

MTR070273

MITRE TECHNICAL REPORT

Open Checklist Reporting Language (OCRL) by Example

October 2007

Charles M. Schmidt
Lisa A. Nordman

Sponsor: NSA
Department: G026

Contract: W15P7T-07-C-F600
Project: 0707N60C

Approved for Public Release; Distribution Unlimited.
Case Number 09-0181.

©2009 The MITRE Corporation. All rights reserved.

MITRE

**Center for Integrated Intelligence Systems
Bedford, Massachusetts**

Intentionally Blank

Acknowledgments

This work was produced by the System Security Analysis Project, an NSA-sponsored project of The MITRE Corporation. Charles Schmidt was the task lead. Charles Schmidt and Leonard LaPadula were the lead technical developers. The team would also like to recognize the valuable contributions of Shaan Foltz, Lisa Nordman, and Matthew Wojcik in this development effort.

Intentionally Blank

Table of Contents

Section	Page
1 Introduction	1
1.1 The Purpose of the Open Checklist Reporting Language	1
1.2 The High-Level Organization of a Report Definition File	1
1.3 The Open Checklist Reporting Language Data Model	2
1.4 The Examples	4
2 Example: Single WMI Source	5
3 Example: Multiple WMI Sources	9
3.1 Variation: The "hidden" Attribute	13
3.2 Variation: Complex Qualifiers	14
3.3 Variation: Using <Qualifier> with <Select> Elements	16
4 Example: Simple File Source	21
4.1 Variation: Executable Data Sources	26
5 Example: Structured File Source	29
5.1 Variation: Combining <Line> and <PatternMatch> Elements	33
5.2 Variation: Declaring Multiple Instances	34
6 Example: Nested Structured File Source	41
6.1 Variation: Using <InstanceReference> Elements	45
7 Example: OVAL Source	49
8 Conclusion	55

List of Figures

Figure		Page
1	The Open Checklist Reporting Language Data Model	3
2	OVAL Registry Object Items	52
3	OVAL Active Directory Object Items	53

List of Tables

Table		Page
1	One Property from One WMI Query	5
2	Multiple Data Sources from Multiple WMI Queries	9
3	The "hidden" Attribute	13
4	A Complex Qualifier	15
5	Using <Qualifier> Elements with <Select> Elements	16
6	A <Qualifier> Element in a <Select> Element with an Expanded Body	17
7	Extracting from a File Data Source	21
8	An Executable Data Source	26
9	Extracting Structure from Files	29
10	Combining <Line> and <PatternMatch> Elements	33
11	Multiple <Instance> Elements in a DataSource	34
12	Reporting Multiple <Instance> Elements	36
13	Reporting Multiple <Instance> Elements Separately	37
14	Extracting Nested Structure from Files	41
15	Using the <InstanceReference> Element	46
16	OVAL Data Source	50

Section 1

Introduction

This document provides annotated examples of uses of the Open Checklist Reporting Language (hereafter referred to as OCRL). The intention is to guide OCRL authors through the structure of report documents by example. This document assumes the reader has a passing familiarity with XML in that they understand what elements and attributes are and how to write them. Readers who prefer a more comprehensive technical guide to OCRL are directed to the document entitled "Report Schema: A Language for Automated Generation of Reports for Security Guidance".

1.1 The Purpose of the Open Checklist Reporting Language

OCRL defines an XML language for defining reports. Report definitions are structured to enable a software tool to collect system information from one or more sources and then organize this information for presentation in a report file. A tool reads the *report definition file*, gathers the specified system information, and then generates the *report file*. A user reads the report file and uses that to manually evaluate compliance or non-compliance with a given policy recommendation. Each generated report file corresponds to one recommendation.

OCRL can be used in cases where current benchmark technology does not support automatic evaluation of compliance, but where it is still possible to automatically collect some or all of the information needed for an informed user to perform such an evaluation. In other words, the information needed to make the compliance decision can be mechanically gathered, but the final compliance decision currently requires human judgment.

Given this, the purpose of a report definition file is twofold. First, it identifies where to find the necessary information that is to be presented in the report. This takes the form of identifying the data repository that is the source of relevant data and defining what structures should be extracted from this repository. Secondly, the report definition file describes how the data extracted from the data repositories should be laid out in the report file. OCRL elements that support these activities are described below.

1.2 The High-Level Organization of a Report Definition File

Each OCRL file may contain one or more report definitions. A report definition corresponds to a single policy recommendation and, as a result, each report file produced by an interpreter corresponds to one policy recommendation. Each report definition is bounded by <ReportDefinition> elements. (The root element of the file is the <ReportDefinitions> element. This element would surround all <ReportDefinition> elements.) The <ReportDefinition> element itself contains a number of high-level attributes:

- id – This string is the id of the report definition (present to support external references).
- filename – This string is concatenated with the date and time to form the name of the file to which the report should be written.
- title – This is a title of the report file (often displayed at the top of the file).
- recommendation – This is the recommendation for which the report is being generated. It is the compliance with this recommendation that the user is attempting to evaluate.
- instruction – This field describes how to use the information presented in the report file to evaluate compliance with the recommendation.
- display – This is an optional attribute that allows a report author to suggest that a given report be presented on-screen to the user rather than being used to create an output file.

The <ReportDefinition> element contains one to two child elements: <DataSources> and <Reporting>. The <DataSources> element is used to identify the data sources from which the interpreter gathers system information. The <Reporting> describes how the gathered data should be set out in the report file. It is these two sections on which the majority of this document focuses.

1.3 The Open Checklist Reporting Language Data Model

Before getting into the details of the <DataSources> and <Reporting> sections, it is necessary to understand something of the data model that OCRL uses to structure the information it is tasked with collecting and reporting. This model can be viewed as having three levels of abstraction.

The simplest structure in the OCRL data model is called a "Property". A Property represents a name-value pair. The value of a Property is extracted by the OCRL interpreter when it reads data from a source identified in a data source's definition. The name of a Property provides a handle for this value and is used within the <Reporting> section to identify which Property's value should be processed at a given point.

The next structure in the OCRL data model is called an "Instance". An Instance is a collection of Properties. Properties that share an Instance are connected in some way. This connection may be in scope, in subject, in time, or some other aspect. Thus, Instances preserve the context of Properties within a data source. For example, consider a medical device monitoring a patient. Every minute, the machine records the time and the patient's temperature. In the OCRL data model, these would be represented by two Properties, possibly with the names of "time" and "temp". Assume we have an hour's worth of material from this device. If the information we extract is going to be useful to us, each "temp"

Property must be associated with a "time" Property since, otherwise, we end up with sixty "temp" Properties without any context telling us their order. OCRL uses the Instance structure to record this correlation between Properties. Specifically, the "time" and "temp" Properties at each monitoring interval are stored into one Instance, giving us sixty Instances over an hour's worth of monitoring. When we examine the data we recorded, we know the "time" Property corresponding to each "temp" Property because they share an Instance.

Finally, the highest level structure in the OCRL data model is called a DataSource. A DataSource holds one or more Instances. In the above example, the DataSource contains sixty Instances, corresponding to the entire hour of collected data. A single report definition may utilize many DataSource structures.

The data model is shown in Figure 1. Note the hierarchy: a single Report Definition contains one or more DataSources, each DataSource contains one or more Instances, each Instance contains one or more Properties, and each Property contains a name and value.

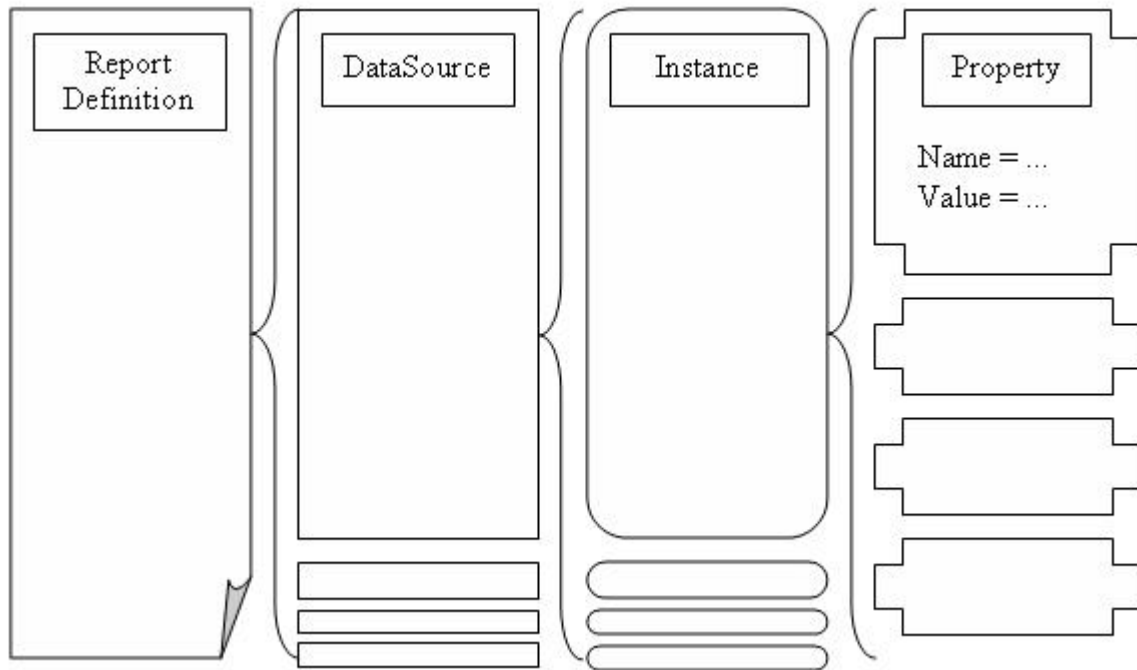


Figure 1. The Open Checklist Reporting Language Data Model

It is important to note that the OCRL data model does not necessarily correspond to any physical representation. That is to say, the OCRL interpreter will not necessarily create any file or database with actual components corresponding to DataSources, Instances, and Properties. Instead, the data model is simply a way to visualize the data collected and

reported by an interpreter. Throughout this document, capitalized "DataSource", "Instance", and "Property" will refer to structures within the OCRL data model.

1.4 The Examples

Sections 2 through 7 walk the reader through examples of report definition files. Each example builds on the previous examples.

For those who wish to jump right into development, example files, corresponding to examples provided below, accompany this document. Users can begin work by copying and editing these files. If any questions arise during this process, simply consult the appropriate sections of this document for detailed commentary on the construction of these files:

- **SingleWMISource.xml** – Report from a single WMI query. (Section 2)
- **MultipleWMISources.xml** – Report from multiple WMI queries. (Section 3)
- **SimpleFileSource.xml** – Report on properties in a file with a simple structure. (Section 4)
- **StructuredFileSource.xml** – Report on properties in a structured file in a way that preserves this structure. (Section 5)
- **NestedStructureFileSource.xml** – Report on properties in a hierarchically structured file. (Section 6)
- **OvalSource.xml** – Report on information gathered through the use of the OVAL language. (Section 7)

Note that all of these examples could have been placed together in a single file, but this was not done in order to make them easier to use as examples.

Section 2

Example: Single WMI Source

Table 1 below is a simple example of a report definition. It collects data from a single WMI query and formats it into a report.

Table 1. One Property from One WMI Query

Line #	XML
1	<ReportDefinitions>
2	<ReportDefinition id="SMS-9-17" filename="Report-SMS-9-17" title="Users who can create new collections of targets." recommendation="Limit the number of administrators who can create new collections." instruction="Review the following account information to decide whether any of the listed users do not need to create new collections:">
3	<DataSources>
4	<WmiDataSource id="wmi-5">
5	<Namespace>root\SMS\site-MEL</Namespace>
6	<Class>SMS_UserClassPermissionNames</Class>
7	<Property>UserName</Property>
8	<Where>ObjectKey = 1 AND PermissionName = "Create"</Where>
9	</WmiDataSource>
10	</DataSources>
11	<Reporting>
12	<Text>The following accounts have Create permission on the Collections class:</Text>
13	<Select datasourceid="wmi-5" instances="all">
14	<Item name="UserName" />
15	<Text/>
16	</Select>
17	</Reporting>
18	</ReportDefinition>
19	</ReportDefinitions>

Stepping through this line by line:

- Line 1 - The opening tag for the report document.
- Line 2 - Start a report definition. This also provides, via its attributes, some general information about this report. Specifically, the attributes indicate:
 - The id of this report is "SMS-9-17". (The "id" attribute.)

- The report file created by the interpreter for this report should be named "*date_and_timestamp_Report-SMS-9-17*" for some date and timestamp. (The "filename" attribute.)
 - The title of the report should be "Users who can create new collections of targets." (The "title" attribute.)
 - The recommendation for which this report was created is "Limit the number of administrators who can create new collections." (The "recommendation" attribute.)
 - The instructions to the user in the output report file are "Review the following account information to decide whether any of the listed users do not need to create new collections:" (The "instruction" attribute.)
- Line 3 - Starts the DataSources section of the document. This section identifies the data that is used to generate the report file.
 - Line 4 - Starts the specification of one DataSource that uses a WMI query to guide the extraction of data. The required "id" attribute is the name used to identify this DataSource; this name is used in the Reporting section of the document to reference the desired data. All DataSources must have their own unique id. This DataSource's id is "wmi-5".
 - Line 5 - The <Namespace> element provides the namespace from which the WMI query should select data.
 - Lines 6-8 - In a WMI DataSource, the data is gathered using a WMI query. These lines describe the query that should be sent to the WMI provider. In this case, the resulting WMI query reads as follows:

```
SELECT UserName FROM SMS_UserClassPermissionNames
WHERE ObjectKey = 1 AND PermissionName = "Create"
```

The "UserName" comes from the <Property> element, the "SMS_UserClassPermissionNames" comes from the <Class> element, and the WHERE clause of the query comes from the <Where> element. The <Where> element is optional in a WMI query. The <Property> element can also be omitted, in which case all WMI properties in the specified class are gathered. The author may also specify multiple <Property> elements within a WMI DataSource, in which case the comma-separated list of these properties appears after the SELECT in the final query. The <Class> element must appear exactly once. These elements, if present, must appear in the order given in the example: first <Class>, then any <Property> elements, and finally a <Where> element, if used.

- Line 9 - Closes off the definition of the WMI DataSource with id "wmi-5".

- Line 10 - Closes off the definition of data sources for this report.
- Line 11 - Starts the Reporting section of the report definition file. This section is responsible for selecting data gathered in the DataSources section and organizing it for display.
- Line 12 - A <Text> element. This element identifies literal text that should appear in the report file.
- Line 13 - A <Select> element. The <Select> element is responsible for selecting Instances from a DataSource. (Recall the DataSources/Instances/Properties data model used by the tool.) The "datasourceid" attribute of the <Select> element identifies the DataSource from which Instances should be extracted (in this case, "wmi-5", the only DataSource in the document). The "instances" attribute identifies which of the Instances in this DataSource should be selected. In this case, a value of "all" indicates that every top-level Instance in the DataSource should be selected. Note that, since the Instances in the OCRL data model can be arranged hierarchically, the "top-level" Instances would not include Instances that were children of other Instances. This document will talk more about parent and child Instances in Section 6. Other valid values for the "instances" attribute are "first" and "last", in which case only the first or last Instance is selected, respectively.
WMI queries can return multiple WMI instances of the named class. Each of these WMI instances corresponds to an Instance in the OCRL data model. The <Select> block can be translated as saying, "Select the Instances from the given DataSource (modulo any limitation on that given by the "instance" attribute) then process the body of the <Select> for each of these selected Instances.
- Line 14 - An <Item> element. This element is used to identify the Property of an Instance for output. The "name" attribute of the <Item> gives the name of the Property of interest (in this case, the "UserName" Property). This line dictates that this Property should appear in the resulting report file.
Because <Item> elements identify Properties of Instances they must appear within a <Select> element since it is the <Select> that identifies an Instance. Because of this, if the <Select> element iterates over multiple Instances from a DataSource, any <Item> element in its body is applied to each selected Instance individually. If four Instances were selected, the contained <Item> element grabs the named Property four times, once from each of these Instances in turn.
- Line 15 - A <Text> element with no body. This simply places an empty line of text in the report file (a newline). This is used to make the formatting of the file more understandable.
- The remaining lines simply close out the XML of the document.

Below is an example of what a report produced by the above definition might look like. It is important to note that the appearance of the report file below is just one example – different interpreters may use colors, tables, or other structures to format the output.

```
*****
*****
* Users who can create new collections of targets. *
*****
*****

Recommendation: Limit the number of administrators who can create new
collections.
*****

Instructions: Review the following account information to decide whether
any of the listed users do not need to create new collections:

The following accounts have Create permission on the Collections class:
UserName: Lisa

UserName: Len

UserName: Charles

UserName: Shaan

UserName: Linda
```

In the report generated by this particular implementation of the OCRL interpreter, the title, recommendation, and instructions appear at the beginning of the file. Following this information is the contents of the <Text> block in line 12 of the <Reporting> section of the report definition file. Below this appear five Property name-value pairs. What happened here was that the DataSource "wmi-5" collected five WMI instances in response to its query. Each of these became one Instance within the wmi-5 DataSource. In the <Reporting> section, the <Select> tag selects "all" the Instances in the wmi-5 DataSource. The <Select> block then iterated over all five of these selected Instances applying its body to each Instance in turn. The body of the <Select> simply consisted of the single <Item> element, which indicated the "UserName" Property should be reported in the report file, and the empty <Text> element, which is responsible for the blank line that separates each of the listed Properties. As a result, the report file displays the "UserName" Property for the first of the five Instances (UserName: Lisa) followed by the "UserName" Property for the second Instance (UserName: Len) followed by the "UserName" Property for the third Instance (UserName: Charles) and so on.

Section 3

Example: Multiple WMI Sources

Sometimes we wish to include more information in generated reports by combining the data from multiple data sources into a single report. This example expands on the previous report definition file to do just that. The document shown in Table 2 uses two <WmiDataSource> elements to create two DataSources and combines them in the <Report> section.

Table 2. Multiple Data Sources from Multiple WMI Queries

Line #	XML
1	<ReportDefinitions>
2	<ReportDefinition id="SMS-9-17" filename="Report-SMS-9-17" title="Users who can create new collections of targets." recommendation="Limit the number of administrators who can create new collections.">
3	<DataSources>
4	<WmiDataSource id="wmi-5">
5	<Namespace>root\SMS\site-MEL</Namespace>
6	<Query>SELECT UserName FROM SMS_UserClassPermissionNames WHERE ObjectKey = 1 AND PermissionName = "Create"</Query>
7	</WmiDataSource>
8	<WmiDataSource id="wmi-6">
9	<Namespace>root\SMS\site-MEL</Namespace>
10	<Class>SMS_R_User</Class>
11	<Property>UserGroupName</Property>
12	<Property>UniqueUserName</Property>
13	</WmiDataSource>
14	</DataSources>
15	<Reporting>
16	<Text>The following accounts have Create permission on the Collections class:</Text>
17	<Select datasourceid="wmi-5" instances="all">
18	<Item name="UserName" setvariable="theUniqueUserName"/>
19	<Text>This account belongs to the following groups:</Text>
20	<Select datasourceid="wmi-6" instances="all">
21	<Item name="UserGroupName">
22	<Qualifier operator="equal" operatortype="string">
23	<Property name="UniqueUserName"/>
24	<Variable name="theUniqueUserName"/>
25	</Qualifier>

26	</Item>
27	</Select>
28	<Text />
29	</Select>
30	</Reporting>
31	</ReportDefinition>
32	</ReportDefinitions>

A line-by-line analysis of this report definition follows. Lines that are unchanged from the previous example receive minimal attention:

- Line 1 – Start the file.
- Line 2 – Start a report definition.
- Line 3 – Start the DataSources section.
- Lines 4-7 – Define the first WmiDataSource (wmi-5). This is the same WMI query as in the previous example, but this time instead of using the <Property>, <Class> and <Where> elements to set the individual pieces of the WMI query, we simply use the <Query> element to give the entire query itself. These two methods are equivalent and both definitions of this data source produce the same results.
- Lines 8-13 – This defines a second WmiDataSource with the id "wmi-6". This also connects to the "root\SMS\site-MEL" namespace. The resulting WMI query looks like the following:

```
SELECT UserGroupNames, UniqueUserName FROM SMS_R_User
```

Note that this query identifies two <Property> values, "UserGroupNames" and "UniqueUserName". Note that it also has no <Where> element.

- Line 14 – Close the DataSource definition block.
- Line 15 – Begin the Reporting block.
- Line 16 – Start the report with a text string.
- Line 17 – Select all instances from the DataSource "wmi-5".
- Line 18 – As before, we call out the Property named "UserName" to appear in the report file. In addition, the optional "setvariable" attribute has been added and set to "theUniqueUserName". This creates a new Variable with the name "theUniqueUserName" and a value equal to the value of the "UserName" Property of this Instance.
- Line 19 – This is another <Text> line. The contained text appears in the report file immediately after each "UserName" Property is displayed.

- Line 20 – This is another <Select> block that selects "all" the Instances within the "wmi-6" DataSource. Note that because it appears within the <Select> block for the "wmi-5" DataSource, this <Select> block is processed once for every Instance in the "wmi-5" DataSource.
- Line 21 – This is an <Item> tag. Because it appears within the "wmi-6" <Select> block, it identifies a Property within an Instance from the "wmi-6" DataSource.
- Line 22 – This is a <Qualifier> element tag. A <Qualifier> block identifies a condition. Its parent <Item> tag contributes to the report file only if this condition is met. The <Qualifier> tag itself has two attributes: "operator", which is used to specify the comparison operation that must be met, for example, "equal" or "greater than", and "operatortype", which identifies the operands as being either "string" or "number".
- Lines 23 & 24 – These represent the operands of the <Qualifier>. In particular, the value of the "UniqueUserName" Property is compared against the value of the "theUniqueUserName" Variable. (Recall that the "theUniqueUserName" Variable was defined in line 18.)
- Lines 25-27 – These lines close out the inner <Select>.
- Line 28 – An empty <Text> element to add a blank line to the report at the end of each iteration of the outer <Select>.
- Lines 29-32 – These lines simply close out the remaining elements in the document.

The report generated by such a file might look like the following:

```
*****
*****
* Users who can create new collections of targets. *
*****
*****
```

```
Recommendation: Limit the number of administrators who can create new
collections.
```

```
*****
```

```
Instructions: Review the following account information to decide whether
changes are needed for compliance with the recommendation:
```

```
The following accounts have Create permission on the Collections class:
```

```
UserName: Lisa
```

```
This account belongs to the following groups:
```

```
UserGroupName: Admins
UserGroupName: Administrator
```

```
UserName: Len
This account belongs to the following groups:
UserGroupName: Administrator
UserGroupName: SMS Operator
```

```
UserName: Charles
This account belongs to the following groups:
```

```
UserName: Shaan
This account belongs to the following groups:
```

```
UserName: Linda
This account belongs to the following groups:
```

Again, we see the file starts with the information provided in the `<ReportDefinition>` element as well as the `<Text>` string ("The following accounts have...") just as before. Below this we see the same five `UserName` values we saw in the previous example but each of these is now followed by the second `<Text>` string ("This account belongs to...").

Below the first two `UserName` entries we also see `UserGroupName` Properties. The `UserGroupName` Properties come from the inner `<Select>` block. To understand what is happening here, it may be helpful to view the nested `<Select>` blocks as nested loops within a program:

```
DISPLAY Text "The following accounts have Create permission on the Collections class:"
FOR-EACH Instance in wmi-5
{
  DISPLAY Property UserName
  DISPLAY Text "This account belongs to the following groups:"
  FOR-EACH Instance in wmi-6
  {
    IF (Property UniqueUserName == Variable theUniqueUserName)
      DISPLAY Property UserGroupName
  }
  DISPLAY Text ""
}
```

The `wmi-6` `DataSource` recovered several Instances from its WMI query and the inner `<Select>` loop iterates through all these Instances. On each iteration of the inner `<Select>` the interpreter reports the value of the `UserGroupName` Property, but only if the `<Qualifier>` condition was met. (The qualifier condition being that the value of the `UniqueUserName`

Property of the current Instance matched the value of the theUniqueUserName Variable). We can see that this condition was met in two wmi-6 Instances of the first Instance of wmi-5 (UserName: Lisa) and was met in two wmi-6 Instances of the second Instance of wmi-5 (UserName: Len), but that it was not met for any of the other combinations of Instances of wmi-5 and wmi-6. This is why we see two UserGroupName Properties listed after the first two UserName Properties but no UserGroupName Properties after the last three UserName Properties.

3.1 Variation: The "hidden" Attribute

In the previous example, the UserName property does double duty: it is written to the report file and it is used to set the value of a Variable. Sometimes, an author may wish the latter behavior without the former. That is, they may wish to set a Variable, but not write the value of the Property at the same time. This can be accomplished using the "hidden" attribute of the <Property> element as demonstrated in Table 3. By setting the value of this attribute to "true" the Variable is set without this Property being added to the report file. (As we have seen, this attribute is optional. By default, its value is false.)

Table 3. The "hidden" Attribute

Line #	XML
15	<Reporting>
16	<Text>The following accounts have Create permission on the Collections class:</Text>
17	<Select datasourceid="wmi-5" instances="all">
18	<Item name="UserName" setvariable="theUniqueUserName" hidden="true"/>
19	<Text>This account belongs to the following groups:</Text>
20	<Select datasourceid="wmi-6" instances="all">
21	<Item name="UserGroupName">
22	<Qualifier operator="equals" operatortype="string">
23	<Property name="UniqueUserName" />
24	<Variable name="theUniqueUserName" />
25	</Qualifier>
26	</Item>
27	</Select>
28	<Text />
29	</Select>
30	</Reporting>

The above XML just adds the "hidden" attribute to the <Item> element that grabs the "UserName" Property. The resulting report file for this report definition is the same as before except that the UserName Property does not appear in the output report:

```
*****
*****
* Users who can create new collections of targets. *
*****
*****
```

Recommendation: Limit the number of administrators who can create new collections.

```
*****
```

Instructions: Review the following account information to decide whether changes are needed for compliance with the recommendation:

The following accounts have Create permission on the Collections class:
This account belongs to the following groups:
UserGroupName: Admins
UserGroupName: Administrator

This account belongs to the following groups:
UserGroupName: Administrator
UserGroupName: SMS Operator

This account belongs to the following groups:

This account belongs to the following groups:

This account belongs to the following groups:

3.2 Variation: Complex Qualifiers

In the original Section 3 example we showed how the presence of an Item in a report could be predicated on some single condition as expressed in a <Qualifier> element. However, sometimes there may be multiple conditions that we wish to consider when deciding whether a particular Property should appear. Fortunately, OCRL is capable of supporting these situations through the use of complex qualifiers.

Complex qualifiers use logical <AND>, <OR>, and <NOT> elements to combine multiple <Qualifier> elements. These logical elements can be nested in a variety of ways to produce sophisticated conditions. Consider Table 4, which contains a variation of the <Reporting> section from Table 2.

Table 4. A Complex Qualifier

Line #	XML
15	<Reporting>
16	<Text>The following accounts have Create permission on the Collections class:</Text>
17	<Select datasourceid="wmi-5" instances="all">
18	<Item name="UserName" setvariable="theUniqueUserName"/>
19	<Text>This account belongs to the following groups:</Text>
20	<Select datasourceid="wmi-6" instances="all">
21	<Item name="UserGroupName">
	<AND>
22	<Qualifier operator="equal" operatortype="string">
23	<Property name="UniqueUserName" />
24	<Variable name="theUniqueUserName" />
25	</Qualifier>
26	<NOT>
27	<OR>
28	<Qualifier operator="equal" operatortype="string">
29	<Property name="OverrideField" />
30	<FixedValue>yes</FixedValue>
31	</Qualifier>
32	<Qualifier operator="greater than" operatortype="number">
33	<Property name="ExceptionCases" />
34	<FixedValue>3</FixedValue>
35	</Qualifier>
36	</OR>
37	</NOT>
38	</AND>
39	</Item>
40	</Select>
41	<Text />
42	</Select>
43	</Reporting>

The above example contains three qualifiers. To facilitate discussions, this document gives them the following names: Qualifier_A (lines 22 – 25), Qualifier_B (lines 28 – 31), and Qualifier_C (lines 32 – 35). The Item with the name "UserGroupName" only appears if Qualifier_A is true and neither Qualifier_B nor Qualifier_C are true. In pseudocode, this might look like:

(Qualifier_A AND NOT(Qualifier_B OR Qualifier_C))

The above example also provides an example of the third type of operand a <Qualifier> element may have: the <FixedValue> operand. A <FixedValue> operand simply uses its

body as the value against which the value of the other operand is compared. In the case of Qualifier_B, the <Qualifier> evaluates to true if the value of the OverrideField Property within the current Instance was equal to "yes".

Qualifier_C also uses a <FixedValue> operand, but in this <Qualifier> the operator is "greater than" and the operatortype is "number". In this case, the value of both operands is converted into a numeric value and the <Qualifier> evaluates to true only if the numeric value of the Property named "ExceptionCases" in the current Instance is greater than the value of the <FixedValue> operand of 3.

3.3 Variation: Using <Qualifier> with <Select> Elements

In the previous example, we showed the use of a <Qualifier> condition to determine whether an <Item> tag contributes to the result file. Conditions can also be applied to <Select> blocks themselves. A <Qualifier> in this context is used to filter the Instances through which the <Select> iterates. Consider Table 5, which contains a modified excerpt from Table 2:

Table 5. Using <Qualifier> Elements with <Select> Elements

Line #	XML
15	<Reporting>
16	<Text>The following accounts have Create permission on the Collections class:</Text>
17	<Select datasourceid="wmi-5" instances="all">
18	<Item name="UserName" setvariable="theUniqueUserName" />
19	<Text>This account belongs to the following groups:</Text>
20	<Select datasourceid="wmi-6" instances="all">
21	<Qualifier operator="equal" operatortype="string">
22	<Property name="UniqueUserName" />
23	<Variable name="theUniqueUserName" />
24	</Qualifier>
25	<Item name="UserGroupName" />
26	</Select>
	</Text>
27	</Select>
28	</Reporting>

The <Reporting> section from Table 2 has been duplicated, but the <Qualifier> has been moved from within the <Item> element to within the <Select> element. In this new position, the <Qualifier> screens Instances as a whole. This is to say, as the <Select> tag iterates through all the Instances in the DataSource, for each Instance the body of the <Select> is processed only if the <Qualifier> condition is met.

In the above example, the generated report file looks exactly the same as the report generated by the report definition in Table 2. This is because the <Item> element calling out the UserGroupName Property is the entirety of the inner <Select> element's body. As a result, excluding any Instances that do not meet the <Qualifier> element's condition has exactly the same effect as printing the one line of output for that Instance only when the same condition was met. Table 6 contains a more interesting example by including a <Text> element and multiple <Item> elements in the body of the <Select>:

Table 6. A <Qualifier> Element in a <Select> Element with an Expanded Body

Line #	XML
20	<Select datasourceid="wmi-6" instances="all">
21	<Qualifier operator="equal" operatortype="string">
22	<Property name="UniqueUserName" />
23	<Variable name="theUniqueUserName" />
24	</Qualifier>
25	<Text>User and group:</Text>
26	<Item name="UniqueUserName" />
27	<Item name="UserGroupName" />
28	</Select>

Now we have a <Text> element and two <Item> elements in the body of the inner <Select>. In this case, for each Instance in the DataSource, if the <Qualifier> element's condition is met, we see the designated text followed by both Properties in the report document. If the <Qualifier> element's condition is not met for a given Instance, neither the text nor the Properties are printed for that Instance. The report generated by this report definition might look like the following:

```

*****
*****
* Users who can create new collections of targets. *
*****
*****

Recommendation: Limit the number of administrators who can create new
collections.
*****

Instructions: Review the following account information to decide whether
changes are needed for compliance with the recommendation:

The following accounts have Create permission on the Collections class:

```

```
UserName: Lisa
This account belongs to the following groups:
User and group:
UniqueUserName: Lisa
UserGroupName: Admins
User and group:
UniqueUserName: Lisa
UserGroupName: Administrator
```

```
UserName: Len
This account belongs to the following groups:
User and group:
UniqueUserName: Len
UserGroupName: Administrator
User and group:
UniqueUserName: Len
UserGroupName: SMS Operator
```

```
UserName: Charles
This account belongs to the following groups:
```

```
UserName: Shaan
This account belongs to the following groups:
```

```
UserName: Linda
This account belongs to the following groups:
```

Note that in the above report nothing appears in the group listing for the users Charles, Shaan, or Linda. This happens even though the `<Text>` and `<Item>` elements within the inner `<Select>` do not contain their own `<Qualifier>` elements. (In fact, `<Text>` elements cannot contain `<Qualifier>` elements in the first place.) There is nothing listed after these users because the `<Qualifier>` applied to the inner `<Select>` skips all instances in the inner `<Select>` element's `DataSource` that do not meet the given criteria. As we saw from the report produced by the original example, the `<Qualifier>` element's condition is never met for any of these three users. As a result, none of the elements in the body of the inner `<Select>`, even the `<Text>` element, are printed to the report.

Complex qualifiers, as described in the previous variation, can also be used to constrain `<Select>` elements. The structure and function of complex qualifiers is the same for both `<Item>` and `<Select>` elements.

Finally, it should be noted that OCRL also accepts a `<Where>` tag as being equivalent to a `<Qualifier>` tag when constraining a `<Select>` element. This was done to conform to the familiar syntax of SQL where a `SELECT` can be filtered by using a `WHERE` clause. The

syntax and functionality of a <Where> is identical to that of the <Qualifier> element and authors may use whichever one they feel comfortable. Note however, that while a <Where> may be used instead of a complex qualifier, it may not be used within a complex qualifier. The child elements of logical elements, <AND>, <OR>, and <NOT>, must either be a <Qualifier> element or another logical element.

Intentionally Blank

Section 4

Example: Simple File Source

We now turn to using a file as a DataSource. Table 7 is a simple example of extracting and reporting two Properties from a file:

Table 7. Extracting from a File Data Source

Line #	XML
1	<ReportDefinitions>
2	<ReportDefinition id="SWT-1" filename="Walkthrough Report" title="Network Discovery" recommendation="Enable network discovery and set the ICMP Ping Timeout to a value appropriate for your network." instruction="Review the following information: network discovery should be enabled (set to TRUE); ensure that the ICMP Ping Timeout value is appropriate and modify it in the system's configuration file if necessary.">
3	<DataSources>
4	<FileDataSource id="discovery-1">
5	<Filename directory="C:\Discovery Report">NestingTest.txt</Filename>
6	<InstanceStructures>
7	<InstanceDeclarations>
8	<Instance classid="WholeFile">
9	<Property name="Discovery Enabled">
10	<PatternMatch>PROPERTY <Discovery Enabled><([^\>]*)></PatternMatch>
11	</Property>
12	<Property name="Ping Timeout">
13	<PatternMatch>PROPERTY <ICMP Ping Timeout><([^\>]*)></PatternMatch>
14	</Property>
15	</Instance>
16	</InstanceDeclarations>
17	</InstanceStructures>
18	</FileDataSource>
19	</DataSources>
20	<Reporting>
21	<Select datasourceid="discovery-1" instances="all">
22	<Item name="Discovery Enabled"/>
23	<Item name="Ping Timeout"/>
24	</Select>
25	</Reporting>
26	</ReportDefinition>
27	</ReportDefinitions>

Going through line-by-line:

- Line 1 – Start of the file.
- Line 2 – Start of the report definition. The meanings of the various attributes are the same as with the previous examples.
- Line 3 – Begin the DataSources section.
- Line 4 – Begin defining a new FileDataSource.
- Line 5 – The <Filename> element identifies the name of the file that should be parsed for configuration information ("NestingTest.txt"). The "directory" attribute further identifies that this file is located in the "C:\Discovery Report" directory.
- Line 6 - Unlike WMI where Instances are predefined in a query's return values, the division of a file into logical blocks is more complicated. Because different files may use different markers to indicate boundaries between logical units, it is necessary to explicitly call out the divisions of Instances. The <InstanceStructures> element encapsulates this information.
- Line 7 – Like variables in a programming language, Instance structures can be defined (given a meaning for later use), declared (applying a defined Instance in a particular context), or both. The <InstanceDeclarations> holds information that is being declared or both declared and defined. This document talks more about definition vs. declaration during the discussion of <InstanceReference> elements in Section 6.1.
- Line 8 –This is the start of an <Instance> element. <Instance> elements describe an Instance-class. An Instance-class is a description of how to locate an Instance within a source file, what information one might find in the portion of the source file associated with this Instance, and how to recognize and extract this information. Because this <Instance> element appears within the <InstanceDeclarations> element, it both defines an Instance-class with certain structures and declares that this Instance-class can be located in a particular context. The structures that make up the defined Instance-class are described by the <Instance> element's attributes and child elements as outlined below. Because the <Instance> element appears as an immediate child of the <InstanceDeclarations> element, the context where the OCRL interpreter looks for this Instance is the body of the source file rather than just within the bounds of some parent Instance. (This document discusses Instances within parent Instances in Section 6.)
In this example, the <Instance> element's "classid" attribute has a value of "WholeFile". The purpose of this attribute is to provide an identifier for a given set of Instances. This can be useful to distinguish between Instances created using different definitions, although in this case, the "classid" is superfluous since no other <Instance> is defined.

An <Instance> element's first two children can be <Begin> and <End> elements. These child elements describe markers in the file that indicate where an Instance begins or ends, respectively. All the Properties extracted from the file between those points belong to a single Instance. The absence of these two child elements in the current example indicates that the Instance created in this DataSource begins at the start of the file and ends at its end. (In other words, there is only one Instance and it represents the entire file.) The example in Section 5 describes how to use the <Begin> and <End> elements to parse files into multiple Instances. For now, it is simply necessary to understand that the lack of any <Begin> or <End> elements within the <Instance> element indicates that all Properties recovered from the file are placed within the same, single Instance.

- Line 9 – Since different files may store relevant information in different ways, report definition authors must describe how to identify this information within a file. The <Property> element and its children are responsible for this along with providing a name by which this information may be referenced. Within the <Property> element itself, the "name" attribute assigns a name to a Property. Child elements of <Property> are used to locate the value of this named Property within the file.
- Line 10 – As mentioned above, it is necessary to describe how to recognize a particular Property within a file. Here, we use the <PatternMatch> element to identify the value of this Property. A <PatternMatch> contains a regular expression. If a line matching the regular expression is found, the "grab" from the regular expression (the part of the pattern contained in parenthesis) becomes the value of the Property. If there is no "grab" section in the pattern, the entire matching line becomes the value of the Property. Here the pattern to identify the value is:

```
PROPERTY <Discovery Enabled><([>]*)>
```

Note that in the example, in order to wrap this regular expression in XML, it was necessary to encode the < and > as < and > respectively. This regular expression dictates that the text between the < and > immediately following the string "PROPERTY <Discovery Enabled>" should become the value of this Property. If no line exists that matches the regular expression, the interpreter tool assumes that no such Property exists within the enclosing Instance.

- Line 11 – Closes this Property definition.
- Lines 12-14 – Defines a second property with the name "Ping Timeout" and a value that is located using the regular expression:

```
PROPERTY <ICMP Ping Timeout><([>]*)>
```

- Line 15 – Closes out the Instance definition.

- Line 16-17 – Closes out the Instance declarations and the description of Instance structures for this data source.
- Line 18 – Closes out the definition of this FileDataSource.
- Line 19 – Closes out the DataSources definition section.
- Line 20-25 – This describes how to construct the report file. Its structure should be familiar: for each Instance in the DataSource, write the Discovery Enabled and Ping Timeout properties to the report. As described above, this particular DataSource only has one Instance corresponding to the entire file. As such, the <Select> only iterates through once.
- Line 26-27 – Close out the file.

In short, this report definition file identifies a file and describes how to identify two Properties within the file. The order in which the Properties are defined in the <Instance> element is unimportant – they are matched in the source file in whatever order they happen to occur there. These two Properties are then displayed in the report file. Assume the file NestingTest.txt looks like the following:

```

BEGIN_COMPONENT
  <SMS_NETWORK_DISCOVERY>
  <6>
  <SMS-2003-SERVER>
  PROPERTY <Discovery Enabled><TRUE><><0>
  PROPERTY <NetToMediaTable Retrieval><ALL_DEVICES><><0>
  PROPERTY <Maximum Number Outstanding ICMP Pings><50><><0>
  PROPERTY <ICMP Ping Timeout><1000><><0>
  PROPERTY <Number Concurrent Device Sessions><10><><0>
  PROPERTY <SNMP Retry Count><1><><0>
  PROPERTY <SNMP Retry Timeout><1000><><0>
  PROPERTY <Subnet Include Local><TRUE><><0>
  PROPERTY <DHCP Include Local><FALSE><><0>
  PROPERTY <Domain Include Local><TRUE><><0>
END_COMPONENT

```

The regular expression pattern for the Property with the name "Discovery Enabled" matches line 5 of the file while the regular expression pattern for the Property with the name "Ping Timeout" matches line 8. Below is a sample report file given this source file:

```
*****
*****
* Network Discovery *
*****
*****
```

Recommendation: Enable network discovery and set the ICMP Ping Timeout to a value appropriate for your network.

```
*****
```

Instructions: Review the following information: network discovery should be enabled (set to TRUE); ensure that the ICMP Ping Timeout value is appropriate and modify it in the system's configuration file if necessary.

```
Discovery Enabled: TRUE
Ping Timeout: 1000
```

Sometimes a source file may contain multiple lines that match a regular expression for a given Property. When that happens, all matching lines map to Properties of the given name within the same Instance. For example:

```
BEGIN_COMPONENT
  <SMS_NETWORK_DISCOVERY>
  <6>
  <SMS-2003-SERVER>
  PROPERTY <Discovery Enabled><TRUE-January><><0>
  PROPERTY <Discovery Enabled><TRUE-February><><0>
  PROPERTY <Discovery Enabled><FALSE-March><><0>
  PROPERTY <Discovery Enabled><TRUE-April><><0>
  PROPERTY <NetToMediaTable Retrieval><ALL_DEVICES><><0>
  PROPERTY <Maximum Number Outstanding ICMP Pings><50><><0>
  PROPERTY <ICMP Ping Timeout><1000><><0>
  PROPERTY <Number Concurrent Device Sessions><10><><0>
  PROPERTY <SNMP Retry Count><1><><0>
  PROPERTY <SNMP Retry Timeout><1000><><0>
  PROPERTY <Subnet Include Local><TRUE><><0>
  PROPERTY <DHCP Include Local><FALSE><><0>
  PROPERTY <Domain Include Local><TRUE><><0>
END_COMPONENT
```

This file contains multiple lines that match the pattern associated with the Discovery Enabled Property. The exact way in which the OCRL interpreter displays multiple Properties

with the same name in a report can vary, but it should not result in an error or loss of data. One possible way to handle this case is to simply add each instance of the Property to the report file where the corresponding Item dictates:

```
*****
*****
* Network Discovery *
*****
*****
```

Recommendation: Enable network discovery and set the ICMP Ping Timeout to a value appropriate for your network.

```
*****
```

Instructions: Review the following information: network discovery should be enabled (set to TRUE); ensure that the ICMP Ping Timeout value is appropriate and modify it in the system's configuration file if necessary.

```
Discovery Enabled: TRUE-January
Discovery Enabled: TRUE-February
Discovery Enabled: FALSE-March
Discovery Enabled: TRUE-April
Ping Timeout: 1000
```

Other interpreters might display such multiply-instantiated Properties in a comma-separated list or some other structure.

4.1 Variation: Executable Data Sources

Using the output of command-line executables as data sources is supported by OCRL and is very similar in structure to using a file as a data source. Consider the DataSource definition in Table 8:

Table 8. An Executable Data Source

Line #	XML
4	<ExecutableDataSource id="discovery-1">
5	<Executable directory="C:\OraHome\bin">checkProperties.exe</Executable>
6	<Arguments>-oL -p</Arguments>
7	<InstanceStructures>
8	<InstanceDeclarations>
9	<Instance classid="WholeFile">
10	<Property name="Discovery Enabled">

11	<code><PatternMatch>PROPERTY &lt;Discovery Enabled&gt;&lt;([^\&gt;]*)&gt;</PatternMatch></code>
12	<code></Property></code>
13	<code><Property name="Ping Timeout"></code>
14	<code><PatternMatch>PROPERTY &lt;ICMP Ping Timeout&gt;&lt;([^\&gt;]*)&gt;</PatternMatch></code>
15	<code></Property></code>
16	<code></Instance></code>
17	<code></InstanceDeclarations></code>
18	<code></InstanceStructures></code>
19	<code></ExecutableDataSource></code>

In the above example, we define a new `<ExecutableDataSource>`. The `<Executable>` element (line 5) serves a similar role to the `<Filename>` element in a `<FileDataSource>` except that it identifies an executable file to run rather than a text file to read. The optional "directory" attribute provides the path for the executable if the executable is not in the user's regular system path. In line 6 we have the optional `<Arguments>` element, whose body holds any arguments that should be appended to the executable when it is run.

The rest of the body of the `<ExecutableDataSource>` element consists of the same `<InstanceStructures>` element that was in the previous `<FileDataSource>` example. OCRL assumes that the given executable (with the given arguments) produces text output to the standard output window. The OCRL interpreter runs the named executable and parses this output text just like it parses a file's content. As such, after the `<Executable>` and `<Arguments>` elements, the body of an `<ExecutableDataSource>` can have the same structures as a `<FileDataSource>`.

Intentionally Blank

Section 5

Example: Structured File Source

We noted earlier that when a top-level <Instance> element lacks a <Begin> and/or <End> child element, the entire source file is treated as a single Instance. However, files often contain organized blocks of information that we want to treat as Instances for the purposes of reporting. OCRL supports this by allowing users to specify patterns that mark the boundaries of logical units of a file. As the OCRL interpreter parses a file data source, it looks for these bounding markers, creating a new Instance every time they are encountered. The <Begin> and <End> elements of an <Instance> are what are used to describe these bounding markers. Consider the report definition file shown in Table 9:

Table 9. Extracting Structure from Files

Line #	XML
1	<ReportDefinitions>
2	<ReportDefinition id="SWT-1" filename="Walkthrough Report" title="Network Discovery" recommendation="Enable network discovery and set the ICMP Ping Timeout appropriately." instruction="Review the following information: network discovery should be enabled (set to TRUE); ensure that the ICMP Ping Timeout value is appropriate and modify it in the system's configuration file if necessary.">
3	<DataSources>
4	<FileDataSource id="discovery-1">
5	<Filename directory="C:\Discovery Report">NestingTest.txt</Filename>
6	<InstanceStructures>
7	<InstanceDeclarations>
8	<Instance classid="System">
9	<Begin caseInsensitive="true">BEGIN_COMPONENT</Begin>
10	<End caseInsensitive="true">END_COMPONENT</End>
11	<Property name="Instance Name">
12	<Line>1</Line>
13	</Property>
14	<Property name="Discovery Enabled">
15	<PatternMatch>PROPERTY <Discovery Enabled><([^\>]*)></PatternMatch>
16	</Property>
17	<Property name="ICMP Ping Timeout">
18	<PatternMatch>PROPERTY <ICMP Ping Timeout><([^\>]*)></PatternMatch>
19	</Property>
20	</Instance>

21	<code></InstanceDeclarations></code>
22	<code></InstanceStructures></code>
23	<code></FileDataSource></code>
24	<code></DataSources></code>
25	<code><Reporting></code>
26	<code><Select datasourceid="discovery-1" instances="all"></code>
27	<code><Item name="Instance Name"/></code>
28	<code><Item name="Discovery Enabled"/></code>
29	<code><Item name="ICMP Ping Timeout"/></code>
30	<code><Text /></code>
31	<code></Select></code>
32	<code></Reporting></code>
33	<code></ReportDefinition></code>
34	<code></ReportDefinitions></code>

This report definition file is similar to the previous example. Going through line-by-line:

- Lines 1-7 – These are the same as the previous example.
- Line 8 – The `<Instance>` element indicates that we are starting the definition of an Instance-class, as before, and that this Instance-class can appear at the "top-level" of a file. (By "top-level" we mean to indicate that this Instance-class does not need to appear within the body of some parent Instance-class. Parent and child Instances are discussed in Section 6.) This Instance-class's classid is given as "System".
- Line 9 – The `<Begin>` element identifies the pattern that marks the beginning of an instance within the file. This is a regular expression, although in this case it requires none of the special regular-expression characters to describe the pattern. The optional "caseInsensitive" attribute is present and set to true, indicating that the pattern-matcher should perform a case-insensitive match.
- Line 10 – The `<End>` element identifies the pattern that marks the ending of an instance within the file. Just like begin, the body of this element is a regular expression and the element uses the optional "caseInsensitive" attribute to indicate that the matching should be case-insensitive.
- Lines 11-13 – These lines define a Property within this Instance-class called "Instance Name". Unlike the previous Property definitions we have discussed, instead of having a `<PatternMatch>` define how to extract the value of the Property, this Property definition has a `<Line>` element as its body. Sometimes information we are interested in does not have a unique pattern we can use but its position within an Instance may be well defined. The `<Line>` element indicates the line number of this Instance from which this Property is extracted. In this case, the first line of the Instance (that is, the first line after the line that matches the `<Begin>` pattern) becomes the value of the "Instance Name" property.

Note that the location of the Property is relative to the bounds of the encapsulating Instance. In fact, the parser only looks for the Properties that are the child of the parent Instance if it is parsing between the Instance's boundaries.

- Lines 14-16 – This defines a Property within this Instance-class named "Discovery Enabled" whose value is set by a regular expression. As noted earlier, the interpreter only creates a Property according to this definition if the matching line of text is located between the bounds of the parent Instance. If a line is found that matches the pattern in the <Property> but that line is not between the stated <Begin> and <End> patterns, then this does not result in the creation of a Property.
- Lines 17-19 – This defines a Property within this Instance-class named "ICMP Ping Timeout" whose value is set by a regular expression.
- Line 20 – Close out this <Instance> element.
- Line 21 – Close out the declaration of Instances (<InstanceDeclarations> element).
- Line 22 – Closes out the block describing the structure of the DataSource's Instances (<InstanceStructures> element).
- Lines 23-24 – Closes out this DataSource definition and the DataSources section, respectively.
- Lines 25-32 – The Reporting section contains a single <Select> block which reports three Properties for each instance.
- Lines 33-34 – Close out this document.

In summary, this report definition file defines an Instance-class and delineates it with the <Begin> and <End> patterns. It further declares that this Instance may be found at the top-level of the source file. The interpreter tool goes through the named file attempting to locate a line that matches the <Begin> pattern. Once such a line is located, it attempts to match each of the Property definitions until it reaches the <End> pattern. Since a new Instance is created whenever the <Begin> pattern is located, the file DataSource can now contain multiple Instances.

Consider the following source file:

```
BEGIN_FILE_DEFINITION
  <1>
  <MEL>
  <453>
  PROPERTY <Discovery Enabled><TRUE><><0>
  BEGIN_PROPERTY_LIST
```

```

        <Deltas>
        <Set,"Client Component","Remote Control">
    END_PROPERTY_LIST
END_FILE_DEFINITION
BEGIN_COMPONENT
    <SMS_NETWORK_DISCOVERY>
    <6>
    <SMS-2003-SERVER>
    PROPERTY <Discovery Enabled><TRUE-April><><0>
    PROPERTY <NetToMediaTable Retrieval><ALL_DEVICES><><0>
    PROPERTY <Maximum Number Outstanding ICMP Pings><50><><0>
    PROPERTY <ICMP Ping Timeout><1000><><0>
    PROPERTY <Number Concurrent Device Sessions><10><><0>
    PROPERTY <SNMP Retry Count><1><><0>
END_COMPONENT
BEGIN_COMPONENT
    <SMS_HOST_DISCOVERY>
    <6>
    <SMS-2003-SERVER>
    PROPERTY <Discovery Enabled><TRUE><><0>
    PROPERTY <NetToMediaTable Retrieval><ALL_DEVICES><><0>
    PROPERTY <Maximum Number Outstanding ICMP Pings><50><><0>
    PROPERTY <ICMP Ping Timeout><1000><><0>
    PROPERTY <SNMP Retry Count><1><><0>
END_COMPONENT
EOF

```

In this file we have two lines that match the begin pattern for the Instance-class. As a result, we can expect two Instances to be extracted from this DataSource. Note also that there are three lines that match the pattern for the "Discovery Enabled" Property – one in each Instance block and one preceding them. The first match, outside the Instances, is ignored since the "Discovery Enabled" Property belongs in an Instance and this match is not between the bounding patterns of this Instance. The other two matches set the values of the "Discovery Enabled" Property in each of the respective Instances.

An example report file for this input file appears below:

```

*****
*****
* Network Discovery *
*****
*****

```

Recommendation: Enable network discovery and set the ICMP Ping Timeout to a value appropriate for your network.

Instructions: Review the following information: network discovery should be enabled (set to TRUE); ensure that the ICMP Ping Timeout value is appropriate and modify it in the system's configuration file if necessary.

Instance Name: <SMS_NETWORK_DISCOVERY>
Discovery Enabled: TRUE-April
ICMP Ping Timeout: 1000

Instance Name: <SMS_HOST_DISCOVERY>
Discovery Enabled: TRUE
ICMP Ping Timeout: 1000

Note that, as expected, we see two Instances in the report. Note also that the "Instance Name" Property's value is the value of the first line after the start of each Instance.

5.1 Variation: Combining <Line> and <PatternMatch> Elements

From examination of the source file, it is clear that the "Instance Name" Property (collected from the first line of each Instance) has no distinguishing characteristics that set it apart from other properties within an instance: It is just a "<" followed by some text and closed by a ">". This describes many lines within an Instance block and so is an insufficient pattern to uniquely identify this Property. For this reason, we use the <Line> element to specify that it is the first line of the Instance that is of interest to us. However, we might only want a subset of this line rather than the entire line. In this particular case, we might only want the text without the enclosing spaces and angle brackets. This can be accomplished by using both the <Line> and <PatternMatch> elements together within the Property definition as shown below in Table 10:

Table 10. Combining <Line> and <PatternMatch> Elements

Line #	XML
11	<Property name="Instance Name">
12	<Line>1</Line>
13	<PatternMatch><([>]+)></PatternMatch>
14	</Property>

In this case, to be identified as defining the value of the named Property, a line in the file must pass both tests: it must match the line number in the <Line> element and it must match the regular expression in the <PatternMatch> element. Here the pattern match tag is not very precise and matches most lines within the Instance, but the <Line> element constrains us only to the first line of the Instance. Because the <PatternMatch> uses a grab (parenthesis),

only the grabbed subset of the first line of each Instance is used to set the value of the Property. The report file now looks like the following:

```

*****
*****
* Network Discovery *
*****
*****

Recommendation: Enable network discovery and set the ICMP Ping Timeout to
a value appropriate for your network.
*****

Instructions: Review the following information: network discovery should
be enabled (set to TRUE); ensure that the ICMP Ping Timeout value is
appropriate and modify it in the system's configuration file if necessary.

Instance Name: SMS_NETWORK_DISCOVERY
Discovery Enabled: TRUE-April
ICMP Ping Timeout: 1000

Instance Name: SMS_HOST_DISCOVERY
Discovery Enabled: TRUE
ICMP Ping Timeout: 1000

```

Note that now the "Instance Name" Property's value is now just the text between the angle brackets of the first line of each Instance rather than the entire line.

5.2 Variation: Declaring Multiple Instances

The above example only includes one <Instance> element in the DataSource, but this does not need to be the case. Any number of <Instance> elements can be used within a DataSource, each of them describing a different set of bounding criteria for the Instance structure within the file. Consider Table 11, which contains a modification of the previous <FileDataSource>:

Table 11. Multiple <Instance> Elements in a DataSource

Line #	XML
4	<FileDataSource id="discovery-1">
5	<Filename directory="C:\Discovery Report">NestingTest.txt</Filename>
6	<InstanceStructures>
7	<InstanceDeclarations>

8	<Instance classid="System">
9	<Begin caseInsensitive="true">BEGIN_COMPONENT</Begin>
10	<End caseInsensitive="true">END_COMPONENT</End>
11	<Property name="Instance Name">
12	<Line>1</Line>
13	</Property>
14	<Property name="Discovery Enabled">
15	<PatternMatch>PROPERTY <Discovery Enabled><([^\>]*)></PatternMatch>
16	</Property>
17	</Instance>
18	<Instance classid="Settings">
19	<Begin caseInsensitive="true">BEGIN_SETTING</Begin>
20	<End caseInsensitive="true">END_SETTING</End>
21	<Property name="Instance Name">
22	<Line>1</Line>
23	</Property>
24	<Property name="Port Lock">
25	<PatternMatch>SETTING <Port Lock><([^\>]*)></PatternMatch>
26	</Property>
27	</Instance>
28	</InstanceDeclarations>
29	</InstanceStructures>
30	</FileDataSource>

In this example we have two <Instance> elements in a single DataSource. Note that both of them have different <Begin> and <End> patterns. Both <Instance> definitions contain two <Property> definitions. Note also that both <Instance> definitions have a property named "Instance Name". Because these Properties are in different Instance-classes, however, there is no conflict regarding duplication of Property names.

Assume the following source file:

```

BEGIN_FILE_DEFINITION
  <1>
  <MEL>
END_FILE_DEFINITION
BEGIN_SETTING
  <HTTP>
  SETTING <Port Lock><OFF><><0>
END_SETTING

```

```

BEGIN_COMPONENT
  <SMS_NETWORK_DISCOVERY>
  <6>
  <SMS-2003-SERVER>
  PROPERTY <Discovery Enabled><TRUE-April><><0>
  PROPERTY <NetToMediaTable Retrieval><ALL_DEVICES><><0>
END_COMPONENT
BEGIN_COMPONENT
  <SMS_HOST_DISCOVERY>
  <6>
  <SMS-2003-SERVER>
  PROPERTY <Discovery Enabled><TRUE><><0>
  PROPERTY <NetToMediaTable Retrieval><ALL_DEVICES><><0>
END_COMPONENT
EOF

```

The above file contains two Instances corresponding to the "System" <Instance> element, as identified by its "classid" attribute, and one Instance corresponding to the "Setting" <Instance> element. The respective Properties are extracted as usual.

When reporting, note that a <Select> element iterates over all the Instances of a DataSource. Assume the <Reporting> section shown in Table 12:

Table 12. Reporting Multiple <Instance> Elements

Line #	XML
25	<Reporting>
26	<Select datasourceid="discovery-1" instances="all">
27	<Item name="Instance Name"/>
28	<Item name="Discovery Enabled"/>
29	<Item name="Port Lock"/>
30	<Text />
31	</Select>
32	</Reporting>

Note that there is only one <Select> element in this Reporting block. Note also that it contains three <Item> tags to print out three Properties even though both types of Instances only contain two named Properties. This is because, as noted earlier, the <Select> iterates over all top-level Instances in the DataSource regardless of their class. The output from this Reporting section might look like the following:

```

*****
*****

```

```
* Network Discovery *
*****
*****
```

Recommendation: Enable network discovery and set the ICMP Ping Timeout to a value appropriate for your network.

```
*****
```

Instructions: Review the following information: network discovery should be enabled (set to TRUE); ensure that the ICMP Ping Timeout value is appropriate and modify it in the system's configuration file if necessary.

```
Instance Name: HTTP
Port Lock: OFF
```

```
Instance Name: SMS_NETWORK_DISCOVERY
Discovery Enabled: TRUE-April
```

```
Instance Name: SMS_HOST_DISCOVERY
Discovery Enabled: TRUE
```

Note that the Instances are printed out in the order in which they are read from the file, with the "Setting" Instance first and the two "System" Instances next. Note also that Properties that do not exist within a given Instance do not result in any data being added to the report.

Of course, we might not wish to intermingle different classes of Instances in a Report. OCRL can handle this by using the "classid" attribute. As noted earlier, every Instance created through an <Instance> element automatically creates a Property named ClassId whose value is the value of that <Instance> element's "classid" attribute. We also noted in an earlier variation that we can use a <Qualifier> element within a <Select> element to ignore any Instances in that <Select> element's DataSource that do not meet some set of conditions. Using these two facts we can change the Reporting block so that it includes two <Select> elements each of which only displays one class of Instance. An example of this behavior is shown in Table 13.

Table 13. Reporting Multiple <Instance> Elements Separately

Line #	XML
25	<Reporting>
26	<Text>*** System Values ***</Text>
27	<Select datasourceid="discovery-1" instances="all">
28	<Qualifier operator="equal" operatortype="string">
29	<Parameter name="ClassId"/>

30	<FixedValue>System</FixedValue>
31	</Qualifier>
32	<Item name="Instance Name"/>
33	<Item name="Discovery Enabled"/>
34	<Text />
35	</Select>
36	<Text>*** Settings ***</Text>
37	<Select datasourceid="discovery-1" instances="all">
38	<Qualifier operator="equal" operatortype="string">
39	<Parameter name="ClassId"/>
40	<FixedValue>Settings</FixedValue>
41	</Qualifier>
42	<Item name="Instance Name"/>
43	<Item name="Port Lock"/>
44	<Text />
45	</Select>
46	</Reporting>

The Reporting section above contains two <Select> blocks, each preceded by a <Text> label. Both of these <Select> blocks iterate over the Instances of the same DataSource, but the only processes Instances where the ClassId Property is equal to "System" while the second only processes Instances where the ClassId Property is equal to "Settings". Using the same input file as before, the report looks like the following:

```
*****
*****
* Network Discovery *
*****
*****
```

```
Recommendation: Enable network discovery and set the ICMP Ping Timeout to
a value appropriate for your network.
```

```
*****
```

```
Instructions: Review the following information: network discovery should
be enabled (set to TRUE); ensure that the ICMP Ping Timeout value is
appropriate and modify it in the system's configuration file if necessary.
```

```
*** System Values ***
Instance Name: SMS_NETWORK_DISCOVERY
Discovery Enabled: TRUE-April
```

```
Instance Name: SMS_HOST_DISCOVERY  
Discovery Enabled: TRUE
```

```
*** Settings ***  
Instance Name: HTTP  
Port Lock: OFF
```

In this output report we see the two "System" Instances being printed out first (by the first <Select> element) and the one "Settings" Instance printed out last (by the second <Select> element). The use of qualifiers allowed us to organize different classes of Instances from a single DataSource so they appeared separately in the final report.

Intentionally Blank

Section 6

Example: Nested Structured File Source

Sometimes files describe structures within structures and the report author may wish to capture this in their reports. This can be done by defining nested Instance-classes within a report definition. A nested Instance-class is simply an <Instance> element within the body of another <Instance> element. In the report interpreter, the child Instances remain associated with a specific parent Instance in the same way that Properties are associated with a specific parent Instance. Because all of this structure is retained, it is possible to recreate the relationships between parent and child Instances within the report file. Consider the report definition document shown in Table 14:

Table 14. Extracting Nested Structure from Files

Line #	XML
1	<ReportDefinitions>
2	<ReportDefinition id="SWT-1" filename="Ping" title="Network Discovery" recommendation="Set the Ping Timeout" instruction="Network discovery should be TRUE with an appropriate ICMP Ping Timeout value.">
3	<DataSources>
4	<FileDataSource id="discovery-1">
5	<Filename directory="C:\Discovery Report">NestingTest.txt</Filename>
6	<InstanceStructures>
7	<InstanceDeclarations>
8	<Instance classid="System">
9	<Begin>BEGIN_COMPONENT</Begin>
10	<End>END_COMPONENT</End>
11	<Property name="Instance Name">
12	<Line>1</Line>
13	</Property>
14	<Property name="Discovery Enabled">
15	<PatternMatch>PROPERTY <Discovery Enabled><([^\>]*)></PatternMatch>
16	</Property>
17	<Instance classid="Server">
18	<Begin>BEGIN_PROPERTY</Begin>
19	<End>END_PROPERTY</End>
20	<Property name="Property Name">
21	<Line>1</Line>
22	</Property>
23	<Property name="Domain Include Local">

24	<PatternMatch>PROPERTY <Domain Include Local><([^\>]*)></PatternMatch>
25	</Property>
26	</Instance>
27	<Property name="ICMP Ping Timeout">
28	<PatternMatch>PROPERTY <ICMP Ping Timeout><([^\>]*)></PatternMatch>
29	</Property>
30	</Instance>
31	</InstanceDeclarations>
32	</InstanceStructures>
33	</FileDataSource>
34	</DataSources>
35	<Reporting>
36	<Select datasourceid="discovery-1" instances="all">
37	<Item name="Instance Name"/>
38	<Select instances="all">
39	<Item name="Property Name"/>
40	<Item name="Domain Include Local"/>
41	<Text />
42	</Select>
43	<Item name="Discovery Enabled"/>
44	<Item name="ICMP Ping Timeout"/>
45	<Text />
46	</Select>
47	</Reporting>
48	</ReportDefinition>
49	</ReportDefinitions>

The report definition provided above is the same as that provided in Table 9 with two exceptions: the original Instance-class now contains another Instance-class within its body and the Reporting section now contains an inner <Select> block that has no "datasourceid" attribute. As these are the only significant differences, this discussion focuses on these two points rather than going through the whole document line-by-line.

The inner <Instance> element starts on line 17 of the file. The format of this <Instance> element is just like a normal <Instance> element. The fact that it is nested within another <Instance> element simply means that the <Begin> pattern only matches if the pattern is found within the portion of the source file corresponding to the parent Instance-class. In this case, the "BEGIN_PROPERTY" string indicates the beginning of an Instance only if it appears between "BEGIN_COMPONENT" and "END_COMPONENT" lines. Otherwise, the match is discarded just as if we found a line that matched a Property's regular expression, but the match occurred outside the body of an Instance.

The other difference here is that the inner <Select> has no "datasourceid" attribute. Recall that the "datasourceid" attribute identifies a DataSource and the <Select> iterates over all the Instances of that DataSource. The absence of a "datasourceid" attribute in the inner <Select> indicates that the inner <Select> should iterate over all the child Instances in the current Instance of the outer <Select>. For example, consider the following source document:

```

BEGIN_FILE_DEFINITION
  <1>
  <MEL>
  BEGIN_PROPERTY
    <Deltas>
    PROPERTY <Domain Include Local><0><><0>
  END_PROPERTY
END_FILE_DEFINITION
BEGIN_COMPONENT
  <SMS_NETWORK_DISCOVERY>
  <6>
  <SMS-2003-SERVER>
  PROPERTY <Domain Include Local><6><><0>
  BEGIN_PROPERTY
    <Source>
    PROPERTY <Domain Include Local><6><><0>
  END_PROPERTY
  PROPERTY <Discovery Enabled><TRUE-January><><0>
  PROPERTY <NetToMediaTable Retrieval><ALL_DEVICES><><0>
  PROPERTY <Maximum Number Outstanding ICMP Pings><50><><0>
  PROPERTY <ICMP Ping Timeout><1000><><0>
  BEGIN_PROPERTY
    <Destination>
    PROPERTY <Domain Include Local><7><><0>
  END_PROPERTY
END_COMPONENT
BEGIN_COMPONENT
  <SMS_HOST_DISCOVERY>
  <6>
  <SMS-2003-SERVER>
  PROPERTY <Discovery Enabled><TRUE><><0>
  PROPERTY <NetToMediaTable Retrieval><ALL_DEVICES><><0>
  PROPERTY <Maximum Number Outstanding ICMP Pings><50><><0>
  PROPERTY <ICMP Ping Timeout><1000><><0>
  BEGIN_PROPERTY
    <Source>
    PROPERTY <Domain Include Local><5><><0>
  END_PROPERTY
  BEGIN_PROPERTY
    <Destination>

```

```
PROPERTY <Domain Include Local><2><><0>
END_PROPERTY
END_COMPONENT
EOF
```

The above file includes two Instances that match the outer Instance-class and each of these Instances contains two child Instances matching the inner Instance-class. Note that the <Begin> pattern for the inner Instance matches Line 4 of the source file, but because this is not between a BEGIN_COMPONENT and END_COMPONENT, and thus not within an Instance that matches the outer Instance-class, this does not result in a new Instance.

Note also that the fourth line of the first Instance (four lines after the first BEGIN_COMPONENT) matches the pattern for the "Domain Include Local" Property described in the inner Instance-class. However, because this line does not appear between a BEGIN_PROPERTY and END_PROPERTY it is not inside an Instance that matches the inner Instance-class and it is ignored.

An example report for the above source file appears below:

```
*****
*****
* Network Discovery *
*****
*****
```

Recommendation: Set the Ping Timeout

```
*****
```

Instructions: Network discovery should be TRUE with an appropriate ICMP Ping Timeout value.

```
Instance Name:      <SMS_NETWORK_DISCOVERY>
Property Name:     <Source>
Domain Include Local: 6
```

```
Property Name:     <Destination>
Domain Include Local: 7
```

```
Discovery Enabled: TRUE-January
ICMP Ping Timeout: 1000
```

```
Instance Name:      <SMS_HOST_DISCOVERY>
Property Name:     <Source>
Domain Include Local: 5
```

```
Property Name:          <Destination>
Domain Include Local: 2

Discovery Enabled: TRUE
ICMP Ping Timeout: 1000
```

As we can see, we report the Properties from the two parent Instances (Instance Name, Discovery Enabled, and ICMP Ping Timeout). We also report the Properties of the child Instances of both of these parent Instances (Property Name and Domain Include Local). Note that the order in which Properties appear in the report is controlled by the `<Reporting>` section of the report definition file rather than the order in which those properties are read from the source file. Note also that when we iterate over the Instances in the DataSource "discovery-1", we only iterate over the top-level Instances and do not include Instances that are children of other Instances. These children are only accessed by iterating over the children of other Instances.

6.1 Variation: Using `<InstanceReference>` Elements

Once we start looking at nested Instances we can run into the case where a single Instance-class might exist in multiple contexts. For example, the same Instance-class might exist at the top-level of a file and might also exist as the child of one or more other Instances. Moreover, sometimes an Instance might even be capable of containing other Instances of its own class. For example, we could use XML tags as the bounds of an Instance definition and those XML tags could be self-nesting. OCRL handles these cases through the use of the `<InstanceDefinitions>` block and `<InstanceReference>` elements.

The `<InstanceDefinitions>` block holds `<Instance>` elements that only define a particular Instance-class. Unlike `<Instance>` elements in the `<InstanceDeclarations>` block, `<Instance>` elements in the `<InstanceDefinitions>` block do not provide a context in which to look for the associated Instance in the source file. However, `<Instance>` elements in the `<InstanceDefinitions>` block can be referenced within either the `<InstanceDefinitions>` and `<InstanceDeclarations>` blocks by `<InstanceReference>` elements.

The `<InstanceReference>` element allows a report definition author to reference an `<Instance>` element that appears in the `<InstanceDefinitions>` block. Placing an `<InstanceReference>` in the Reporting block is equivalent to adding the named `<Instance>` element there. The advantage is that only one `<Instance>` element needs to exist for a given class and from then on, `<InstanceReference>` elements can be used to place that `<Instance>` in any context – even within itself. Consider the Table 15:

Table 15. Using the <InstanceReference> Element

Line #	XML
3	<FileDataSource id="discovery-1">
4	<Filename directory="C:\Discovery Report">NestingTest.txt</Filename>
5	<InstanceStructures>
6	<InstanceDefinitions>
7	<Instance classid="Server">
8	<Begin>BEGIN_PROPERTY</Begin>
9	<End>END_PROPERTY</End>
10	<Property name="Property Name">
11	<Line>1</Line>
12	</Property>
13	<Property name="Domain Include Local">
14	<PatternMatch>PROPERTY <Domain Include Local><([^\>]*)></PatternMatch>
15	</Property>
16	</Instance>
17	</InstanceDefinitions>
18	<InstanceDeclarations>
19	<Instance classid="System">
20	<Begin>BEGIN_COMPONENT</Begin>
21	<End>END_COMPONENT</End>
22	<Property name="Instance Name">
23	<Line>1</Line>
24	</Property>
25	<Property name="Discovery Enabled">
26	<PatternMatch>PROPERTY <Discovery Enabled><([^\>]*)></PatternMatch>
27	</Property>
28	<InstanceReference targetClassId="Server"/>
29	<Property name="ICMP Ping Timeout">
30	<PatternMatch>PROPERTY <ICMP Ping Timeout><([^\>]*)></PatternMatch>
31	</Property>
32	</Instance>
33	</InstanceDeclarations>
34	</InstanceStructures>
35	</FileDataSource>

In this example we provide two <Instance> definitions: The <Instance> with the "classid" of "Server" appears in the <InstanceDefinitions> block while the <Instance> with the "classid" of "System" appears in the <InstanceDeclarations> block. By placing the

"Server" <Instance> element in the <InstanceDefinitions> block, we are indicating that we only want to define this Instance-class but not set a context for it.

The "System" <Instance> element appears in the <InstanceDeclarations> block as before indicating that it both defines an Instance-class and declares that this Instance could appear at the top-level of the source file. In addition, at line 28 we include an <InstanceReference> element whose "targetClassId" attribute is "Server", which is the name of the <Instance> definition in the <InstanceDefinitions> block. This indicates that the parser should look for the "Server" Instance in this particular context, namely within a "System" Instance.

Given the original output file, this report definition produces exactly the same report as the original Section 6 example. Just as in the original report definition, this variation looks for just one top level Instance ("System") and within any "System" Instances it finds it looks for Instances of class "Server". In this particular case the use of <InstanceReference> did not save us any work, but one might imagine a case, where there were multiple top-level <Instance> elements all of which could have a "Server" Instance as a child. In this case, having a single definition of "Server" and having all the top-level <Instance> definitions reference it is much easier for an author to write.

Intentionally Blank

Section 7

Example: OVAL Source

OCRL is also capable of using an OVAL Object as a DataSource. OVAL, which stands for Open Vulnerability and Assessment Language, is an XML-based language for locating and evaluating system settings on a computer. OCRL takes advantage of OVAL's ability to locate information and uses that to create Instances within a DataSource.

To understand how an OVAL Object becomes an OCRL DataSource it is helpful to understand some of the OVAL System Characteristics (SC) language. This language is used to hold system information that is discovered by an OVAL Object. Each OVAL Object is associated with one or more "Items" within the SC file. Each Item corresponds to the data of a specific system location (one registry key, one Active Directory property, one file's metadata, etc.). Since a single Object can end up identifying multiple locations, some Objects may have more than one Item. For example, an OVAL Object might use regular expressions to identify a set of registry keys. Each Item contains a set of XML child elements corresponding to the information about that location that is used by an OVAL State to evaluate a system setting.

OCRL uses this structure to create a DataSource from an OVAL Object. The Report Definition file names an OVAL Object. Every OVAL SC Item that this Object would produce is mapped to an Instance within the DataSource. Every child element within an Item is mapped to a Property of that Item's Instance, where the Property name is the name of the XML element and the Property value is the body of this element. In addition, all Items have a "status" attribute that indicates the success of locating the relevant information for an Object.¹ The "status" attribute also becomes a Property of each Instance with the name of "Status" and a value equal to the value of the "status" attribute.

Consider the report definition in Table 16:

¹ Technically, "status" is an optional attribute so not all Items in an OVAL SC file have it. However, it has a default value so even if the attribute itself is absent, there is still a value for it. As such, we can always fill in a value for the Status Property.

Table 16. OVAL Data Source

Line #	XML
1	<ReportDefinitions>
2	<ReportDefinition id="OvalExample" filename="OvalOutputReport.txt" instruction="Verify that the users listed as registry keys at a given location match those in the active directory objects" recommendation="All users with registry keys identified in the OVAL Object with the identifier 'oval:org.mitre.1:obj:29' must also appear in the Active Directory object identified in the OVAL Object with the identifier 'oval:org.mitre.1:obj:33'" title="Sample OVAL report">
3	<DataSources>
4	<OvalDataSource id="RegistryOval">
5	<OvalObjectId filename="source_oval.xml">oval:org.mitre.1:obj:29</OvalObjectId>
6	</OvalDataSource>
7	<OvalDataSource id="ADOval">
8	<OvalObjectId filename="source_oval.xml">oval:org.mitre.1:obj:33</OvalObjectId>
9	<SystemCharacteristicsFilename directory="sc_files">source_oval_sc.xml</SystemCharacteristicsFilename>
10	</OvalDataSource>
11	</DataSources>
12	<Reporting>
13	<Text>The following users have registry keys:</Text>
14	<Select datasourceid="RegistryOval">
15	<Item name="name"/>
16	</Select>
17	<Text />
18	<Text>The following users are listed in Active Directory:</Text>
19	<Select datasourceid="ADOval">
20	<Item name="value"/>
21	</Select>
22	</Reporting>
23	</ReportDefinition>
24	</ReportDefinitions>

A line-by-line analysis follows:

- Lines 1-3 – These are the familiar opening lines to a report definition.
- Line 4 – Start an OVAL DataSource definition.
- Line 5 – The <OvalObjectId> element identifies the OVAL Object that should be used for this DataSource, namely oval:org.mitre.1:obj:29. (All OVAL Objects have an "id" attribute, and it is this attribute that is used to identify the Object.) An optional "filename" attribute identifies the file in which this OVAL Object is defined. If the

filename is omitted, the OCRL interpreter looks in some defined directory for the file defining the OVAL Object.

- Line 6 – Closes this OVAL DataSource.
- Line 7 – Opens a second OVAL DataSource definition.
- Line 8 – Identifies an OVAL Object for this DataSource.
- Line 9 – The <SystemCharacteristicsFilename> element identifies an OVAL SC file that has already been created for this OVAL Object, possibly through a prior execution of the OVAL Definition Interpreter tool. Normally, the OCRL interpreter attempts to parse the OVAL Object and look up the indicated locations itself. The presence of this element indicates that this is not necessary, and it can simply read all the requisite information about the Object from the given file. An optional "directory" attribute can provide the path for the named file. Without this attribute, the OCRL interpreter looks in the current working directory.
- Lines 10-11 – Close out this OVAL DataSource definition and the DataSources block.
- Lines 12-22 – Report both of the DataSources in turn. The contents of this section should be familiar from prior examples.
- Lines 23-24 – Close out the document.

In this example we define two DataSources and display all their Instances in turn. Each DataSource names an OVAL Object to guide its data collection. The contents of each DataSource depend on the nature of the OVAL Objects themselves as well as the contents of the locations that those Objects identify. For example, the Object associated with the "RegistryOval" DataSource might result in the Items shown in Figure 2:

```

<registry_item id="57">
  <hive>HKEY_LOCAL_MACHINE</hive>
  <key>SOFTWARE\Microsoft\Windows\CurrentVersion\Users\{F9B3}</key>
  <name>cmschmidt</name>
  <type>reg_sz</type>
  <value>9.00.3042.00</value>
</registry_item>
<registry_item id="58">
  <hive>HKEY_LOCAL_MACHINE</hive>
  <key>SOFTWARE\Microsoft\Windows\CurrentVersion\Users\{F250}</key>
  <name>ljl</name>
  <type>reg_sz</type>
  <value>1.00.0001</value>
</registry_item>
<registry_item id="59">
  <hive>HKEY_LOCAL_MACHINE</hive>
  <key>SOFTWARE\Microsoft\Windows\CurrentVersion\Users\{E9F4}</key>
  <name>DisplayVersion</name>
  <type>Inordman</type>
  <value>9.00.3042.00</value>
</registry_item>

```

Figure 2. OVAL Registry Object Items

This OVAL SC output results in three Instances for this DataSource, one for each Item. Each Instance has the following Properties: hive, key, name, type, and value. The values of these Properties are the bodies of the corresponding XML elements in each Item/Instance. All Instances also have a "Status" Property as well. Since none of the Items here actually have the "status" attribute, the value of all the Status Properties is the default value of the "status" attribute: "exists".

The second DataSource might result in the Items shown in Figure 3:

```

<activedirectory_item id="128">
  <naming_context>configuration</naming_context>
  <relative_dn>cn=ljl, Users, cn=MUA, cn=Configuration, dc=mitre, dc=org
</relative_dn>
  <attribute>username</attribute>
  <adstype>ADSTYPE_CASE_IGNORE_STRING</adstype>
  <object_class>MUA_users</object_class>
  <value>ljl</value>
</activedirectory_item>
<activedirectory_item id="129">
  <naming_context>configuration</naming_context>
  <relative_dn>cn=lnordman, Users, cn=MUA, cn=Configuration, dc=mitre,
  dc=org </relative_dn>
  <attribute>username</attribute>
  <adstype>ADSTYPE_CASE_IGNORE_STRING</adstype>
  <object_class>MUA_users</object_class>
  <value>lnordman</value>
</activedirectory_item>

```

Figure 3. OVAL Active Directory Object Items

This OVAL SC file produces two Instances for this DataSource because only two Items appear. Each Instance has the following Properties taken from the child XML element names: naming_context, relative_dn, attribute, adstype, object_class, and value. In addition, the Status property has the default value of "exists" for both Instances.

Note that the OCRL interpreter does not necessarily create an SC file with the above information. It is up to the implementer of the interpreter as to whether the SC file is created or whether the information is used directly by the interpreter for parsing without creating any file.

An example of a report using the above report definition appears below:

```

*****
*****
* Sample OVAL Report *
*****
*****

Recommendation: All users with registry keys identified in the OVAL Object
with the identifier 'oval:org.mitre.1:obj:29' must also appear in the
Active Directory object identified in the OVAL Object with the identifier
'oval:org.mitre.1:obj:33'.
*****

```

Instructions: Verify that the users listed as registry keys at a given location match those in the active directory objects.

The following users have registry keys:

name: cmschmidt

name: ljl

name: lnordman

The following users are listed in Active Directory:

value: ljl

value: lnordman

As dictated in the Reporting block of the Report Definition, we print out the "name" Property for each Instance (Item) in the first DataSource and the "value" Property for each Instance (Item) in the second DataSource.

Section 8

Conclusion

This document has presented an overview of many of the capabilities offered in OCRL. While it has covered most of the main structures and features, this document does not claim to be a comprehensive description of the language. Instead, it is hoped that this document serves as a springboard for new users of OCRL allowing them to quickly get up to speed or to clarify concepts they find confusing. It is recommended that users consult "Open Checklist Reporting Language (OCRL): A Language for Automated Generation of Reports for Security Guidance Evaluation" next for a more detailed and comprehensive understanding of OCRL and its capabilities.

Please do not delete these paragraphs or the final end-of-section mark in your document.
They are important for correct functioning of the RoboTech technical document template.
RoboTech: Version 3.0