Document Number: DSP0231

Date: 2009-07-14

Version: 1.0.0

CIM Simplified Policy Language (CIM-SPL)

Document Type: Specification

Document Status: DMTF Standard

Document Language: E

CONTENTS

Figures

Tables

Foreword

The *Common Information Model Simplified Policy Language (CIM-SPL)* specification (DSP0231) was prepared by the DMTF Policy Working Group.

DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems management and interoperability.

Acknowledgments

The authors wish to acknowledge the following people.

Editor:

- Jorge Lobo – IBM

Participants from the DMTF Policy Working Group:

- Dakshi Agrawal – IBM
- Seraphin Calo – IBM
- Kang-Won Lee – IBM
- Jorge Lobo – IBM
- Andrea Westerinen (Cisco Systems, now at Microsoft)

Introduction

This document presents the *CIM Simplified Policy Language (CIM-SPL)*, a proposed standard submitted by the DMTF Policy Working Group. The objective of CIM-SPL is to provide a means for specifying *if-condition-then-action* style policy rules to manage computing resources using constructs defined by CIM.

The design of CIM-SPL is inspired by existing policy languages and models including policy definition language (PDL) from Bell Laboratories, the Ponder policy language from Imperial College, and autonomic computing policy language (ACPL) from IBM Research. One of the main design points of CIM-SPL is to provide a policy language compatible with the CIM Policy Model and the CIM Schema.

CIM Simplified Policy Language (CIM-SPL)

1 Scope

The objective of CIM-SPL is to provide a means for specifying *if-condition-then-action* style policy rules to manage computing resources using constructs defined by the underlying model of CIM Information Model. Using CIM-SPL, one can write policy rules whose condition may consist of CIM data that contains the properties of managed resources. The CIM data may be available through various types of CIM data repositories. The action part of a CIM-SPL policy can invoke any operations or function calls in general. In particular, the action part can contain operations on the CIM data repository to change the properties of a CIM instance. This document provides several examples drawn from storage provisioning and network management to illustrate the usage of CIM-SPL.

The basic unit of a CIM-SPL policy is a policy rule. A CIM-SPL policy rule consists of a condition, an action, and other supporting fields (for example, Import). Multiple policy rules can be grouped into a policy group. Policy groups can be nested (that is, a policy group can contain other policy groups). In general, the structure of a policy group reflects a hierarchy of managed resources. For the specification of the policy condition, CIM-SPL provides the following rich set of operators described in sections 9 and 11, all based on the intrinsic CIM types:

- signed and unsigned short, regular, and long integers
- float and long float
- string
- Boolean
- calendar

This document presents a detailed description of the basic CIM-SPL operators with examples. These operations can also be used to compute the arguments passed as parameters to the policy actions.

2 Normative References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

2.1 Approved References

DMTF DSP0004, *CIM Infrastructure Specification 2.3*,
http://www.dmtf.org/standards/published_documents/DSP0004_2.3.pdf

2.2 Other References

ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards,*
http://isotc.iso.org/livelink/livelink.exe?func=ll&objId=4230456&objAction=browse&sort=subtype

3 Terms and Definitions

For the purposes of this document, the following terms and definitions apply.

3.1

can

used for statements of possibility and capability, whether material, physical, or causal

3.2

cannot

used for statements of possibility and capability, whether material, physical, or causal

3.3

conditional

indicates requirements to be followed strictly in order to conform to the document when the specified conditions are met

3.4

mandatory

indicates requirements to be followed strictly in order to conform to the document and from which no deviation is permitted

3.5

may

indicates a course of action permissible within the limits of the document

3.6

need not

indicates a course of action permissible within the limits of the document

3.7

optional

indicates a course of action permissible within the limits of the document

3.8

shall

indicates requirements to be followed strictly in order to conform to the document and from which no deviation is permitted

3.9

shall not

indicates requirements to be followed in order to conform to the document and from which no deviation is permitted

3.10

should

indicates that among several possibilities, one is recommended as particularly suitable, without mentioning or excluding others, or that a certain course of action is preferred but not necessarily required

3.11

should not

indicates that a certain possibility or course of action is deprecated but not prohibited

4 Symbols and Abbreviated Terms

4.1
CIMOM
CIM object manager

4.2
HBA
host bus adapter

4.3
IP
Internet protocol

4.4
MOF
managed object format

4.5
SAN
storage area network

4.6
SNIA
Storage Networking Industry Association

4.7
ssh
secure shell

4.8
UTF
Unicode Transformation Format

5 CIM Policy Model

This section briefly summarizes the CIM Policy Model, on which CIM-SPL is based. The CIM Policy Model is an information model defined by the DMTF to describe policy management systems. At its core, it provides a model for policy systems where the administrator can specify if-condition-then-action style policies for various distributed capabilities (for example, network filters and access control).

The highest-level constructs of the CIM Policy Model are the CIM_Policy class, the CIM_PolicySet class, the CIM_PolicyRule class, the CIM_PolicyGroup class, the CIM_PolicyTimePeriodCondition class, and the associations among them. In addition, the CIM_PolicyRule class is associated with the CIM_PolicyCondition and CIM_PolicyAction classes, which specify policy conditions and actions. See Figure 1, which shows the top portion of the hierarchy.

The information model of CIM-SPL is derived from the CIM Policy Model, that is, a policy rule in CIM-SPL is a subclass of CIM_PolicyRule called CIM_SPL_PolicyRule and contains a string property called PolicyString. The PolicyString property stores a policy written in CIM-SPL. No separately defined and associated conditions or actions may need to exist for this PolicyRule. Conditions and actions are embedded in the text of the CIM-SPL policy in the PolicyString. CIM policies are either a policy rule or an aggregation of policy rules in a PolicyGroup. This aggregation can contain policy rules or other policy groups, and either a policy rule or a policy group can be applied to managed elements to govern their operations. In practice, grouping policy rules that are commonly applied to the same kind of managed resources makes sense. Thus, it is important to have a way to define policy groups to simplify authoring and managing of policies. Mechanisms to define CIM-SPL Policies based on the combination of separately defined conditions, actions and policy groups may be created but are not described in this document.

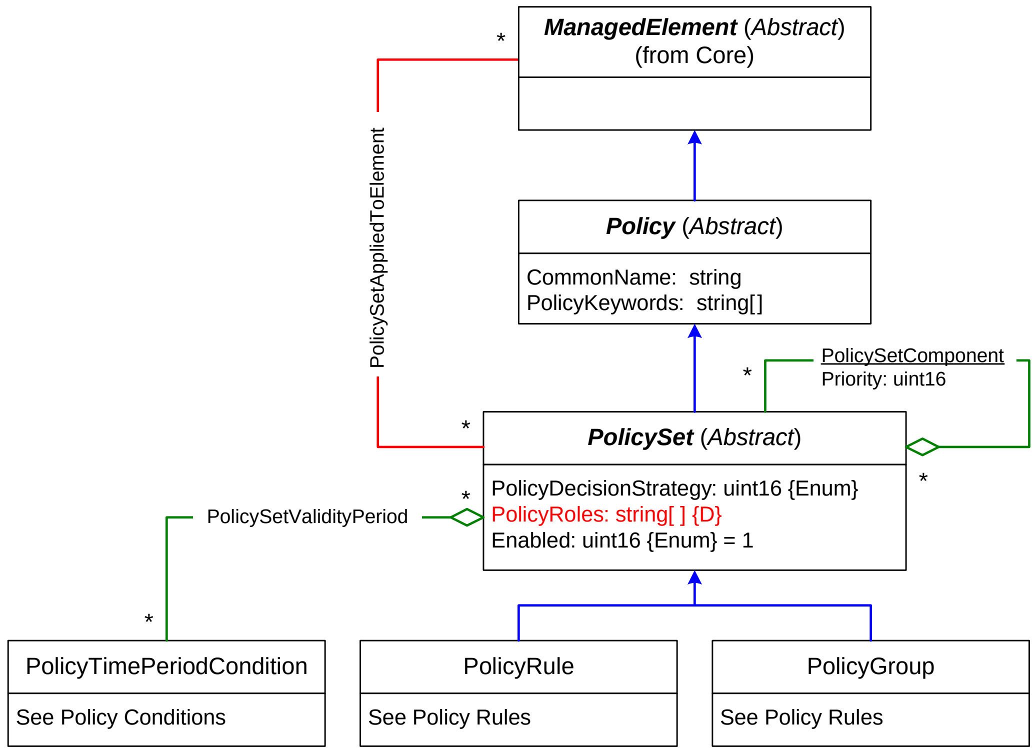

Figure 1 – CIM Policy Information Model

6 Usage Models

This section outlines some of the ways in which CIM-SPL can be used. The usage models described in this section are not intended to be exhaustive; rather they are presented here for illustrative purposes.

6.1 Best Practice Checker

Policy-based best practice checking is a promising area in validating network configurations. In this section, the examples are drawn from storage area network (SAN) management. One of the main challenges in SAN management is the complexity encountered during system setup and reconfiguration. Typically a SAN consists of a large number of components from multiple manufacturers, and many of the components may have interoperability constraints. For example, a storage device from a certain vendor may work with only certain types of SAN switches (with certain firmware levels). Such interoperability constraints are usually documented and published by device vendors. In addition, over time, SAN administrators have developed best practices for avoiding typical problems associated with misconfigured devices. Following is a short list of sample best practices from field practitioners:

- All zones should be configured so that the same host bus adapter (HBA) cannot talk to both tape and disk devices.
- Both Windows server and Linux server should not be members of the same zone.

- Every active and connected port should be a member of at least one active zone.

To enforce these best practices, the storage management software queries the configuration and status of all the devices in a SAN and stores the configuration information in a CIM database. The policy management system can check for violations of the best practices that have been encoded in CIM-SPL. To ensure correct operation, the system administrator may run a configuration checker to validate all best practices for a particular device after making changes. The configuration checker can also run on a schedule (for example, every day at midnight), when it pulls the current configuration information from the database and checks it against best practices.

Alternatively, the policy evaluation may be triggered manually by the system administrator after making a configuration change. For example, after replacing an old HBA, the system administrator may want to validate the best practices against the new HBA. In this case, the administrator can evaluate only the policies relevant to an HBA and only for the configuration of the new HBA, as opposed to evaluating all the policies for the entire SAN.

6.2 Routing in Networks

Typical security policies for networks are implemented during the configuration of network devices such as switches, routers, or firewalls. The following list provides a few examples of these policies:

- Allow telnet connections within the local network.
- Block any connection from locations outside the local network.
- Block any telnet connection to locations outside the local network.
- Allow ssh connections to locations outside the local network.

To capture these policies, most systems provide support for accepting configuration entries in the form of “if-then” rules. For example, given the prefix of the local network, the first rule can be written as follows: “If the input connection comes from an IP address with the local prefix and the destination port is the telnet port, then accept the connection.” Similarly, the second rule can be written as follows: “If the input connection comes from an IP address with no local prefix, then drop the connection.” The specification of these rules in CIM-SPL is straightforward. The implementation most likely depends on the device enforcing the policy. For example, routers that may directly support an interpreter for CIM-SPL will accept the CIM_Policies and reprogram themselves accordingly. Other systems, such as a computer running Linux, can translate the rules into *iptables* filter rules and perhaps dynamically load the rules into the operating system kernel. This file with rules is read by the system kernel, and the rules are applied at the appropriate time.

7 SPL Policy Rules

A policy in CIM-SPL is always a policy group but the most basic element in a CIM-SPL policy is a policy rule. A CIM-SPL policy rule is essentially a stream of characters that specifies a Condition/Action policy rule. To store, transmit, and represent policy strings in a byte-oriented medium or protocol, the characters need to be encoded in a byte format. CIM-SPL parsers shall support at least UTF-8 encoding of characters. A parser may support additional encodings, such as GB18030, the official character set of the People's Republic of China, to specify identifiers and strings.

The following example illustrates the CIM-SPL format. A detailed description of the syntax is provided in Section 13.

```
4.1 # This is an example of a CIM-SPL policy.
# 2005/07/15
Import SAMPLE CIM_V_2_8_CIM_Core28-Final::PhysicalElement;
Strategy Execute_All_Applicable;
```

```
Declaration{
   InstallDate="ManagedSystemElement.InstallDate";
   Macro { Name = Age;
        Type = Long;
        Arguments Born:DATETIME;
        Procedure = getYear(CurrentDate) – getYear(Born)
      }
}
Policy {
Condition {  4 > Age(InstallDate) AND
                          VendorEquipmentType == "switch"}
Decision {   Upgrade (SKU)    }
}:1
# End of Policy
```

As shown in the preceding example, a policy string comprising a single rule has four components:

1) ***Import statement****:* The Import statement in the example refers to a CIM class that is relevant to the policy string. In the remainder of the policy string, a policy rule is written as if an instance, called instance under evaluation, of this class is available for manipulation. The rule may be able to access other objects by traversing the references in associations where the instance under evaluation participates. An Import statement is required in each policy string. Section 8.1.1 elaborates on the manner in which the object instance may be obtained.

2) ***Strategy statement****:* The Strategy statement indicates how many policy rules can be executed. In the example, this statement can be ignored because the example contains a single rule.

3) ***Declaration section****:* A Declaration section defines named constants and macros that can be used in the policy section of the policy string. In this way, the actual policy specification can be clearer and easier to understand. The Declaration section is optional.

4) ***Policy section****:* The Policy section contains the main body of the policy string, with a condition statement, a decision statement, and priority. Both the condition and the decision can refer to the named constants and use the macros defined in the declaration section. The priority helps to determine what policy rule to execute in case multiple rules are triggered.

In addition, a policy string can have comment statements. Each comment statement starts on a new line with the # character as the first non-space character. Comment statements can occur anywhere in the policy string. Comment statements are for human users and for maintenance; they are ignored by the policy compiler.

Following are conventions and rules that are observed unless specified otherwise:

- Each policy string consists of multiple lines[1].
- Consecutive white space characters[2] in a line are treated as a single space character.
- Blank lines or lines with only white space characters are ignored.
- Reserved keywords in CIM-SPL are not case-sensitive.
- As the preceding example shows, each policy string may contain multiple sections.
- Each section is separated from the others by a label and opening and closing curly brackets.

[1] Lines in a policy string are delimited by line separators that include LF(u000A), CR(u000D), NEL(u0085), FF(u000C), LS(u2028), and PS(u2029).

[2] White space characters include characters in the space separator category (Zs) in the Unicode specification.

- The order of the data inside the sections is not important.
- A policy string may refer to identifiers that are either the named constants or macros defined in the declaration section, or are properties or methods of the instance under consideration or any property or method of an object that can be reached traversing associations. Identifiers are described in Section 9.70.
- Semicolons are used at the end of the Import statement and at the end of primitive statements within the Declaration section and the Policy declaration section (see section 7.1.3) where no grouping characters (parentheses and curly brackets) occur.

7.1 Policy String Components

In the following three subsections the different parts of a simple policy, that is, a policy group with a single rule, are described in detail. Groups are described in Section 8. Normative grammar is given in Section 13.

7.1.1 Import Statement

When a policy rule is evaluated, the evaluation shall be done for a *target set* of managed elements. A policy rule, being part of a policy set, is meant to be applied to a managed element. The target set shall define instances of PolicySetAppliesToElement.

Every policy group shall have an Import statement and it shall refer to a CIM MOF file and a class included in that file. For brevity, the class referred to by the Import statement of a policy will be called the *import class* of the policy. The object instances in the target shall all be instances of the import class of the policy and they may further filter to only the objects that satisfy the optional simple Boolean condition in the Import statement. The syntax for the simple Boolean condition is defined later in Section 10. Note that the method used for an evaluator of a policy o get access to the managed elements in the target set of the policy is outside the scope of the language definition and is an implementation issue.

Format:

```
Import <name> CIM_V<major>_<minor>_<release><final or preliminary><mof file
name w/o extension>::<class name>:<simple Boolean condition> ;
```

The Import statement specifies that an instance of the class specified in the statement is available during the evaluation of the policy rules. Policy rules may reference properties of this instance, including the properties in its super classes. Any other object that a policy rule may refer to in any part of the rule shall be accessible through reference associations related to this managed element. Operators are available to traverse associations in which this element participates to get access to other elements and their properties. The name is an identifier for the policy. It can be any sequence of letters or numbers always starting with a letter.

7.1.2 Declaration Section

The Declaration section contains declarations for named constants and macro procedures. For example, InstallDate can be defined as "PhysicalElement.ManagedSystemElement.InstallDate", and the InstallDate constant can be used when specifying the installDate in the Age macro. InstallDate refers to a property of the super class ManagedSystemElement of PhysicalElement. Details of how identifiers are interpreted are in Section 9.70. Macro procedures are used for common operations that may appear repeatedly in the policy sections. The Declaration section is optional.

Format:

```
Declaration {
    <List of constant definitions> (Optional)
    <List of macro definitions>     (Optional)
```

```
}
The names of constants and macros shall be different from the policy name.
Constant Definition
Format:
    <constant name> = <constant value>;
Macro Definition
Format:
Macro {
Name=<string that is the macro's name>; (Required)
Type=type; (Required)
Argument= name1:type1[,name2:type2]*; (Optional)
    Procedure=<expression> (Required)
}
```

Name is the identifier of a macro and Type is the return type of the macro call. Each argument is a Name:Type pair. Procedure defines the expression that is used as a result of a call to this macro. Here <expression> can be any valid CIM-SPL expression. See section 7.1.3.3 for CIM-SPL expressions and operators. The expression can include a macro call as long as the macro name has already been defined. See Section 9.70 for the definition of a macro call.

7.1.3 Policy Section

The Policy section contains the main body of a policy rule. It consists of the Policy Declaration, Condition, and Decision sections.

Format:

```
Policy {
  Declaration {
    <List of constant definition> (Optional)
    <List of macro definitions> (Optional)
   }
   Condition {      (Optional)
        <If Condition>
   }
   Decision {       (Required)
        <Then Decision>
   }
}: Priority
```

7.1.3.1 Declaration

The meaning of this section is the same as the global Declaration section except that the scope of the policy rule declarations is within the policy rule only. The policy declarations override global declarations if the names happen to clash.

7.1.3.2 Condition

The Condition section is an optional subsection of the Policy section. This is the "if" condition part of the policy rule. If the Condition section is omitted, the policy is considered always active (that is, an "unconditional" policy or a policy with a "true" condition).

Format:

```
Condition {
    <Boolean Expression> (An expression that results in a Boolean constant
    after evaluation)
}
```

Following is a summary of the operators and the functions that can be used to create Boolean expressions used in a Condition section. A more detailed description is provided in section 9.1

7.1.3.3 Predefined Operators and Functions

With few exceptions (for example, the minus operator, which can be either a unary or binary operator), each operator has a fixed number of typed arguments. CIM-SPL is a strongly-typed language (that is, the types of the arguments shall match the types supported by the operators). For example, numeric operators can take only numeric arguments; string operators can take only string arguments, and so on. Any named constants and macros, and other expressions, can be arguments to CIM-SPL operators.

Alpha, *beta*, and *gamma* in the examples shown in Table 1, Table 2 and

Table 5 represent numeric constants, and *time, date, date1, date2* shown in Table 6 represent a date time constant.

Table 1 – Numeric Operators

Operator	Example
+	(alpha + 2)
–	(alpha – 2), – alpha
*	(alpha * beta)
/	(alpha / beta)

Table 2 – Boolean Operators

Operator	Example
&&	(alpha < 10) && (beta > 3)
||	(alpha < 10) || (beta > 3)
^	(alpha < 10) ^ (beta > 3)
!	!(alpha)

Alpha, *beta*, and *gamma* in the examples shown in Table 3 are either all numeric values or all strings.

Table 3 – Relational Operators

Operator	Example
==	(alpha == beta)
!=	(alpha != beta)
>=	(alpha >= beta)
<=	(alpha <= beta)
>	(alpha > beta)
<	(alpha < beta)

Table 4 – String Functions

Function	Example
stringLength	stringLength("John Doe") ; returns 8
toUpper	toUpper("John Doe") ; returns “JOHN DOE”
toLower	toLower("John Doe") ; returns “john doe”
concatenate	concatenate("John ", "Doe") ; returns “John Doe”
substring	substring("John Doe", 1, 5) ; returns “ohn ”
matchesRegExp	matchesRegExp(IP,"\d{1,3}+\.\d{1,3}+\.\d{1.3}+")
leftSubstring	leftSubstring("Mississippi", 4, "LeftToRight") ; returns “Miss" leftSubstring("Mississippi", 4, "RightToLeft") ; returns "Mississ"
rightSubstring	rightSubstring("Mississippi", 4, "LeftToRight") ; returns "issippi" rightSubstring("Mississippi", 4, "RightToLeft") ; returns "ippi"
middleSubstring	middleSubstring("Mississippi", 4, 5, "LeftToRight") ; returns "issip" middleSubstring("Mississippi", 4, 5, "RightToLeft") ; returns "ippi"
replaceSubstring	replaceSubstring("Illinois", "nois", "i") ; returns ”Illini”
toUINT8	toUINT8("2")
toSINT8	toSINT8("2")
toSINT16	toSINT16("12")
toUINT16	toUINT16("12")
toSINT32	toSINT32("-12341234")
toUINT32	toUINT32("12341234")
toSINT64	toSINT64("-1234")
toUINT64	toUINT64("1234")
toREAL64	toREAL64("123.45")
toREAL32	toREAL32("12345.678")
toBoolean	toBoolean("true")
word	word(alpha, " ", 3)
startsWith	startsWith("Just a test", "Just") ; returns true
endsWith	endsWith("Just a test", "test") ; returns true
contains	contains("Just a test", "t a t") ; returns true
containsOnlyDigits	containsOnlyDigits("1234")
containsOnlyLetters	containsOnlyLetters("aBcD")
containsOnlyLettersOrDigits	containsOnlyLettersOrDigits("a1b2C3")

Table 5 – Numeric Functions

Function	Example
min	min(alpha, beta, gamma)
max	max(alpha, beta, gamma)
remainder	remainder(alpha, beta)
power	power(alpha, beta)
abs	abs(alpha)
toBoolean	toBoolean(0)
round	round(alpha)
exp	exp(alpha)
log	log(alpha)
sqrt	sqrt(alpha)
floor	floor(alpha)
ceiling	ceiling(alpha)

Table 6 – Time Functions

Function	Example
getMillisecond	getMillisecond(time)
getSecond	getSecond(time)
getMinute	getMinute(time)
getHour12	getHour12(time)
getHour24	getHour24(time)
getDayOfWeek	getDayOfWeek(date)
getDayOfWeekInMonth	getDayOfWeekInMonth(date)
getDayOfMonth	getDayOfMonth(date)
getDayOfYear	getDayOfYear(date)
getWeekOfMonth	getWeekOfMonth(date)
getWeekOfYear	getWeekOfYear(date)
getMonth	getMonth(date)
getYear	getYear(date)
isWithin	isWithin(date, date1, date2)
toMilliseconds	toMilliseconds(time)

7.1.3.4 Decision

The Decision section is a required subsection of the Policy section. It contains the then-action clause of the "if-condition-then-action" policy statement. The statement describes which CIM PolicyActions are called when the "if" condition is true. If some part of the action block encounters an error and thus the execution could not complete successfully, a CIM_ERROR_POLICY_EXECUTION should be thrown. In the implementation, a failure may be defined by a time out, that is, if an action does not complete within a predefined time, then it is considered a failure.

Format:

```
Decision {
    <action block>
}
```

An <action block> may take one of the following forms:

<policy action name>* ( ) *<cop> <constant>

A single PolicyAction evaluation without arguments.

<policy action name>* (*<expression>*[, *<expression>*]*) *<cop> <constant>

A single PolicyAction evaluation with at least one argument. The argument expression types shall match the argument types of the concrete PolicyAction being evaluated. If this is not the case, a CIM_ERROR_POLICY_EXECUTION may be thrown. ***<cop>*** is one of the comparison operators ==, != , <, <=, >, >=, and ***<constant>*** is a numeric constant. The ***<cop> <constant>*** pair is optional.

***<cascaded policy name>* (Collection)**

The cascaded policy name is an identifier that shall refer to a PolicySet element that will be evaluated as a result of the current policy execution. The collection shall be an expression that results in a collection of managed elements that are used during the evaluation of the cascaded policy (that is, it represents the target set). See section 7.1.1.

<action block1> -> <action block2>

This represents a sequence of action evaluations, where action block 1 is executed first, and then action block 2 is executed if action block 1 executes successfully. If action block 1 does not complete successfully, action block 2 should not be executed and the whole block returns failure. If the first block succeeds, the second block is evaluated and the whole block returns whatever the second block returns.

<action block1>* | | *<action block2>

This represents the concurrency "some" semantics, where at least one of the action blocks (action block 1 or action block 2) should be executed. In this case, both the blocks should be executed concurrently (without any particular order), and the whole block succeeds as soon as one of the two action blocks succeeds.

<action block1>* && *<action block2>

This represents the concurrency "all" semantics. Both action blocks should be executed, but there is no explicit sequence defined for the execution. In this case, both the blocks should be evaluated concurrently (without any particular order), and the whole block succeeds if both internal blocks return success.

<action block1>* | *<action block2>

This represents the conditional semantics. If action block 1 completes successfully, then the whole block succeeds, and action block 2 is not executed. If action block 1 could not complete successfully, action block 2 will be executed, and the whole block returns whatever the second block returns.

(*<action block>*)

The parentheses are used to change the association precedence of combination operators. In the action block, all decisions have equal precedence and are evaluated left to right by default. When enclosed in parentheses, an action block is evaluated as a single block (see the following example).

A PolicyAction name can take one of the following forms:

1) Set.Identifier, with the identifier referring to a managed element. The argument names shall be properties of the element, and the effect is to set all the properties passed as arguments to the values returned by the expressions.

2) Identifier.MethodName, with the identifier referring to a managed element. The method name shall be a method of the managed element to which the identifier refers. The arguments shall match the signature of the method. The <*cop*> <*constant*> pair can only appear in this case.

Any evaluation of a concrete PolicyAction (in the form <policy *action name*> (<*expression*> …)) or the setting of properties in a PolicySet instance return either success or failure. For an <*action block*>, if the block is just an action or a cascaded policy instance, the block returns whatever the action or the policy set returns if no <*cop*> <*constant*> pair appears after the action. If the pair appears, the execution will be considered a failure if the value returned by the action does not match the condition.

To see the effect of parentheses, consider the following decision example: $a \rightarrow b \mid c$. The association left to right makes this expression equivalent to $((a \rightarrow b) \mid c)$. In this case, if the evaluation of *a* succeeds, *b* is evaluated; otherwise *c* is evaluated. If *b* is evaluated and succeeds, nothing else is evaluated. On the other hand if *b* fails, *c* is evaluated. So *c* is evaluated whenever *a* or *b* fails. In the expression $a \rightarrow (b \mid c)$, *c* is evaluated only if *a* succeeds and *b* fails.

The same syntax for expressions pertains to specifying arguments for policy action invocations as applies in condition clauses (see section 7.1.3.3).

7.1.4 Strategy and Priorities

The Strategy statement and policy priorities are explained in Section 8, in which policy groups are introduced.

8 SPL Policy Groups

Policies in CIM are not only individual policy rules — they can be policy groups. A policy group in the CIM Policy Model aggregates policy rules and other policy groups using PolicySetComponent aggregations (see Figure 1).

This section provides the syntax for writing policy groups and describes how the *target set* for the evaluation of the rules inside a policy group is determined. Similar to CIM-SPL policy rules, a CIM-SPL policy group is represented by a policy string. The policy string for a policy group has the format presented in the following section.

8.1 Policy Group Components

A policy group has the following format:

```
Import CIM_V<major>_<minor>_<release><final or preliminary><mof file name w/o
extension>::<class name>:<simple Boolean condition> ;
Strategy [Execute_All_Applicable | Execute_First_Applicable] ;  (Required)
Declaration {
    <List of constant definition> (Optional)
    <List of macro definitions> (Optional)
}
Policy { … } : Priority; (Optional)
Policy { … } : Priority; (Optional)
```

```
Policy { … } : Priority; (Priority is required)
…
PolicyGroup:[Association Name(Property1,Property2)] { … }: Priority;
(Optional)
PolicyGroup:[Association Name(Property1,Property2)] { … }: Priority;
(Optional)
PolicyGroup:[Association Name(Property1,Property2)] { … }: Priority;
(Optional)
…
```

At least one policy rule or one policy group shall be part of a policy group. The priorities are positive integers. The order of policies and policy groups is immaterial; they can be intermixed, but all priorities shall be different.

8.1.1 Suggested Mechanisms of Invocation: Import Statements and Indications

The Import statement in a policy group plays the same role as the Import statement in a single policy rule. This Import statement indicates the class of the object instance under consideration to each policy rule in the group. How the Import statement affects the evaluation of a policy group that is contained in another policy group is described later in this section.

A policy (rule or group) is evaluated on a *target set* of managed elements. All these managed elements shall be instances of the import class. This set defines instances of the PolicySetAppliesToElement association in the CIM Policy Model.

The method used for an evaluator of a policy rule to get access to the managed elements in the target set of the policy rule is outside the scope of the language definition and is an implementation issue. This allows different policy invocation methods to be applied in different systems environments.

In one situation, a policy enforcement point may directly request (probably from a CIM Object Manager (CIMOM) the evaluation of all policies relevant to the resource with which it is associated. The time of invocation is decided by the policy enforcement point. This evaluation mechanism is often designated as *solicited policy evaluation*. For a solicited evaluation of a policy, the target set may consist of all instances of the import class that match the simple Boolean condition of the Import statement of the policy. If the simple Boolean condition is not specified in the Import statement, the target set consists of all instances of the import class. CIM-SPL does not define how the instances of the import class are gathered; that is, it does not describe the scope of the operation that gathers instances. For example, the scope of gathering data could be limited to a particular CIMOM or to all locations in the world-wide IT infrastructure of an enterprise. At the time of activating or installing a policy, the scope of data gathering should be explicitly or implicitly specified for the policy. Thus, this issue falls outside the CIM-SPL language definition.

In another situation, policy evaluation may be triggered by an event in the system. This mechanism is often designated as *unsolicited policy evaluation*. For an unsolicited evaluation of a policy, the *target set* for evaluation can be provided implicitly by the instances of CIM_InstIndication subclasses that are *consistent* with the Import statement of the policy. The SourceInstance parameter of a CIM_InstIndication instance points to a managed element that was the source of the CIM_InstIndication. A CIM_InstIndication instance is consistent with the Import statement of a policy if the class of the managed element to which the SourceInstance parameter of the CIM_InstIndication points is a subclass of the class in the Import statement. CIM-SPL does not define which compatible instances of CIM_InstIndication initiate an unsolicited evaluation of a policy. For example, when a policy is installed or is activated, a policy server may require that an instance of CIM_IndicationFilter be specified. Such a requirement would be sufficient for the policy server to generate a CIM_IndicationSubscription to CIM_InstIndications that trigger evaluations of the policy. Subsequently, when the policy server receives a CIM_InstIndication instance, the policy server evaluates the corresponding policy on the managed element to which the SourceInstance parameter of the received CIM_InstIndication instance points.

8.1.2 Strategy Statement

A policy is applicable if its condition part evaluates to TRUE. A policy group is applicable if at least one of the policies belonging to the group is applicable. Any CIM-SPL implementation shall support at least two evaluation strategies: *Execute_All_Applicable* and *Execute_First_Applicable.* The strategy shall be specified in the strategy statement of a policy group. The *Execute_All_Applicable* strategy goes one by one over all the policies and policy subgroups, evaluating all applicable policies. The *Execute_First_Applicable* strategy proceeds in the order indicated by the priorities and examines policies and policy subgroups until one that is applicable is evaluated. Implementations may handle other evaluation strategies. If a policy mentions a strategy not supported by the CIM-SPL implementation the evaluation shall return a CIM_ERR_NOT_SUPPORTED error.

8.1.3 Policy Evaluation

Assuming that a set of managed elements have been collected for evaluation by a policy group, the evaluation shall proceed as follows. Consider a policy group **P** whose constituent policy rules and policy groups are given by P_1, P_2, P_3, P_4, … and so on. For each managed element **M** for which the policy group **P** needs to be evaluated, the evaluation of **P** proceeds in two steps (regardless of whether the triggering of the evaluation was solicited or unsolicited): an *Applicability* step and an *Action Evaluation* step. The *Applicability* step returns a set of action blocks (see Section 7.1.3.4). The *Action Evaluation* step takes the set of action blocks output by the *Applicability* step and evaluates the actions. The *Applicability* step proceeds as follows:

First, if the evaluation strategy is *Execute_All_Applicable*, each P_i is processed as follows:

If P_i is a policy rule, the rule is checked if it is applicable on **M.** If during the evaluation of any condition to determine the applicability of a policy rule, the evaluation fails, the policy evaluations fails and returns CIM_ERR_POLICY_EVALUTION.

If P_i is a policy group, a new target set **S** is created as follows:

- If the policy group P_i has the optional association specification *Association Name(P1,P2)* specified with P_i (indicated following the keyword *PolicyGroup*), the target set **S** has managed elements that are associated with **M** through instances of the named association in such a way that **M** is referenced in the instance by property *P1* while the elements in the target set are referenced in the instances by property *P2*. If there is no association with specified name associated with the object the policy evaluation fails and returns CIM_ERR_POLICY_EVALUTION.
- If the optional association is not specified with P_i, the association is assumed to be the association CIM_Component, *Property1* the property GroupComponent, and *Property2* the property PartComponent. As a consequence, the target set **S** consists of all Components of the managed element **M**. If there is no association CIM_Component the policy evaluation fails and returns CIM_ERR_POLICY_EVALUTION.
- Then recursively the *Applicability* step is applied to the policy group P_i for each managed element in **S**. Each evaluation returns a set of action blocks
- If the evaluation strategy is *Execute_First_Applicable*, the P_is shall be processed in the order specified by the priority (lower numbers first), but the processing shall stop at the first time either a policy rule is applicable or the *Applicability* step applied to a policy group returns a non-empty set of action blocks.
- Next, all action blocks of policy rules that have been processed and have been found applicable and all action block sets returned by the processing of policy groups are collected together in a single set of action blocks and the set is returned as the result of the *Applicability* step. If no policy rule has been found applicable and no group has returned a non-empty set of action blocks, the *Applicability* step will return an empty set of action blocks.

- Next the *Action Evaluation* step is applied. In this step each action block in the set shall be processed according to the action execution schema described in Section 7.1.3.4.

The evaluation of a policy group for a managed element works recursively — policy rules in the policy group are applied to the managed element, and, by default, policy subgroups in a policy group are applied to the *components* of the managed element. The default behavior can be changed and the policy subgroups in a policy group can be applied to other managed elements that are associated with the managed element through an association other than CIM_Component. Section 8.2 shows how this evaluation provides a powerful mechanism for specifying and applying policies in a hierarchical manner.

8.2 Policy Group Example

Figure 2 shows a diagram from the SNIA specification SMI-S 1.1.0. It shows a SAN Fabric that has Host, Switch, and Array instances (distinguished by the value of the Dedicated property) as its components.

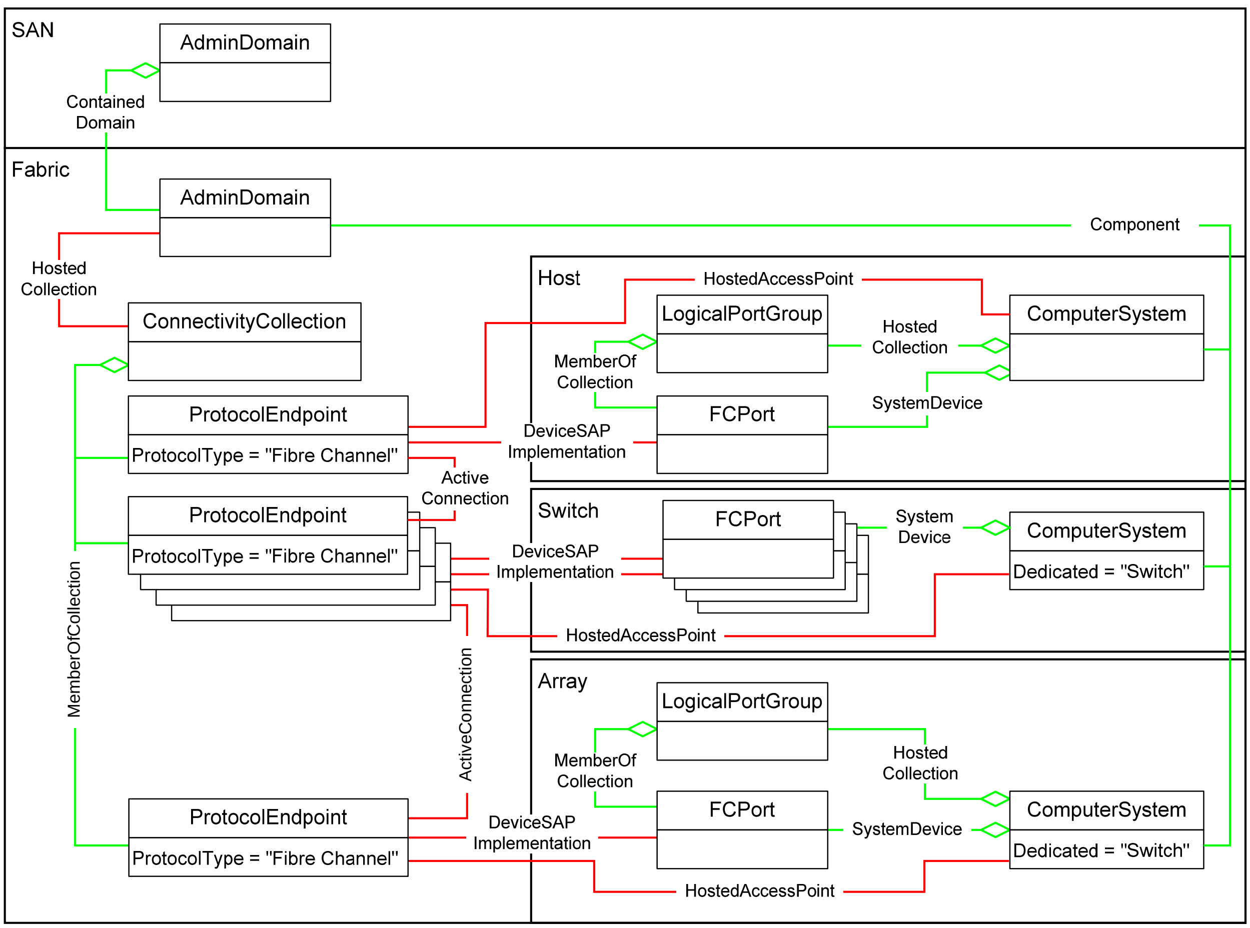

Figure 2 – Fabric Instance Diagram

A policy group for a SAN fabric can comprise two policy subgroups, one for the Switch and the other for the Array. Inside the policy group for the Switch, a policy group can exist for fiber channel ports (FCPort). Schematically, the policy group for the SAN fabric is shown in Figure 3.

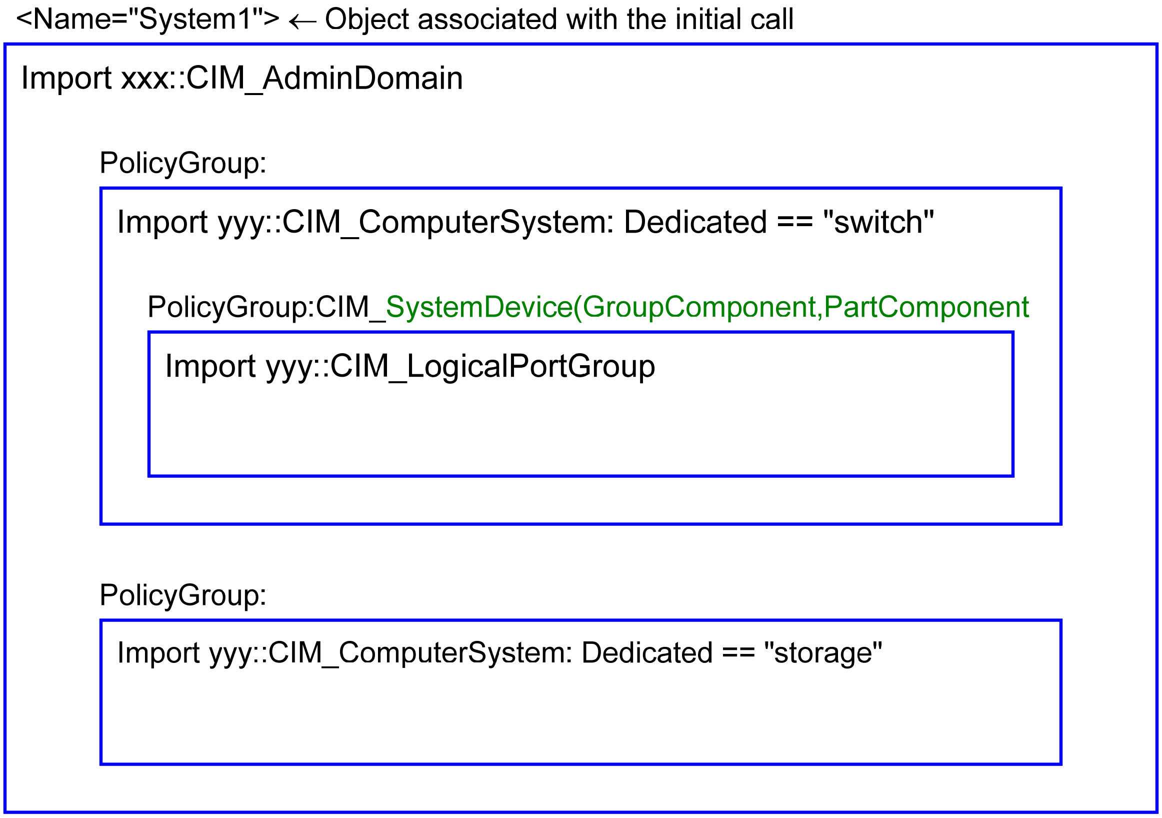

Figure 3 – PolicyGroup Schema

When this policy group is evaluated for a SAN fabric, the first policy subgroup is evaluated for all switches in the fabric (components of type ComputerSystem with the Dedicated property set to “switch”). Similarly, the second policy subgroup is evaluated for all storage arrays in the fabric (components of type ComputerSystem with Property Dedicated set to “storage”). For each switch in the fabric, the innermost policy group is evaluated for all fiber channel ports (FCPort instances that are reached by traversing the association GroupComponent from the switch instance).

When specifying a policy group, certain policies are applicable only to a particular component of the managed element. Or, certain policies within a policy group may be applicable only to managed elements that are associated in a specific manner with the managed element. Such policies can be conveniently collected within a subgroup to ease the specification of a policy group.

9 Expressions

The expression language of CIM-SPL shall support all CIM intrinsic data types, arrays of these data types, references to instances of CIM classes, and arrays to instances of CIM classes. This section describes required operators supported by CIM-SPL.

Unless specified otherwise, a CIM-SPL function shall have the following syntax: **operator***(operand1, operand2, operand3, …)*. The syntax phrase shows the required operator in **bold**. Items that are required arguments are shown in italics, like <*expression*>. Italicized values inside square brackets are optional. These operators shall not be case-sensitive. Spacing between operands, operators, and parentheses is optional. If one of the operands were to evaluate to NULL, then the operator shall evaluate to NULL.

9.1 Abs

Shall return the absolute value of the required numeric argument

abs(<*expression*>**)**

Examples:

abs(nbr1)

abs(4)

9.2 Logical And

Shall return a Boolean value corresponding to the logical AND operation on the required Boolean arguments

(<*expression*> **&&** <*expression*>**)**

Examples:

(a && b)

((stringLength(alpha) >9) && (5<c))

9.3 StartsWith

Shall return TRUE if the first required string argument begins with the second required string argument

startsWith(<*expression1*>, <*expression2*>**)**

where <*expression1*> is the given string, and <*expression2*> is the substring

Example:

startsWith("just a test", "just")

9.4 Ceiling

Shall return the smallest integer that is greater than or equal to the required numeric argument

ceiling(<*expression*>**)**

where <*expression*> is the numeric value

Example:

ceiling(nbr1)

9.5 Concatenate

Shall return the concatenation of two or more required string arguments

concatenate(<*expression1*>, <*expression2*>, ..., <*expressionN*>**)**

Examples:

concatenate(alpha, beta)

concatenate("Entered ", aValue, " in field")

9.6 Contains

Shall return TRUE if the first required string argument contains the second required string argument

contains(<*expression1*>, <*expression2*>**)**

where <*expression1*> is the given string, and <*expression2*> is the substring

Example:

contains("just a test", "t a t")

9.7 ContainsOnlyLettersOrDigits

Shall return TRUE if the required string argument is all letters or digits

containsOnlyLettersOrDigits(<*expression*>**)**

where <*expression*> is the string

Example:

containsOnlyLettersOrDigits("one4theroad")

9.8 ContainsOnlyDigits

Shall return TRUE if the required string argument is all digits

containsOnlyDigits(<*expression*>**)**

where <*expression*> is the string

Example:

containsOnlyDigits('12345')

9.9 ContainsOnlyLetters

Shall return TRUE if the required string argument is all letters

containsOnlyLetters(<*expression*>**)**

where <*expression*> is the string

Example:

containsOnlyLetters("onefortheroad")

9.10 Division

Shall return the result of the first required numeric argument divided by the second required numeric argument with standard convention casting

(<*expression*> / <*expression*>**)**

Examples:

(a/b)
(5/c)

9.11 EndsWith

Shall return TRUE if the first required string argument ends with the second required string argument

endsWith(<*expression1*>, <*expression2*>**)**

where <*expression1*> is the given string, and <*expression2*> is the substring

Example:

endsWith("just a test", "test")

9.12 Equal

Shall return TRUE if the first required argument and the second required argument do not evaluate to the same value

Equality Operator

(<*expression*> **==** <*expression*>**)**

Examples:

(a == b)
(stringLength(alpha) == 5)
(string1 == string2)

9.13 Exp

Shall return the value of *e* (Euler's number, the base of natural logarithms) raised to the power of the value of the required numeric expression

exp(<*expression*>**)**

where <*expression*> is the value of the power

Examples:

exp(nbr1)
exp(2)

9.14 Floor

Shall returns the largest integer that is less than or equal to the required numeric argument

floor(<*expression*>**)**

where <*expression*> is the numeric value

Example:

floor(nbr1)

9.15 GetDayOfMonth

Shall return the day of the required DATETIME argument as a numeric value, for example, the first day of the month has a value of 1, and so on.

getDayOfMonth(<*expression*>**)**

where <*expression*> is the DATETIME value

Example:

getDayOfMonth(aDate)

9.16 GetDayOfWeek

Shall return the day of the week of the required DATETIME argument as a numeric value, for example, Sunday = 1, Monday = 2, and so on

getDayOfWeek(<*expression*>**)**

where <*expression*> is the DATETIME value

Example:

getDayOfWeek(aDate)

9.17 GetDayOfWeekInMonth

Shall return the day of the week in month of the required DATETIME argument as a numeric value, for example, the DAY_OF_MONTH 1 through 7 always correspond to DAY_OF_WEEK_IN_MONTH 1; 8 through 14 correspond to DAY_OF_WEEK_IN_MONTH 2, and so on. DAY_OF_WEEK_IN_MONTH 0 indicates the week before DAY_OF_WEEK_IN_MONTH 1. Negative values count back from the end of the month, so the last Sunday of a month is specified as DAY_OF_WEEK = SUNDAY, DAY_OF_WEEK_IN_MONTH = -1.

getDayOfWeekInMonth(<*expression*>**)**

where <*expression*> is the DATETIME value

Example:

getDayOfWeekInMonth(aDate)

9.18 GetDayOfYear

Shall return the day within the year of the required DATETIME argument as a numeric value

getDayOfYear(<*expression*>**)**

where <*expression*> is the DATETIME value

Example:

getDayOfYear(aDate)

9.19 GetHour12

Shall return the hour of the required DATETIME argument in a 12-hour clock as a numeric value

getHour12(<*expression*>**)**

where <*expression*> is the DATETIME value

Example:

getHour12(aDate)

9.20 GetHour24

Shall return the hour of the required DATETIME argument in a 24-hour clock as a numeric value

getHour24(<*expression*>**)**

where <*expression*> is the DATETIME value

Example:

getHour24(aDate)

9.21 GetMillisecond

Shall return the millisecond within the second of the required DATETIME argument as a numeric value

getMillisecond(<*expression*>**)**

where <*expression*> is the DATETIME value

Example:

getMillisecond(aDate)

9.22 GetMinute

Shall return the minute within the hour of the required DATETIME argument as a numeric value

getMinute(<*expression*>**)**

where <*expression*> is the DATETIME value

Example:

getMinute(aDate)

9.23 GetMonth

Shall return the month within the year of the required DATETIME argument as a numeric value

getMonth(<*expression*>**)**

where <*expression*> is the DATETIME value

Example:

getMonth(aDate)

9.24 GetSecond

Shall return the second within the minute of the required DATETIME argument as a numeric value

getSecond(<*expression*>**)**

where <*expression*> is the DATETIME value

Example:

getSecond(aDate)

9.25 GetWeekOfMonth

Shall return the week within the month of the required DATETIME argument as a numeric value

getWeekOfMonth(<*expression*>**)**

where <*expression*> is the DATETIME value

Example:

getWeekOfMonth(aDate)

9.26 GetWeekOfYear

Shall return the week within the year of the required DATETIME argument as a numeric value

getWeekOfYear(<*expression*>**)**

where <*expression*> is the DATETIME value

Example:

getWeekOfYear(aDate)

9.27 GetYear

Shall return the year of the required DATETIME argument as a numeric value

getYear(<*expression*>**)**

where <*expression*> is the DATETIME value

Example:

getYear(aDate)

9.28 Greater

Shall return TRUE if the first required argument is greater than (or comes later in lexicographical order based on UTF-8) the second required argument; returns FALSE otherwise

(<*expression*> **>** <*expression*>**)**

Examples:

(a > 4)
(stringLength(alpha) > stringLength("beta"))

9.29 Greater or Equal

Shall return TRUE if the first required argument is greater than (or comes later in lexicographical order based on UTF-8) or equal to the second required argument; returns FALSE otherwise

(<*expression*> **>=** <*expression*>**)**

Examples:

(a >= 4)
(stringLength(alpha) >= stringLength("beta"))

9.30 IsWithin

Shall return TRUE if the IsWithin checks whether a DATETIME is inside a time period. When taking three DATETIME expressions, the first is the DATETIME, and the remaining two define the start and end of the time period.

isWithin(<*expression1*>, <*expression2*>, <*expression3*>**)**

where <*expression1*> is the DATETIME value to check, <*expression2*> is the start DATETIME value, and <*expression3*> is the end DATETIME value

Examples:

isWithin(aDate1, aDate2, aDate3)

isWithin(2005-01-29T09:40:00 TZ=America/Chicago, aDate1, aDate2)

isWithin(2005-01-29T09:40:00 TZ=America/Chicago, 2006-01-29T09:40:00 TZ=America/Chicago, 2006-01-29T09:40:00 TZ=America/Chicago)

isWithin(aDate, aDate, 2006-01-29T09:40:00 TZ=America/Chicago)

9.31 Less

Shall return TRUE if the first required string or numeric argument is less than (or comes earlier in lexicographical order based on UTF-8) the second required string or numeric argument; returns FALSE otherwise. Both arguments shall be of the same datatype.

(<*expression*> **<** <*expression*>**)**

Examples:

(a < 4)
(stringLength(alpha) < stringLength("beta"))

9.32 Less or Equal

Shall return TRUE if the first required string or numeric argument is less than (or comes earlier in lexicographical order based on UTF-8) or equal to the second required string or numeric argument; returns FALSE otherwise. Both arguments shall be of the same datatype.

(<*expression*> **<=** <*expression*>**)**

Examples:

(a <= 4)
(stringLength(alpha) <= stringLength("beta"))

9.33 Ln

Shall return the natural logarithm of the given required numeric expression (logarithm base e)

ln(<*expression*>**)**

where <*expression*> is the numeric value

Example:

ln(nbr1)

9.34 Max

Shall return the maximum value of the required numeric or string arguments. All arguments shall be either numeric type or string type.

max(<*expression*>, <*expression*>, [<*expression*>…]**)**

where <*expression*>s are the numeric values to compare

Examples:

```
max(nbr1, nbr2)
max(aNbr, 4, ToSINT16("2"))
```

9.35 Min

Shall return the minimum value of the required numeric or string arguments. All arguments shall be either numeric type or string type.

min(<*expression*>, <*expression*>, [<*expression*>…]**)**

where <*expression*>s are the numeric values to compare

Examples:

```
Min(nbr1, nbr2)
min(aNbr, 4, ToSINT16("2"))
```

9.36 Subtraction

Shall return the result of the first required numeric argument minus the second optional numeric argument if two arguments are present; otherwise, returns the unary minus of the first required numeric argument. The data type of the result value shall follow standard JAVA casting conventions.

(<*expression*> **-** <*expression*>**)**

Examples:

```
(a - b)
(stringLength(alpha) - 5)
```

9.37 Not Equal

Shall return TRUE if the first required argument and the second required argument do not evaluate to the same value

(<*expression*> **!=** <*expression*>**)**

Examples:

```
(a != b)
(stringLength(alpha) != c)
```

9.38 Logical Not

Shall return a Boolean value that corresponds to the logical NOT operation on the required Boolean argument

!(<*expression*>**)**

Examples:

```
!(alpha)
!(true)
```

9.39 Logical Or

Shall return a Boolean value that corresponds to the logical OR operation on the required Boolean arguments

(<*expression*> **||** <*expression*>**)**

Examples:

```
(a || b)
((stringLength(alpha) < 5) || (5+b))
```

9.40 Addition

Shall return the sum of the required numeric arguments. The data type of the result value shall follow the standard JAVA casting conventions.

(<*expression*> **+** <*expression*>**)**

Examples:

```
(a + b)
(stringLength(alpha) + 5)
```

9.41 Power

Shall return the value of the first required numeric argument raised to the power of the second required numeric argument.

power(<*expression1*>, <*expression2*>**)**

where <*expression1*> is the value raised to the power of <*expression2*>

Examples:

```
power(nbr1, nbr2)
power(2, 4)
```

9.42 Product

Shall return the product of the required numeric arguments. The data type of the result value shall follow the standard JAVA casting conventions.

(<*expression*> ***** <*expression*>**)**

Examples:

```
(a * b)
(stringLength(alpha) * c)
```

9.43 Mod

Shall return the remainder from an operation of dividing the first required numeric argument by the second required numeric argument

mod(<*expression1*>, <*expression2*>**)**

where <*expression1*> is the value divided by <*expression2*>

Examples:

```
mod(nbr1, nbr2)
mod(aNbr, 4)
```

9.44 Round

Shall return the closest SINT32 value to the required numeric argument. The return type of the result shall follow Java conventions for rounding and unary numeric promotion.

round(<*expression*>**)**

where <*expression*> is the value to round

Examples:

round(nbr1)
round(ToREAL32(aNbr))

9.45 SquareRoot

Shall return the square root of the required numeric argument

squareRoot(<*expression*>**)**

where <*expression*> is the numeric value

Example:

squareRoot(nbr1)

9.46 StringLength

Shall return the number of characters in the required string argument

stringLength(<*expression*>**)**

where <*expression*> is the string

Examples:

stringLength(alpha)
stringLength("hello world")

9.47 MatchesRegExp

Shall return TRUE if the required first string argument matches the regular expression defined by the required second string argument. The second string argument shall be interpreted as a regular expression.

matchesRegExp(<*expression1*>, <*regExp*>**)**

where <*expression1*> shall return a string and <*regExp*> shall be a regular expression that shall follow the syntax and semantics of regular expressions in the Pattern class of the java.util.regex core package in Java 2 SE 5.0.

Example:

matchesRegExp(IP,"\d{1,3}+\.\d{1,3}+\.\d{1.3}")

9.48 Substring Operations

The following set of string operations can be implemented using the MatchesRegExp operator. However, CIM-SPL provides for these additional substring operations for readability and possibly more efficient implementations.

9.48.1 Substring

The substring operator takes either two or three arguments. The first and second arguments of this operator are required while the third argument is optional. The first argument of this operator shall be a string argument, while the second and third argument shall be integer argument.

This operator shall return the substring of the first string argument, starting at the position indicated by the second numeric argument and going to the end of the string or the position indicated by the third numeric argument - 1. The position of a character is determined as follows: The first character is at position 0, the second character is at position 1, and so on.

The second numeric argument shall be greater than or equal to 0. The third numeric argument shall be greater than the second numeric argument if the third argument is present. If the starting position given by the second numeric argument is greater than the length of the string, an empty string shall be returned. If the third numeric position is not present, the string starting at the second numeric position until the end of the string shall be returned.

substring(<*expression1*>, <*expression2*>, [<*expression3*>]**)**

where <*expression1*> is a string argument, and <*expression2*> and <*expression3*> are integers (UINT32) argument.

Examples:

substring(”Robert Hancock”, 2, 8) returns “bert H”.

9.48.2 LeftSubstring

The LeftSubstring operator returns a prefix of a given string argument by taking three arguments. How to compute the prefix is determined by the arguments. The first argument shall be a string and it indicates the given string, the second argument shall be either an integer or a string indicating an offset, and the third argument shall be a string indicating a direction and is either “LeftToRight” or “RightToLeft”.

When the offset is given by a number, the prefix is determined by counting the character position by the offset from either left to right (from the beginning of the string) or from right to left (from the end of the string). In particular, if the direction is "LeftToRight", the offset indicates the number of characters to return from the beginning of the string. If the direction is "RightToLeft", the offset indicates the number of characters to skip from the end of the string.[3] For example, leftSubstring("Mississippi", 4, "LeftToRight") returns "Miss", and leftSubstring("Mississippi", 4, "RightToLeft") returns "Mississ".

When the offset is given by a string, the prefix is determined by searching for the offset string in the original string in the direction specified by the third parameter. The returned substring consists of the characters on the left side of the offset string. For example, leftSubstring("Mississippi", "ss", "LeftToRight") returns "Mi", and leftSubstring("Mississippi", "ss", "RightToLeft") returns "Missi".

leftSubstring(<*expression1*>, <*expression2*>, <*expression3*>**)**

where <*expression1*> is a string, <*expression2*> is an integer, and <*expression3*> is a string constant that indicates either "LeftToRight" or "RightToLeft"

leftSubstring(<*expression1*>, <*expression2*>, <*expression3*>**)**

where <*expression1*> and <*expression2*> are strings, and <*expression3*> is a string constant that indicates either "LeftToRight" or "RightToLeft"

[3] If the offset value is a negative number, the entire string is returned in either case.

Examples:

leftSubstring(StateSymbolAndZip, 2, "LeftToRight") // to get the state symbol
leftSubstring(FirstAndLastName, " ", "LeftToRight") // to get the first name

9.48.3 RightSubstring

The RightSubstring operator returns a suffix of a given string argument by taking three arguments. How to compute the suffix is determined by the arguments. The first argument shall be a string and it indicates the given string, the second argument shall be either an integer or a string indicating an offset, and the third argument shall be a string indicating a direction and is either “LeftToRight” or “RightToLeft”.

When the offset is given by a number, the suffix is determined by simply counting the character position by the offset from either left to right (from the beginning of the string) or from right to left (from the end of the string). In particular, if the direction "RightToLeft", the offset indicates the number of characters to return as a suffix. If the direction is "LeftToRight", the offset indicates the number of characters to skip from the beginning of the string. For example, rightSubstring("Mississippi", 4, "LeftToRight") returns "issippi", and rightSubstring("Mississippi", 4, "RightToLeft") returns "ippi".

When the offset is given by a string, the suffix is determined by searching for the offset string in the original string in the direction specified by the third parameter. The returned substring consists of the characters on the right side of the offset string. For example, rightSubstring("Mississippi", "ss", "LeftToRight") returns "issippi", and rightSubstring("Mississippi", "ss", "RightToLeft") returns "ippi".

rightSubstring(<*expression1*>, <*expression2*>, <*expression3*>**)**

where <*expression1*> is a string, <*expression2*> is an integer, and <*expression3*> is a string constant that indicates either "LeftToRight" or "RightToLeft"

rightSubstring(<*expression1*>, <*expression2*>, <*expression3*>**)**

where <*expression1*> and <*expression2*> are strings, and <*expression3*> is a string constant that indicates either "LeftToRight" or "RightToLeft"

Examples:

rightSubstring(StateSymbolAndZip, 5, RightToLeft) // to get the zip code
rightSubstring(FirstAndLastName, " ", "LeftToRight) // to get the last name

9.48.4 MiddleSubstring

The MiddleSubstring operator returns a middle portion of a given string using various arguments as filters. How to compute the suffix is determined by the arguments. MiddleSubstring takes four arguments: original string, first offset, second offset, and direction string. The first and second offsets shall be specified either by a number or a string. The direction string can be either "LeftToRight" or "RightToLeft". The meaning of the first offset is similar to that in the rightSubstring: it indicates where the resulting substring starts scanning, either from the left or from the right based on the direction string. The meaning of the second offset is as follows: if it is a number, it simply indicates the number of characters to return; if it is a string, it specifies where the substring should end. For example:

middleSubstring(<*expression1*>, <*expression2*>, <*expression3*>, <*expression4*>**)**

where <*expression1*> is a string, <*expression2*> and <*expression3*> are integers, and <*expression4*> is a string constant that indicates either "LeftToRight" or "RightToLeft"

middleSubstring(<*expression1*>, <*expression2*>, <*expression3*>, <*expression4*>**)**

where <*expression1*> is a string, <*expression2*> is an integer, <*expression3*> is a string, and <*expression4*> is a string constant that indicates either "LeftToRight" or "RightToLeft"

middleSubstring(<*expression1*>, <*expression2*>, <*expression3*>, <*expression4*>**)**

where <*expression1*> is a string, <*expression2*> is a string, <*expression3*> is an integer, and <*expression4*> is a string constant that indicates either "LeftToRight" or "RightToLeft"

middleSubstring(<*expression1*>, <*expression2*>, <*expression3*>, <*expression4*>**)**

where <*expression1*> is a string, <*expression2*> and <*expression3*> are strings, and <*expression4*> is a string constant that indicates either "LeftToRight" or "RightToLeft"

Examples:

middleSubstring("Mississippi", 4, 5, "LeftToRight") = "issip"

middleSubstring("Mississippi", 4, 5, "RightToLeft") = "ippi"

middleSubstring("Mississippi", "ss", 5, "LeftToRight") = "issip"

middleSubstring("Mississippi", "ss", 5, "RightToLeft") = "ippi"

middleSubstring("Mississippi", 4, "ss", "LeftToRight") = "i"

middleSubstring("Mississippi", 4, "ss", "RightToLeft") = ""

middleSubstring("Mississippi", "ss", "ip", "LeftToRight") = "iss"

middleSubstring("Mississippi", "ss", "ip", "RightToLeft") = "Missi"

9.48.5 ReplaceSubstring

The ReplaceSubstring operator shall take two or three string arguments. It replaces one substring with another substring in a given string. The first argument specifies the given string, the second argument specifies a from-string, and the third argument specifies a to-string. Note that it is a purely functional form with no side-effect — that is, none of the string arguments are modified.

replaceSubstring(<*expression1*>, <*expression2*>, [<*expression3*>]**)**

where <*expression1*>, <*expression2*>, and <*expression3*> are strings

Example:

replaceSubstring(Name, "Jim", "James")

9.49 ToBoolean

Shall return Boolean TRUE if the required argument is a string argument that equates to “true” ignoring the case; shall return Boolean TRUE if the required argument is a numeric argument that evaluates to a non-zero; otherwise, it shall return FALSE.

Note: toBoolean(1) returns TRUE. toBoolean("1") returns FALSE, because this is passing in a string.

toBoolean(<*expression*>**)**

where <*expression*> is the value to be converted

Examples:

toBoolean("true")
toBoolean(1)

9.50 ToREAL32

Shall return a Real32 value corresponding to the one required argument. The required argument shall be either of type string or numeric. The conversion shall be done according to the Java conventions.

ToREAL32(<*expression*>**)**

where <*expression*> is the string

Examples:

ToREAL32("25.")
ToREAL32(alpha)

9.51 ToSINT32

Shall return a SINT32 value corresponding to the one required argument. The required argument shall be either of type string or numeric. The conversion shall be done according to the Java conventions.

ToSINT32(<*expression*>**)**

where <*expression*> is the string

Examples:

ToSINT32("257")
ToSINT32(alpha)

9.52 ToSINT16

Shall return a SINT16 value corresponding to the one required argument. The required argument shall be either of type string or numeric. The conversion shall be done according to the Java conventions.

ToSINT16(<*expression*>**)**

where <*expression*> is the value to convert

Examples:

ToSINT16("25")
ToSINT16(alpha)

9.53 ToSINT64

Shall return a SINT64 value corresponding to the one required argument. The required argument shall be either of type string or numeric. The conversion shall be done according to the Java conventions.

ToSINT64(<*expression*>**)**

where <*expression*> is the value to convert

Examples:

ToSINT64("2556")
ToSINT64(255)
ToSINT64(alpha)

9.54 ToLower

Shall return the required string argument converted into lowercase

toLower(<*expression*>**)**

where <*expression*> is the string

Examples:

```
toLower(alpha)
toLower("Hello World")
```

9.55 ToMilliseconds

Shall return the number of milliseconds since the standard base time known as "the epoch," namely January 1, 1970, 00:00:00 GMT, corresponding to the required DATETIME argument.

toMilliseconds(<*expression*>**)**

where <*expression*> is the DATETIME value

Example:

```
toMilliseconds(aDate)
```

9.56 ToSINT8

Shall return a SINT18 value corresponding to the one required argument. The required argument shall be either of type string or numeric. The conversion shall be done according to the Java conventions.

ToSINT8(<*expression*>**)**

where <*expression*> is the value to convert

Examples:

```
ToSINT8("25")
ToSINT8(25)
ToSINT8(alpha)
```

9.57 ToString

Shall return a String value corresponding to the one required argument. The required argument shall be either of type Boolean or numeric. The conversion shall be done according to the Java conventions.

Converts the numeric and the Boolean arguments into a string value

toString(<*expression*>**)**

where <*expression*> is the value to convert

Examples:

```
toString(nbr1)
toString(1)
toString(true)
```

9.58 ToUINT32

Shall return a UINT32 value corresponding to the one required argument. The required argument shall be either of type string or numeric. The conversion shall be done according to the Java conventions.

ToUINT32(<*expression*>**)**

where <*expression*> is the value to convert

Examples:

```
ToUINT32("2556")
ToUINT32(2550)
ToUINT32(alpha)
```

9.59 ToUINT16

Shall return a UINT16 value corresponding to the one required argument. The required argument shall be either of type string or numeric. The conversion shall be done according to the Java conventions.

ToUINT16(<*expression*>**)**

where <*expression*> is the value to convert

Examples:

```
ToUINT16("2556")
ToUINT16(2550)
ToUINT16(alpha)
```

9.60 ToUINT64

Shall return a UINT64 value corresponding to the one required argument. The required argument shall be either of type string or numeric. The conversion shall be done according to the Java conventions.

ToUINT64(<*expression*>**)**

where <*expression*> is the value to convert

Examples:

```
ToUINT64("2556")
ToUINT64(2550)
```

9.61 ToUINT8

Shall return a UINT8 value corresponding to the one required argument. The required argument shall be either of type string or numeric. The conversion shall be done according to the Java conventions.

ToUINT8(<*expression*>**)**

where <*expression*> is the value to convert

Examples:

```
ToUINT8("25")
ToUINT8(25)
```

9.62 ToUpper

Shall return an uppercase version of the required string argument.

toUpper(*<expression>***)**

where <*expression*> is the string

Examples:

toUpper(alpha)
toUpper("hello world")

9.63 Word

This operator shall take three arguments. The first two arguments shall be of type string, and the third argument shall be of type number. This operator shall extract *n* words from the first string argument where the third argument specifies the number *n*. Words are defined as text between the separator substring given by the second argument.

word(<*expression1*>, <*expression2*>, <*expression3*>**)**

where <*expression1*> is the given string, <*expression2*> is the separator substring, and <*expression3*> is the number

Example:

Word(alpha, " ", 3)

9.64 Logical XOR

Shall return a Boolean value that corresponds to the logical XOR operation on the bit representation of the two required numeric arguments

(<*expression*> **^** <*expression*>**)**

Examples:

(a ^ b)
(Netmask ^ (IP))

9.65 StringConstant

Values inside double quotes are converted to StringConstants.

Example:

alpha = = "22"

9.66 LongConstant

UINT32 (unquoted) values that do not contain decimals are converted to UINT64 constants.

Example:

alpha == 22

9.67 DoubleConstant

Numeric (unquoted) values that contain decimal points are converted to Real64 constants. This includes numeric values ending in a decimal point (for example, 22).

Example:

alpha == 22.25

9.68 DATETIMEConstant

Unquoted values in the format "yyyy-mm-ddThh:mm:ss TZ=javaTimezoneID" are interpreted as a DATETIMEConstant following the XML standard semantics.

Example:

alpha > 2004-01-29T09:40:00 TZ=America/Chicago

9.69 BooleanConstant

Unquoted strings 'true' or 'false' inside the clauses are converted to BooleanConstants in the resulting XML.

Example:

alpha == true

9.70 Identifier

Identifier can be either simple or multi-level. A **simple identifier** shall be any of the following values:

- Name of a named constant. It evaluates to the value of the named constant as defined in the declaration sections.
- Name of a named macro. It evaluates to the value of the named macro as defined in the declaration sections.
- <classname.propertyname>, where classname is the class name of the Import statement or any super class of that class, and propertyname is the name of a property of the classname. The prefix "classname." is optional. It is required only to disambiguate the name of a property if the same name is used in any super class and it is not overridden. It evaluates to the value of the property of the class instance under consideration.
- Any of the preceding three values followed by an index enclosed in square brackets. In this case, the type of the named constant, macro, or property should be an array of an intrinsic CIM data type or CIM references. The index can only be an expression that evaluates to an integer (UINT32) value. The first index is always 0. If any of these conditions is not true, the policy parser returns an error; otherwise, the expression evaluates to the value of the data in the position indicated by the index in the array identified by the named constant, macro, or property.
- A qualifier that is an expression that evaluates to a reference of an instance of a CIM class. This can be the reserved word *Self* that refers to the object instance under consideration or the member of a collect operator that returns collections of references (see section 11.2).

A **multi-level identifier** has the form <qualifier.simpleIdentifier>, where simpleIdentifier is a simple identifier that is not a qualifier.

A simple identifier that is not prefixed by a qualifier shall be evaluated under the scope of the managed element that is made available to the rule based on the Import statement. If the qualifier appears, the scope of the simple identifiers shall be the object referenced by the evaluation of the qualifier.

10 Simple Boolean Condition

A simple Boolean condition shall be a CIM-Expression with the following two properties:

- The expression evaluates to a Boolean constant.
- Any identifier that appears in the expression is a simple identifier that is not a qualifier.

11 Collection Operations

In addition to operators that apply to the CIM intrinsic data types, CIM-SPL also supports operations over arrays. In addition to handling arrays of basic CIM intrinsic types, CIM-SPL operations also manipulate arrays of references to CIM object instances. All the arrays returned by a CIM-SPL operation behave as a CIM Indexed array. These operations are referred to as collection operations, and they are described in the following subsections.

11.1 Basic Collection

Shall return an array of intrinsic CIM data type objects, all of the same type

[<*expression1*>, ,*expression2*>, … , <*expressionN*> **]**

where the N <*expressions*> are of the same type. At least one expression is required.

Example:

[2 , 2, 3+StringLength("abc")]

11.2 Collect

Shall return an array of an intrinsic CIM data type or references to object instances all of the same class. This operator has two signatures:

collect(<*RefExpression*>,<*association*>, <*role*>, <*resultRole*>,<*CIM class Name*>, <*expression*>**)**

collect(<*RefExpression*>,<*association*>, <*role*>, <*resultRole*>, <*CIM class Name*>, <*CIM class property name*>, <*expression*>**)**

where <*RefExpression*> shall be an expression that evaluates to an object reference, <association> shall be the name of a CIM association, <*role*> and <*resultRole*> shall be reference names in the <*association*>, <*CIM class Name*> shall be the class of the <*resultRole*> (this argument can be null if there is no ambiguity on the possible classes of the <*resultRole*>), <*CIM class property name*> shall be the name of a property of objects that might be referenced by the <*resultRole*> reference in instances of the <*association*>, the <*expression*> is a Boolean expression, and the <*identifiers*> appearing in the <*expression*> are evaluated under the scope of the objects referenced by the <*resultRole*> reference in instances of the <*association*>.

In the first signature, any instance of the <*association*> in which the reference returned by the evaluation of the <*RefExpression*> appears as the value of the <*role*> reference, the object reference by the <*resultRole*> is inspected and, if the expression evaluated under the scope of this object evaluates to TRUE, the object is returned in the collection. In the second signature, instead of returning a reference to the object, only the value of the property identified by the <*CIM class property name*> is returned. If this property is an array, this operation returns an array with the first values of all collected object properties.

Examples:

collect(Self, Realizes, PhysicalElement, LogicalDevice, CIM_LogicalDevice, TotalPowerOnHours > 5)

collect(Self , Realizes, PhysicalElement, LogicalDevice, CIM_LogicalDevice, TotalPowerOnHours, TotalPowerOnHours > 5)

Self references an instance of the CIM_PhysicalElement class. To traverse multiple associations, collect operators can be nested, as in the following example:

```
collect(
    collect(Self , Realizes,
        PhysicalElement, LogicalDevice, CIM_LogicalDevice,
        true)[0],
    DeviceServiceImplementation,
    LogicalDevice,
    CIM_Service,
    True)[1].Started
```

Starting in a PhysicalElement (see Figure 4), the Realizes association is traversed to get a collection of LogicalDevice elements. Using one of these logical devices as an anchor (the first one in the collection), the DeviceServiceImplementation association is traversed to get a collection of Service elements. The expression takes the second element in this collection ([1]), and the value of the Started property is returned.

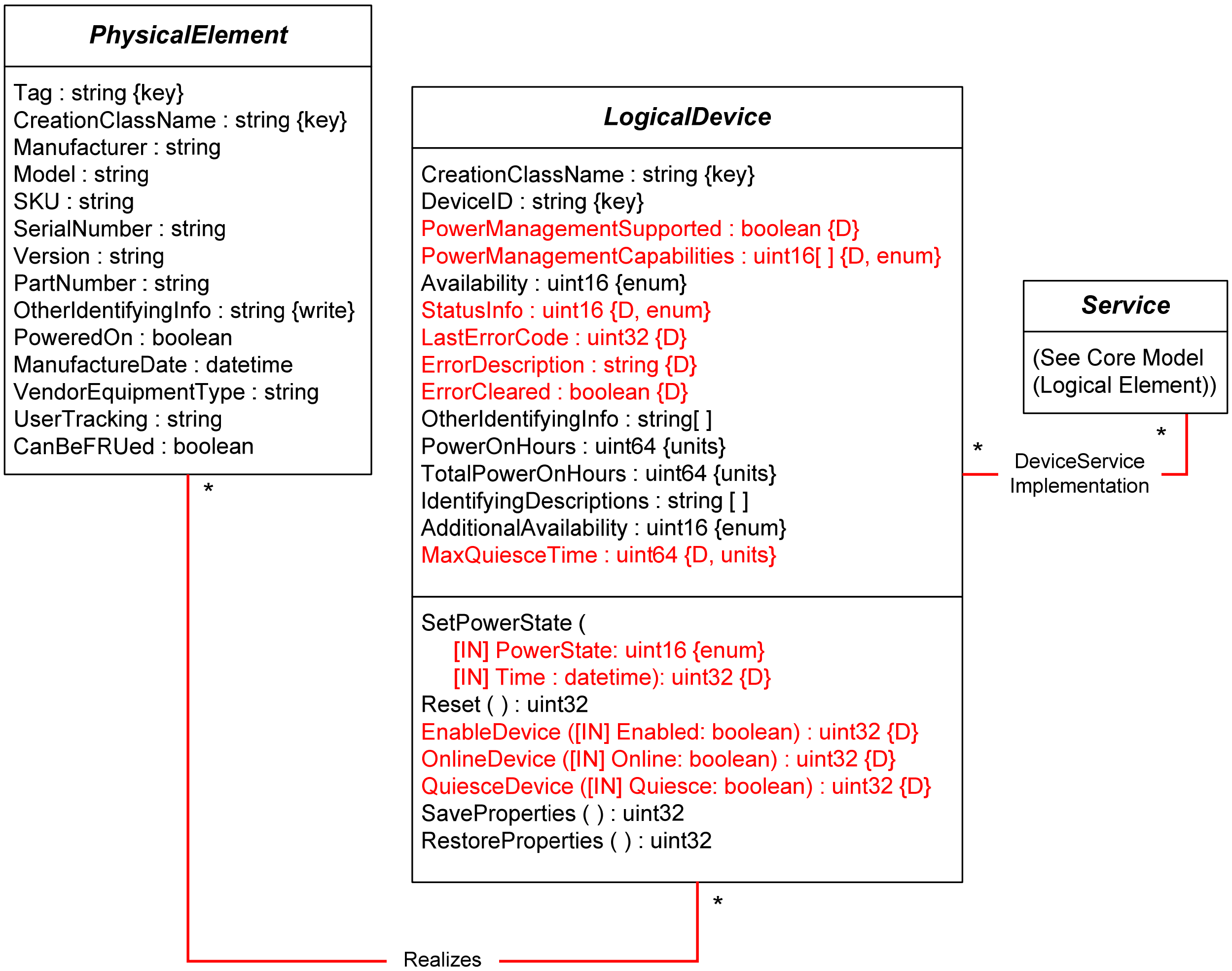

Figure 4 – Example of CIM Associations

11.3 InCollection

Checks whether an object is a member of a collection

inCollection(<*expression*>, <*collection*>**)**

Shall returns TRUE if the value returned by <*expression*> appears in <*collection*>. The type of the <*expression*> shall the same type as the arguments in the <*collection*>.

Example:

inCollection(100, collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice, CIM_LogicalDevice, TotalPowerOnHours, TotalPowerOnHours > 5))

11.4 Union

Shall results in a new collection that is the union of the two required collections in the arguments. Object repetitions are preserved.

union(<*collection*>, <*collection*>**)**

Example:

union([100],collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice, CIM_LogicalDevice, TotalPowerOnHours, TotalPowerOnHours > 5))

11.5 SubCollection

Checks whether a collection is a subcollection of another collection

subCollection(<*collection1*>, <*collection2*>**)**

Shall return TRUE if every member in <*collection1*> appears in <*collection2*>

Example:

subCollection([100,150],collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice, CIM_LogicalDevice, TotalPowerOnHours, TotalPowerOnHours > 5))

11.6 EqCollections

Checks whether two collections are equal

eqCollections(<*collection1*>, <*collection2*>**)**

Shall return TRUE if <*collection1*> is a subcollection of <*collection2*>, <*collection2*> is a subcollection of <*collection1*>, and the repetitions of objects in both collections is identical

Example:

eqCollections([100,150],collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice, CIM_LogicalDevice, TotalPowerOnHours, TotalPowerOnHours > 5))

11.7 AnyInCollection

Checks whether an object with a given property exists in a collection

anyInCollection(<*expression*> <*op*> <*collection*>**)**

where <*op*> shall be either a Boolean or relational operator. If <*op*> is a Boolean operator, <*expression*> shall be Boolean. If <*op*> is relational, <*expression*> shall be of a type compatible with the operator. The operation shall return TRUE if there is an object <*oj*> in <*collection*> such that <*expression*> <*op*> <*oj*> is true.

Example:

anyInCollection(240 > collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice, CIM_LogicalDevice, TotalPowerOnHours, TotalPowerOnHours > 5))

11.8 AllInCollection

Checks whether all objects in a collection have a given property

allInCollection(<*expression*> <*op*> <*collection*>**)**

where <*op*> shall be either a Boolean or relational operator. If <*op*> is a Boolean operator, <*expression*> shall be Boolean. If <*op*> is relational, <*expression*> shall be of a type compatible with the operator. The operation shall return TRUE if for every object <*oj*> in <*collection*>, <*expression*> <*op*> <*oj*> returns TRUE.

Example:

anyInCollection(240 > collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice, CIM_LogicalDevice, TotalPowerOnHours, TotalPowerOnHours > 5))

11.9 ApplyToCollection

Applies an arithmetic operation to each element in a collection and shall return a numeric collection

applyToCollection(<*expression*> <*op*> <*collection*>**)**

where <*op*> shall be a binary numeric operator and <*expression*> is a numeric expression. The operation shall return a collection similar to <*collection*> but in which every object <*oj*> in <*collection*> is replaced by the result of the expression <*expression*> <*op*> <*oj*>.

Example:

applyToCollection(1024 + collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice, CIM_LogicalDevice, TotalPowerOnHours, TotalPowerOnHours > 5))

11.10 Sum

Shall return the sum of a collection of numeric CIM data elements. The <*collection*> shall be a collection of numeric values.

sum(<*collection*>**)**

Example:

sum(collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice, CIM_LogicalDevice, TotalPowerOnHours, TotalPowerOnHours > 5))

11.11 MaxInCollection

Shall return the maximum object from a collection of totally ordered CIM data objects

maxInCollection(<*collection*>**)**

Example:

maxInCollection(collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice, CIM_LogicalDevice, TotalPowerOnHours, true))

Strings are ordered lexicographically based on UTF-8.

11.12 MinInCollection

Shall return the smallest object from a collection of totally ordered CIM data objects

minInCollection(<*collection*>**)**

Example:

minInCollection(collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice, CIM_LogicalDevice, TotalPowerOnHours, true))

11.13 AvrgInCollection, MedianInCollection, sdInCollection

Shall return the average/median/standard deviation in a double from a collection of numeric CIM data objects. The <*collection*> shall be a collection of numeric values.

avrgInCollection(<*collection*>**) / medianInCollection(**<*collection*>**) / sdInCollection(**<*collection*>**)**

Example:

avrgInCollection(collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice, CIM_LogicalDevice, TotalPowerOnHours, true))

11.14 CollectionSize

Shall return the size of a collection in a UINT32

collectionSize(<*collection*>**)**

Example:

collectionSize(collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice, CIM_LogicalDevice, TotalPowerOnHours, true))

12 Policy Example

The following example shows a policy that is invoked when a file system is 85 percent full. The policy expands the storage pool by 25 percent.

```
Import   CIM_X_XX_XXXX::CIM_LocalFileSystem;
Strategy Execute_All_Applicable;
Policy {
 Declaration { /* Macros to traverse HostedService associations to get */
                   /* to FileSystemConfigurationService for ModifyFile and */
                   /* StorageConfigurationService for CreateOrModify…   */
     computer_system = collect(Self, CIM_HostedFileSystem,
                                PartComponent, GroupComponent, null, true)[0];
     storage_config_service  =
                      collect(computer_system, CIM_HostedService, Antecedent,
                                  Dependent, CIM_StorageConfigurationService,
                                  true)[0];
     logical_disk = collect(Self, CIM_ResidesOnExtent,
                              Dependent, Antecedent, null, true)[0];
     storage_pool = collect(logical_disk, CIM_AllocatedFromStoragePool,
```

```
                              Dependent, Antecedent, null, true)[0];
     fs_goal = collect(Self, CIM_ElementSettingData, ManagedElement,
                         SettingData, CIM_FileSystemSetting true)[0];
    }
Condition {
         (AvailableSpace / FileSystemSize) < 0.25
    }
Decision {
     storage_conf_service.CreateOrModifyElementFromStoragePool("LogicalDisk",
                                                /* ElementType Volume */
                                                8, job, fs_goal,
                                                1.25 * FileSystemSize,
                                                storage_pool,
                                                logical_disk)
        // CreateOrModifyElementFromStoragePool defined in pp. 1024
        // of SMI-S 1.1.0 SNIA Standard document
        // ElementType value list can be found in pp. 1116
            | DoSomethingOnFailure()
    }
}:1;
```

13 CIM-SPL Grammar

In the following grammar, non-terminal symbols are represented by sequences of uppercase letters in boldface. Alternatives in the production rules are represented by "|", except for the use of "||" in Boolean expressions and "||| and "|" in action blocks. All blanks but one are ignored in the rules. Blanks do not appear in any of the terminal symbols.

CIMPOLICY →
    Import **IMPORTSTATEMENT** ;
    Strategy **STRATEGYSTATEMENT** ;
    DECLARATIONSTATEMENT
    POLICYSTATEMENTS
IMPORTSTATEMENT →
    MOFFILENAME :: CLASSNAME OPTIONALBOOLEANCONDITION
MOFFILENAME →
    cim_v**MAJOR_MINOR_RELEASE_TYPE_FILENAME**
MAJOR →
    DECIMALNUMBER
MINOR →
    DECIMALNUMBER
RELEASE →
    DECIMALNUMBER
TYPE →
    preliminary | final
FILENAME →

```
    <a MOF file name without the extension>
CLASSNAME →
    <the name of a CIM class name defined in the MOF file>
OPTIONALBOOLEANCONDITION →
    <> | SIMPLEBOOLEANEXPRESSION
STRATEGYSTATEMENT →
    Execute_All_Applicable | Execute_First_Applicable
DECLARATIONSTATEMENT →
    <> | Declaration { DECLARATIONS }
DECLARATIONS →
    CONSTANTDECLARATION MACRODECLARATION
CONSTANTDECLARATION →
    <> | NAME  = EXPRESSION ; CONSTANTDECLARATION
MACRODECLARATION →
    <> | Macro { MACRO } MACRODECLARATION
MACRO →
    Name = NAME ; type = CIMTYPE ; ARGUMENTS procedure = EXPRESSION
ARGUMENTS →
    <> | argument = NAME : CIMTYPE MOREARGUMENTS ;
MOREARGUMENTS →
    <> | , NAME : CIMTYPE MOREARGUMENTS
POLICYSTATEMENTS →
    POLICY ; MOREPOLICYSTATEMENTS  | POLICYGROUP ; MOREPOLICYSTATEMENTS
MOREPOLICYSTATEMENTS →
    <> | POLICYSTATEMENTS
POLICY →
    Policy { DECLARATIONSTATEMENT CONDITIONSTATEMENT DECISION } : PRIORITY
CONDITIIONSTATEMENT →
    <> | Condition { BOOLEANEXPRESSION }
DECISION →
    Decision { ACTIONBLOCK }
PRIORITY →
    DECIMALNUMBER
EXPRESSION →
    BOOLEANEXPRESSION | ARITHMETICEXPRESSION | STRINGEXPRESSION |
    DATETIMEEXPRESSION
BOOLEANEXPRESSION →
    TRUE | FALSE | IDENTIFIER | FUNCTIONCALL |
    BOOLEANEXPRESSION BOOLEANOPERATOR BOOLEANEXPRESSION |
    ARITHMETICEXPRESSION RELATIONALOPERATOR ARITHMETICEXPRESSION |
    STRINGEXPRESSION RELATIONALOPERATOR STRINGEXPRESSION |
    BOOLEANEXPRESSION EQOPERATOR BOOLEANEXPRESSION |
    ( BOOLEANEXPRESSION ) | ! ( BOOLEANEXPRESSION )
BOOLEANOPERATOR →
```

```
    && | || | ^
RELATIONALOPERATOR →
    EQOPERATOR | >= | <= | > | <
EQOPEATOR →
    == | !=
ARITHMETICEXPRESSION →
    NUMBER  | IDENTIFIER | FUNCTIONCALL |
    ARITHMETICEXPRESSION * ARITHMETICEXPRESSION |
    ARITHMETICEXPRESSION / ARITHMETICEXPRESSION |
    ARITHMETICEXPRESSION + ARITHMETICEXPRESSION |
    ARITHMETICEXPRESSION - ARITHMETICEXPRESSION |
    ( ARITHMETICEXPRESSION )
NUMBER →
    UNSIGNINTEGER | INTEGER | REAL
UNSIGNINTEGER →
    0UDECIMALNUMBER
INTEGER →
    SIGN DECIMALNUMBER
REAL →
    INTEGER DECIMAL EXP
DECIMAL →
    <> | .DECIMALNUMBER
EXP →
    <> | EINTEGER
SIGN →
    <> | + | -
DECIMALNUMBER →
    DIGIT MOREDIGITS
DIGIT →
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
MOREDIGITS →
    <> | DIGIT MOREDIGITS
STRINGEXPRESSION →
    <any sequence of Unicode characters in between '> |
    IDENTIFIER | FUNCTIONCALL
DATETIMEEXPRESSION →
    DATATIME | IDENTIFIER | FUNTIONCALL
DATATIME →
    DIGIT DIGIT DIGIT DIGIT-DIGIT DIGIT-DIGIT DIGIT T
    DIGIT DIGIT:DIGIT DIGIT:DIGIT DIGIT TZ=<javaTimezoneID>
IDENTIFIER →
    Self | SIMPLEIDENTIFIER | COMPLEXIDENTIFIER
SIMPLEIDENTIFIER →
    NAME INDEX | NAME.NAME INDEX
```

```
NAME →
    < any sequence of letters, numbers, and "_" that starts with a letter>
INDEX →
    <> | [ ARITHMETICEXPRESSION ]
COMPLEXIDENTIFIER →
    FUNCTIONCALL.SIMPLEIDENTIFIER
FUNCTIONCALL →
    NAME ( PARAMETERS ) |
    collect( PARAMETERS )[ ARITHMETICEXPRESSION ]. SIMPLEIDENTIFIER
PARAMETERS →
    <> | EXPRESSION MOREPARAMETERS
MOREPARAMETERS →
    <> | , PARAMETERS
ACTIONBLOCK →
    IDENTIFIER ( ACTIONARGS ) COMP |
    ACTIONBLOCK -> ACTIONBLOCK |
    ACTIONBLOCK && ACTIONBLOCK
    ACTIONBLOCK || ACTIONBLOCK
    ACTIONBLOCK | ACTIONBLOCK |
    ( ACTIONBLOCK )
ACTIONARGS →
    <> | EXPRESSIONLIST
EXPRESIONLIST →
    EXPRESSION | EXPRESSION, EXPRESSIONLIST
COMP →
    <> | COP INTEGER
COP →
    == | != | <= | < | > | >=
POLICYGROUP →
    Policygroup:ASSONAME { CIMPOLICY }:PRIORITY
ASSONAME →
    <> | NAME  ( NAME , NAME )
CIMTYPE →
    SHORT | USHORT | INTEGER | LINTEGER |
    REAL | LREAL | STRING | BOOL | CALENDAR
SIMPLEBOOLEANEXPRESSION →
    <a BOOLEANEXPRESSION where all the identifiers are limited to NAMES>
```

ANNEX A
(informative)

Change Log

Version	Date	Author	Description
1.0.0	2009-07-14		DMTF Standard Release

Bibliography

DMTF DSP0107, *CIM Event Model White Paper 2.1*, http://www.dmtf.org/standards/documents/CIM/DSP0107.pdf

DMTF DSP0108, *CIM Policy Model White Paper* 2.7, http://www.dmtf.org/standards/documents/CIM/DSP0108.pdf