return>` step specifies a value, but no *returns* attribute was specified for the block, a check-in error results.

Attributes for the `<return>` Step

The `<return>` step has one attribute value of type String. This value is the value to return. If no value is specified, the block stops execution without returning a value.

EXAMPLE 2–4 Using the `<return>` Step in `<try>` and `<finally>` Blocks

The following example shows <return> steps in `<try>` and `<finally>` blocks. The value `b` will be returned (as in Java).

```
<try>
    <block>
        <return value="a"
    </block>
    <catch/>
    <finally>
         <return value="b">
    </finally>
</try>
```

EXAMPLE 2–5 Using the `<return>`Step Conditionally

The following is an example of the `<return>` step used conditionally (line numbers were added for emphasis). If `var1` is `true`, then the `<return>` step at line 5 would skip the remainder of the steps in the block. Lines 8 and 9 would not be executed at all. If `var1` is not `true`, then the `<return>` step at line 9 would be the last step executed in the block.

EXAMPLE 2–5 Using the <return>Step Conditionally *(Continued)*

```
1. <control name = "blah" returns="true">
2.   <varList>
3.   <var name="var1" default="val1"/>
4.   <var name="var2" default="val2"/>
5.   </varList>
6.   <if>
7.      <condition><istrue value=":[var1]"/></condition>
8.      <then>
9.         <return value=":[var2]">
10.     </then>
11.  </if>
12.  <assign varName="var2" value="new value"/>
13.  <return value=":[var2]"/>
14.</control>
```

EXAMPLE 2–6 Using the <return> Step Conditionally Within a <try> Block

In contrast, the following example shows a conditionally used <return> step within a <try> block. Regardless of whether or not var1 is true, line 15 will always be the last line executed.

```
1. <control name="blah" returns="true">
2.   <varList>
3.   <var name="var1" default="val1"/>
4.   <var name="var2" default="val2"/>
5.   </varList>
6.   <try>
7.      <block>
8.          <if>
9.              <condition><istrue value=":[var1]"/></condition>
10.             <then>
11.                 <return value=":[var2]">
12.             </then>
13.         </if>
14.         <assign varName="var2" value="new value"/>
15.         <return value=":[var2]"/>
16.     </block>
17.  <catch/>
18.  <finally>
19.      <pause delaySecs="5"/>
20.  </finally>
21.   </try>
22.</control>
```

EXAMPLE 2–7 Using the <return> Step Within an inlineSubplan

The following example shows a <return> step occurring within the <SimpleSteps> of an inlineSubplan or execSubplan body. Control is returned to the next step in the set of steps that call the inlineSubplan or execSubplan. The next step after the <return> step at line 4 is line 7.

```
1. <complexSteps>
2.   <inlineSubplan>
3.       <simpleSteps>
4.           <return/>
5.       </simpleSteps>
6.   </inlineSubplan>
7.   <inlineSubplan>
8.       <simpleSteps>
9.            <pause delaySecs="1"/>
10.      </simpleSteps>
11.  </inlineSubplan>
12.</complexSteps>
```

<sendCustomEvent> Step

The <sendCustomEvent> step is used to generate a custom event with a particular message. This step can be used in conjunction with the notification rules module to send an email any time this step is encountered on a particular host.

The <sendCustomEvent> step has one required attribute, *message*, which is the message to include as the event text. This attribute can reference simple substitution variables.

<transform> Step

The <transform> step is used to perform a text-based transformation to a file on the target host. Currently, the provisioning system supports Perl-type and XSLT-based transformations.

If the output file exists on the target host before the transformation, the permissions and ownership of the file are preserved. However, if the output file is new, it inherits the default RA permissions.

The <transform> child elements specify the transformation to be applied to the input file. The child elements can be any one of the following elements:

- A single <stylesheet> element that defines the XSLT transformation to apply to the input source
- One or more <subst> elements that define Perl-like substitution patterns to apply sequentially to the input source
- A single <source> element that names the external file that contains the transformation
- Empty, which means that the contents of the input file are copied directly to the output file

You might use an empty element to extract from or write to `zip` archive files.

Attributes for the `<transform>` Step

The `<transform>` step has the following attributes:

- *input* – An optional attribute that is the generalized path of the file on the target host on which to apply the transformation. If this attribute is omitted, input is read from the output file. This attribute can reference simple substitution variables.
- *output* – A required attribute that is the generalized path of the file on the target host to which to write the result of the transformation. This attribute can reference simple substitution variables.

The *input* and *output* attributes can reference the same or distinct files. The value of these attributes is a generalized path that can include `zip` archives or `zip` derivatives, such as JAR, as directory elements. For example, a path might be `webapp/myapp.jar/config.xml`.

`<stylesheet>` Element

The `<stylesheet>` element is a child of the `<transform>` step. This element specifies an XSLT transformation to apply to the input source. At most one `<stylesheet>` element can appear as a child of a particular `<transform>` element. You cannot use the `<stylesheet>` element in conjunction with other child elements.

The `<stylesheet>` element is an XSLT Version 1.0 element as defined by the `http://www.w3.org/1999/XSL/Transform` name space. For more information, see Version 1.0 of the *XSL Transformations (XSLT)* specification at `http://www.w3.org/TR/xslt.html`. Only the XSLT `<stylesheet>` element is accepted as a child of the `<transform>` element. In particular, neither the XSLT synonym `<transform>` nor the simplified XSLT transform syntax described in Section 2.3 of the XSLT specification is supported as a child of the `<transform>` element.

The `<stylesheet>` element body can include substitution variable references if the body is still a valid XSLT without first undergoing variable substitution.

When a `<stylesheet>` element is used as a transformation, the input file must be written in XML. For more information, see Version 1.0 of the *XSL Transformations (XSLT)* specification.

EXAMPLE 2–8 Using the `<stylesheet>` Element

This transformation changes each `a` to a `b` in the `/etc/hosts` file. The `hosts` file is overwritten with these changes.

```
<transform output="/etc/hosts">
    <xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:template match="/">
            <xsl:for-each select="a">
                <xsl:value-of select="b"/>
            </xsl:for-each>
        </xsl:template>
```

EXAMPLE 2–8 Using the <stylesheet> Element *(Continued)*

```
    </xsl:stylesheet>
</transform>
```

<subst> Element

The <subst> element is a child of the <transform> step. This element specifies a Perl-like substitution pattern to apply as a transformation. One or more <subst> elements can appear as children of the <transform> element, but they cannot be used in conjunction with other child elements. When more than one <subst> element appears, they are applied sequentially.

All occurrences of the pattern in the input file are replaced, including multiple occurrences on a line.

For information about the supported syntax, see Java documentation (class `java.util.regex.Pattern`).

Attributes for the <subst> Element

The <subst> element has the following attributes:

- *match* – A required attribute that is a case-sensitive Perl-like regular expression that is sought after in the input. This attribute can reference simple substitution variables.
- *replace* – A required attribute that is a Perl-like replacement value that is substituted for each occurrence of the pattern given by *match*. This value is not interpreted verbatim: the $*n* construct is interpreted as the *n*th parenthetical expression inside the matching expression. This attribute can reference simple substitution variables.

EXAMPLE 2–9 Using the <subst> Element

The following transformation converts all occurrences of the string `127.0.0.`*xxx* to `10.10.0.`*xxx* in the `/etc/hosts` file:

```
<transform output=”/etc/hosts”>
    <subst match="127\.0\.0\.(\d+)"
    replace="10.10.0.$1"/>
</transform>
```

<source> Element

The <source> element is a child of the <transform> step. This element specifies an external file on the target host that contains the transformation to be applied to the input file. At most one <source> element can appear as a child of a particular <transform> element. You cannot use the <source> element in conjunction with other child elements.

Configuration generation is not performed on the specified source file as part of the <transform> step. However, the specified source file can be a config-type resource file that is deployed as part of a component installation. In such as case, substitution variables that are contained in the source file would have been substituted when the file was deployed.

Attributes for the <source> Element

The <source> element has the following attributes:

- *type* – A required element that is the type of transformation that is contained in the specified file. The following values are permitted:
  - PERL – A Perl-like transformation that is similar to that of the <subst> element. In this case, the format of the specified file should be similar to the following format:

    ```
    <?xml version='1.0'?>
    <transform>
        <subst match="127\.0\.0\.(\d+)"
        replace="10.10.0.$1"/>
    </transform>
    ```

    Perl-type external transformation files can contain any number of <subst> elements.
  - XSLT – An XSLT transformation. In this case, the specified file contains a standard XSLT Version 1.0 transformation as defined by the name space http://www.w3.org/1999/XSL/Transform. Unlike inline transformations, which only permit the XSLT <stylesheet> element, XSLT transformations that are contained in external source files can include any valid top-level XSLT transformation element. Such elements are <stylesheet>, <transform>, and simplified XSLT syntax. The simplified XSLT syntax is described in Section 2.3 of the XSLT specification.
- *name* – A required attribute that is the name of the file on the target host that contains the transformation. The contents of the file must correspond to the type defined by the *type* attribute. The name cannot include zip archives as directory elements. This attribute can reference simple substitution variables.

<try> Step

The <try> step is used to specify typical error handling and cleanup logic for a block of steps. This step has no attributes. This step has a required <block> element, and two optional elements: <catch> and <finally>.

The <try> step has the following child elements:

- <block> – A required element that consists of the steps initially executed.
- <catch> – An optional element that contains the steps to execute in case of a typical error. You must specify this element if the <finally> element is not specified. If you specify this element, it can only appear one time.
- <finally> – An optional element that contains the steps to execute regardless of typical errors. You must specify this element if the <catch> element is not specified. If you specify this element, it can only appear one time.

The <try> step is executed as follows:

- The steps in the <block> element are executed in order until all have completed or until a step fails.
- If and only if a <catch> element is defined and the execution of the block element terminated with a step failure other than plan abort or plan timeout, the steps in the <catch> element are executed in order until all have completed or a step fails.
- If a <finally> element is defined, its steps are executed in order until all have completed or a step fails. These steps are executed regardless of the success of the previous two elements unless either of them failed due to plan abort or plan timeout.
- If any step within the <try> block fails with a plan abort or plan timeout, execution fails. No other steps in the <try> block are executed and the execution of the <try> step ends with a failure.

The <catch> element is used to suppress and recover from errors that are encountered in the <block> element. The <finally> element is used to unconditionally perform some cleanup, regardless of whether typical errors were encountered.

The <try> step either succeeds or fails as follows:

- If the <try> step contains *only* a <finally> element, it fails if the execution of either the <block> or the <finally> element fails.
- If the <try> step contains *only* a <catch> element, it fails only if the <catch> element executes and fails, or the <block> element fails due to plan abort or plan timeout.
- If the <try> step contains both a <catch> and a <finally> element, it fails only if one of the following situations occurs:
  - The <catch> or the <finally> element executes and fails.
  - The <finally> element executes and fails.
  - The <block> element fails due to plan abort or plan timeout.

Failures in the <block> element are suppressed by the presence of a <catch> element.

<block> Element

The <block> element is a child element of the <try> step. This element specifies the primary steps that are executed by the <try> step. The <block> element contains one or more steps that are permitted within the scope of the block that contains the <try> step.

<catch> Element

The <catch> element is a child element of the <try> step. This element specifies the steps to execute if a typical error occurs while executing the steps of the <block> element. The <catch> element can contain any number of steps that are permitted within the scope of the block that contains the <try> step.

The <catch> element suppresses typical errors of the <block> element and defines typical error-recovery actions. When the <catch> element is empty, it only suppresses typical errors.

<finally> Element

A child element of the <try> step. <finally> specifies the steps to execute regardless of whether a typical error occurred earlier within the <try> or <catch> steps. An *atypical error* can be a plan abort or a plan timeout, in which case the steps of the <finally> element are skipped. <finally> can contain any number of steps that are permitted within the scope of the block that contains the <try> step.

The <finally> element is used primarily to specify cleanup steps that should always be run, regardless of an error. To run cleanup steps in response to an error, use the <catch> element instead.

EXAMPLE 2–10 Using the <try> Step

In this example, the component installation consists of resource deployment followed by a restart. In this case, the component is considered to be installed after its resources are deployed, and a failure during restart should not affect its installed state. Typical errors can be suppressed during the restart as follows:

```
<installSteps blockName="default">
    <deployResource/>
    <try>
        <block>
            <call blockName="restart"><thisComponent/></call>
        </block>
        <catch/><!-- suppress all typical errors -->
    </try>
</installSteps>
```

You can use <try> blocks to model intelligent auto-upgrades. Version 1.1 of a component has two different installation routines. One performs a fresh installation of the component. The other performs an upgrade installation if Version 1.0 of that component was previously installed. You can model this situation in a single installation block, as follows:

```
<installSteps blockName="default">
    <try>
        <block>
          <checkDependency>
              <installedComponent name="foo" version="1.0"/>
          </checkDependency>
          <!-- 1.0 installation exists, do upgrade -->
        </block>
        <catch>
          <!-- 1.0 installation doesn't exist, do fresh installation -->
        </catch>
    </try>
</installSteps>
```

The <finally> block is most often used to clean up temporary resources. The following example creates a temporary file, processes it, and then removes it.

EXAMPLE 2–10 Using the <try> Step *(Continued)*

```
<control blockName="default">
    <varList>
        <var name="file" default="/tmp/file.txt"/>
    </varList>
    <execNative outputFile=":[file]">
        <exec cmd="ls"><arg value="-l"/></exec>
    </execNative>
    <try>
        <block>
            <!-- process file in some way -->
        </block>
        <finally>
            <execNative>
                <exec cmd="rm"><arg value=":[file]"/></exec>
            </execNative>
        </finally>
    </try>
</control>
```

<urlTest> Step

This step is used to verify that the contents of a particular URL match an expected pattern. If the desired pattern is not matched, the step fails and execution stops.

Attributes for the <urlTest> Step

The <urlTest> step has the following attributes:

- *delaySecs* – A required attribute of type `positiveInteger`, which is the number of seconds to wait before testing the URL contents.
- *timeoutSecs* – A required attribute of type `positiveInteger`, which is the number of seconds to wait to receive the URL contents before failing. Time starts after the delay has completed.
- *URL* – A required attribute that is the URL whose contents should be tested. Currently, only the HTTP protocol is supported. This attribute can reference simple substitution variables.
- *pattern* – A required attribute that is a glob-style pattern that is expected to match the contents of the URL that is being tested. This value supports multibyte encoding. This attribute can reference simple substitution variables.

Installed Component Targeters

This section describes the elements that specify an installed component as the target of a step, such as a control service call. All targeters cannot be used with all targeted steps. Each targeter specifies the steps with which it can be used.

<installedComponent> Installed Component Targeter

The <installedComponent> element identifies a particular installed component that is assumed to be installed on the target host.

This element can be used as a targeter for the <checkDependency>, <createDependency>, <call>, <uninstall>, and <addSnapshot> steps.

This targeter matches the specified component directly and cannot be used to match derived instances of that component. To target components that are derived from a particular type, use the <systemType> targeter.

Attributes for the <installedComponent> Targeter

This targeter has the following attributes:

- *name* – A required attribute of type `entityName`, which is the name of the installed component.
- *path* – An optional attribute of type `pathReference`, which is the path of the component. If this attribute is omitted, the path of the containing entity is assumed.
- *version* – An optional attribute of type `version`, which is the version of the installed component. If this attribute is omitted, the most recently installed component, regardless of the version, is used.
- *versionOp* – An optional attribute that specifies the operator to use when comparing the *version* attribute with versions of components that are installed on the target host. If more than one installed component applies, the most recently installed component is used. These values are permitted: =, >=, and >. The >= operator is used by default. If *version* is not specified, *versionOp* is ignored.

- *onlyCompat* – An optional attribute that specifies the components that should match.

  If the value is `true`, only components that are call compatible with the component of version *version* should be matched. A component of version *version* must exist. By default, the value is `false`. If *version* is omitted, this element is ignored.
- *installPath* – An optional attribute that is the install path of the installed component. If this attribute is omitted, the most recently installed component in any path is used. The value is converted to universal format prior to component resolution. See "Universal Install Path Format" on page 52. This attribute can reference simple substitution variables.
- *host* – An optional attribute that is the host on which the component is installed. By default, *host* is the current host. See the description of the *host* attribute in "Attributes for the <retarget> Step" on page 36. This attribute can reference simple substitution variables.

<systemService> Installed Component Targeter

The <systemService> element identifies a particular system service component that is assumed to be installed on the current physical host.

This element can be used as a targeter for the <checkDependency>, <createDependency>, <call>, <uninstall>, and <addSnapshot> steps.

Use of the <systemService> targeter implicitly retargets to the root physical host of the current host. If you need to target a system service on a different host, a <retarget> step must be used. You cannot otherwise specify a new host within the <systemService> targeter.

Attributes for the <systemService> Targeter

The <systemService> targeter has one required attribute of type `systemName`, *name*, which is the name of the system service component. If the system service is defined by a plug-in, the system service name should be prefixed with the plug-in name, such as *pluginName#serviceName*.

<systemType> Installed Component Targeter

The <systemType> element identifies a component that is an instance of a particular type that is assumed to be installed on the target host. If more than one installed component matches the specified criteria, the component that was most recently installed is used.

This element can be used as a targeter for the <checkDependency>, <createDependency>, <call>, <uninstall>, and <addSnapshot> steps.

Attributes for the <systemType> Targeter

The <systemType> targeter has the following attributes:

- *name* – A required attribute of type `systemName`, which is the name of the system type component. If the system type is defined by a plug-in, the system type name should be prefixed with the plug-in name, such as *pluginName#typeName*.

- *installPath* – An optional attribute that is the install path of the desired component. The value is converted to universal format prior to component resolution. See "Universal Install Path Format" on page 52. This attribute can reference simple substitution variables.
- *host* – An optional attribute that is the host on which the component is installed. By default, *host* is the current host. See the description of the *host* attribute in "Attributes for the <retarget> Step" on page 36. This attribute can reference simple substitution variables.

<thisComponent> Installed Component Targeter

The <thisComponent> element specifies that the component that contains the step should be used as the target of the step. Only steps that are contained in a component can use this targeter. This element has no attributes.

This element can be used as a targeter for the <call>, <uninstall>, and <addSnapshot> steps.

If the steps listed do not contain a component targeter element, <thisComponent> is assumed by default.

<superComponent> Installed Component Targeter

<superComponent> specifies that the base component of the component that contains the step should be used as target of the step. Only steps that are contained in a derived component can use this targeter. This element has no attributes.

This element can be used as a targeter for the <call>, <uninstall>, and <addSnapshot> steps.

This targeter always binds to the base component's definition of the step in question, even if the derived component overrides it.

<nestedRef> Installed Component Targeter

The <nestedRef> element identifies a nested component reference that is declared or inherited by the current composite component. Only steps that are in a composite component can use this targeter.

This element can be used as a targeter for the <checkDependency>, <call>, <uninstall>, and <addSnapshot> steps.

The specified component reference must already be installed by the calling component. If the referenced component is not installed, an error is issued. If the nested component reference was installed on a host other than the current target host, use of the <nestedRef> targeter implicitly retargets the associated step to that host.

Attributes for the <nestedRef> Targeter

The <nestedRef> targeter has one required attribute of type identifier, *name*, which is the name of a nested component reference in this component.

<allNestedRefs> Installed Component Targeter

The <allNestedRefs> element identifies the set of all nested component references that are declared or inherited by the current composite component. Only steps within composite components can use this targeter.

This element can be used as a targeter for the <call>, <uninstall>, and <addSnapshot> steps.

This targeter can identify any number of components. If it identifies no components, the step is a no-op. If it identifies more than one component, the step is semantically expanded as if a separate occurrence of the step that uses the <nestedRef> targeter exists for each of the identified components. The steps are executed serially rather than in parallel. The ordering of the steps varies based on the step type. If the execution of the step on one of the components causes an error, the step is not executed on the remaining matching components.

When used as a targeter for a <call> or <addSnapshot> step, this targeter matches all of the nested component references that are currently installed by this component. The component matches are in the order of installation.

When used as a targeter for an <uninstall> step, this targeter matches all of the nested component references that are currently installed by this component. The component matches are in the reverse order of installation.

<toplevelRef> Installed Component Targeter

The <toplevelRef> element identifies a top-level component reference that is declared or inherited by the current composite component. Only steps that are in composite components can use this targeter.

This element can be used as a targeter for the <checkDependency>, <createDependency>, <call>, <uninstall>, and <addSnapshot> steps.

This targeter is semantically equivalent to the <installedComponent> targeter, except that the *name*, *path*, and *version* attribute values are predefined based on the referenced component. See "<installedComponent> Installed Component Targeter" on page 47.

Attributes for the <toplevelRef> Targeter

The <toplevelRef> targeter has the following attributes:

- *name* – A required attribute of type identifier, which is the name of a top-level component reference in this component.

- *versionOp* – An optional attribute that specifies the operator to use when comparing the version of the referenced component with versions of components that are installed on the target host. If more than one installed component applies, the most recently installed component is used.

  These values are permitted: =, >=, and >. If this attribute is omitted, >= is used.
- *onlyCompat* – An optional attribute that specifies the components that are matched. If `true`, this attribute specifies that only components that are call compatible with the referenced component should be matched. The default value is `false`.
- *installPath* – An optional attribute that is the install path of the referenced component. If this attribute is omitted, the most recent installation of the referenced component in any path is used. The value is converted to universal format prior to component resolution. This attribute can reference simple substitution variables.
- *host* – An optional attribute that is the host on which the referenced component is installed. By default, *host* is the current host. See the description of the *host* attribute in "Attributes for the <retarget> Step" on page 36. This attribute can reference simple substitution variables.

<dependee> Installed Component Targeter

The <dependee> element identifies an installed component on which the calling component has a declared dependency that was created by <createDependency>. Only steps that are in components can use this targeter.

This element can be used as a targeter for the <call>, <uninstall>, and <addSnapshot> steps.

The <dependee> targeter has one required attribute of type `identifier`, *name*, which is the name of a dependency that is created by this component.

<allDependants> Installed Component Targeter

The <allDependants> element identifies the set of installed components that have a declared dependency on the calling component. These dependencies were created by <createDependency>. Only steps in components can use this targeter.

This element can be used as a targeter for the <call>, <uninstall>, and <addSnapshot> steps.

This targeter functions similarly to the <allNestedRefs> targeter in that it causes the containing step to be mapped over all of the matching components. The order of the mapping over the dependant components is unspecified.

The <allDependants> targeter has one required attribute of type `identifier`, *name*, which is the name of a dependency that is created on this component by other components.

<targetableComponent> Installed Component Targeter

The <targetableComponent> element identifies a targetable component that is associated with a particular component targeting host.

This element can be used as a targeter for the <call>, <uninstall>, <checkDependency>, <createDependency>, and <addSnapshot> steps.

The <targetableComponent> targeter has one optional attribute, *name*, which is the name of a component targeting host. If this attribute is omitted, the value is the current target host. This attribute can reference simple substitution variables.

Universal Install Path Format

You can specify an install path within an installed component reference. In these cases, the *installPath* attribute value is converted to universal format before the installed component reference is resolved. This conversion occurs because the install path of the installed component is also stored in universal format.

In universal format, all occurrences of the path separator in the install path are replaced by a slash (/). The path separator is specific to the operating system that is running on the master server. Trailing slashes are dropped. The root install path (/) is not converted to the empty path.

When the Master Server application is running on UNIX based systems, trailing slashes are ignored. Thus, both /opt/apache/ and /opt/apache can be used to refer to the component that is installed in the /opt/apache directory.

Repository Component Targeters

This section describes the elements that specify a particular component that resides in the master server repository as the target of a step, such as install. All targeters cannot be used with all targeted steps. Each targeter specifies the steps with which it can be used.

<component> Repository Component Targeter

<component> specifies a particular component that is assumed to exist in the component repository. Only steps that are contained in a simple plan can use this targeter.

This element can be used as a targeter for the <install> step.

Attributes for the <component> Targeter

The <component> targeter has the following attributes:

- *name* – A required attribute of type entityName, which is the name of the component.
- *path* – An optional attribute of type pathReference, which is the path of the component. If this attribute is omitted, the path of the containing entity is assumed.
- *version* – An optional attribute of type version, which is the version of the component. If this attribute is omitted, the latest version of the component is used.
- *host* – An optional attribute that is the host on which the component should be installed. By default, *host* is the current host. See the description of the *host* attribute in "Attributes for the <retarget> Step" on page 36. This attribute can reference simple substitution variables.

<thisComponent> Repository Component Targeter

<thisComponent> specifies that the component that contains the step should be used as the target of the step. Only steps in a component can use this targeter. This element has no attributes.

This element can be used as a targeter for the <install> step.

If the steps listed do not contain a component targeter element, <thisComponent> is assumed by default.

<superComponent> Repository Component Targeter

<superComponent> specifies that the base component of the component that contains the step is the target of the step. Only steps that are contained in a derived component can use this targeter. This element has no attributes.

This element can be used as a targeter for the <install> step.

This targeter always binds to the base component's definition of the step in question, even if the derived component overrides it.

<nestedRef> Repository Component Targeter

<nestedRef> specifies a nested component reference that is declared or is inherited by the current composite component. Only steps in a composite component can use this targeter.

This element can be used as a targeter for the <install> step.

The specified component reference should not have been previously installed by the calling component. Otherwise, an error is issued. If a nested referenced component is to be installed on a different host, use a <retarget> step instead. You cannot otherwise specify a new host within the <nestedRef> targeter. You cannot install a nested component reference on more than one host for a given containing component.

The <nestedRef> targeter has one required attribute of type identifier, *name*, which is the name of a nested component reference that is in this component.

<allNestedRefs> Repository Component Targeter

<allNestedRefs> specifies the set of all nested component references that are declared or inherited by the current composite component. Only steps in a composite component can use this targeter.

This element can be used as a targeter for the <install> step.

This targeter can specify more than one component. If it specifies no components, the step is a no-op. If it specifies more than one component, the step is semantically expanded as if a separate occurrence of the step that uses the <nestedRef> targeter exists for each of the specified components. The steps are executed serially rather than in parallel. The ordering of the steps varies based on the step type. If the execution of the step on one of the components causes an error, the step is not executed on the remaining matching components.

When used as a targeter for an <install> step, this targeter matches all of the nested component references that are declared in this component. The component matches are in the order of declaration.

<toplevelRef> Repository Component Targeter

<toplevelRef> specifies a top-level component reference that is declared or inherited by the current composite component. Only steps in a composite component can use this targeter.

This element can be used as a targeter for the <install> step.

This targeter is semantically equivalent to the <component> targeter, except that the *name*, *path*, and *version* attribute values are predefined based on the referenced component. A top-level component reference can be installed one or more times on one or more hosts.

Attributes for the <toplevelRef> Targeter

The <toplevelRef> targeter has the following attributes:

- *name* – A required attribute of type `identifier`, which is the name of a top-level component reference in this component.
- *host* – An optional attribute that is the host on which the referenced component is to be installed. By default, *host* is the current host. See the description of the *host* attribute in "Attributes for the <retarget> Step" on page 36. This attribute can reference simple substitution variables.

Boolean Operators

This section describes elements that serve as Boolean operators. These elements appear in the <condition> element of an <if> step. See "<if> Step" on page 33. Boolean operators can evaluate only to true or false.

<istrue> Boolean Operator

This Boolean operator is used to determine whether a particular value is true. <istrue> has no child elements. <istrue> evaluates to true only if *value* equals `true`. The comparison is case-insensitive.

Attributes for the <istrue> Boolean Operator

The <istrue> operator has one required attribute, *value*, which is the value to compare to the string `true`. This attribute can reference simple substitution variables.

EXAMPLE 2–11 Using the <istrue> Boolean Operator

The following examples show how <istrue> is used and the results:

- The following statement evaluates to true.

```
<istrue value="True"/>
```

- The following statement evaluates to false.

```
<istrue value="yes"/>
```

- The following statement evaluates to true if *var* is `true`.

```
<istrue value=":[var]"/>
```

<equals> Boolean Operator

This Boolean operator is used to determine whether a particular value is equal to another value. This operator has the *value1*, *value2*, and *exact* attributes. This operator has no child elements, and evaluates to true only if *value1* and *value2* are equal. If *exact* is true, the values must be exactly the same, including case. If *exact* is false, the comparison is case-insensitive.

`<istrue value="..."/>` is a syntactic shorthand for the following statement:

```
<equals value1="..." value2="true"/>
```

Attributes for the <equals> Boolean Operator

The `<equals>` operator has the following attributes:

- *value1* – A required attribute that is a value to be compared. This attribute can reference simple substitution variables.
- *value2* – A required attribute that is the value to be compared. This attribute can reference simple substitution variables.
- *exact* – An optional attribute of type `boolean`, which is `true` if a case-sensitive match should be performed, `false` otherwise. Defaults to `false`.

EXAMPLE 2–12 Using the `<equals>` Boolean Operator

The following examples show how `<equals>` is used and the results:

- The following statement evaluates to true.

  ```
  <equals value1="True" value2="true"/>
  ```

- The following statement evaluates to false.

  ```
  <equals value1="True" value2="true" exact="true"/>
  ```

- The following statement evaluates to true.

  ```
  <equals value1="apple" value2="apple" exact="true"/>
  ```

- The following statement evaluates to false.

  ```
  <equals value1="apple" value2="orange"/>
  ```

- The following statement evaluates to true if *var1* is equal to *var2*.

  ```
  <equals value1=":[var1]" value2=":[var2]"/>
  ```

<matches> Boolean Operator

This Boolean operator is used to determine whether a particular value matches a pattern. This operator has the *value*, *pattern*, and *exact* attributes. This operator has no child elements and evaluates to true only if the value of *value* matches the glob-style pattern contained in *pattern*. If *exact* is true, the values must be a case-sensitive match. Otherwise, the values can be a case-insensitive match.

Attributes for the <matches> Boolean Operator

The <matches> operator has the following attributes:

- *value* – A required attribute that is the value to be matched against the pattern. This attribute can reference simple substitution variables.
- *pattern* – A required attribute that is the pattern to be matched. This attribute can reference simple substitution variables.
- *exact* – An optional attribute that is true if a case-sensitive match should be performed, false otherwise. Defaults to false.

EXAMPLE 2–13 Using the <matches> Boolean Operator

The following examples show how <matches> is used and the results:

- The following statement evaluates to true.

```
<matches value="True" pattern="true"/>
```

- The following statement evaluates to true.

```
<matches value="True" pattern="t*"/>
```

- The following statement evaluates to false.

```
<matches value="blue" pattern="*u"/>
```

- The following statement evaluates to true.

```
<matches value="True" pattern="t?ue"/>
```

- The following statement evaluates to false.

```
<matches value="Tue" pattern="t?ue"/>
```

- The following statement evaluates to false.

```
<matches value="True" pattern="t*" exact="true"/>
```

- The following statement evaluates to true if *var1* matches the pattern of *var2*.

```
<matches value=":[var1]" pattern=":[var2]"/>
```

<not> Boolean Operator

This Boolean operator negates the result of another Boolean operator. This operator has no attributes and has a single child element, which is one of the other Boolean operators. This operator evaluates to true only if the value of its contained operator is not true.

EXAMPLE 2–14 Using the <not> Boolean Operator

The following examples show how <not> is used and the results:

- The following statement evaluates to false.

```
<not><istrue value="True"/></not>
```

EXAMPLE 2–14 Using the <not> Boolean Operator *(Continued)*

- The following statement evaluates to true.

```
<not><equals value1="apple" value2="orange"/></not>
```

<and> Boolean Operator

This Boolean operator logically ANDs the results of other Boolean operators. This operator has no attributes and can contain any number of child elements, which are the other Boolean operators. The <and> operator evaluates to true only if all of its child elements evaluate to true.

EXAMPLE 2–15 Using the <and> Boolean Operator

The following examples show how <and> is used and the results:

- The following statement evaluates to true.

```
<and/>
```

- The following statement evaluates to true.

```
<and><istrue value="True"/></and>
```

- The following statement evaluates to false.

```
<and><equals value1="apple" value2="orange"/></and>
```

- The following statement evaluates to true.

```
<and>
    <matches value="apple" value2="ap*e"/>
    <istrue value="TRUE"/>
    <not><equals value1="apple" value2="orange"/></not>
</and>
```

- The following statement evaluates to false.

```
<and>
    <matches value="apple" value2="ap*e"/>
    <istrue value="TRUE"/>
    <equals value1="apple" value2="orange"/>
</and>
```

<or> Boolean Operator

This Boolean operator logically ORs the results of other Boolean operators. This operator has no attributes and can contain any number of child elements, which are the other Boolean operators. The <or> operator evaluates to true only if it contains at least one child element that evaluates to true.

EXAMPLE 2–16 Using the <or> Boolean Operator

The following examples show how <or> is used and the results:

- The following statement evaluates to false.

```
<or/>
```

- The following statement evaluates to true.

```
<or><istrue value="True"/></or>
```

- The following statement evaluates to false.

```
<or><equals value1="apple" value2="orange"/></or>
```

- The following statement evaluates to false.

```
<or>
    <matches value="apple" value2="p*e"/>
    <istrue value="FALSE"/>
    <equals value1="apple" value2="orange"/>
</or>
```

- The following statement evaluates to true.

```
<or>
    <matches value="apple" value2="p*e"/>
    <not><istrue value="FALSE"/></not>
    <equals value1="apple" value2="orange"/>
</or>
```

◆ ◆ ◆ CHAPTER 3

Component Schema

This chapter describes the XML schema used by components, and covers these topics:

Unless indicated, attributes described in this chapter cannot reference component-scoped substitution variables.

For an overview of the XML schema architecture, see Chapter 1.

<component> Element Overview

A component is enclosed within the <component> element. All versions of a component must have the same name and path. This element's attributes can reference component-scoped substitution variables.

Attributes for the <component> Element

The <component> element has the following attributes:

- *xmlns* – A required string that has a value of `http://www.sun.com/schema/SPS`.

- *xmlns:xsi* – A required string that has a value of `http://www.w3.org/2001/XMLSchema-instance`.
- *xsi:schemaLocation* – An optional string. The recommended value is `http://www.sun.com/schema/SPS component.xsd`.
- *access* – An optional attribute that specifies the accessibility of the component, which is how the component can be referenced by other components. These are the legal values:
  - `PATH` – The component can only be referenced by other components in the same path as this component. When `access="PATH"`, you cannot directly install the component, and it can only be included in other components by using a nested reference.
  - `PUBLIC` – The component can be referenced by any component and is not bound by the restrictions imposed by `PATH` access. This is the default value.
- *modifier* – An optional value of type `modifierEnum`, which specifies the following override requirements for the component:
  - `ABSTRACT` – Identifies the component as an *abstract component*. An abstract component serves only as a base component for other components to extend and cannot be installed. Only an abstract component is permitted to declare abstract child elements.
  - `FINAL` – Identifies the component as a *final component*, which means that the component cannot be extended by another component.

  If this attribute is omitted, the component can be extended and installed, which is the default.
- *name* – A required value of type `entityName`, which is the name of the component.
- *path* – An optional value of type `pathName`, which is the absolute path of the component. If this attribute is omitted, the root path (`/`) is the default value. The value must name a folder that exists at the time that the component is saved.
- *description* – An optional string that is a description of the component.
- *label* – An optional string, maximum length of 32 characters, that is a brief description of the component.
- *softwareVendor* – An optional string that is the vendor name of the software application that is modeled by the component.
- *author* – An optional string that is the name of the component's author.
- *version* – A required value of type `schemaVersion`, which is the version of the component schema. Currently, the only permitted values are `5.0`, `5.1`, and `5.2`.

  The 5.2 version of the schema is backward compatible with the 5.0 and 5.1 versions.
- *platform* – An optional string that specifies the name of the host set that includes the hosts that are valid physical targets on which this component can be installed.

  If this attribute is omitted, any host that contains the Remote Agent application and that is a supported platform is a valid physical target. Otherwise, the physical targets for any plan that installs this component must be a subset of the hosts that are contained in the specified host set. If the physical targets include a host that is not part of the specified host set, the plan issues a runtime error. These plan runtime errors are reported as preflight errors. A component save time

error is issued if you specify a name that does not correspond to a supported platform host set. Platform host sets are all prefixed with the *system#* plug-in name. If a component platform host set is unsupported, a new version of the component cannot be checked in until the platform is changed. Any operation on the existing component version that refers to the unsupported platform host set will fail.

- *limitToHostSet* – An optional string that specifies the name of the host set that contains the hosts that are valid targets for this plan.

  If this element is omitted, all hosts are valid targets. Otherwise, the specified targets must be a subset of the hosts that are included in the named host set. If the targets include a host that is not part of the specified host set, the plan issues a runtime error. These plan runtime errors are reported as preflight errors. A component save time error occurs when you specify a name that does not correspond to an existing, supported host set. If the specified host set is one that is defined by a plug-in, *pluginName* must be a prefix to the host set name, such as *pluginName#hostSetName*.

  These are the two main differences between the *platform* and the *limitToHostSet* attributes:

  - *platform* names one of the predefined platform host sets, whereas *limitToHostSet* names a user-defined host set. Therefore, if you want to limit installation based on a custom host set, use *limitToHostSet*.
  - When a component is targeted at a virtual host, *limitToHostSet* is tested against the virtual host, whereas *platform* is tested against the root physical host of that virtual host.

  Therefore, if you set *limitToHostSet* but not *platform*, a component can be installed on a particular set of virtual hosts that might reside on different physical platforms (as is the case with WebLogic applications). However, if you set *platform* but not *limitToHostSet*, a component can be installed on any host that is rooted by a physical host with the given platform. If you set both, you can constrain both degrees.
- *installPath* – A required string that is only for nonderived components. This path is used when the component is installed. For simple components, this value also serves as the root directory in which to install the component's resources. The path is stored in universal format when an instance of this component is installed. See "Universal Install Path Format" on page 52.

Except for *installPath* and *limitToHostSet*, component attributes are not inherited.

The *installPath* attribute is inherited and cannot be overridden by derived components. However, the base component can use component variables when specifying its value, and the value of these variables can be overridden.

The *limitToHostSet* attribute is inherited and can be overridden by derived components only if the base component did not specify *limitToHostSet*.

Because the *limitToHostSet* value names a mutable, user-managed entity, the relation of host set cannot be reasoned about in the same way as platforms.

The *platform* attribute is not inherited. However, the *platform* attribute value of a derived component can be no more general than that of the *base component*. If *platform* is not specified in a derived component, *platform* cannot be specified (or must be specified as any) in the base component.

Child Elements of the <component> Element

The <component> element has the following child elements, which must appear in the order shown. These child elements might have their own child elements, attributes, or both.

- <extends> – Declares the base component from which the component is derived
- <varList> – Lists the component-scoped variables that are used by the component and its resources
- <targetRef> – Declares that the component is "targetable"
- <resourceRef> – Specifies the resource managed by the component
- <componentRefList> – Lists the components that are referenced by this component
- <installList> – Contains one or more named blocks of <install> steps
- <uninstallList> – Contains one or more named blocks of <uninstall> steps
- <snapshotList> – Contains one or more named <snapshot> blocks
- <controlList> – Lists the <control> blocks that are available for the component
- <diff> – Lists the directives that are used by the comparison engine to perform comparisons on this component

<extends> Element

The <extends> element is an optional child of the <component> element. This element is used to declare the base component from which this component is derived. The base component cannot be final. If used, this element can only appear one time.

This component automatically inherits the attributes and elements of the base component. The component can selectively override certain aspects of the inherited data. Inheritance and override allowances are described by the description of the attribute or element.

A component is an *instance of* the component that it extends. A component is also an instance of the components that the base component is an instance of.

The <extends> element has one required child element, <type>, which specifies the base component. The <type> element must be used exactly one time per <component> element.

<type> Element

The <type> element names the base component type of this component. This element is a child of the <extends>, <componentRefList>, and <componentRef> elements.

The <type> element has one required attribute of type systemName, *name*, which is the name of the system type component that serves as the base type. If the specified type is one that is defined by a plug-in, *pluginName* must be prefixed to the type name, such as *pluginName#typeName*.

<varList> Element

The <varList> element is an optional child of the <component> element. This element declares the list of component-scoped substitution variables that are used by this component and by its configuration resources. If used, this element can only appear one time.

The <varList> element has one required child element, <var>, which declares a component substitution variable.

By default, a derived component inherits the accessible <varList> element contents of its base component. When a derived component declares a <varList>, its contents are effectively merged with those of the base component. The derived component can declare new <var> elements to override inherited ones, but a derived component cannot remove elements that are declared by the base component.

<var> Element

The <var> element is a child of the <varlist> element, which is a child of the <component> element. The <var> element declares a component substitution variable. For each substitution variable that you want to declare, you must specify the name of the variable and its default value.

By default, a *derived component* inherits all of the accessible variables from its base component, including access mode, modifier, default value, and prompt. The <var> element can appear one or more times in the <varList> element.

A derived component can define additional variables by using names that are not among those variables that have been inherited from the base component. A derived component can override the prompt, default value, modifier, and access mode of a nonfinal inherited variable by re-declaring a variable with the same name. When a variable is overridden, the entire contents of the variable must be re-declared, including the prompt, default value, access mode, and modifier. Only specify the default value if the overriding variable is nonabstract. The access mode can be no more restrictive than that of the base component.

When a variable is overridden, all references to the variable evaluate to the overridden value, even those that appear in the base component.

If the derived component is declared as nonabstract, any abstract variables declared by the base component must be overridden by the derived component.

Attributes for the <var> Element

The <var> element has the following attributes:

- *access* – An optional value of type accessEnum, which specifies the accessibility of the variable. This attribute can have one of the following values:
  - PUBLIC – Access is not restricted in any way, which is the default.
  - PROTECTED – Access is limited to derived components and entities that are in the same path.
  - PATH – Access is limited to entities that are in the same path.
  - PRIVATE – Access is limited to this component.
- *modifier* – An optional value of type modifierEnum, which specifies the override requirements for the variable. This attribute can have the following values:
  - ABSTRACT – The variable's *default* attribute is omitted and must be specified by a variable in nonabstract derived components. Variables can only be declared as abstract if the component is also declared as abstract. Abstract variables cannot be private. Nonabstract variables must declare a default value.
  - FINAL – The variable cannot be overridden by derived components.

  If the attribute is omitted, derived components can choose whether to override the variable.
- *name* – A required value of type identifier, which is the name of the substitution variable. Each variable name declared by a <var> element in the <varList> element must be unique.
- *default* – A required string for nonabstract variables, which is the default value of the substitution variable. This value can include references to other substitution variables, session variables, target host attributes, and installed component variables. However, an abstract variable cannot define a default value, so this attribute cannot be used for abstract variables.
- *prompt* – An optional string that is a user-readable description of the variable.

<targetRef> Element

The <targetRef> element is an optional child of the <component> element. This element declares that the component is targetable. A *targetable component* is one that automatically creates a physical host or a virtual host that is associated with the component. This host is created when the component is installed. If used, this element can appear only one time, and it must appear immediately after the <varList> element. This element can be used by both simple and composite components. However, the <varList> element can only be used in components that are being installed as top-level components.

The <targetRef> element has one optional child element, <agent>, which indicates whether the associated host is a physical host or a virtual host. If the <agent> element is present, the host is a physical host. The <agent> element body defines the configuration of the remote agent. If the element is not present, the host is a virtual host, which is the default.

Attributes for the <targetRef> Element

The <targetRef> element has the following attributes:

- *hostName* – A required value, which is the name of the host to create when you install this component. The name must be unique at the time of installation and must be a valid host name. This value can include component-scoped substitution variable references.
- *typeName* – An optional value of type systemName, which is the name of the host type to use for the associated host. The specified host type must exist at the time that the component is saved. If this host type is defined by a plug-in, the name should include *pluginName* as a prefix, such as *pluginName#typeName*.

  If this attribute is omitted, the value is system#crhost.

<agent> Element

The <agent> element is a child of the <targetRef> element. This element indicates that the associated host is a physical host. This element is optional and can be used only one time.

If this element is omitted, the associated host is created as a virtual host. If this element is used, the associated host is created as a physical host, and the <agent> element specifies the configuration of the associated remote agent.

This element is inherited by derived components. A derived component can declare a local <targetRef> element only if the base component did not. This means that a derived component cannot override an inherited <targetRef> element.

Attributes for the <agent> Element

The <agent> element has the following attributes. These attributes can reference component-scoped substitution variables.

- *connection* – A required value that specifies the connection type that is used to connect to the remote agent. This attribute can have one of the following values:
  - RAW
  - SSL
  - SSH
- *ipAddr* – A required value that is the IP address of the physical host. This value can be either a server name or an IP address. Server names must be resolvable to an IP address by the master server.
- *port* – An optional value that is the port on which the remote agent is listening. If *connection* is RAW or SSL, the default value of *port* is 1131. If *connection* is SSH, this attribute is ignored.
- *params* – An optional value that is at least one parameter that is used to connect to the remote agent.

<resourceRef> Element

The <resourceRef> element is an optional child of the <component> element. This element specifies the resource that is managed by the component. This element can only be used by a simple component. This element cannot be used in conjunction with the <componentRefList> element, which can only be used by a composite component. The configurable attributes of this element and its children can reference component substitution variables. Resources have implicit PUBLIC access mode. If used, this element can only appear one time.

A component is a simple component if it is derived from a simple component or if it is a nonderived component that contains a <resourceRef> element. A derived component can only contain a <resourceRef> element if it is derived from a simple component.

The <resourceRef> element has child elements, which must appear in the following order:

- <installSpec> – A required element for nonderived components that specifies how to install the resource. This element cannot be included in derived components.
- <resource> – A required element for nonabstract components that identifies the associated resource. This element cannot be included in abstract components.

By default, a derived component inherits the <resourceRef> element of its base component.

A derived component can override the modifier and the <resource> element of a nonfinal inherited <resourceRef> element by re-declaring the <resourceRef> element. When a <resourceRef> element is overridden, the <installSpec> element is omitted, as its contents cannot be overridden. The <resource> element is specified only if the overriding <resourceRef> is nonabstract.

When a <resourceRef> is overridden, all uses of the resource (including <deployResource> and <addResource>) resolve to the overridden value, even uses in the base component.

If the derived component is declared as nonabstract and the <resourceRef> element of the base component is abstract, the derived component must override the <resourceRef> element.

Attributes for the <resourceRef> Element

The <resourceRef> element has one optional attribute, *modifier*, which has a value of type modifierEnum. The *modifier* attribute specifies the following override requirements for the resource:

- ABSTRACT – The <resource> element of <resourceRef> is omitted and must be specified by a nonabstract derived component. A <resourceRef> can only be declared abstract if the component is also declared abstract. A nonabstract <resourceRef> must declare a <resource> element.
- FINAL – <resourceRef> cannot be overridden by derived components.

If this attribute is omitted, derived components can choose whether to override the <resourceRef>.

<installSpec> Element

The <installSpec> element is a child of the <resourceRef> element. This element specifies the way in which the associated resource is to be installed. This element is inherited by derived components and cannot be overridden. However, the base component can use component variables to specify values for <installSpec> attributes. The values of these variables can also be overridden.

Attributes for the <installSpec> Element

The <installSpec> element has the following attributes. These attributes can reference component-scoped substitution variables.

- *name* – A required string that is the name to use for the resource when it is installed.
- *path* – An optional string that is the path in which to install the resource. Relative directories are considered relative to the *installPath* attribute of the containing component. If this argument is omitted, the component's *installPath* attribute is used by default.
- *permissions* – An optional string that indicates the permissions to assign to the resource when installed.

  The string is in the format of an octal triplet, as defined by the UNIX `chmod` command. See the `chmod`(1M) man page. If this attribute is omitted, the resource is installed with default permissions.
- *user* – An optional string that is the owner of this resource when it is installed. If this attribute is omitted, the user is determined by the plan executor.
- *group* – An optional string that is the group to assign to this resource when it is installed. If this attribute is omitted, the group is determined by the plan executor.
- *deployMode* – An optional attribute that specifies the way in which the associated directory resource is deployed. This attribute is ignored if the resource is not a directory.
  - `ADD_TO` – The directory contents are added to any existing files in the target directory.
  - `REPLACE` – The directory contents replace all existing files in the target directory.

  If this argument is omitted, the default value, `REPLACE`, is used.
- *diffDeploy* – An optional value of type `boolean`, which specifies whether the resource should be deployed in differential deploy mode. If this attribute is omitted, differential deploy mode is disabled. If differential deploy mode is enabled, only resources that have not previously been deployed are deployed.

<resource> Element

The <resource> element is a child of the <resourceRef> element. This element identifies the resource to be deployed by the component.

If the referenced resource is a configurable resource, it can contain substitution variable references to any component-scoped variable that is accessible to the containing component.

Attributes for the <resource> Element

The <resource> element has the following attributes:

- *name* – A required string that is the full path name of a resource contained in a checked-in component.
- *version* – A required value of type version, which is the version of the resource that has been created by an earlier component check-in.

<componentRefList> Element

The <componentRefList> element is an optional child of the <component> element. This element specifies the list of components that are referenced by the component. This element cannot be used in conjunction with the <resourceRef> element. Configurable attributes of this element and its children can reference component substitution variables. If used, this element can only appear one time.

A component is a *composite component* if it is derived from a composite component or if it is a nonderived component that does not contain a <resourceRef> element. A *derived component* can only contain a <componentRefList> element if it is derived from a composite component.

The <componentRefList> element has the following optional child elements:

- <type> – Specifies the type that all referenced components must be instances of. If this element is omitted, referenced components can be of any type.

  This element is optional. If used, this element can appear only one time per <componentRefList> element.
- <componentRef> – A reference to a component. This element is optional. If used, this element can appear more than once.

By default, a derived component inherits the contents of the <componentRefList> element from its base component. When a derived component declares a <componentRefList>, its contents are effectively merged with those of the base component. The derived component can declare new <componentRef> elements and override inherited ones. However, the derived component cannot remove elements that are declared by the base component.

A derived component can override the <type> element that is declared by the <componentRefList> element of the parent component. This override is achieved by re-declaring the <type> in its <componentRefList>. In this case, the overridden type must be an instance of the original type or the original must not be specified. Furthermore, all referenced components must be instances of the overridden type, including those that are inherited from the base component.

Attributes for the <componentRefList> Element

The <componentRefList> element has one optional attribute, *modifier*, which specifies the override requirements of the resource. If this attribute is specified, the value must be FINAL, which means that derived components cannot declare new <componentRef> elements.

If this attribute is omitted, derived components can add new <componentRef> elements. In either case, derived components can override the <type> element and nonfinal inherited <componentRef> elements. If the base component's <componentRefList> *modifier* attribute is FINAL, the *modifier* attribute of the derived component must also be FINAL.

When *modifier* is FINAL for the <componentRefList> element, the *modifier* attribute of each contained <componentRef> is not necessarily FINAL.

<componentRef> Element

The <componentRef> element is a child of the <componentRefList> element. This element specifies a component that is referenced by this component. This element implies PUBLIC access.

The <componentRef> element has child elements, which must appear in the following order:

- <type> – Specifies the component type of which the referenced component must be an instance. It must be an instance of the component type that is specified by the enclosing <componentRefList>.

  This element is optional. If used, this element can appear only one time per <componentRef> element.

  If this element is omitted, the component type specified by the enclosing <componentRefList> is used.
- <argList> – An optional element that is a list of values to be used as component variable settings for the referenced component when it is installed.

  If used, this element can appear only one time as a child of the <componentRef> element.
- <component> – A required element that specifies the referenced component. You cannot use this element in abstract <componentRef> elements.

By default, a derived component inherits all the component references of its base component.

If the <componentRefList> element of the base component is nonfinal, a derived component can define additional component references by using names that are not among those of the component references that are inherited from the base component.

A derived component can override a component reference of a nonfinal inherited component reference by re-declaring a component reference that has the same name. When a component reference is overridden, the entire contents of the component reference must be re-declared. The overriding *installMode* must be the same as that of the original reference. The overriding <type>

element must be an instance of the original type. The overriding <argList> element is merged with the original. See "<argList> Element" on page 73. The <component> element is specified only if the overriding reference is nonabstract.

When a component reference is overridden, all uses of the component reference evaluate to the overridden value, including those in the base component.

If the derived component is declared as nonabstract, any abstract component references that are declared by the base component must be overridden by the derived component.

Attributes for the <componentRef> Element

The <componentRef> element has the following attributes:

- *modifier* – An optional attribute of type modifierEnum that specifies the override requirements of the component reference. This attribute has the following values:
  - ABSTRACT – The <component> child element of the <componentRef> element is omitted and must be specified by nonabstract derived components. A <componentRef> can only be declared abstract if the component is also declared abstract. A nonabstract <componentRef> must declare a <component> element.
  - FINAL – The <componentRef> cannot be overridden by derived components.

  If this attribute is omitted, derived components can choose whether to override the component reference.
- *name* – A required attribute of type identifier that specifies a local name for the referenced component. This name must be unique among all sibling <componentRef> elements.
- *installMode* – An optional attribute that specifies the way in which the referenced component should be installed and targeted thereafter. If this attribute is omitted, the value is NESTED.

  This attribute has the following values:
  - TOPLEVEL – If the referenced component is installed in this way, it can be used by any other component just as if it had been directly installed by a plan.
  - NESTED – If the referenced component is installed in this way, its installation is implicitly scoped to that of the referencing component. Its services are only available to the referencing component.

  A nested referenced component logically defines a finer-grained unit of functionality that is required by the referencing component. This functionality is not otherwise useful to other components. A top-level referenced component defines services that are used by the referencing component, but can also be used by other components.

  The lifetime of a nested referenced component is implicitly scoped to that of the referencing component. The nested referenced component can only be installed during the installation of the referencing component, and is implicitly uninstalled when the referencing component is uninstalled. In contrast, the lifetime of a top-level referenced component is not tied to that of the referencing component. The referencing component can install a top-level referenced

component when it is installed, or by other means if the top-level referenced component is already installed. When the referencing component is uninstalled, a top-level referenced component remains installed unless it is explicitly uninstalled by the referencing component. Other components are also permitted to uninstall the referencing component.

To refer to a component that defines a <targetRef> element, a TOPLEVEL <componentRef> must be used. A NESTED <componentRef> cannot be used.

<argList> Element

This <argList> element is a child of the <componentRef> element. This element specifies a list of values to be used as component variable settings for the referenced component when it is installed. The format of this <argList> is the same as that of the <argList> child element of the <call> step. See "<call> Step" on page 24.

If the reference is ABSTRACT, each attribute of the <argList> element names a component variable in the referenced component or the declared type. The value of the attribute for the <argList> element is the override value that should be used for the named component variable when the referenced component is installed.

When a component reference is overridden by a derived component, the <argList> element of the base and derived components are effectively merged. This merge is accomplished by applying the contents of the <argList> of the base component to the referenced component, then applying the <argList> of the derived component. When processing the <argList> of the base component reference, only variables that are defined in the declared type of the base component reference are considered.

Component variables that are not named in the <argList> use their default value when installed. If no <argList> is specified, the referenced component is installed and uses default values for all its variables. The variables of the referenced component named in the <argList> must be accessible to the referencing component. Furthermore, the variables must be declared with the PUBLIC or PROTECTED access mode, and cannot be FINAL.

For top-level referenced components, the <argList> element is only used if the referenced component is installed by the referencing component. The referenced component could have been installed by other means, in which case the <argList> has no bearing.

<component> Element

This <component> element is child of the <componentRef> element. This element identifies the referenced component. This element has the same structure as the <component> repository component targeter, except that the *host* attribute is not permitted. The referenced component version must exist in the repository at the time that the containing component is saved.

If the *version* attribute is not specified, the version is resolved to be the latest version of the referenced component that exists at the time that the containing component is saved. A save-time error occurs if no versions of the referenced component exist. After the containing component is saved, the versions of all the components it references are locked. The referenced components cannot be modified without creating a new version of the container component.

<installList> Element

The <installList> element is a child of the <component> element. This element contains one or more named blocks of install steps. Each block provides a different way to install the component. Many components have only one install block as a child of the <installSteps> element, described later in this section.

This element is required for nonderived components and optional for derived components. If used, this element can only appear one time.

You can use more than one install block when different steps are required for different installation environments. For example, you might create one install block to deploy an EJB application to a server cluster, another to deploy to a single managed server, another for initial install, and one to upgrade the application.

By default, a derived component inherits the accessible <installList> element contents of its base component. When a derived component declares an <installList>, its contents are effectively merged with those of the base component. The derived component can declare new <installSteps> elements and override inherited ones, but it cannot remove elements that are declared by the base component.

The <installList> element has one child element, <installSteps>, which lists the sequence of steps to be executed to install the component. The <installSteps> element can appear one or more times.

<installSteps> Element

The <installList> element has one child element, <installSteps>, which lists the sequence of steps to be executed to install the component. When an <install> step causes this component to be installed, the steps listed here are executed in order. Typically, the install steps of a simple component include a <deployResource> step. The install steps of a composite component include one or more <install> steps to install referenced components.

The <installSteps> element children consist of an optional <paramList> element followed by the body, which consists of an optional local <varList> element. The local <varList> element is followed by zero or more “shared” or “component install-only” steps. The body is not included if the install block is declared abstract.

By default, a derived component inherits all accessible install blocks of its base component.

A derived component is permitted to define additional install blocks by using names that are not among those of the install blocks inherited from the base component. A derived component can override a nonfinal inherited install block by re-declaring a block with the same name. Blocks are overridden by using name alone, and they cannot be overloaded based on parameters. When a block is overridden, the entire contents of the block must be re-declared, including the access mode, modifier, and parameters. The body is specified only if the overriding block is nonabstract. The access mode can be no more restrictive than that of the base component.

The signature of the overriding block in the derived component must be compatible with that of the base component. That is, any arguments that are acceptable to the base block must also be acceptable to the derived block.

A derived block is compatible with a base block when it does not declare a new required parameter and does not redefine a parameter to be required if that parameter is optional in the parent block.

The following signature changes are compatible:

- Removing a required or optional parameter
- Making a required parameter optional
- Adding an optional parameter

When a block is overridden, all references to the block evaluate to the overridden value, including those in the base component.

If the derived component is declared as nonabstract, any abstract blocks declared by the base component must be overridden by the derived component.

A block in a derived component is permitted to explicitly call into a block that is defined by the base component even if the derived component overrides the block that uses the `<superComponent>` targeter.

Attributes for the `<installSteps>` Element

The `<installSteps>` element has the following attributes:

- *access* – An optional attribute of type `accessEnum`, which specifies the accessibility of the install block. The following values are permitted:
  - `PUBLIC` – Access is unrestricted and is the default access mode.
  - `PROTECTED` – Access is limited to derived components and entities that are in the same path.
  - `PATH` – Access is limited to entities that are in the same path.
  - `PRIVATE` – Access is limited to this component.

  Only `PUBLIC` blocks can be run directly from the component.
- *modifier* – An optional attribute of type `ModifierEnum`, which specifies the override requirements for the install block. The following values are permitted:
  - `ABSTRACT` – The block cannot include a body. The body must be specified by nonabstract derived components. Install blocks can only be declared abstract if the component is also declared abstract. Abstract blocks cannot be private. Nonabstract blocks must declare a body.
  - `FINAL` – The install block cannot be overridden by derived components.

  If the *modifier* attribute is omitted, derived components can choose whether to override the block.
- *name* – A required attribute of type `entityName`, which is the name of the install block. The name must be unique among all install blocks in the containing `<installList>`.

- *description* – An optional attribute that is a string that describes the install block. This attribute is useful for documentation purposes.
- *returns* – An optional attribute that indicates that the block must return a value. Valid values are true or false. If `returns=true` is specified, then each execution step within the block must end with either a <return> step or an uncaught <raise> step. The default is `false`.

<paramList> Element

The <paramList> element is a child of the <installSteps>, <uninstallSteps>, <snapshot>, and <control> elements. This element declares a list of parameters that can be used by the steps of the enclosing element. The value of the parameters are defined by the caller based on the contents of the caller's <argList> element. For example, in the case of a <paramList> within an <installSteps> block, parameter values are defined based on the <argList> of the <install> step that invoked the <installSteps> block.

The steps of the enclosing element can use the following variables and parameters:

- Locally scoped variables that are declared in the local <varList> element
- Parameters that are declared in the <paramList> element
- Component-scoped variables that are declared in the component <varList> element of the enclosing component

If a <paramList> parameter has the same name as a component <varList> variable, the value of the parameter is used. In this case, the parameter is said to "hide" the component variable. Hiding is not permitted between local variables and parameters because their names must be distinct.

The <paramList> element has one child, <param>, which is required. This child element is a parameter declaration, which includes a name and a default value. Specify one <param> element for each parameter that you want to define.

<param> Element

The <param> element is a child of the <paramList> element. This element declares a parameter, which includes a name and a default value. The default value is only used if the caller does not explicitly pass a value for this parameter. If the default value is unspecified and the caller does not explicitly pass a value for this parameter, a preflight error occurs at plan runtime.

The <param> element includes the following attributes. Use the *prompt* and *displayMode* attributes when the containing install, uninstall, or control block is invoked directly by the user rather than from a plan or another component.

- *name* – A required attribute of type `identifier` that is the name of the parameter. The name must be unique among every other local variable and parameter that is declared by the enclosing element.
- *prompt* – An optional attribute that is a string that specifies the text to display in the user interface when prompting for the parameter value. If this attribute is omitted, the value of *name* is used.

- *default* – An optional attribute that is a string that specifies the default value of the parameter. The parameter can include references to component variables, target host attributes, session variables, and installed component variables, but not to other parameters.
- *displayMode* – An optional attribute that specifies the display mode of the parameter. The legal values are as follows:
  - `PASSWORD` – The user-specified value is hidden, which means that the password is not shown or is replaced by asterisks.
  - `BOOLEAN` – The parameter is specified by means of a check box.
  - `CLEAR` – The value is displayed as entered.

  If the value is `CLEAR` or `BOOLEAN`, it can be safely displayed as entered. If the attribute is omitted, the value is `CLEAR`.

Local <varList> Element

The local <varList> element is a child of the <installSteps>, <uninstallSteps>, <snapshot>, and <control> elements. This element declares a list of variables that can be used by the steps of the enclosing element. The values of the variables are defined at the point of declaration. They cannot be redefined.

The steps of the enclosing element can use the following variables and parameters:

- Locally scoped variables that are declared in the local <varList> element
- Parameters that are declared in the <paramList> element
- Component-scoped variables that are declared in the component <varList> element of the enclosing component

If a local <varList> variable has the same name as a component <varList> variable, the value of the local variable is used. In this case, the local variable is said to "hide" the component variable. Hiding is not permitted between local variables and parameters because their names must be distinct.

The local <varList> element has one required child element, <var>, which is a local variable declaration that includes a name and a default value. You can specify more than one <var> element.

Local <var> Element

The local <var> element is a required child of the local <varList> element and is used to declare a local variable name and its value.

The local <var> element has the following attributes:

- *name* – A required attribute of type `identifier`, which specifies the name of the local variable. The name must be unique among every other local variable and parameter declared by the enclosing element.
- *default* – A required attribute of type `String`, which is the default value of the local variable. This local variable can include the following references:

- Other local variables that were declared earlier
- Parameters
- Component variables
- Target host attributes
- Session variables
- Installed component variables

<uninstallList> Element

The <uninstallList> element is a child of the <component> element. This element contains one or more named <uninstall> step blocks, each of which provides a different way to uninstall the component. Many components have only one uninstall block as a child of this element. You can use more than one uninstall block when different steps are required for different installation environments.

For example, you might create one uninstall block to undeploy an EJB application to a server cluster and another to undeploy to a single managed server. Uninstall blocks often correspond one-to-one with install blocks, and in such cases, by convention use the same name to indicate the correspondence.

This element is required for nonderived components and optional for derived components. If used, this element can only appear one time.

The <uninstallList> element has a required child element, <uninstallSteps>. This child element is a named uninstall block that contains steps that can be executed to uninstall the component. Specify one <uninstallSteps> element for each way that you want to uninstall the component.

By default, a derived component inherits the accessible <uninstallList> element contents of its base component. When a derived component declares an <uninstallList>, its contents are effectively merged with those of the base component. The derived component can declare new <uninstallSteps> elements and override inherited ones, but it cannot remove elements that are declared by the base component.

<uninstallSteps> Element

The <uninstallSteps> element is a child of the <uninstallList> element. This element lists the sequence of steps to be executed to uninstall the component. When an <uninstall> step causes this component to be uninstalled, the steps listed in this element are executed in order. The <uninstallSteps> element of a simple component is permitted to include an <undeployResource> step, though it is not required. The <uninstallSteps> element of a composite component is permitted include one or more <uninstall> steps to uninstall the referenced components, though these steps are not required.

The <uninstallSteps> element children consist of an optional <paramList> element followed by a body. The body consists of an optional local <varList> element that is followed by an optional

<dependantCleanup> block, which is followed by zero or more “shared” or “component uninstall-only” steps. The body is not included if the uninstall block is declared abstract.

The following example shows the contents of a sample <uninstallSteps> element. Everything after </paramList> defines the body.

```
<uninstallSteps name="default">
  <paramList>
     <param name="param1"/>
   </paramList>
  <varList>
    <var name="var1" default="my var 1"/>
  </varList>
  <dependantCleanup>
    <uninstall blockName="default">
        <allDependants name="child2parent"/>
    </unisntall>
  </dependantCleanup>
  <undeployResource/>
</uninstallSteps>
```

By default, a derived component inherits all of the accessible uninstall blocks of its base component. Semantics for overriding an uninstall block are the same as those for overriding an install block.

Attributes for the <uninstallSteps> Element

The <uninstallSteps> element has the following attributes:

- *access* – An optional attribute of type accessEnum, which specifies the accessibility of the uninstall block. The following values are permitted:
  - PUBLIC – Access is unrestricted, which is the default.
  - PROTECTED – Access is limited to derived components and entities that are in the same path.
  - PATH – Access is limited to entities that are in the same path.
  - PRIVATE – Access is limited to this component.

  Only PUBLIC blocks can be run directly from the component.
- *modifier* – An optional attribute of type modifierEnum, which specifies the override requirements for the uninstall block. The following values are permitted:
  - ABSTRACT – The block cannot include a body because it must be specified by nonabstract derived components. Uninstall blocks can only be declared abstract if the component is also declared abstract. Abstract blocks cannot be private. Nonabstract blocks must declare a body.
  - FINAL – The uninstall block cannot be overridden by derived components.

  If this attribute is omitted, derived components can choose whether to override the block.
- *name* – A required attribute of type entityName, which is the name of the uninstall block. The name must be unique among all uninstall blocks in the containing <uninstallList>.

- *description* – An optional attribute, which is a string that describes the uninstall block. This attribute is useful for documentation purposes.
- *returns* – An optional attribute that indicates that the block must return a value. Valid values are `true` or `false`. If *returns*=`true` is specified, then each execution step within the block must end with either a <return> step or an uncaught <raise> step. The default is `false`.

<dependantCleanup> Element

The <dependantCleanup> element is a child of the <uninstallSteps> element. The <dependantCleanup> element specifies a set of steps to be executed to remove components that currently depend on the calling component. This element has no attributes and can include any number of steps that are permitted within the scope of the containing uninstall block.

When included, this element causes the check for dependant components to be deferred until after the contents of the block have been executed. If dependant components still remain after the block has been executed, the uninstall fails and the component remains installed. If no dependant components remain, the uninstall proceeds with the remaining steps.

If the containing component is targetable, the block can be used to remove components that are installed on its associated component targeting host. If installed components remain on the associated host after this block completes, the uninstall fails.

If a <dependantCleanup> block is not included in an uninstall block, the block fails immediately if dependant components exist.

The <dependantCleanup> block is often used in conjunction with the <allDependants> targeter to uninstall all dependant components at one time.

<snapshotList> Element

The <snapshotList> element is an optional child of the <component> element. This element contains one or more named snapshot blocks. Each snapshot block provides a different way to capture the installed state of this component on the target host. One or more snapshot blocks can be used to capture different aspects of the installed state. This results in a fine-grained comparison of the captured installed state and the current state of the component. If used, this element can only appear one time.

This element has one required child element, <snapshot>, which is a named snapshot block that can be executed to capture the installed state of this component. You can use more than one <snapshot> element.

By default, a derived component inherits the accessible <snapshotList> element contents of its base component. When a derived component declares a <snapshotList>, its contents are effectively merged with those of the base component. The derived component can declare new <snapshot> elements and override inherited ones, but it cannot remove elements declared by the base component.

<snapshot> Element

The <snapshot> element is a child of the <snapshotList> element. This element defines a sequence of steps to be executed to capture the installed state this component. When a <createSnapshot> or <addSnapshot> step names this snapshot block, the steps within the prepare block are executed in order. Then, files named with the capture block are captured within the capture area of the target server. Finally, steps within the cleanup block are executed in order.

A snapshot block is also used to compare the current state of a component against its state at the time of installation. In particular, the prepare steps are re-executed on the target server. Then, files captured at install time are compared to the current state of the files, and the cleanup steps are re-executed.

The <snapshot> element has the following child elements:

- <paramList> – An optional element that is a list of parameters to be used by the prepare, capture, and cleanup blocks of this snapshot. This element can only appear one time.
- <varList> – An optional element that is a list of local variables to be used by the prepare, capture, and cleanup blocks of this snapshot. This element can only appear one time.
- <prepare> – An optional element that contains steps to be executed in preparation for the file capture or comparison. This element can only appear one time.
- <capture> – An optional element that contains a list of files and directories to be captured as part of this snapshot. This element can only appear one time.
- <cleanup> – An optional element that contains steps to be executed after the capture or comparison has completed. This element can only appear one time.

If this snapshot is to be called from a <createSnapshot> step, it cannot declare any required parameters in its <paramList> element.

The <varList>, <prepare>, <capture>, and <cleanup> elements collectively define the body of the snapshot. The body is not included if the snapshot block is declared abstract.

By default, a derived component inherits all of the accessible snapshot blocks of its base component. Semantics for overriding a snapshot block are the same as those for overriding an install block.

You cannot call the base component snapshot block's prepare block from a derived component snapshot block's prepare block. This restriction applies to cleanup, as well. To make this sort of call, the base component must factor its <prepare> and <cleanup> steps into a control block that can be called by the derived component.

Attributes for the <snapshot> Element

This element has the following attributes:

- *access* – An optional attribute of type accessEnum, which specifies the accessibility of the snapshot block. The following values are permitted:
  - PUBLIC – Access is unrestricted, which is the default.

- PROTECTED – Access is limited to derived components and entities that are in the same path.
- PATH – Access is limited to entities that are in the same path.
- PRIVATE – Access is limited to this component.

- *modifier* – An optional attribute of type modifierEnum, which specifies the override requirements for the snapshot block. The following values are permitted:
  - ABSTRACT – The block cannot include a body because it must be specified by nonabstract derived components. Snapshot blocks can only be declared abstract if the component is also declared abstract. Abstract blocks cannot be private. Nonabstract blocks must declare a body.
  - FINAL – The snapshot block cannot be overridden by derived components.

  If this attribute is omitted, derived components can choose whether to override the block.
- *name* – A required attribute of type entityName, which is the name of the snapshot block. The name must be unique among all snapshot blocks in the containing <snapshotList>.
- *description* – An optional attribute that is a string that describes the snapshot block. This attribute is useful for documentation purposes.

<prepare> Element

The <prepare> element is a child of the <snapshot> element. This element defines a sequence of steps to prepare to capture or compare files. These steps are executed both as a result of a <createSnapshot> or <addSnapshot> step that targets this snapshot and a comparison run that targets this snapshot. In all cases, these steps are executed prior to capturing any files or performing any comparisons.

The <prepare> element children consist of one or more <call>, <execNative>, and <transform> steps. No other steps are permitted. The contained steps can reference local parameters and variables that are declared by the snapshot block, as well as unhidden component substitution variables.

<capture> Element

The <capture> element defines the files and resources that are to be captured as part of this snapshot. The capture occurs only after the steps in the <prepare> block have executed. When the capture is complete, the steps in the <cleanup> block will execute.

The <capture> element children consist of one or more <addFile>, <addSnapshot>, and <addResource> elements. The files and directories that are listed in this block are captured in the order specified. The contained children can reference local parameters and variables that are declared by the snapshot block, as well as unhidden component substitution variables.

<addFile> Element

The <addFile> element is a child of the <capture> element and specifies a file that is to be captured as part of the containing snapshot.

ATTRIBUTES FOR THE <addFile> ELEMENT

The <addFile> element includes the following attributes:

- *path* – A required attribute that specifies the path name of a file or directory on the file system of the target host. This attribute can reference simple substitution variables.
- *ownership* – An optional attribute that specifies the ownership option for the captured file.

  One aspect of the installed state that is captured by the snapshot is file and directory ownership. This ownership is not the same as UNIX permissions. The ownership is more closely tied to the concept of reference counts. Specifically, a file or directory can be captured as being owned by one or more snapshots.

  If the file owner later changes as a result of another component installation, this change can be recognized and reported when the file is compared against its initial state. This feature helps to track down differences that result from one component unintentionally overwriting files associated with another component. Snapshot ownership information is captured in a repository on the target host, which is known as the owners table.

  The values of the ownership attribute have the following semantics:

  - `SET_SELF` – When the ownership is set in this way, the owners table is updated to contain a single entry for the associated file or directory. That entry lists the executing installed component and snapshot as owners. The entry also lists the capture area ID of the captured contents of the file or directory.
  - `ADD_SELF` – When the ownership is set in this way, the installed component and snapshot are added as additional owners and share the existing capture area ID as the previous owners.
  - `ADD_TEMP` – This value is like `ADD_SELF` except that a new capture is always created and its ID is always used for the new entry rather than sharing the ID of the other owners.

  If this attribute is omitted, the default value is `SET_SELF`.
- *filter* – An optional attribute of type `boolean` that describes whether to capture files, directories, or both.

  If the value of *path* is a directory, this attribute is used to indicate whether the directory itself, the files it contains, or both should be captured. If the value of *path* is not a directory, this attribute is ignored and the file is captured directly. If this attribute is omitted, the default `BOTH` is used.

  The values for this attribute are as follows:

  - `DIRECTORIES`
  - `FILES`
  - `BOTH`
- *recursive* – An optional attribute that indicates whether subdirectories should be recursively captured by using the current filter settings.

  If the value of *path* is a directory, this attribute indicates whether subdirectories should be recursively captured by using the current filter settings. If the value of *path* is not a directory, this attribute is ignored and the file is captured directly. The default value is `true`.

- *displayName* – An optional attribute that is a string to be included in the display when this snapshot entry is compared against another.

  This attribute can reference simple substitution variables.

<addSnapshot> Element

The <addSnapshot> element denotes that an external snapshot block should be executed and that its contents should be added to this snapshot.

Using an <addSnapshot> step is semantically equivalent to all of the following scenarios:

- Adding the <prepare> steps of the called snapshot to the end of the calling snapshot's prepare block
- Adding the <cleanup> steps of the called snapshot to the start of the calling snapshot's cleanup block
- Adding the <capture> steps of the called snapshot to the calling snapshot's capture block in place of the <addSnapshot> step

Make sure that the files that are referenced by the called and calling snapshot blocks do not conflict.

Any number of <addSnapshot> steps can appear within a <capture> element. The called snapshot itself can contain any number of <addSnapshot> steps within its <capture> element.

When files are added to a snapshot indirectly by using <addSnapshot> callouts, the topmost component that initiated the snapshot capture is considered to be the owner of the files, as opposed to the component that contained the <addFile> directive. Similarly, when a comparison is performed on a snapshot, only <diff> element <ignore> directives of the topmost component that initiated the snapshot are considered. The <diff> element <ignore> directives that are contained on components visited as a result of <addSnapshot> callouts are not considered.

The <addSnapshot> element has one required attribute of type `entityName`, *blockName*, which is the name of the external snapshot block to execute.

The <addSnapshot> element has the following child elements:

- <argList> – An optional element that is a list of arguments to pass to the snapshot block. This element can only appear one time.
- <Installed component targeter> – An optional element that identifies the component that contains the snapshot block. If this element is omitted, <thisComponent> is used.

<addResource> Element

The <addResource> element denotes a resource that is associated with the component that is to be captured as part of the containing snapshot. This element can only be included in simple components.

This element serves as a syntactic shorthand for an equivalent <addFile> element, as follows:

- If the associated resource is a directory resource with *deployMode*=ADD_TO, <addResource> is equivalent to the following statement:

  ```
  <addFile path="path-of-deployed-directory" filter="FILES"
  displayName="resourceSourcePath"/>
  ```

- If the associated resource is a directory resource with *deployMode*=REPLACE, <addResource> is equivalent to the following statement:

  ```
  <addFile path="path-of-deployed-directory"
  displayName="resourceSourcePath"/>
  ```

- Otherwise, the associated resource is a file-based resource, and <addResource> is equivalent to the following statement:

  ```
  <addFile path="path-of-deployed-file"
  displayName="resourceSourcePath"/>
  ```

<cleanup> Element

The <cleanup> element is a child of the <snapshot> element and defines a sequence of steps to be executed after a file capture or comparison has completed. These steps are executed both as a result of a <createSnapshot> or <addSnapshot> step that targets this snapshot and a comparison run that targets this snapshot. In all cases, these steps are executed after capturing all files or performing comparisons. Use cleanup blocks to remove any temporary files that are created by the prepare block.

The <cleanup> element children consist of one or more <call>, <execNative>, and <transform> steps. No other steps are permitted. The contained steps can reference local parameters and variables that are declared by the snapshot block, as well as unhidden component substitution variables.

<controlList> Element

The <controlList> element is an optional child of the <component> element. This element lists the control blocks that are available for the component. If used, this element can only appear one time.

This element has a required child element, <control>, which is a control block. You can specify more than one <control> element.

By default, a derived component inherits the accessible <controlList> element contents of its base component. When a derived component declares a <controlList>, its contents are effectively merged with those of the base component. The derived component can declare new <control> elements and override inherited ones, but it cannot remove elements that are declared by the base component.

<control> Element

The <control> element defines a control block that is available for this component. A control block is a sequence of steps that can be performed after the component has been installed. For example, a

component for a database application might include control blocks to start or shut down the database. A <call> step can invoke a control block by name, which causes the control block steps to be executed in order.

The <control> element children consist of an optional <paramList> element followed by a body, which consists of an optional local <varList> element followed by zero or more "shared" steps. See "Shared Steps" on page 23. The body is not included if the control block is declared abstract.

By default, a derived component inherits all accessible control blocks of its base component. Semantics for overriding a control block are the same as those for overriding an install block.

Attributes for the <control> Element

The <control> element has the following attributes:

- *access* – An optional attribute of type accessEnum, which specifies the accessibility of the control block. These are the possible values:
  - PUBLIC – Access is unrestricted, which is the default value.
  - PROTECTED – Access is limited to derived components and entities that are in the same path.
  - PATH – Access is limited to entities that are in the same path.
  - PRIVATE – Access is limited to this component.

  Only PUBLIC blocks can be run directly from the component.
- *modifier* – An optional attribute of type modifierEnum, which specifies the override requirements for the control block. These are the possible values:
  - ABSTRACT – The block cannot include a body. Instead, the body must be specified by nonabstract derived components. A control block can only be declared abstract if the component is also declared abstract. An abstract block cannot be private. Nonabstract blocks must declare a body.
  - FINAL – The control block cannot be overridden by derived components.

  If this attribute is omitted, derived components can choose whether to override the block.
- *name* – A required attribute of type entityName, which is the name of the control block. This is referenced from a <call> step to execute the control.
- *description* – An optional attribute that is a string that describes the control block.
- *returns* – An optional attribute that indicates that the block must return a value. Valid values are true or false. If returns=true is specified, then each execution step within the block must end with either a <return> step or an uncaught <raise> step. The default is false.

<diff> Element

The <diff> element is an optional child of the <component> element. This element contains a list of directives that are used by the comparison engine to run comparisons on this component. If used, this element can only appear one time.

The <diff> element has one child element, <ignore>, which is required. This element specifies a directory path to ignore during comparison. You can use the <ignore> element one or more times.

A derived component automatically inherits all of the ignore directives declared by its base component. The component can also declare additional ignore directives in its own <diff> element. Inherited directives cannot be removed.

<ignore> Element

The <ignore> element is a child of the <diff> element and specifies a file name path pattern to ignore when using this component in a comparison. This element is typically for files and directories that are created by the installed application, such as log files. The configurable attributes of this element can reference component substitution variables.

The <ignore> element has one required attribute, *path*, which is a glob-style pattern that matches the file name paths to ignore, for example, `/logs/*.log`. This attribute can reference component-scoped substitution variables.

Install-Only Steps for Components

This section describes steps that can only be used within an install block of a component.

All steps except the <deployResource> step can be used by both simple and composite components. The <deployResource> step can only be used by simple components.

<createDependency> Step

This step creates a persistent dependency of the current component on another component. When executed, this step first checks for an installed component that matches the given criteria. The step fails if no such component exists (just as with the <checkDependency> step). If a match is found, a persistent dependency is created between the matching component, the "dependee," and the calling component, the "dependant."

If more than one installed component matches the criteria, which is possible if no install path is specified, the latest match is used as the dependee. The persistent component is created with the name that is specified in the step. The name must be unique among all dependencies created in the install block.

After the persistent dependency is created, it remains until the dependant component is uninstalled. If the installation of the dependant component fails in a subsequent step after having created a persistent dependency, the dependency is removed immediately at the time of failure.

A given component might depend on any number of other components by executing a `<createDependency>` step for each. The `<createDependency>` steps should appear as the first steps within an install block so that the installation will fail prior to performing any real work if the dependencies are not satisfied.

The `<createDependency>` step has one required child element, which identifies the dependee component. The child element is an installed component targeter. See "Installed Component Targeters" on page 47.

Attributes for the `<createDependency>` Step

The `<createDependency>` step has one required attribute of type `identifier`, *name*, which is the name of the dependency to create. The name must be unique among all dependencies created by the current component.

Uninstallation Implications for the `<createDependency>` Step

A component cannot be uninstalled if installed components depend on it. When an uninstall block of a component is encountered, if one or more persistent dependencies exist for which the component is the dependee, the uninstallation fails immediately.

However, if component B is being uninstalled by another component A, dependencies created by A on B will not prevent B from being uninstalled, and will be implicitly removed when B is successfully uninstalled.

The dependee component can specify actions to uninstall its dependants by using a `<dependantCleanup>` block.

Reinstallation Implications for the `<createDependency>` Step

A component installation is considered to be a reinstallation if a preexisting component in the same version tree is installed on the same host and install path. A component can be reinstalled only if the new component also satisfies all the dependants of the original component. A component can always be reinstalled with the same version component. However, a component can only be reinstalled with a different version if the new version also matches the constraints specified in the `<createDependency>` step that created the persistent dependency.

When a simple install block of a component is encountered, the N1 SPS determines if the installation will overwrite an existing installation. If so, the N1 SPS finds all persistent dependencies for which

the existing component is the dependee, and reverifies that the constraints of the dependency are still satisfied with the new component being installed. If any constraints are not satisfied, the installation of the new component fails, and the original component remains installed.

Otherwise, if and when the new component successfully completes its installation, the component becomes the new dependee on all persistent dependencies. The original component is considered to be uninstalled and all of the persistent dependencies for which it was the dependant are removed. This implies that the new component is responsible for recreating dependencies, as needed.

Naming Conventions for the <createDependency> Step

Dependency names follow this format: *xxx*2*yyy*. The *xxx* indicates the name of the dependant component, and *yyy* indicates the name of the dependee component.

For example, a WebLogic managed server might have a dependency called `server2domain` on its admin server and a `server2cluster` dependency on its containing cluster. This convention facilitates self-documentation of the nature of the dependency relationship and makes it readable for both the dependee and dependant relationships of a particular component.

<createSnapshot> Step

This step creates a snapshot of the current installed state of the component being installed. You can specify any number of <createSnapshot> steps within an install block.

The named snapshot block cannot declare any required parameters because the <createSnapshot> step does not support argument passing. Argument passing is not supported because it would also require support for argument collection and passing during a later comparison on the resulting snapshot.

A comparison performed on a composite component includes all snapshots that are created directly by that component. The comparison also includes the complete tree of snapshots that are created by the recursive installation of all nested component references. However, snapshots that are associated with top-level component references are not considered. Thus, such snapshots must be explicitly included in the snapshot of the composite component by using the <addSnapshot> capture directive.

In addition, nested components might have some interdependencies that make it necessary to defer snapshot capture until all such components are installed. One example is a nested component that deploys a directory and a nested component that deploys a file into that directory. In such interdependant cases, the containing component should install the nested components by using parameter passing or special install blocks that tell the nested component not to take the snapshot during their install. Then, have the snapshot block of the containing component include appropriate snapshots of the nested components by using the <addSnapshot> directive.

Attributes for the <createSnapshot> Step

The <createSnapshot> step has one required attribute of type `entityName`, *blockName*, which is the name of the snapshot block to execute within the component.

<install> Step

This step installs a component onto a target host, and causes the steps of the named install block of the targeted component to be executed.

The syntax of this step is as specified for the simple plan <install> step (see "<install> Step" on page 99), except that the component targeter can be omitted, in which case <thisComponent> is assumed.

When used within a component, this step is often used to call into another install block in the same component. In this case, the component is not considered to be installed until the outermost install block has completed its execution. Furthermore, the component is only considered to be installed on the host on which the install step was initially targeted, even if calls to other local install blocks were retargeted to other hosts.

This step can also be used within a composite component to install referenced components. The referenced components can be installed on hosts other than that on which the containing component is being installed. If the installation of the containing component fails, any nested referenced components that were successfully installed prior to the failure are implicitly uninstalled without executing an uninstall block. However, any top-level referenced components that were successfully installed prior to the failure remain installed.

<deployResource> Step

This step deploys the resource of the containing component and can only be used in an install block of a simple component. This step has no attributes or child elements.

A directory type resource where deployMode=ADD_TO has each of its contained files copied while preserving the directory structure. New directories are created, as needed. The existing directory structure and contents is unchanged, other than copying of resource contents. Individual file permissions and ownership are updated, as appropriate.

A directory type resource where deployMode=REPLACE is treated the same as deployMode=ADD_TO, except that any preexisting directory is recursively removed prior to deployment.

All other resources are copied, then have their permissions and ownership updated, as appropriate. Resources that are checked in as configurable undergo variable substitution prior to being copied. Configurable resources can reference any variable that is accessible to the component in which the resource was declared.

Uninstall-Only Steps for Components

This section describes the <uninstall> and <undeployResource> steps, which can only be used within an uninstall block of a component.

<uninstall> Step

This step is used to uninstall a component from a target host, and can be used in an uninstall block of any component. This step causes the steps of the named uninstall block of the targeted component to be executed.

The syntax of this step is as specified for the simple plan <uninstall> step, except that the component targeter can be omitted, in which case <thisComponent> is assumed.

When used within a component, this step is often used to call into another existing uninstall block in the same component. In this case, the component is not uninstalled until the outermost uninstall block has completed its execution. Furthermore, the component is only uninstalled on the host on which the uninstall step was initially targeted, even if calls to other local uninstall blocks were retargeted to other hosts.

This step can also be used within composite components to uninstall referenced components. When a composite component is uninstalled, all of its nested referenced components that were not explicitly uninstalled are implicitly uninstalled by the system without running an uninstall block. However, top-level referenced components that were not explicitly uninstalled remain installed.

<undeployResource> Step

This step is used to remove a resource from the containing component. The step can only be used in an uninstall block of a simple component and has no attributes or child elements.

A directory type resource where deployMode=ADD_TO will have its files removed, but its subdirectories will remain.

A directory type resource where deployMode=REPLACE will have the entire directory and its contents removed.

All other resources are treated as simple files and are removed.

The resource is removed regardless of whether it was originally deployed during the installation of the component. Even if the component was installed using an install block that did not contain a <deployResource> step, the <undeployResource> step removes the resource as if it had been originally installed by the component.

◆ ◆ ◆ CHAPTER 4

Plan Schema

This chapter describes the XML schema for the N1 SPS plans. The chapter covers these topics:

Unless indicated, attributes described in this chapter cannot reference component-scoped substitution variables.

For an overview of the XML schema architecture, see Chapter 1.

<executionPlan> Element Overview

An entire plan is enclosed by the <executionPlan> element.

Plans are either simple or composite. A *simple plan* is a sequential list of steps that are executed on a particular set of target servers. A simple plan does not contain or call other plans. A *composite plan* is composed solely of other subplans. A composite plan is not directly targeted because each simple subplan can run on a different set of targets.

Attributes for the <executionPlan> Element

The <executionPlan> element has the following attributes:

- *xmlns* – A required string that has the following value:

  `http://www.sun.com/schema/SPS`

- *xmlns:xsi* – A required string that has the following value:

```
http://www.w3.org/ 2001/XMLSchema-instance
```

- *xsi:schemaLocation* – An optional string that has the following recommended value:

  ```
  http://www.sun.com/schema/SPS plan.xsd
  ```

- *name* – A required attribute of type `entityName`, which is the name of the execution plan.
- *path* – An optional attribute of type `pathName`, which is the absolute path of the execution plan. If this attribute is omitted, the root path (`/`) is used. The value must name a folder that exists at the time that the plan is saved.
- *description* – An optional attribute that is a string that describes the execution plan.
- *version* – A required attribute of type `schemaVersion`, which is the version of the plan schema that is being used. The only permitted values are `5.0` and `5.1`.

  The only permitted values are `5.0` and `5.1`.

Child Elements of the <executionPlan> Element

The <executionPlan> element has the following optional child elements, which must appear in the order shown. These child elements might have their own child elements, attributes, or both.

- <paramList> – Declares a list of parameters for use by steps contained in the plan and any components that they reference
- <varList> – Declares a list of variables for use by the steps contained in the plan and any components they reference
- <simpleSteps> – Contains one or more “shared” or “simple plan only” steps
- <compositeSteps> – Contains one or more “composite plan-only” steps

<paramList> Element

The <paramList> element is an optional child of the <executionPlan> element. This element is used to declare a list of parameters for use by the steps contained in the plan and any components that the steps reference. If specified, this element can only appear one time.

When this plan is run as a top-level plan, the caller is prompted to enter values for all parameters declared in this list. When this plan is invoked as the result of an <execSubplan> step in another plan, the calling plan must explicitly pass values for all parameters that are declared by <paramList> that do not have default values.

The <paramList> element has one required child element, <param>, which is a plan parameter declaration. The declaration includes a name, a prompt, and a default value. You can specify one <param> element for each parameter that you want to declare.

<param> Element

The <param> element is a child of the plan <paramList> element and is used to declare a parameter for use within the plan.

Attributes for the <param> Element

The <param> element has the following attributes:

- *name* – A required attribute of type `identifier`, which is the name of the plan parameter. The name must be unique among all top-level plan parameters and variables.
- *prompt* – An optional attribute that is a string that is displayed in the user interface when prompting for the value of the parameter. If this attribute is omitted, the value of *name* is used.
- *default* – An optional attribute that is a string that is the default value of the parameter. The default value can include references to session variables.
- *displayMode* – An optional attribute that specifies the display mode of the parameter. The following legal values are permitted:
  - `PASSWORD` – The user-specified value is hidden, which means that the password is not shown or is replaced by asterisks.
  - `BOOLEAN` – The parameter is specified by means of a check box.
  - `CLEAR` – The value is displayed as entered.

  If the value is `CLEAR` or `BOOLEAN`, the value can be safely displayed as entered. If the attribute is omitted, the value is `CLEAR`.

<varList> Element

The <varList> element is an optional child of the <executionPlan> element and the <inlineSubplan> step. For information about the latter element, see "<inlineSubplan> Step" on page 98. The <varList> element is used to declare a list of variables for use by the steps contained in the plan and any components they reference. The values of the variables are defined at the point of declaration, and cannot be redefined. If specified, this element can only appear one time.

The <varList> element has one required child element, <var>, which is a plan variable declaration. A declaration includes a name and a value. Specify a <var> element for each variable that you want to declare.

<var> Element

The <var> element is a child of the plan <varList> element and is used to declare a plan variable including name and value.

Attributes for the <var> Element

This element has the following attributes:

- *name* – A required attribute of type `identifier`, which is the name of the local variable. The name must be unique among every variable in the containing <varList>. Variables associated with the top-level <executionPlan> must also be unique among the plan parameters.
- *default* – A required attribute that is a string that is the default value of the plan variable. This value can include references to other plan variables that have been declared earlier, to session variables, and to plan parameters. If this plan is a simple plan, you can include references to target host attributes and to installed component variables.

<simpleSteps> Element

The <simpleSteps> element is an optional child of the <executionPlan> element and the <inlineSubplan> step. For information about the latter element, see "<inlineSubplan> Step" on page 98. The <simpleSteps> element contains one or more "shared" or "simple plan only" steps. The presence of a <simpleSteps> element indicates that the plan is a simple, not composite, plan. If specified, this element can only appear one time.

When run, the steps within this element are sequentially executed on a set of logical target hosts that are chosen by the caller. These hosts are called the *initial target hosts*. While this plan executes, its steps can be redirected to execute on a host other than the initial host. The host on which the plan actually executes is known as the *current target host*. If a step is not redirected, the current host and the initial host are the same. The initial host can be either virtual or physical, as can the current host. The *physical host* is the root parent host of the current host, which is the same as the current host if the current host is physical.

The <simpleSteps> element children consist of one or more "shared" or "simple plan only" steps. These steps can include references to plan parameters and variables. See "Plan-Only Steps for Simple Plans" on page 99.

Attributes for the <simpleSteps> Element

The <simpleSteps> element has the following attributes:

- *executionMode* – An optional attribute that indicates whether the contained steps should be executed in a series or in parallel over the target hosts. The legal values are as follows:
  - `PARALLEL`
  - `SERIES`

  If this attribute is omitted, the value is `PARALLEL`.
- *limitToHostSet* – An optional attribute that specifies the name of the host set that contains the hosts that can be valid targets for this plan.

If this attribute is omitted, all hosts can be valid targets. Otherwise, the targets that you specify must be a subset of the hosts that are contained in the named host set. If the targets include a host that is not contained in the named host set, the plan issues a runtime error. The plan also issues a runtime error if you specify a name that does not correspond to an existing, supported host set. If the specified host set is defined by a plug-in, *pluginName* must be prefixed to the host set name, such as *pluginName#hostSetName*. These plan runtime errors are reported during validation before the preflight starts.

<compositeSteps> Element

The <compositeSteps> element is an optional child of the <executionPlan> element and the <inlineSubplan> step. For information about the latter element, see "<inlineSubplan> Step" on page 98. The <compositeSteps> element contains one or more "composite plan-only" steps. The presence of a <compositeSteps> element indicates that the plan is a composite plan. The <compositeSteps> element does not have any attributes. If specified, this element can only appear one time.

The <compositeSteps> element children consist of one or more "composite plan-only" steps. These steps can include references to plan parameters and variables. See "Plan-Only Steps for Composite Plans" on page 97.

Plan-Only Steps for Composite Plans

This section describes the steps that can only be used within a composite plan. The attributes of some of the steps that are contained within a composite plan can include references to plan variables and parameters.

<execSubplan> Step

The <execSubplan> step executes another plan. The <execSubplan> step can only appear as the child of the <compositeSteps> element.

The <execSubplan> step has an optional child element, <argList>, which is a list of arguments to pass to the called plan. For each parameter in the <paramList> section of the called plan for which no default value is declared, a corresponding argument must be declared by this <argList>. See "<argList> Element" on page 24. If specified, this element can only appear one time.

Attributes for the <execSubplan> Step

The <execSubplan> step has the following attributes:

- *planName* – A required attribute of type entityName, which is the name of the plan to execute. When this step is run, a corresponding top-level <executionPlan> with this name must be specified. This name cannot reference an inline subplan.

- *planPath* – An optional attribute of type `pathReference`, which is the path of the plan to execute. If this attribute is omitted, the path of the containing plan is assumed.
- *planVersion* – An optional attribute of type `Version`, which is the version of the plan to execute. If this attribute is omitted, the latest version of the named plan is executed.

<inlineSubplan> Step

The <inlineSubplan> step executes a sequential series of steps. This step can only appear as the child of the <compositeSteps> element.

An <inlineSubplan> step is similar to an <execSubplan> step. However, while the <execSubplan> steps names an external plan to execute, the <inlineSubplan> step directly contains the plan to execute as a child element.

The primary difference between an inline subplan and a top-level plan is that inline subplans are not saved as distinctly named entities. Thus, inline subplans cannot be externally referenced by <execSubplan> steps. Top-level plans are distinctly named entities and can be referenced from <execSubplan> steps.

Inline subplans are useful when the content is concise, directly tied to the context and logic of the calling plan, and does not otherwise make sense as a stand-alone plan. In these cases, having all steps in one self-contained unit can facilitate plan maintenance and readability.

Unlike top-level plans, inline subplans cannot declare parameters. They implicitly inherit the parameters and variables of all enclosing plans. They can declare additional variables that are local to the inline subplan, which can hide variables and parameters of the enclosing plans. A subplan variable hides the variable of an enclosing plan if both have the same name. In this case, only the value of the variable that is declared by the innermost subplan is available for use by its steps.

The <inlineSubplan> step consists of an optional <varList> followed by one additional child element, either <simpleSteps> or <compositeSteps> depending on the type of inline subplan: simple or composite.

The <inlineSubplan> step has the following child elements:

- <varList> – An optional element that is a list of plan variables for use within the inline subplan. If specified, this element can only appear one time.
- <simpleSteps> – An optional element that contains a list of simple steps. Only one <simpleSteps> or <compositeSteps> element can be present. If specified, this element can only appear one time.
- <compositeSteps> – An optional element that contains a list of composite steps. Only one <compositeSteps> or <simpleSteps> element can be present. If specified, this element can only appear one time.

Attributes for the <inlineSubplan> Step

The <inlineSubplan> step has the following attributes:

- *planName* – A required attribute of type `entityName`, which is a name used to identify the inline subplan. This name is used primarily for display purposes, and need not be distinct from names of other plans (inline or top-level).
- *description* – An optional attribute that is a string description of the inline subplan. This attribute is useful for documentation purposes.

Plan-Only Steps for Simple Plans

This section describes steps that can only be used within a simple plan. Steps that are contained within a plan can reference any of the variables that are declared by that plan. The steps can also reference any of the unhidden variables and parameters of all of the enclosing plans.

<install> Step

The <install> step installs a component onto the target host. This causes the steps of the named <installSteps> element of the associated component to be executed. This step can only appear as the child of the <simpleSteps> element.

The <install> step has the following child elements:

- <argList> – An optional element that is a list of arguments to pass to the <installSteps> block. If specified, this element can only appear one time. See "<argList> Element" on page 24.
- <Repository component targeter> – A required element that identifies the component to install. See "Repository Component Targeters" on page 52.
- <assign> — An optional element that specifies the name of the local variable to assign the value returned by the called block. The syntax and behavior of the <assign> child element is the same as that of the "<assign> Step" on page 24.

Attributes for the <install> Step

The <install> step has one required attribute of type `entityName`, *blockName*, which is the name of the install block to execute within the target component.

<uninstall> Step

The <uninstall> step uninstalls the resources of a component that is currently installed on the target host. This causes the steps of the named <uninstallSteps> element of the associated component to be executed. This step can only appear as the child of the <simpleSteps> element.

The <uninstall> step has the following child elements:

- <argList> – An optional element that is a list of arguments to pass to the <uninstallSteps> block. If specified, this element can only appear one time. See "<argList> Element" on page 24.

- `<Installed component targeter>` – A required element that identifies the component to uninstall. See "Installed Component Targeters" on page 47.
- `<assign>` — An optional element that specifies the name of the local variable to assign the value returned by the called block. The syntax and behavior of the `<assign>` child element are the same as that of the "`<assign>` Step" on page 24.

Attributes for the `<uninstall>` Step

The `<uninstall>` step has one required attribute of type `entityName`, *blockName*, which is the name of the uninstall block to execute within the target component.

◆ ◆ ◆ CHAPTER 5

Resource Descriptor Schema

This chapter describes the XML schema for the resource descriptor files, and covers these topics:

Using a Resource Descriptor File

A *resource descriptor file* specifies the owner, group, and permission settings to use for the files and directories that comprise the resource of a simple component. This resource descriptor is an XML file. By using a resource descriptor file, you can override the permissions that are determined at component check-in time.

When a resource descriptor file is not used, a resource uses the owner, group, and permission settings it has at check-in time. This situation is the default case when you perform the check-in on a UNIX system. When you check in a component on a Windows system, the default settings are as if you used the `:NONE:` value for each of the settings in a resource descriptor file.

When you use a resource descriptor file, a resource uses the owner, group, and permission settings specified by the resource descriptor. If an `<entry>` element is specified for a resource, the settings are taken from that entry. If the entry does not specify all of the settings, the missing setting values are taken from the `<defaultEntry>`, if present. If those setting values are not specified in a `<defaultEntry>`, the resource uses the setting values it had at check-in time.

If no `<entry>` element is specified for a resource, the resource uses the settings specified in `<defaultEntry>`, if present. If no `<defaultEntry>` is present, the resource uses the settings it had at check-in time.

Use the `:NONE:` value to tell N1 SPS to use the default settings from the file system on which the component will be deployed. You can specify the `:NONE:` value for any setting specified in a `<defaultEntry>` block or in an `<entry>` for a resource.

A resource descriptor file is only used when deploying a component to a UNIX system. If a component is deployed to a Windows system, the resource descriptor file is ignored. So, if your component only applies to Windows systems, do not create a resource descriptor file.

A resource descriptor file can be used by simple components that extend the `system#file` and `system#directory` component types. A resource descriptor file can also be used by a component that extends the `com.sun.linux#rpm` component type, which is part of the Linux plug-in.

You check in the resource descriptor file at the same time that you check in your component. When you attempt a checkin-current for a component that used a resource descriptor file for its last checkin, N1 SPS expects to find the resource descriptor in the original check-in location. Thus, if you move the file to a different location and attempt a checkin-current for the component, the check-in operation fails.

You can download the resource descriptor for a simple component that has been checked in to see the settings for every file that is part of the component's resource.

You might use this download feature to update the file and check in an updated version of the component. First, you download the resource descriptor file and then check out the associated component's resource. Then, you modify the resource descriptor file and use it to check in a new version of the component.

The resource descriptor you download might differ from the resource descriptor you used to check in the component. Differences might appear because the descriptor you use to check in a component is not required to have information about every file in the resource, or to have full information for every entry. Notice that no `<defaultEntry>` block appears in the downloaded resource descriptor file. Instead, every file is described in its own `<entry>`.

<resourceDescriptor> Element Overview

An entire resource descriptor is enclosed by the `<resourceDescriptor>` element.

The resource descriptor file is encoded in UTF-8 format unless a byte order mark (BOM) is present. If present, the BOM is used to determine the encoding of the file.

Attributes for the <resourceDescriptor> Element

The `<resourceDescriptor>` element has the following attributes:

- *xmlns* – A required string that has the following value:

  `http://www.sun.com/schema/SPS`

- *xmlns:xsi* – A required string that has the following value:

  `http://www.w3.org/2001/XMLSchema-instance`

- *xsi:schemaLocation* – An optional string that has the following recommended value:

```
http://www.sun.com/schema/SPS resourceDescriptor.xsd
```

- *schemaVersion* – A required attribute of type `schemaVersion`, which is the version of the resource descriptor schema that is being used. The only permitted values are `5.0`, `5.1` and `5.2`.

Child Elements of the <resourceDescriptor> Element

The <resourceDescriptor> element has one required child element, <entryList>, which is the list of files and directories contained by this resource. The entry for each file or directory includes the associated owner, group, and permission settings.

<entryList> Element

The <entryList> element is a required child of the <resourceDescriptor> element. This element is used to list the files and directories included in this resource and describes the settings that are associated with the files and directories.

The <entryList> element has two optional child elements that, if present, must appear in this order:

- <defaultEntry>
- <entry>

If specified, the <defaultEntry> element can only appear one time. The <entry> element describes the settings for a single file or directory. Thus, you can specify more than one <entry> element in the resource descriptor file.

<defaultEntry> Element

The <defaultEntry> element is an optional child of the <entryList> element. This element specifies the default owner, group, and permission settings for resource files that are not listed in <entry> elements in the resource descriptor file.

If this element is omitted, the file and directories listed in this <entryList> block use the setting values determined at check-in time. If specified, this element can only appear one time.

<settings> Element

The <settings> element is a required child element of the <defaultEntry> element and of the <entry> element.

The <settings> element behaves differently when used as a child of these elements:

- <defaultEntry> element – Specifies the setting values to be used as defaults for any file that is not associated with its own <entry> element
- <entry> element – Specifies the setting values to be used by the file associated with this <entry> element

If any attributes are omitted, the corresponding attribute value in the <defaultEntry> element for this <entryList> is used. If no <defaultEntry> element is specified, the check-in time settings are used.

Attributes for the <settings> Element

The <settings> element has these optional attributes:

- *group* – An optional string that is the group of this entry. The group must be a group name, not a group ID.
- *owner* – An optional string that is the owner of this entry. The owner must be a user name, not a user ID.
- *permissions* – An optional string that is the permissions settings for a file or directory. The string is in the format of an octal triplet, as defined by the UNIX chmod command. See the chmod(1M) man page.

You can specify a value of :NONE: for any of these attributes. Use the :NONE: value to tell N1 SPS to use the default settings from the file system on which the component will be deployed.

<entry> Element

The <entry> element is an optional child element of the <entryList> element. This element specifies the owner, group, and permission settings for one entry in the resource. The <entry> element has one required element, <settings>. See "<settings> Element" on page 103.

Files and directories that do not appear as <entry> elements in the <entryList> block use the following:

- Setting values specified in the <defaultEntry> block
- Setting values determined at check-in time

Attributes for the <entry> Element

The <entry> element has one required attribute, *name*. The *name* value is a string that defines the name of this entry, relative to the root of the resource. The name of the top-level resource should always be root.

For example, if the resource created a directory hierarchy starting with topDir, the *name* attribute for the top level directory is root. The *name* attribute for a nested directory subDir would be root/subDir. For a file resource named file.txt, the name attribute would be root.

Note – Trailing slash (/) characters are not permitted.

See "Sample XML for the <resourceDescriptor> Element" on page 105.

Sample XML for the <resourceDescriptor> Element

The following example shows an <resourceDescriptor> element for a directory resource.

```
<?xml version ="1.0" encoding="UTF-8"?>
<resourceDescriptor
      xmlns="http://www.sun.com/schema/SPS"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.sun.com/schema/SPS
                          resourceDescriptor.xsd"
      schemaVersion="5.2">
  <entryList>
    <defaultEntry>
      <settings
        owner="root"
        group="wheel"
        permissions="664"/>
    </defaultEntry>

    <!-- This directory overrides all of the default settings -->
    <entry
         name="root">
      <settings
        owner="gprabhu"
        group="bin"
        permissions="777"/>
    </entry>

   <!-- This directory overrides the owner and group, but will have
       perms of "664" from the defaultEntry -->
    <entry
        name="root/nestedDirectory" >
      <settings
        owner="gprabhu"
        group="wheel"/>
    </entry>

    <!-- This file overrides the group and perms, but will have owner
     of "root" from the defaultEntry -->
    <entry
         name="root/nestedDirectory/fileThatWillUseSomeDefaults.txt" >
      <settings
        group="bin"
        permissions="777"/>
    </entry>

    <!-- This file overrides none of the settings in the defaultEntry,
        so it will inherit all of them.
```

```
        In practice, this entry would probably be omitted entirely. -->
    <entry
        name="root/nesteddirectory/fileThatWillUseAllDefaults.txt" >
      <settings/>
    </entry>
  </entryList>
</resourceDescriptor>
```

The following example shows an <resourceDescriptor> element for a file resource.

```
<?xml version ="1.0" encoding="UTF-8"?>
<resourceDescriptor xmlns="http://www.sun.com/schema/SPS"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.sun.com/schema/SPS resourceDescriptor.xsd"
 schemaVersion="5.2">
  <entryList>
    <!-- There is no <defaultEntry> in this <entryList>, so anything not
        specified in the <settings> element for this entry uses the
        settings determined at check-in time. -->
    <entry name="root">
      <settings owner="terry" group="bin" permissions="777"/>
    </entry>
  </entryList>
</resourceDescriptor>
```

◆ ◆ ◆ CHAPTER 6

Plug-In Descriptor Schema

This chapter describes the XML schema that you use to define a plug-in. The chapter contains the following topics:

<plugin> Element Overview

The <plugin> element is the top-level element in the plug-in schema. The <plugin> element identifies the parts of the plug-in.

Attributes for the <plugin> Element

The <plugin> element has the following attributes:

- *name* – The name of the plug-in. The *name* attribute has a maximum length of 64. Plug-in names follow a structure similar to `com.sun.solaris`.
- *description* – An optional description of the plug-in.
- *vendor* – The provider of the plug-in.
- *version* – The version of the plug-in. Version numbers follow the standard *x.y* (*major.minor*) format.
- *schemaVersion* – A required attribute of type `schemaVersion`, which is the version of the plug-in XML schema being used. The only permitted values are `5.0`, `5.1` and `5.2`.

  The 5.2 version of the schema is backward compatible with the 5.0 and 5.1 versions.

- *previousVersion* – An optional attribute that is the version of this plug-in expected to be on the system. If not specified, an initial install is assumed. If specified, the value represents the version from which the plug-in is to be upgraded.
- *xmlns* – A required string that has the following value:

  `http://www.sun.com/schema/SPS`
- *xmlns:xsi* – A required string that has the following value:

  `http://www.w3.org/2001/XMLSchema-instance`
- *xsi:schemaLocation* – An optional string that has the following recommended value:

  `http://www.sun.com/schema/SPS plugin.xsd`

Child Elements of the <plugin> Element

The <plugin> element may include the following child elements:

- <readme> – Optional path to the readme file written by the plug-in author
- <serverPluginJAR> – Optional path to the JAR file that contains server-side plug-in code to run on the master server
- <gui> – Optional GUI extensions for the plug-in
- <dependencyList> – Optional list of external plug-ins on which this plug-in depends
- <memberList> – Optional list of member objects to create as part of the plug-in

  These member objects can include any number of folder, host type, host set, host search, component, and plan objects. These member objects can appear in any order.

<readme> Element

The <readme> element is a child of the <plugin> element and is used to declare the location of a readme.txt file in the plug-in JAR. This readme.txt file is expected to be a Unicode-encoded text file. The byte order mark (BOM) is used to specify the Unicode encoding. If no BOM is present, UTF-8 encoding will be used by default.

The <readme> element has one attribute, *jarPath*, that contains the path name to the readme file. The path name of the readme file is relative to the root of the plug-in JAR file. Leading slash (/) or period (.) characters are not permitted in the *jarPath*.

<serverPluginJAR> Element

The <serverPluginJAR> element is a child of the <plugin> element and is used to declare the location of a server side plug-in JAR file that contains code to run on the Master Server. The location of this JAR file is relative to the root of the plug-in JAR file. This JAR file will typically contain component exporter implementations for the component types defined by the plug-in. Externally written code that is to run in the Master Server (for example, exporter classes) need not be in the same JAR file as the code that runs on the agent for the plug-in, though it is permissible.

The <serverPluginJAR> element has one attribute, *jarPath*, that contains the path name to the server-side plug-in JAR file. The path name is relative to the root of the plug-in JAR file. Leading slash (/) or period (.) characters are not permitted in the *jarPath*.

<gui> Element

The <gui> element is an optional child of the <plugin> element and is used to declare the location of a separate plug-in UI descriptor file for a set of GUI extensions in support of this plug-in. The syntax of this plug-in UI descriptor file is described in Chapter 7.

The <gui> element has one attribute, *jarPath*, that contains the path name to the plug-in UI descriptor file. The path name is relative to the root of the plug-in JAR file. Leading slash (/) or period (.) characters are not permitted in the *jarPath*.

<dependencyList> Element

The <dependencyList> element is a child of the <plugin> element and is used to declare a list of other plug-ins on which this plug-in depends. It has no attributes and contains one or more <pluginRef> elements that identify a dependency for this plug-in. When the plug-in is deployed, the N1 SPS software checks against these dependencies.

<pluginRef> Element

The <pluginRef> element is a child of the <dependencyList> element and is used to declare a reference to another plug-in. The <pluginRef> element has two required attributes:

- *name* – The name of the referenced plug-in. The *name* attribute has a maximum length of 64. Plug-in names follow a structure similar to com.sun.solaris.
- *version* – The minimum version of the referenced plug-in. The referenced plug-in must be installed on the system with this version or higher.

The <pluginRef> element contains no child elements.

<memberList> Element

The <memberList> element is a child of the <plugin> element and is used to declare a list of system objects that are part of this plug-in. These objects can appear in any order.

The <memberList> element has no attributes and contains at least one of the following child elements:

- <folder> – A folder declaration
- <hostType> – A host type declaration
- <hostSet> – A host set declaration
- <hostSearch> – A host search declaration
- <component> – A component declaration
- <plan> – A plan declaration

<folder> Element

The <folder> element is a child of the <memberList> element and is used to declare a folder to be referenced by the plug-in.

A plug-in can specify a folder to be created in the form of a full path name. For example, /a/b/c. In this example, a and b are interior folders, and c is a leaf folder. The plug-in owns the leaf folder. The *admin* group is listed as the folder owner group, and the folder is identified as being owned by the plug-in. A plug-in may only create components and plans in a folder that it owns. Users cannot create components, plans or subfolders in a plug-in owned folder.

If an interior folder does not exist when a plug-in is loaded, it is implicitly created. Interior folders may not be owned by a plug-in. The owner group for a plug-in created interior folder is the *admin* group, but the folder is not identified as belonging to any plug-in. If a plug-in author intends for interior folders to be explicitly owned by a plug-in, the folders should be created individually. In the above example, folder /a would be created first, followed by folder /a/b, then folder /a/b/c.

Unowned interior folders may not be created under an owned interior folder. This requirement prevents plug-in authors from creating components and plans in a folder between owned folders in the folder hierarchy, which would complicate the deletion semantics.

If an interior folder exists and is unowned when a plug-in is loaded, the interior folder is used directly. If an interior folder exists and is owned by a plug-in, the interior folder must be owned either by the current plug-in or by a plug-in on which the current plug-in has a direct dependency. This requirement enables multiple cooperative plug-ins to be distributed separately by a plug-in vendor. By obeying Java package style naming conventions when creating folders, vendors can avoid folder name space collisions.

Attributes for the <folder> Element

The <folder> element has two attributes:

- *name* – The path name of the folder
- *description* – An optional description of the folder

<hostType> Element

The <hostType> element is a child of the <memberList> element and is used to declare a host type to be referenced by the plug-in. The name of the host type will be implicitly prefixed with the plug-in name when the host type is created in the system.

Attributes for the <hostType> Element

The <hostType> element has two attributes:

- *name* – The name of the host type. The *name* attribute has a maximum length of 32. The name must begin with either a Unicode letter or an underscore character (_), followed by either Unicode letters, numbers, or an underscore character (_), dot (.), or dash (-).
- *description* – An optional description of the host type

<varlist> Element

The <hostType> element contains an optional <varlist> child element. The <varlist> element specifies a list of variables to be added to the <hostType> element and later used by hosts in configuration.

The <varlist> child element contains one or more <var> child elements. The <var> element provides <hostType> element variable declaration through two required attributes:

- *name* – The name of the variable
- *default* – The default value of the variable

<hostSet> Element

The <hostSet> element is a child of the <memberList> element and is used to declare a host set to be referenced by the plug-in. The <hostSet> element cannot contain hosts, since plug-ins cannot define hosts. Platform host sets cannot be defined by any plug-in other than the system plug-in. The name of the host set will be implicitly prefixed with the plug-in name when the host set is created in the system.

The <hostSet> element contains two optional child elements:

- <hostSetRef>
- <hostSearchRef>

Attributes for the <hostSet> Element

The <hostSet> element has three attributes:

- *name* – The name of the host set. The *name* attribute has a maximum length of 32. The name must begin with either a Unicode letter or an underscore character (_), followed by either Unicode letters, number, or an underscore character (_), dot (.), or dash (-).
- *description* – An optional description of the host set
- *unsupported* – An optional attribute that, when true, means that the host set is not supported. The default is false.

<hostSetRef> Element

The <hostSetRef> element is a child of the <hostSet> element. It specifies a sub-host set. This host set must have been previously defined either in this plug-in or in a plug-in on which this plug-in directly depends. References to host sets defined in another plug-in must include the plug-in name, for example, com.foo.other#hostSetName. Unqualified references are assumed to be objects created by this plug-in.

Attributes for the <hostSetRef> Element

The <hostSetRef> element has one attribute, *name*. This attribute provides the name of the host set reference. The *name* attribute has an optional *pluginName* that has a maximum length of 64, followed by a # separator, followed by a *hostEntityName* that has a maximum length of 32.

<hostSearchRef> Element

The <hostSearchRef> element is a child of the <hostSet> element. It specifies a sub-host search. This host search must have been previously defined either in this plug-in or in a plug-in on which this plug-in directly depends. References to host searches defined in another plug-in must include the plug-in name, for example, com.foo.other#hostSearchName. Unqualified references are assumed to be objects created by this plug-in.

Attributes for the <hostSearchRef> Element

The <hostSearchRef> element has one attribute, *name*. This attribute provides the name of the host search reference. The *name* attribute has an optional *pluginName* that has a maximum length of 64, followed by a # separator, followed by a *hostEntityName* that has a maximum length of 32.

<hostSearch> Element

The plug-in <hostSearch> element is a child of the <memberList> element and is used to declare a host search to be referenced by the plug-in. The name of the host search will be implicitly prefixed with the plug-in name when the host search is created in the system.

The <hostSearch> element contains at least one of the following child elements:

- <criteriaList>
- <appTypeCriteria>
- <physicalCriteria>

Note – Although the <criteriaList>, <appTypeCriteria>, and <physicalCriteria> elements are each optional, one of the three must be provided.

Attributes for the <hostSearch> Element

The <hostSearch> element has two attributes:

- *name* – The name of the host search. The *name* attribute has a maximum length of 32. The name must begin with either a Unicode letter or an underscore character (_), followed by either Unicode letters, numbers, or an underscore character (_), dot (.), or dash (-).
- *description* – An optional description of the host search.

<criteriaList> Element

The <criteriaList> element is a child of the <hostSearch> element. It specifies a list of criteria to be added to the <hostSearch> element. The <criteriaList> element must be specified if <appTypeCriteria> and <physicalCriteria> are not specified.

The <criteriaList> element contains one or more <criteria> elements. The <criteria> element specifies a search criteria, including name, match type, and pattern.

Attributes for the <criteriaList> Element

The <criteriaList> element has three attributes:

- *name* – The name of the host variable to match.
- *pattern* – The pattern to match.
- *match* – The match type of the criteria. Valid values are EQUALS or CONTAINS. The default is EQUALS.

<appTypeCriteria> Element

The <appTypeCriteria> element is a child of the <hostSearch> element. It specifies a list of application type criteria to be added to the <hostSearch> element. The arguments of the <appTypeCriteria> element are expressed as attributes, and order is not important. If all values are false or the element is empty or unspecified, the search disregards this criteria when performing the search. The <appTypeCriteria> element must be specified if <criteriaList> and <physicalCriteria> are not.

Attributes for the <appTypeCriteria> Element

The <appTypeCriteria> element has three optional attributes:

- *ms* – If true, match MasterServer application type in host search. The default is false.
- *ld* – If true, match LocalDistributor application type in host search. The default is false.
- *ra* – If true, match RemoteAgent application type in host search. The default is false.

<physicalCriteria> Element

The <physicalCriteria> element is a child of the <hostSearch> element. It specifies a list of physical type criteria to be added to the <hostSearch> element. The arguments of the <physicalCriteria> element are expressed as attributes, and order is not important. If all values are false or the element is empty or unspecified, the search disregards this criteria when performing the search. The <physicalCriteria> element must be specified if <criteriaList> and <appTypeCriteria> are not.

Attributes for the <physicalCriteria> Element

The <physicalCriteria> element has two optional attributes:

- *physical* – If true, match physical host types in host search. The default is false.
- *virtual* – If true, match virtual host types in host search. The default is false.

<component> Element

The <component> element is a child of the <memberList> element and is used to declare a component in the plug-in JAR file. All objects referenced by this component must have been previously defined either in this plug-in or in a plug-in on which this plug-in directly depends.

The <component> element contains three optional child elements:

- <systemService>
- <componentType>
- <resource>

Attributes for the <component> Element

The <component> element has two attributes:

- *jarPath* – The location of the component XML file, relative to the root of the plug-in JAR (leading / or . characters are not permitted). The format of the component XML is specified by the Plan and Component Language specification. See Chapter 3 for more information.
- *majorVersion* – An optional attribute that determines whether to check in the component as a new major version. The default is false.

<systemService> Element

The <systemService> element is a child of the <component> element and is used to declare a system service backed by the containing component. This element may not be used with the <componentType> element. When the <systemService> element is used in a <component> element, a component is loaded and a <systemServiceRef> that references that component is created. The name of the system service is prefixed with the plug-in name when the system service is created in the system.

Attributes for the <systemService> Element

The <systemService> element has two attributes:

- *name* – The name of the system service

  A name has a maximum of 64 characters. The name must start with a letter or underscore, followed by any number of letters, digits, or special characters, such as underscore (_), period (.), plus (+), minus (-), and space ( ). Unicode letters and digits are permitted.
- *description* – An optional description of the system service

<componentType> Element

The <componentType> element is a child of the <component> element and is used to declare a component type backed by the containing component. The <componentType> element may not be used with the <systemService> element. When the <componentType> element used in a <component> element, a component is loaded and a component type that is backed by that component is created. The name of the component type is prefixed with the plug-in name when the component type is created in the system.

Component types are grouped by plug-in, and ordered by the component type order within these groupings. Groupings are ordered according to the plug-in order. Within a particular plug-in, the component types are indented under distinct group names as defined by the component types.

Attributes for the <componentType> Element

The <componentType> element has five attributes:

- *name* – The name of the component type.

  A name has a maximum of 64 characters. The name must start with a letter or underscore, followed by any number of letters, digits, or special characters, such as underscore (_), period (.), plus (+), minus (-), and space ( ). Unicode letters and digits are permitted.
- *description* – An optional description of the component type.
- *group* – The group name of the component type, if this component type is part of a hierarchy of component types.

  Group names follow the same requirements as the component type name. In addition, a group can be declared as hidden, which prevents the type from displaying in the component type drop down list on the component list page.

- *order* – A number that identifies where to put this component type in the drop-down list of component types in the browser interface.

  The order is a maximum of 18 characters. In addition to Unicode letters and digits, any character that you can type on an ASCII keyboard is permitted. The order should be sufficient to sequence all of the types that are defined within a particular plug-in.
- *indentLevel* – A number between 0 and 10 that identifies the level to which to indent this component type in a hierarchy of component types in the browser interface.

<resource> Element

The <resource> element is a child of the <component> element. It specifies a resource file name and location in the JAR file. A resource is always checked in as a simple file-typed resource. The component that contains the <resource> element must be a simple component whose <resourceRef> element refers to the resource created by the <resource> element.

Attributes for the <resource> Element

The <resource> element has three attributes:

- *jarPath* – The location of the resource file, relative to the root of the plug-in JAR file. Leading / or . characters are not permitted. For directory-type resources, this path is assumed to be a directory, and is expected to end with a /. Everything in this directory defines the contents of this resource.
- *majorVersion* – An optional attribute that determines whether to check in the resource as a new major version. The default is false.
- *name* – An optional attribute that is the name of the resource. If not specified, the name will default to the absolute *jarPath*, which is converted to absolute if specified as relative.
- *config* – An optional attribute that specifies whether this resource is a configuration template. The default is false.
- *type* – An optional attribute that specifies whether the resource is a file or a directory. Use FILE for a file resource. Use DIRECTORY for a directory resource. The default is FILE.
- *checkInMode* – An optional attribute that specifies whether a directory-type resource should be replaced or appended. Use REPLACE if the check in of this resource should replace an existing version. Use ADD_TO if the check in should add to the resource. The default is REPLACE. This attribute only applies to a resource that has a *type* of DIRECTORY.
- *descriptorPath* – An optional attribute that specifies the path to a resource descriptor file, relative to the root of the plug-in JAR file. Leading / or . characters are not permitted. The format of the resource descriptor file follows the Resource Descriptor schema, as described in Chapter 5.

  If no resource descriptor file is specified, permissions information is used from the default file system settings of the N1 SPS master server. In this case, owner and group are not stored. This is also the case for settings that are omitted from a descriptor (either no entry or a partial entry for a file within the resource).

<plan> Element

The <plan> element is a child of the <memberList> element and is used to declare a plan in the plug-in JAR. All objects referenced by this plan must have been previously defined either in this plug-in or in a plug-in on which this plug-in directly depends.

Attributes for the <plan> Element

The <plan> element has two attributes:

- *jarPath* – The location of the plan XML, relative to the root of the plug-in JAR file. Leading / or . characters are not permitted. The format of the plan XML is specified by the Plan and Component Language specification. See Chapter 4 for more information.
- *majorVersion* – An optional attribute that determines whether to check in the plan as a new major version. The default is false.

Sample XML for the <plugin> Element

EXAMPLE 6–1 Sample Plug-in Descriptor File

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin name="com.bigCo.logic.pluginName"
       description="imitation WL plugin"
        vendor="bigCo"
        version="1.3"
        previousVersion="1.2"
        xmlns="http://www.sun.com/schema/SPS"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.sun.com/schema/SPS
                            plugin.xsd"
        schemaVersion="5.2">
  <readme jarPath="docs/readme.txt"/>
 <serverPluginJAR jarPath="lib/appserver/serverCode.jar"/>
 <gui jarPath="custom/weblogic/gui/pluginUI.xml"/>
 <dependencyList>
 <pluginRef name="webLogicUtils" version="1.0"/>
 <pluginRef name="otherPlugin" version="1.3"/>
 </dependencyList>
 <memberList>
 <folder name="/com/bea/weblogic/6.0" description="Weblogic 6.0 plugin folder"/>
 <folder name="/folder2" description="second place sees dust"/>
 <hostType name="WL Admin Server" description="Host Type for Weblogic Admin Servers">
 <varList>
 <var name="adminPort" default="7001"/>
 <var name="adminUser" default="weblogic"/>
 <var name="secureConnect" default="false"/>
```

EXAMPLE 6–1 Sample Plug-in Descriptor File *(Continued)*

```
 </varList>
 </hostType>
 <hostSet name="Weblogic Admin Servers" description="The Weblogic Admin Servers">
 <hostSetRef name="WL boxes"/>
 <hostSearchRef name="WL Admin Search"/>
 </hostSet>
 <hostSearch name="WL box search" description="matches Weblogic boxes">
 <criteriaList>
 <criteria name="sys.OS" match="CONTAINS" pattern="SunOS"/>
 <criteria name="sys.OSVersion" pattern="5.9"/>
 </criteriaList>
 <appTypeCriteria ms="false" ld="false" ra="true"/>
 <physicalCriteria physical="true" virtual="true"/>
 </hostSearch>
 <hostSet name="Weblogic Servers" description="All Weblogic Servers">
 <hostSetRef name="Weblogic Admin Servers"/>
 <hostSetRef name="com.bigCo.logic.cluster#Weblogic Clusters"/>
 </hostSet>
 <component jarPath="comps/system/weblogic/foo.xml" majorVersion="true">
 <componentType name="contained EJB CT"
 description="contained ejb comp type ref"
 group="hidden"
 order="001-003-002"
 indentLevel="2"/>
 </component>
 <component jarPath="weblogic/system/comps/bar.xml">
 <systemService name="WebLogic SS" description="WL service ref"/>
 </component>
 <component jarPath="weblogic/system/comps/baz.xml"/>
 <plan jarPath="weblogic/system/plans/bar.xml" majorVersion="false"/>
 <component jarPath="weblogic/system/comps/dee.xml">
 <resource jarPath="weblogic/system/plugin-core.jar" majorVersion="true"/>
 </component>
 <component jarPath="weblogic/system/comps/boo.xml">
   <resource jarPath="weblogic/system/bigDir/" majorVersion="true"
      name="com/sun/boo" type="DIRECTORY" checkInMode="ADD_TO"
         descriptorPath="resources/bigDir.manifest" />
 </component>
  </memberList>
</plugin>
```

◆ ◆ ◆ CHAPTER 7

Plug-In User Interface Schema

This chapter describes the XML schema that you use to define the user interface for a plug-in. The chapter contains the following topics:

<pluginUI> Element Overview

The <pluginUI> element enables a plug-in author to describe a limited set of functionality to appear on a custom shortcuts page. The set of functionality exposed via the shortcuts is divided into the following categories:

- Component type shortcuts:
  - List all components that extend this component type
  - Create a component that extends this component type
- Component shortcuts:
  - Manage the specified component (link to component details page)
  - List virtual and physical hosts where this component is installed
- Plan shortcuts:
  - Manage the specified plan (link to plan details page)

Attributes for the <pluginUI> Element

The <pluginUI> element has the following attributes:

- *menuItem* – The text to display in the menu of the browser interface. The name should be 20 characters or less in length, although the actual character limit is defined by the attribute type.
- *tooltip* – An optional tooltip to display for the menu item in the browser interface. The *menuItem* includes the icon, if you choose to provide an icon.
- *xmlns* – A required string that has the following value:

  `http://www.sun.com/schema/SPS`
- *xmlns:xsi* – A required string that has the following value:

  `http://www.w3.org/2001/XMLSchema-instance`
- *xsi:schemaLocation* – An optional string that has the following recommended value:

  `http://www.sun.com/schema/SPS pluginUI.xsd`
- *schemaVersion* – A required attribute of type `schemaVersion`, which is the version of the plug-in XML schema being used. The only permitted values are `5.0`, `5.1` and `5.2`.

  The 5.2 version of the schema is backward compatible with the 5.0 and 5.1 versions.

Child Elements of the <pluginUI> Element

The <pluginUI> element contains the following elements:

- <icon> – Provides the path to a graphic (icon) that you want displayed within the interface for this plug-in
- <customPage> – Defines the contents of the custom page linked to by the menu item in the browser interface

<icon> Element

The <icon> element is a child of the <pluginUI> element. The <icon> element declares the location of the plug-in icon. The icon is expected to be in GIF or JPEG file format. The icon should have dimensions of 32 pixels wide by 26 pixels high.

Attributes for the <icon> Element

The <icon> element has one required attribute *jarPath*. The *jarPath* attribute specifies the location of the plug-in icon, relative to the root of the plug-in JAR file. Leading slash (`/`) or dot (`.`) characters are not permitted.

<customPage> Element

The <customPage> element is a child of the <pluginUI> element and defines the contents of the custom page linked to by the menu item in the browser interface. . The <customPage> element contains one or more <section> elements and has a *name* attribute.

Attributes for the <customPage> Element

The <customPage> element has one required attribute, *name*. The *name* attribute is used in the breadcrumb and title sections of the custom page.

<section> Element

The <section> element is a child of the <customPage> element and defines a section for the custom page. The <section> element contains one or more <entry> or <section> elements and has two attributes:

- *title* – Section title
- *description* – An optional section description, secondary text

---

Note – To create a nested custom page structure, put a <section> element within another <section> element. Any nested <section> elements must appear after all <entry> elements within the <section>.

<entry> Element

The <entry> element is a child of the <section> element and defines an entry point for user actions. An <entry> element contains zero or more <action> elements and has the following attributes:

- *title* – Entry title
- *description* – An optional entry description

<action> Element

The <action> element is a child of the <entry> element and defines a user action. Each <action> element must contain exactly one child element.

The <action> element has two attributes:

- *text* – Text to be rendered for the link of the user action
- *tooltip* – An optional tooltip to be used for the link of the user action

Each <action> element must contain one of the following child elements:

`<compCreate>`

The `<compCreate>` element is a child of the `<action>` element and defines a link to the component create page for the named component type.

The `<compCreate>` element has one required attribute:

- *typeName* – The name of the component type. The component type must be contained in the plug-in or in a plug-in on which this plug-in directly depends. The component type cannot be defined as `hidden`. The `pluginName` must be prefixed to the component type name, for example, `fullPluginName#componentTypeName`.

`<compDetails>`

The `<compDetails>` element is a child of the `<action>` element and defines a link to the component details page for the latest version of the named component.

The `<compDetails>` element has two required attributes:

- *path* – The absolute path of the component.
- *name* – The name of the component. The component must be contained in the plug-in or in a plug-in on which this plug-in directly depends.

`<compList>`

The `<compList>` element is a child of the `<action>` element and defines a link to the component list page filtered by the named component type. If the optional *path* or *flatView* attributes are omitted, their values will be picked up from the user's session when the users clicks on the link. If not omitted, the user's session values will be changed to reflect them after clicking on the link.

The `<compList>` element has three attributes:

- *typeName* – The required name of the component type. The component type must be contained in the plug-in or in a plug-in on which this plug-in directly depends. The component type cannot be defined as `hidden`. The `pluginName` must be prefixed to the component type name, for example, `fullPluginName#componentTypeName`.
- *path* – An optional full path by which to filter the components list page. The path must start with a slash (`/`), and should not end with a slash (`/`). The folder that the path represents must be a folder either owned by this plug-in or owned by a plug-in on which this plug-in directly depends. Alternatively, the folder can contain a folder owned by this plug-in or by a plug-in on which this plug-in directly depends.
- *flatView* – An optional statement as to whether the list page should enable the flat view filter.

<compWhereInstalled>

The <compWhereInstalled> element is a child of the <action> element and defines a link to the Component Where Installed page for the latest version of the named component.

The <compWhereInstalled> element has two required attributes:

- *path* – The absolute path of the component.
- *name* – The name of the component. The component must be contained in the plug-in or in a plug-in on which this plug-in directly depends.

<hostList>

The <hostList> element is a child of the <action> element and defines a link to the host list page filtered by the named host type.

The <hostList> element has one required attribute:

- *typeName* – The name of the host type. The host type must be contained in the plug-in or in a plug-in on which this plug-in directly depends. The host type cannot be defined as `hidden`. The `pluginName` must be prefixed to the host type name, for example, `fullPluginName#hostTypeName`.

<planDetails>

The <planDetails> element is a child of the <action> element and defines a link to the plan details page for the latest version of the named plan.

The <planDetails> element has two required attributes:

- *path* – The absolute path of the plan.
- *name* – The name of the plan. The plan must be contained in the plug-in or in a plug-in on which this plug-in directly depends.

<compProcedureRun>

The <compProcedureRun> element is a child of the <action> element and enables you to run a component procedure directly.

The <compProcedureRun> element has four required attributes:

- *name* – The name of the component that contains the procedure to run. The component must be contained in this plug-in or in a plug-in on which this plug-in directly depends.
- *path* – The absolute path to the component.
- *procedureName* – The name of the component procedure.
- *procedureType* – The type of the component procedure. Valid values include `INSTALL`, `UNINSTALL` or `CONTROL`.

<external>

The <external> element is a child of the <action> element and defines a link to an arbitrary URL that is outside of the N1 SPS product. For example, you might use this element to provide a link to a corporate web page that contains additional information about a specific plug-in or feature.

The <external> element has one attribute, *url*, which supplies a well-formed URL. The URL must conform to the Internet standard for Uniform Resource Identifiers (URIs) as specified in RFC3986 Uniform Resource Identifier (URI): Generic Syntax (http://www.gbiv.com/protocols/uri/rfc/rfc3986.html).

To ensure that the XML for the plug-in user interface descriptor is well-formed, the *url* attribute may include escaped special characters. Use the following escape sequences for the specified characters:

Character	Escape Sequence
>	>
<	<
"	"
&	&

Sample XML for the <pluginUI> Element

The following sample XML takes fragments from various custom pages to illustrate each element in the <pluginUI> schema.

EXAMPLE 7–1 Sample <pluginUI> Descriptor File

```
<?xml version="1.0" encoding="UTF-8"?>
<pluginUI menuItem="pluginName"
          toolTip="view wl server pages"
          xmlns="http://www.sun.com/schema/SPS"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.sun.com/schema/SPS
                              pluginUI.xsd"
          schemaVersion="5.2">
  <icon jarPath="custom/gui/img/WLicon-small.gif"/>
  <customPage name="WebLogic">
    <section title="WebLogic application tasks"
             description="capture and edit your WebLogic applications...">
      <entry title="enterprise applications (EARs)"
             description="capture, edit and deploy your enterprise applications">
        <action text="view all" toolTip="view all EARs">
          <compList typeName="com.bigCo.logic.pluginName#WebLogic enterprise application"
               path="/com/bigCo/logic" flatView="true"/>
        </action>
        <action text="create new" toolTip="create new enterprise application">
         <compCreate typeName="com.bigCo.logic.pluginName#WebLogic enterprise application"/>
        </action>
```

EXAMPLE 7–1 Sample <pluginUI> Descriptor File *(Continued)*

```
  </entry>
  <entry title="web applications (WARs)"
         description="capture, edit and deploy web applications">
    <action text="view all" toolTip="view all WARs">
      <compList typeName="com.bigCo.logic.pluginName#WebLogic web application"/>
    </action>
    <action text="create new" toolTip="create new webapp">
      <compCreate typeName="com.bigCo.logic.pluginName#WebLogic web application"/>
    </action>
  </entry>
  <entry title="java archives containing EJBs (JARs)"
         description="capture, edit and deploy your JARS containing EJBs">
    <action text="view all" toolTip="view all JARs">
      <compList typeName="com.bigCo.logic.pluginName#WebLogic EJB"/>
    </action>
    <action text="create new" toolTip="create new java archive containing EJBs">
      <compCreate typeName="com.bigCo.logic.pluginName#WebLogic EJB"/>
    </action>
  </entry>
<section>
  <entry title="entry in a nested section"
       description="this is a an entry in a nested section">
    <action text="view all" toolTip="view all aType comps">
    <compList
          typeName="com.bigCo.logic.pluginName#aType"
        path="/com/bigCo/logic" />
    </action>
  </entry>
</section>
</section>
<section title="WebLogic infrastructure"
         description="create and edit your WebLogic infrastructure...">
  <entry title="admin servers"
         description="WebLogic domains / administration servers">
    <action text="manage admin servers" toolTip="manage WebLogic admin servers">
      <compDetails path="/com/bea/weblogic" name="WL Admin Server 7.0"/>
    </action>
    <action text="view admin servers" toolTip="list of WebLogic admin servers">
      <compWhereInstalled path="/com/bea/weblogic" name="WL Admin Server 7.0"/>
    </action>
  </entry>
  <entry title="clusters"
         description="WebLogic clusters">
    <action text="manage clusters" toolTip="manage WebLogic clusters">
      <compDetails path="/com/bea/weblogic" name="WL Cluster"/>
    </action>
```

EXAMPLE 7–1 Sample <pluginUI> Descriptor File *(Continued)*

```
      <action text="view clusters" toolTip="list of WebLogic clusters">
        <compWhereInstalled path="/com/bea/weblogic" name="WL Cluster"/>
      </action>
    </entry>
    <entry title="managed servers"
           description="WebLogic server instances">
      <action text="manage server instances" toolTip="WebLogic managed servers">
        <compDetails path="/com/bea/weblogic" name="WL Managed Server"/>
      </action>
      <action text="view managed servers" toolTip="list of WebLogic managed servers">
        <compWhereInstalled path="/com/bea/weblogic" name="WL Managed Server"/>
      </action>
      <action text="update managed servers" toolTip="run a plan on managed servers">
        <planDetails path="/com/bea/weblogic/updates" name="updatePlan"/>
      </action>
      <action text="list servers" toolTip="list the apache servers">
        <hostList searchName="com.bigCo.logic.pluginName.WebLogic#apacheHosts"/>
      </action>
       <action text="custom reports" toolTip="view the custom reports">
        <external url="http://reportserver/reports/dec"/>
      </action>
      <action text="start a Managed Server"
              toolTip="run the start control of a ManagedServer component">
        <compProcedureRun path="/com/sun/weblogic" name="WL Managed Server"
             procedureName="start" procedureType="CONTROL"/>
      </action>
  </entry>
  </section>
 </customPage>
</pluginUI>
```

◆ ◆ ◆ APPENDIX A

Component Change Compatibility

This appendix enumerates the kinds of changes that can be made to a component and indicates whether each change is install compatible or call compatible.

The following changes can be made to components:

Changes That Can Be Made To Components

<component> Element Changes

The following table shows the changes that can be made to the <component> element and indicates whether each change is install compatible or call compatible.

Type of Change	Install Compatible	Call Compatible
Nonfinal to final	No	Yes

Type of Change	Install Compatible	Call Compatible
Final to nonfinal	Yes	Yes
Nonabstract to abstract	No	Yes
Abstract to nonabstract	Yes	Yes
More restrictive access	No	No
Less restrictive access	Yes	Yes
Change the value of the *description*, *label*, *softwareVendor*, or *author* attributes	No[1]	Yes
Change the value of the *name* or *path* attributes[2]	No	Yes
Change from a simple component to a composite component	No	No
Change from a composite component to a simple component	No	No

[1] The attribute values are stored in the installed variable settings record.

[2] This change effectively constitutes a change of the version tree and is only possible in situations where a system service is being updated. In this case, the new component must be an instance of the original component.

platform Attribute Changes

The following table shows the changes that can be made to the *platform* attribute and indicates whether each change is install compatible or call compatible.

Type of Change	Install Compatible	Call Compatible
More general platform	Yes	Yes
More specific platform	No	Yes
Unrelated platform	No	Yes

A platform is more specific than another if the first platform is a descendant of the second. A platform is more general if the first platform is an ancestor of the second platform.

limitToHostSet Attribute Changes

The following table shows the changes that can be made to the *limitToHostSet* attribute and indicates whether each change is install compatible or call compatible.

Type of Change	Install Compatible	Call Compatible
Any change to the *limitToHostSet* attribute	No	Yes

Unlike the *platform* attribute, *limitToHostSet* names a generic, user-specified host set over which there is no explicit control. A host set's membership can change at any time, so you cannot specify more-specific or less-specific host sets.

`<extends>` Element Changes

The following table shows the changes that can be made to the `<extends>` element and indicates whether each change is install compatible or call compatible.

Type of Change	Install Compatible	Call Compatible
New base component instance of the original component	No	Yes
Original base component instance of a new component	No	No
New base component that is unrelated to the original component	No	No
New base component is install compatible with the original component	Yes	Yes
New base component is call compatible with the original component	No	Yes

Changes to Variables

The following table shows the changes that can be made to a variable and indicates whether each change is install compatible or call compatible.

Type of Change	Install Compatible	Call Compatible
Add a new variable	Yes[1]	Yes
Remove or rename a nonprivate variable	No	No
Remove or rename a private variable	Yes	Yes
Change the default value of a final variable	No	Yes
Change the default value of a nonfinal variable	Yes[2]	Yes
Change the *prompt* attribute	Yes	Yes
Nonfinal to final	No	Yes
Final to nonfinal	Yes	Yes
Nonabstract to abstract	No	Yes
Abstract to nonabstract	Yes	Yes
More restrictive access	No	No
Less restrictive access	Yes[1]	Yes

[1] A derived component might exist that already defines the variable with a more restrictive access mode. In such a case, a change would make the derived component invalid. A new nonabstract variable can be added to a component type if no derived component has already defined a variable that has the same name, and the default value of the variable can be recomputed for all installed instances of the component.

[2] No reinstall is required because the installed value can be considered an override of the new default value.

`<targetRef>` Element Changes

The following table shows the changes that can be made to the `<targetRef>` element and indicates whether each change is install compatible or call compatible.

Type of Change	Install Compatible	Call Compatible
Remove `<targetRef>` element	No	No
Add `<targetRef>` element	No	Yes
Modify the *hostName* attribute	Yes[1]	Yes

[1] The attributes of hosts that are associated with existing installed components are not updated as a result of such a change.

Type of Change	Install Compatible	Call Compatible
Modify the *typeName* attribute	No	No
Add or remove an <agent> child element	No	No
Modify the *connection* attribute of the <agent> child element	Yes[1]	Yes
Modify the *ipAddr* attribute of the <agent> child element	Yes[1]	Yes
Modify the *port* attribute of the <agent> child element	Yes[1]	Yes
Modify the *params* attribute of the <agent> child element	Yes[1]	Yes

[1] The attributes of hosts that are associated with existing installed components are not updated as a result of such a change.

<componentRefList> Element Changes

The following table shows the changes that can be made to the <componentRefList> element and indicates whether each change is install compatible or call compatible.

Type of Change	Install Compatible	Call Compatible
Nonfinal to final	No	Yes
Final to nonfinal	Yes	Yes
New type instance of the original	No	Yes
Original type instance of the new	No	No
New type that is unrelated to the original	No	No
New type is install compatible with the original	Yes	Yes
New type is call compatible with the original	No	Yes

<componentRef> Element Changes

The following table shows the changes that can be made to the <componentRef> element and indicates whether each change is install compatible or call compatible.

Type of Change	Install Compatible	Call Compatible
Nonfinal to final	Yes[1]	Yes

[1] A derived component might exist that already defines the component reference with a more restrictive access mode or otherwise incompatible difference. In such a case, a change would make the derived component invalid. A new nonabstract component reference can be added to a component type if no derived component has already defined a component reference that has the same name.

Type of Change	Install Compatible	Call Compatible
Final to nonfinal	Yes	Yes
Nonabstract to abstract	No	Yes
Abstract to nonabstract	Yes	Yes
Change the *installMode* attribute	No	No
Add a new component reference	Yes[1],[2]	Yes
Remove or rename a nested `<componentRef>`	No	No
Remove or rename a top-level `<componentRef>`	No	No
Add, modify, or remove arguments from a nested component's `<argList>`	No	Yes
Add, modify, or remove arguments from a top-level component's `<argList>`	Yes[3]	Yes
New type instance of the original	No	Yes
Original type instance of the new	No	No
New type that is unrelated to the original	No	No
New type that is install compatible with the original	Yes	Yes
New type that is call compatible with the original	No	Yes
New nested component instance of the original	No	Yes
Original nested component instance of the new	No	No
New nested component that is unrelated to the original	No	No
New nested component that is install compatible with the original	Yes	Yes
New nested component that is call compatible with the original	No	Yes

[1] A derived component might exist that already defines the component reference with a more restrictive access mode or otherwise incompatible difference. In such a case, a change would make the derived component invalid. A new nonabstract component reference can be added to a component type if no derived component has already defined a component reference that has the same name.

[2] Adding a nested component is technically possible. An existing installation would be treated as if it had been installed without the nested component. Because any functionality that might depend on the installation could break, this change should only be made if the component can function safely without the nested component. Otherwise, it should be treated as being a change that is not install compatible.

[3] You cannot determine whether this component was the component that actually installed the top-level component. Therefore, this component cannot rely on the top-level component having variables that correspond to the `<argList>` values.

Changes to Resources

The following table shows the changes that can be made to a resource and indicates whether each change is install compatible or call compatible.

Type of Change	Install Compatible	Call Compatible
Nonfinal to final	No	Yes
Final to nonfinal	Yes	Yes
Nonabstract to abstract	No	Yes
Abstract to nonabstract	Yes	Yes
Modify the *installPath*, *name*, *group*, or *user* attributes	No	Yes
Modify the *rsrcName* or *rsrcVersion* attributes	No	Yes

`<install>`, `<control>`, and `<uninstall>` Block Changes

The following table shows the changes that can be made to an `<install>`, `<control>`, or `<uninstall>` block and indicates whether each change is install compatible or call compatible.

Type of Change	Install Compatible	Call Compatible
Nonfinal to final	Yes[1]	Yes
Final to nonfinal	Yes	Yes
Nonabstract to abstract	No	Yes
Abstract to nonabstract	Yes	Yes
More restrictive access	No	No
Less restrictive access	Yes[1]	Yes
Add a new nonprivate block	Yes[1]	Yes
Add a new private block	Yes	Yes
Remove or rename a nonprivate block	No	No
Remove or rename a private block	Yes	Yes

[1] A derived component might exist that already defines the block with a more restrictive access mode or otherwise incompatible difference. In such a case, a change would make the derived component invalid. A new nonabstract block can be added to a component type if no derived component has already defined a block that has the same name.

Type of Change	Install Compatible	Call Compatible
Reorder blocks	Yes	Yes
Change the body of a block	Yes[2]	Yes
Add, modify, or remove local block variables	Yes[2]	Yes
Add, modify, or remove private block parameters	Yes	Yes
Add a required parameter to a nonprivate block	No	No
Add an optional parameter	Yes	Yes
Remove an optional or a required parameter	Yes[3]	Yes[3]
Rename an optional parameter	Yes[4]	Yes[4]
Rename a required parameter from a nonprivate block	No	No
Change a parameter from being optional to being required in a nonprivate block	No	No
Change a parameter from being required to being optional	Yes	Yes
Change the *displayMode* attribute of a parameter	Yes[2]	Yes
Change the *prompt* attribute of a parameter	Yes	Yes
Change InstallBlock returns attribute value	No	Yes
Change UninstallBlock returns attribute value	No	No
Change ControlBlock returns attribute value	No	No

[2] The plan run history of prior runs of this block are not updated. Therefore, they might not directly coincide with the new block contents.

[3] Extra arguments that are passed from the caller are ignored, making this change possible.

[4] This change is equivalent to removing and adding an optional parameter. However, the callers might see unexpected results as the originally passed parameter value is now ignored and replaced with the default value.

Changes to <snapshot> Blocks

The following table shows the changes that can be made to the <prepare>, <cleanup>, or <capture> child elements of the <snapshot> element and indicates whether each change is call compatible or install compatible. <snapshot> blocks generally have the same compatibility matrix as the other blocks, except when dealing with their <prepare>, <capture>, and <cleanup> blocks.

Type of Change	Install Compatible	Call Compatible
Add, modify, or remove a <prepare> or <cleanup> block	Yes	Yes

Type of Change	Install Compatible	Call Compatible
Add, modify, or remove a <capture> step	Yes[1]	Yes

[1] The contents of the snapshot <capture> block are evaluated only one time. The evaluation takes place when the snapshot is taken during initial installation. At comparison time, the stored capture contents are used to drive the comparison, ignoring any changes that might have occurred to the <capture> block. Thus, the existing snapshot is not affected by such a change. If the intent of the change was to affect the contents of the existing snapshot, this change must be modeled as a change that is not install compatible.

Changes to <ignore> Child of <diff> Element

The following table shows the changes that can be made to the <ignore> child element of the <diff> element and indicates whether each change is call compatible or install compatible.

Type of Change	Install Compatible	Call Compatible
Add, modify, or remove an <ignore> element	Yes	Yes

The <ignore> element is only considered when you run the comparison and does not affect the state of existing snapshots.

Index

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

X