



**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**

Copyright Notice

Copyright © 2009 Distributed Management Task Force, Inc. (DMTF). All rights reserved.

DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems management and interoperability. Members and non-members may reproduce DMTF specifications and documents, provided that correct attribution is given. As DMTF specifications may be revised from time to time, the particular version and release date should always be noted.

Implementation of certain elements of this standard or proposed standard may be subject to third party patent rights, including provisional patent rights (herein "patent rights"). DMTF makes no representations to users of the standard as to the existence of such rights, and is not responsible to recognize, disclose, or identify any or all such third party patent right, owners or claimants, nor for any incomplete or inaccurate identification or disclosure of such rights, owners or claimants. DMTF shall have no liability to any party, in any manner or circumstance, under any legal theory whatsoever, for failure to recognize, disclose, or identify any such third party patent rights, or for such party's reliance on the standard or incorporation thereof in its product, protocols or testing procedures. DMTF shall have no liability to any party implementing such standard, whether such implementation is foreseeable or not, nor to any patent owner or claimant, and shall have no liability or responsibility for costs or losses incurred if a standard is withdrawn or modified after publication, and shall be indemnified and held harmless by any party implementing the standard from any and all claims of infringement by a patent owner for such implementations.

For information about patents held by third-parties which have notified the DMTF that, in their opinion, such patent may relate to or impact implementations of DMTF standards, visit <http://www.dmtf.org/about/policies/disclosures.php>.

## CONTENTS

|    |                                       |    |
|----|---------------------------------------|----|
| 34 | Foreword .....                        | 7  |
| 35 | Introduction .....                    | 8  |
| 36 | 1 Scope .....                         | 9  |
| 37 | 2 Normative References.....           | 9  |
| 38 | 2.1 Approved References .....         | 9  |
| 39 | 2.2 Other References.....             | 9  |
| 40 | 3 Terms and Definitions .....         | 10 |
| 41 | 4 Symbols and Abbreviated Terms ..... | 11 |
| 42 | 5 CIM Policy Model .....              | 11 |
| 43 | 6 Usage Models .....                  | 12 |
| 44 | 6.1 Best Practice Checker .....       | 12 |
| 45 | 6.2 Routing in Networks.....          | 13 |
| 46 | 7 SPL Policy Rules.....               | 13 |
| 47 | 7.1 Policy String Components .....    | 15 |
| 48 | 8 SPL Policy Groups .....             | 21 |
| 49 | 8.1 Policy Group Components.....      | 21 |
| 50 | 8.2 Policy Group Example .....        | 24 |
| 51 | 9 Expressions.....                    | 26 |
| 52 | 9.1 Abs.....                          | 26 |
| 53 | 9.2 Logical And .....                 | 26 |
| 54 | 9.3 StartsWith .....                  | 26 |
| 55 | 9.4 Ceiling .....                     | 26 |
| 56 | 9.5 Concatenate.....                  | 27 |
| 57 | 9.6 Contains.....                     | 27 |
| 58 | 9.7 ContainsOnlyLettersOrDigits ..... | 27 |
| 59 | 9.8 ContainsOnlyDigits .....          | 27 |
| 60 | 9.9 ContainsOnlyLetters .....         | 27 |
| 61 | 9.10 Division .....                   | 28 |
| 62 | 9.11 EndsWith.....                    | 28 |
| 63 | 9.12 Equal .....                      | 28 |
| 64 | 9.13 Exp.....                         | 28 |
| 65 | 9.14 Floor .....                      | 28 |
| 66 | 9.15 GetDayOfMonth .....              | 29 |
| 67 | 9.16 GetDayOfWeek.....                | 29 |
| 68 | 9.17 GetDayOfWeekInMonth.....         | 29 |
| 69 | 9.18 GetDayOfYear .....               | 29 |
| 70 | 9.19 GetHour12 .....                  | 29 |
| 71 | 9.20 GetHour24 .....                  | 30 |
| 72 | 9.21 GetMillisecond .....             | 30 |
| 73 | 9.22 GetMinute .....                  | 30 |
| 74 | 9.23 GetMonth .....                   | 30 |
| 75 | 9.24 GetSecond .....                  | 30 |
| 76 | 9.25 GetWeekOfMonth .....             | 31 |
| 77 | 9.26 GetWeekOfYear.....               | 31 |
| 78 | 9.27 GetYear.....                     | 31 |
| 79 | 9.28 Greater .....                    | 31 |
| 80 | 9.29 Greater or Equal .....           | 31 |
| 81 | 9.30 IsWithin .....                   | 32 |
| 82 | 9.31 Less.....                        | 32 |
| 83 | 9.32 Less or Equal .....              | 32 |
| 84 | 9.33 Ln .....                         | 32 |

|     |       |                                                            |    |
|-----|-------|------------------------------------------------------------|----|
| 85  | 9.34  | Max .....                                                  | 33 |
| 86  | 9.35  | Min .....                                                  | 33 |
| 87  | 9.36  | Subtraction .....                                          | 33 |
| 88  | 9.37  | Not Equal .....                                            | 33 |
| 89  | 9.38  | Logical Not .....                                          | 33 |
| 90  | 9.39  | Logical Or .....                                           | 34 |
| 91  | 9.40  | Addition .....                                             | 34 |
| 92  | 9.41  | Power .....                                                | 34 |
| 93  | 9.42  | Product .....                                              | 34 |
| 94  | 9.43  | Mod .....                                                  | 34 |
| 95  | 9.44  | Round .....                                                | 35 |
| 96  | 9.45  | SquareRoot .....                                           | 35 |
| 97  | 9.46  | StringLength .....                                         | 35 |
| 98  | 9.47  | MatchesRegExp .....                                        | 35 |
| 99  | 9.48  | Substring Operations .....                                 | 35 |
| 100 | 9.49  | ToBoolean .....                                            | 38 |
| 101 | 9.50  | ToREAL32 .....                                             | 39 |
| 102 | 9.51  | ToSINT32 .....                                             | 39 |
| 103 | 9.52  | ToSINT16 .....                                             | 39 |
| 104 | 9.53  | ToSINT64 .....                                             | 39 |
| 105 | 9.54  | ToLower .....                                              | 40 |
| 106 | 9.55  | ToMilliseconds .....                                       | 40 |
| 107 | 9.56  | ToSINT8 .....                                              | 40 |
| 108 | 9.57  | ToString .....                                             | 40 |
| 109 | 9.58  | ToUINT32 .....                                             | 41 |
| 110 | 9.59  | ToUINT16 .....                                             | 41 |
| 111 | 9.60  | ToUINT64 .....                                             | 41 |
| 112 | 9.61  | ToUINT8 .....                                              | 41 |
| 113 | 9.62  | ToUpper .....                                              | 42 |
| 114 | 9.63  | Word .....                                                 | 42 |
| 115 | 9.64  | Logical XOR .....                                          | 42 |
| 116 | 9.65  | StringConstant .....                                       | 42 |
| 117 | 9.66  | LongConstant .....                                         | 42 |
| 118 | 9.67  | DoubleConstant .....                                       | 43 |
| 119 | 9.68  | DATETIMEConstant .....                                     | 43 |
| 120 | 9.69  | BooleanConstant .....                                      | 43 |
| 121 | 9.70  | Identifier .....                                           | 43 |
| 122 | 10    | Simple Boolean Condition .....                             | 44 |
| 123 | 11    | Collection Operations .....                                | 44 |
| 124 | 11.1  | Basic Collection .....                                     | 44 |
| 125 | 11.2  | Collect .....                                              | 44 |
| 126 | 11.3  | InCollection .....                                         | 46 |
| 127 | 11.4  | Union .....                                                | 47 |
| 128 | 11.5  | SubCollection .....                                        | 47 |
| 129 | 11.6  | EqCollections .....                                        | 47 |
| 130 | 11.7  | AnyInCollection .....                                      | 47 |
| 131 | 11.8  | AllInCollection .....                                      | 48 |
| 132 | 11.9  | ApplyToCollection .....                                    | 48 |
| 133 | 11.10 | Sum .....                                                  | 48 |
| 134 | 11.11 | MaxInCollection .....                                      | 48 |
| 135 | 11.12 | MinInCollection .....                                      | 49 |
| 136 | 11.13 | AvrgInCollection, MedianInCollection, sdInCollection ..... | 49 |
| 137 | 11.14 | CollectionSize .....                                       | 49 |
| 138 | 12    | Policy Example .....                                       | 49 |
| 139 | 13    | CIM-SPL Grammar .....                                      | 50 |

|     |                                        |    |
|-----|----------------------------------------|----|
| 140 | ANNEX A (informative) Change Log ..... | 54 |
| 141 | Bibliography .....                     | 55 |
| 142 |                                        |    |

## 143 **Figures**

|     |                                               |    |
|-----|-----------------------------------------------|----|
| 144 | Figure 1 – CIM Policy Information Model ..... | 12 |
| 145 | Figure 2 – Fabric Instance Diagram.....       | 24 |
| 146 | Figure 3 – PolicyGroup Schema .....           | 25 |
| 147 | Figure 4 – Example of CIM Associations .....  | 46 |
| 148 |                                               |    |

## 149 **Tables**

|     |                                      |    |
|-----|--------------------------------------|----|
| 150 | Table 1 – Numeric Operators.....     | 17 |
| 151 | Table 2 – Boolean Operators.....     | 17 |
| 152 | Table 3 – Relational Operators ..... | 17 |
| 153 | Table 4 – String Functions .....     | 18 |
| 154 | Table 5 – Numeric Functions .....    | 19 |
| 155 | Table 6 – Time Functions.....        | 19 |
| 156 |                                      |    |



158

## Foreword

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

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

## 163 Acknowledgments

164 The authors wish to acknowledge the following people.

165 Editor:

- 166 • Jorge Lobo – IBM

167 Participants from the DMTF Policy Working Group:

- 168 • Dakshi Agrawal – IBM
- 169 • Seraphin Calo – IBM
- 170 • Kang-Won Lee – IBM
- 171 • Jorge Lobo – IBM
- 172 • Andrea Westerinen (Cisco Systems, now at Microsoft)

173

174

## Introduction

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

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

182

# CIM Simplified Policy Language (CIM-SPL)

## 183 1 Scope

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

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

- 198 • signed and unsigned short, regular, and long integers
- 199 • float and long float
- 200 • string
- 201 • Boolean
- 202 • calendar

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

## 205 2 Normative References

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

### 209 2.1 Approved References

210 DMTF DSP0004, *CIM Infrastructure Specification 2.3*,  
211 [http://www.dmtf.org/standards/published\\_documents/DSP0004\\_2.3.pdf](http://www.dmtf.org/standards/published_documents/DSP0004_2.3.pdf)

### 212 2.2 Other References

213 ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards*,  
214 <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

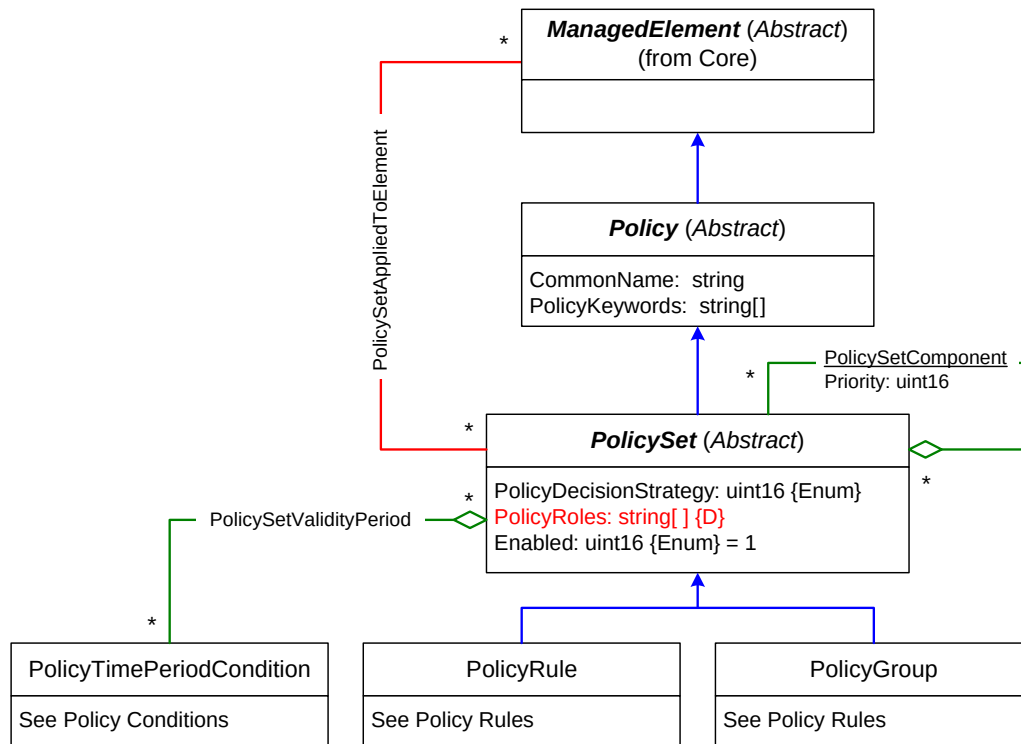
## 5 CIM Policy Model

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;
```

```

364 Declaration{
365     InstallDate="ManagedSystemElement.InstallDate";
366     Macro { Name = Age;
367         Type = Long;
368         Arguments Born:DATETIME;
369         Procedure = getYear(CurrentDate) - getYear(Born)
370     }
371 }
372 Policy {
373     Condition { 4 > Age(InstallDate) AND
374                 VendorEquipmentType == "switch"}
375     Decision { Upgrade (SKU) }
376 } :1
377 # End of Policy

```

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

- 379 1) **Import statement:** The Import statement in the example refers to a CIM class that is relevant to  
380 the policy string. In the remainder of the policy string, a policy rule is written as if an instance,  
381 called instance under evaluation, of this class is available for manipulation. The rule may be  
382 able to access other objects by traversing the references in associations where the instance  
383 under evaluation participates. An Import statement is required in each policy string. Section  
384 8.1.1 elaborates on the manner in which the object instance may be obtained.
- 385 2) **Strategy statement:** The Strategy statement indicates how many policy rules can be executed.  
386 In the example, this statement can be ignored because the example contains a single rule.
- 387 3) **Declaration section:** A Declaration section defines named constants and macros that can be  
388 used in the policy section of the policy string. In this way, the actual policy specification can be  
389 clearer and easier to understand. The Declaration section is optional.
- 390 4) **Policy section:** The Policy section contains the main body of the policy string, with a condition  
391 statement, a decision statement, and priority. Both the condition and the decision can refer to  
392 the named constants and use the macros defined in the declaration section. The priority helps  
393 to determine what policy rule to execute in case multiple rules are triggered.

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

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

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

<sup>1</sup> Lines in a policy string are delimited by line separators that include LF(u000A), CR(u000D), NEL(u0085), FF(u000C), LS(u0208), and PS(u0209).

<sup>2</sup> 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)
```

```

449 }
450 The names of constants and macros shall be different from the policy name.
451 Constant Definition
452 Format:
453     <constant name> = <constant value>;
454 Macro Definition
455 Format:
456 Macro {
457     Name=<string that is the macro's name>; (Required)
458     Type=type; (Required)
459     Argument= name1:type1[,name2:type2]*; (Optional)
460     Procedure=<expression> (Required)
461 }

```

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

### 467 7.1.3 Policy Section

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

```

470 Format:
471 Policy {
472     Declaration {
473         <List of constant definition> (Optional)
474         <List of macro definitions> (Optional)
475     }
476     Condition { (Optional)
477         <If Condition>
478     }
479     Decision { (Required)
480         <Then Decision>
481     }
482 }: Priority

```

#### 483 7.1.3.1 Declaration

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

#### 487 7.1.3.2 Condition

488 The Condition section is an optional subsection of the Policy section. This is the "if" condition part of the  
489 policy rule. If the Condition section is omitted, the policy is considered always active (that is, an  
490 "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),<br>– 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)  |

511

**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"<br>leftSubstring("Mississippi", 4, "RightToLeft") ; returns "Mississ"         |
| rightSubstring              | rightSubstring("Mississippi", 4, "LeftToRight") ; returns "issippi"<br>rightSubstring("Mississippi", 4, "RightToLeft") ; returns "ippi"       |
| middleSubstring             | middleSubstring("Mississippi", 4, 5, "LeftToRight") ; returns "issip"<br>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")                                                                                                         |

512

513

**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)          |

514

**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)         |

**515 7.1.3.4 Decision**

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

522 **Format:**

523 **Decision** {  
 524     <action block>  
 525 }

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

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

528     A single PolicyAction evaluation without arguments.

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

530     A single PolicyAction evaluation with at least one argument. The argument expression types shall  
 531     match the argument types of the concrete PolicyAction being evaluated. If this is not the case, a  
 532     CIM\_ERROR\_POLICY\_EXECUTION may be thrown. <cop> is one of the comparison operators ==,  
 533     != , <, <=, >, >=, and <constant> is a numeric constant. The <cop> <constant> pair is optional.

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

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

539 **<action block1> -> <action block2>**

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

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

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

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

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

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

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

559 **( <action block> )**

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

563 A PolicyAction name can take one of the following forms:

- 564 1) Set.Identifier, with the identifier referring to a managed element. The argument names shall be  
565 properties of the element, and the effect is to set all the properties passed as arguments to the  
566 values returned by the expressions.
- 567 2) Identifier.MethodName, with the identifier referring to a managed element. The method name  
568 shall be a method of the managed element to which the identifier refers. The arguments shall  
569 match the signature of the method. The `<cop> <constant>` pair can only appear in this case.

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

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

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

## 582 7.1.4 Strategy and Priorities

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

## 585 8 SPL Policy Groups

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

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

### 593 8.1 Policy Group Components

594 A policy group has the following format:

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

```
597 Strategy [Execute_All_Applicable | Execute_First_Applicable] ; (Required)
```

```
598 Declaration {
```

```
599     <List of constant definition> (Optional)
```

```
600     <List of macro definitions> (Optional)
```

```
601 }
```

```
602 Policy { ... } : Priority; (Optional)
```

```
603 Policy { ... } : Priority; (Optional)
```

```

604 Policy { ... } : Priority; (Priority is required)
605 ...
606 PolicyGroup:[Association Name(Property1,Property2)] { ... } : Priority;
607 (Optional)
608 PolicyGroup:[Association Name(Property1,Property2)] { ... } : Priority;
609 (Optional)
610 PolicyGroup:[Association Name(Property1,Property2)] { ... } : Priority;
611 (Optional)
612 ...

```

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

### 616 8.1.1 Suggested Mechanisms of Invocation: Import Statements and Indications

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

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

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

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

638 In another situation, policy evaluation may be triggered by an event in the system. This mechanism is  
 639 often designated as *unsolicited policy evaluation*. For an unsolicited evaluation of a policy, the *target set*  
 640 for evaluation can be provided implicitly by the instances of CIM\_InstIndication subclasses that are  
 641 *consistent* with the Import statement of the policy. The SourceInstance parameter of a CIM\_InstIndication  
 642 instance points to a managed element that was the source of the CIM\_InstIndication. A  
 643 CIM\_InstIndication instance is consistent with the Import statement of a policy if the class of the managed  
 644 element to which the SourceInstance parameter of the CIM\_InstIndication points is a subclass of the  
 645 class in the Import statement. CIM-SPL does not define which compatible instances of CIM\_InstIndication  
 646 initiate an unsolicited evaluation of a policy. For example, when a policy is installed or is activated, a  
 647 policy server may require that an instance of CIM\_IndicationFilter be specified. Such a requirement would  
 648 be sufficient for the policy server to generate a CIM\_IndicationSubscription to CIM\_InstIndications that  
 649 trigger evaluations of the policy. Subsequently, when the policy server receives a CIM\_InstIndication  
 650 instance, the policy server evaluates the corresponding policy on the managed element to which the  
 651 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  $P_1, P_2, P_3, P_4, \dots$  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 evaluation 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  $P_i$ s 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.

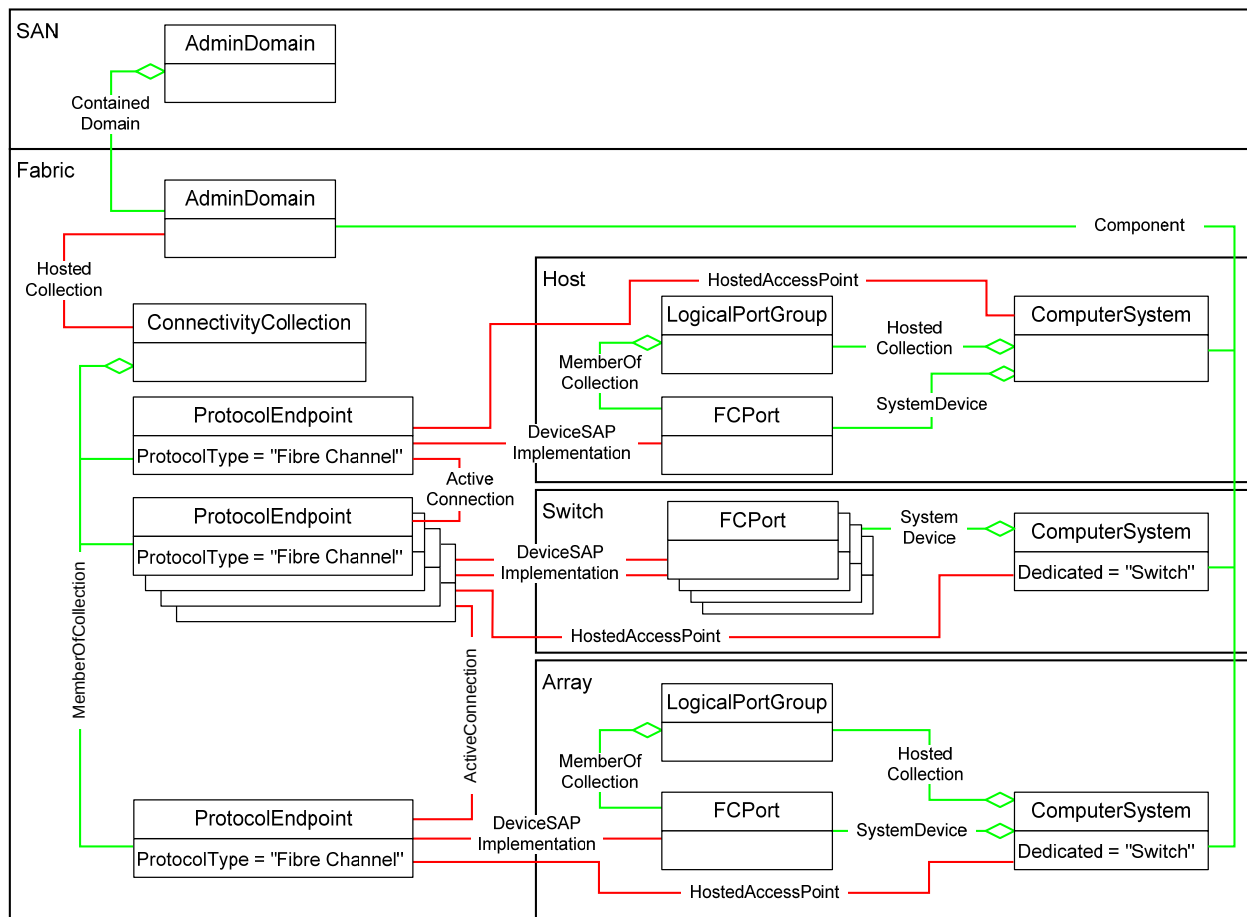
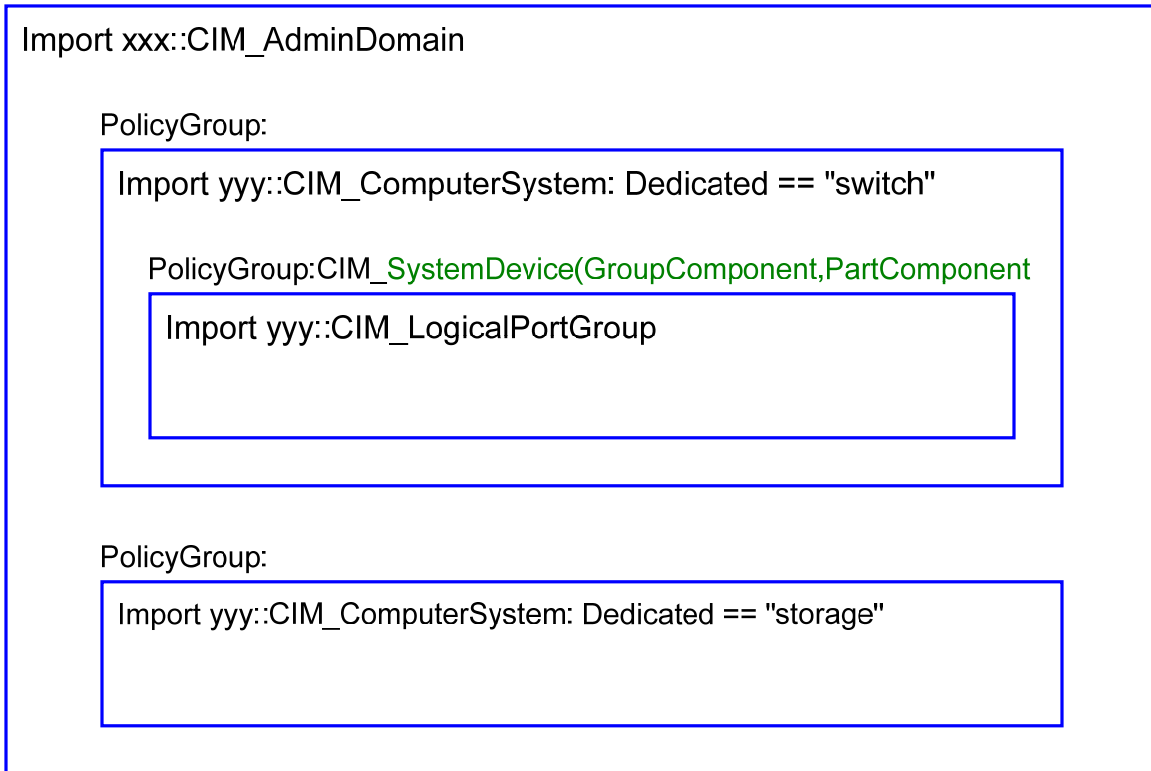


Figure 2 – Fabric Instance Diagram

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

<Name="System1"> ← Object associated with the initial call



716

717

**Figure 3 – PolicyGroup Schema**

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

724 When specifying a policy group, certain policies are applicable only to a particular component of the  
 725 managed element. Or, certain policies within a policy group may be applicable only to managed elements  
 726 that are associated in a specific manner with the managed element. Such policies can be conveniently  
 727 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)

### 1001 9.39 Logical Or

1002 Shall return a Boolean value that corresponds to the logical OR operation on the required Boolean  
1003 arguments

1004 (*<expression> || <expression>*)

1005 **Examples:**

1006 (a || b)

1007 ((stringLength(alpha) < 5) || (5+b))

### 1008 9.40 Addition

1009 Shall return the sum of the required numeric arguments. The data type of the result value shall follow the  
1010 standard JAVA casting conventions.

1011 (*<expression> + <expression>*)

1012 **Examples:**

1013 (a + b)

1014 (stringLength(alpha) + 5)

### 1015 9.41 Power

1016 Shall return the value of the first required numeric argument raised to the power of the second required  
1017 numeric argument.

1018 **power**(*<expression1>*, *<expression2>*)

1019 where *<expression1>* is the value raised to the power of *<expression2>*

1020 **Examples:**

1021 power(nbr1, nbr2)

1022 power(2, 4)

### 1023 9.42 Product

1024 Shall return the product of the required numeric arguments. The data type of the result value shall follow  
1025 the standard JAVA casting conventions.

1026 (*<expression> \* <expression>*)

1027 **Examples:**

1028 (a \* b)

1029 (stringLength(alpha) \* c)

### 1030 9.43 Mod

1031 Shall return the remainder from an operation of dividing the first required numeric argument by the  
1032 second required numeric argument

1033 **mod**(*<expression1>*, *<expression2>*)

1034 where *<expression1>* is the value divided by *<expression2>*

1035 **Examples:**

1036 mod(nbr1, nbr2)

1037 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.

### 1073 9.48.1 Substring

1074 The substring operator takes either two or three arguments. The first and second arguments of this  
 1075 operator are required while the third argument is optional. The first argument of this operator shall be a  
 1076 string argument, while the second and third argument shall be integer argument.

1077 This operator shall return the substring of the first string argument, starting at the position indicated by the  
 1078 second numeric argument and going to the end of the string or the position indicated by the third numeric  
 1079 argument - 1. The position of a character is determined as follows: The first character is at position 0, the  
 1080 second character is at position 1, and so on.

1081 The second numeric argument shall be greater than or equal to 0. The third numeric argument shall be  
 1082 greater than the second numeric argument if the third argument is present. If the starting position given by  
 1083 the second numeric argument is greater than the length of the string, an empty string shall be returned. If  
 1084 the third numeric position is not present, the string starting at the second numeric position until the end of  
 1085 the string shall be returned.

1086 **substring(<expression1>, <expression2>, [<expression3>])**

1087 where <expression1> is a string argument, and <expression2> and <expression3> are integers  
 1088 (UINT32) argument.

1089 **Examples:**

1090 substring("Robert Hancock", 2, 8) returns "bert H".

### 1091 9.48.2 LeftSubstring

1092 The LeftSubstring operator returns a prefix of a given string argument by taking three arguments. How to  
 1093 compute the prefix is determined by the arguments. The first argument shall be a string and it indicates  
 1094 the given string, the second argument shall be either an integer or a string indicating an offset, and the  
 1095 third argument shall be a string indicating a direction and is either "LeftToRight" or "RightToLeft".

1096 When the offset is given by a number, the prefix is determined by counting the character position by the  
 1097 offset from either left to right (from the beginning of the string) or from right to left (from the end of the  
 1098 string). In particular, if the direction is "LeftToRight", the offset indicates the number of characters to return  
 1099 from the beginning of the string. If the direction is "RightToLeft", the offset indicates the number of  
 1100 characters to skip from the end of the string.<sup>3</sup> For example, leftSubstring("Mississippi", 4, "LeftToRight")  
 1101 returns "Miss", and leftSubstring("Mississippi", 4, "RightToLeft") returns "Mississ".

1102 When the offset is given by a string, the prefix is determined by searching for the offset string in the  
 1103 original string in the direction specified by the third parameter. The returned substring consists of the  
 1104 characters on the left side of the offset string. For example, leftSubstring("Mississippi", "ss",  
 1105 "LeftToRight") returns "Mi", and leftSubstring("Mississippi", "ss", "RightToLeft") returns "Missi".

1106 **leftSubstring(<expression1>, <expression2>, <expression3>)**

1107 where <expression1> is a string, <expression2> is an integer, and <expression3> is a string  
 1108 constant that indicates either "LeftToRight" or "RightToLeft"

1109 **leftSubstring(<expression1>, <expression2>, <expression3>)**

1110 where <expression1> and <expression2> are strings, and <expression3> is a string constant  
 1111 that indicates either "LeftToRight" or "RightToLeft"

---

<sup>3</sup> If the offset value is a negative number, the entire string is returned in either case.

1112       **Examples:**

1113           leftSubstring(StateSymbolAndZip, 2, "LeftToRight") // to get the state symbol  
 1114           leftSubstring(FirstAndLastName, " ", "LeftToRight") // to get the first name

1115       **9.48.3 RightSubstring**

1116       The RightSubstring operator returns a suffix of a given string argument by taking three arguments. How to  
 1117       compute the suffix is determined by the arguments. The first argument shall be a string and it indicates  
 1118       the given string, the second argument shall be either an integer or a string indicating an offset, and the  
 1119       third argument shall be a string indicating a direction and is either "LeftToRight" or "RightToLeft".

1120       When the offset is given by a number, the suffix is determined by simply counting the character position  
 1121       by the offset from either left to right (from the beginning of the string) or from right to left (from the end of  
 1122       the string). In particular, if the direction "RightToLeft", the offset indicates the number of characters to  
 1123       return as a suffix. If the direction is "LeftToRight", the offset indicates the number of characters to skip  
 1124       from the beginning of the string. For example, rightSubstring("Mississippi", 4, "LeftToRight") returns  
 1125       "issippi", and rightSubstring("Mississippi", 4, "RightToLeft") returns "ippi".

1126       When the offset is given by a string, the suffix is determined by searching for the offset string in the  
 1127       original string in the direction specified by the third parameter. The returned substring consists of the  
 1128       characters on the right side of the offset string. For example, rightSubstring("Mississippi", "ss",  
 1129       "LeftToRight") returns "issippi", and rightSubstring("Mississippi", "ss", "RightToLeft") returns "ippi".

1130       **rightSubstring(<expression1>, <expression2>, <expression3>)**

1131           where <expression1> is a string, <expression2> is an integer, and <expression3> is a string  
 1132           constant that indicates either "LeftToRight" or "RightToLeft"

1133       **rightSubstring(<expression1>, <expression2>, <expression3>)**

1134           where <expression1> and <expression2> are strings, and <expression3> is a string constant  
 1135           that indicates either "LeftToRight" or "RightToLeft"

1136       **Examples:**

1137           rightSubstring(StateSymbolAndZip, 5, RightToLeft) // to get the zip code  
 1138           rightSubstring(FirstAndLastName, " ", "LeftToRight") // to get the last name

1139       **9.48.4 MiddleSubstring**

1140       The MiddleSubstring operator returns a middle portion of a given string using various arguments as filters.  
 1141       How to compute the suffix is determined by the arguments. MiddleSubstring takes four arguments:  
 1142       original string, first offset, second offset, and direction string. The first and second offsets shall be  
 1143       specified either by a number or a string. The direction string can be either "LeftToRight" or "RightToLeft".  
 1144       The meaning of the first offset is similar to that in the rightSubstring: it indicates where the resulting  
 1145       substring starts scanning, either from the left or from the right based on the direction string. The meaning  
 1146       of the second offset is as follows: if it is a number, it simply indicates the number of characters to return; if  
 1147       it is a string, it specifies where the substring should end. For example:

1148       **middleSubstring(<expression1>, <expression2>, <expression3>, <expression4>)**

1149           where <expression1> is a string, <expression2> and <expression3> are integers, and  
 1150           <expression4> is a string constant that indicates either "LeftToRight" or "RightToLeft"

1151       **middleSubstring(<expression1>, <expression2>, <expression3>, <expression4>)**

1152           where <expression1> is a string, <expression2> is an integer, <expression3> is a string, and  
 1153           <expression4> is a string constant that indicates either "LeftToRight" or "RightToLeft"

1154 **middleSubstring**(<expression1>, <expression2>, <expression3>, <expression4>)  
 1155 where <expression1> is a string, <expression2> is a string, <expression3> is an integer, and  
 1156 <expression4> is a string constant that indicates either "LeftToRight" or "RightToLeft"

1157 **middleSubstring**(<expression1>, <expression2>, <expression3>, <expression4>)  
 1158 where <expression1> is a string, <expression2> and <expression3> are strings, and  
 1159 <expression4> is a string constant that indicates either "LeftToRight" or "RightToLeft"

1160 **Examples:**

1161 middleSubstring("Mississippi", 4, 5, "LeftToRight") = "issip"  
 1162 middleSubstring("Mississippi", 4, 5, "RightToLeft") = "ippi"  
 1163 middleSubstring("Mississippi", "ss", 5, "LeftToRight") = "issip"  
 1164 middleSubstring("Mississippi", "ss", 5, "RightToLeft") = "ippi"  
 1165 middleSubstring("Mississippi", 4, "ss", "LeftToRight") = "i"  
 1166 middleSubstring("Mississippi", 4, "ss", "RightToLeft") = ""  
 1167 middleSubstring("Mississippi", "ss", "ip", "LeftToRight") = "iss"  
 1168 middleSubstring("Mississippi", "ss", "ip", "RightToLeft") = "Missi"

1169 **9.48.5 ReplaceSubstring**

1170 The ReplaceSubstring operator shall take two or three string arguments. It replaces one substring with  
 1171 another substring in a given string. The first argument specifies the given string, the second argument  
 1172 specifies a from-string, and the third argument specifies a to-string. Note that it is a purely functional form  
 1173 with no side-effect — that is, none of the string arguments are modified.

1174 **replaceSubstring**(<expression1>, <expression2>, [<expression3>])  
 1175 where <expression1>, <expression2>, and <expression3> are strings

1176 **Example:**

1177 replaceSubstring(Name, "Jim", "James")

1178 **9.49 ToBoolean**

1179 Shall return Boolean TRUE if the required argument is a string argument that equates to "true" ignoring  
 1180 the case; shall return Boolean TRUE if the required argument is a numeric argument that evaluates to a  
 1181 non-zero; otherwise, it shall return FALSE.

1182 **Note:** toBoolean(1) returns TRUE. toBoolean("1") returns FALSE, because this is passing in a string.

1183 **toBoolean**(<expression>)  
 1184 where <expression> is the value to be converted

1185 **Examples:**

1186 toBoolean("true")  
 1187 toBoolean(1)

## 1188 9.50 ToREAL32

1189 Shall return a Real32 value corresponding to the one required argument. The required argument shall be  
1190 either of type string or numeric. The conversion shall be done according to the Java conventions.

1191 **ToREAL32(<expression>)**

1192 where <expression> is the string

1193 **Examples:**

1194 ToREAL32("25.")

1195 ToREAL32(alpha)

## 1196 9.51 ToSINT32

1197 Shall return a SINT32 value corresponding to the one required argument. The required argument shall be  
1198 either of type string or numeric. The conversion shall be done according to the Java conventions.

1199 **ToSINT32(<expression>)**

1200 where <expression> is the string

1201 **Examples:**

1202 ToSINT32("257")

1203 ToSINT32(alpha)

## 1204 9.52 ToSINT16

1205 Shall return a SINT16 value corresponding to the one required argument. The required argument shall be  
1206 either of type string or numeric. The conversion shall be done according to the Java conventions.

1207 **ToSINT16(<expression>)**

1208 where <expression> is the value to convert

1209 **Examples:**

1210 ToSINT16("25")

1211 ToSINT16(alpha)

## 1212 9.53 ToSINT64

1213 Shall return a SINT64 value corresponding to the one required argument. The required argument shall be  
1214 either of type string or numeric. The conversion shall be done according to the Java conventions.

1215 **ToSINT64(<expression>)**

1216 where <expression> is the value to convert

1217 **Examples:**

1218 ToSINT64("2556")

1219 ToSINT64(255)

1220 ToSINT64(alpha)

## 1221 9.54 ToLower

1222 Shall return the required string argument converted into lowercase

1223 **toLowerCase(<expression>)**

1224 where <expression> is the string

1225 **Examples:**

1226 toLowerCase(alpha)

1227 toLowerCase("Hello World")

## 1228 9.55 ToMilliseconds

1229 Shall return the number of milliseconds since the standard base time known as "the epoch," namely  
1230 January 1, 1970, 00:00:00 GMT, corresponding to the required DATETIME argument.

1231 **toMilliseconds(<expression>)**

1232 where <expression> is the DATETIME value

1233 **Example:**

1234 toMilliseconds(aDate)

## 1235 9.56 ToSINT8

1236 Shall return a SINT18 value corresponding to the one required argument. The required argument shall be  
1237 either of type string or numeric. The conversion shall be done according to the Java conventions.

1238 **ToSINT8(<expression>)**

1239 where <expression> is the value to convert

1240 **Examples:**

1241 ToSINT8("25")

1242 ToSINT8(25)

1243 ToSINT8(alpha)

## 1244 9.57 ToString

1245 Shall return a String value corresponding to the one required argument. The required argument shall be  
1246 either of type Boolean or numeric. The conversion shall be done according to the Java conventions.

1247 Converts the numeric and the Boolean arguments into a string value

1248 **toString(<expression>)**

1249 where <expression> is the value to convert

1250 **Examples:**

1251 toString(nbr1)

1252 toString(1)

1253 toString(true)

## 1254 9.58 ToUINT32

1255 Shall return a UINT32 value corresponding to the one required argument. The required argument shall be  
1256 either of type string or numeric. The conversion shall be done according to the Java conventions.

1257 **ToUINT32(<expression>)**

1258 where <expression> is the value to convert

1259 **Examples:**

1260 ToUINT32("2556")

1261 ToUINT32(2550)

1262 ToUINT32(alpha)

## 1263 9.59 ToUINT16

1264 Shall return a UINT16 value corresponding to the one required argument. The required argument shall be  
1265 either of type string or numeric. The conversion shall be done according to the Java conventions.

1266 **ToUINT16(<expression>)**

1267 where <expression> is the value to convert

1268 **Examples:**

1269 ToUINT16("2556")

1270 ToUINT16(2550)

1271 ToUINT16(alpha)

## 1272 9.60 ToUINT64

1273 Shall return a UINT64 value corresponding to the one required argument. The required argument shall be  
1274 either of type string or numeric. The conversion shall be done according to the Java conventions.

1275 **ToUINT64(<expression>)**

1276 where <expression> is the value to convert

1277 **Examples:**

1278 ToUINT64("2556")

1279 ToUINT64(2550)

## 1280 9.61 ToUINT8

1281 Shall return a UINT8 value corresponding to the one required argument. The required argument shall be  
1282 either of type string or numeric. The conversion shall be done according to the Java conventions.

1283 **ToUINT8(<expression>)**

1284 where <expression> is the value to convert

1285 **Examples:**

1286 ToUINT8("25")

1287 ToUINT8(25)

## 1288 9.62 ToUpper

1289 Shall return an uppercase version of the required string argument.

1290 **toUpper**( *<expression>* )

1291 where *<expression>* is the string

1292 **Examples:**

1293 toUpper(alpha)

1294 toUpper("hello world")

## 1295 9.63 Word

1296 This operator shall take three arguments. The first two arguments shall be of type string, and the third  
 1297 argument shall be of type number. This operator shall extract *n* words from the first string argument where  
 1298 the third argument specifies the number *n*. Words are defined as text between the separator substring  
 1299 given by the second argument.

1300 **word**(*<expression1>*, *<expression2>*, *<expression3>*)

1301 where *<expression1>* is the given string, *<expression2>* is the separator substring, and  
 1302 *<expression3>* is the number

1303 **Example:**

1304 Word(alpha, " ", 3)

## 1305 9.64 Logical XOR

1306 Shall return a Boolean value that corresponds to the logical XOR operation on the bit representation of  
 1307 the two required numeric arguments

1308 (*<expression>* ^ *<expression>*)

1309 **Examples:**

1310 (a ^ b)

1311 (Netmask ^ (IP))

## 1312 9.65 StringConstant

1313 Values inside double quotes are converted to StringConstants.

1314 **Example:**

1315 alpha = "22"

## 1316 9.66 LongConstant

1317 UINT32 (unquoted) values that do not contain decimals are converted to UINT64 constants.

1318 **Example:**

1319 alpha == 22

## 1320 9.67 DoubleConstant

1321 Numeric (unquoted) values that contain decimal points are converted to Real64 constants. This includes  
1322 numeric values ending in a decimal point (for example, 22).

1323 **Example:**

1324       alpha == 22.25

## 1325 9.68 DATETIMEConstant

1326 Unquoted values in the format "yyyy-mm-ddThh:mm:ss TZ=javaTimezoneID" are interpreted as a  
1327 DATETIMEConstant following the XML standard semantics.

1328 **Example:**

1329       alpha > 2004-01-29T09:40:00 TZ=America/Chicago

## 1330 9.69 BooleanConstant

1331 Unquoted strings 'true' or 'false' inside the clauses are converted to BooleanConstants in the resulting  
1332 XML.

1333 **Example:**

1334       alpha == true

## 1335 9.70 Identifier

1336 Identifier can be either simple or multi-level. A **simple identifier** shall be any of the following values:

- 1337       • Name of a named constant. It evaluates to the value of the named constant as defined in the  
1338       declaration sections.
- 1339       • Name of a named macro. It evaluates to the value of the named macro as defined in the  
1340       declaration sections.
- 1341       • <classname.propertyname>, where classname is the class name of the Import statement or any  
1342       super class of that class, and propertyname is the name of a property of the classname. The  
1343       prefix "classname." is optional. It is required only to disambiguate the name of a property if the  
1344       same name is used in any super class and it is not overridden. It evaluates to the value of the  
1345       property of the class instance under consideration.
- 1346       • Any of the preceding three values followed by an index enclosed in square brackets. In this  
1347       case, the type of the named constant, macro, or property should be an array of an intrinsic CIM  
1348       data type or CIM references. The index can only be an expression that evaluates to an integer  
1349       (UINT32) value. The first index is always 0. If any of these conditions is not true, the policy  
1350       parser returns an error; otherwise, the expression evaluates to the value of the data in the  
1351       position indicated by the index in the array identified by the named constant, macro, or property.
- 1352       • A qualifier that is an expression that evaluates to a reference of an instance of a CIM class. This  
1353       can be the reserved word *Self* that refers to the object instance under consideration or the  
1354       member of a collect operator that returns collections of references (see section 11.2).

1355 A **multi-level identifier** has the form <qualifier.simpleIdentifier>, where simpleIdentifier is a simple  
1356 identifier that is not a qualifier.

1357 A simple identifier that is not prefixed by a qualifier shall be evaluated under the scope of the managed  
1358 element that is made available to the rule based on the Import statement. If the qualifier appears, the  
1359 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.

1397       **Examples:**

1398           collect(Self, Realizes, PhysicalElement, LogicalDevice, CIM\_LogicalDevice,  
1399           TotalPowerOnHours > 5)

1400           collect(Self , Realizes, PhysicalElement, LogicalDevice, CIM\_LogicalDevice,  
1401           TotalPowerOnHours, TotalPowerOnHours > 5)

1402 Self references an instance of the CIM\_PhysicalElement class. To traverse multiple associations, collect  
1403 operators can be nested, as in the following example:

1404 **collect**(  
1405     **collect**(Self , Realizes,  
1406         PhysicalElement, LogicalDevice, CIM\_LogicalDevice,  
1407         true)[0],  
1408     DeviceServiceImplementation,  
1409     LogicalDevice,  
1410     CIM\_Service,  
1411     True)[1].Started

1412 Starting in a `PhysicalElement` (see Figure 4), the `Realizes` association is traversed to get a collection of  
 1413 `LogicalDevice` elements. Using one of these logical devices as an anchor (the first one in the collection),  
 1414 the `DeviceServiceImplementation` association is traversed to get a collection of `Service` elements. The  
 1415 expression takes the second element in this collection ([1]), and the value of the `Started` property is  
 1416 returned.

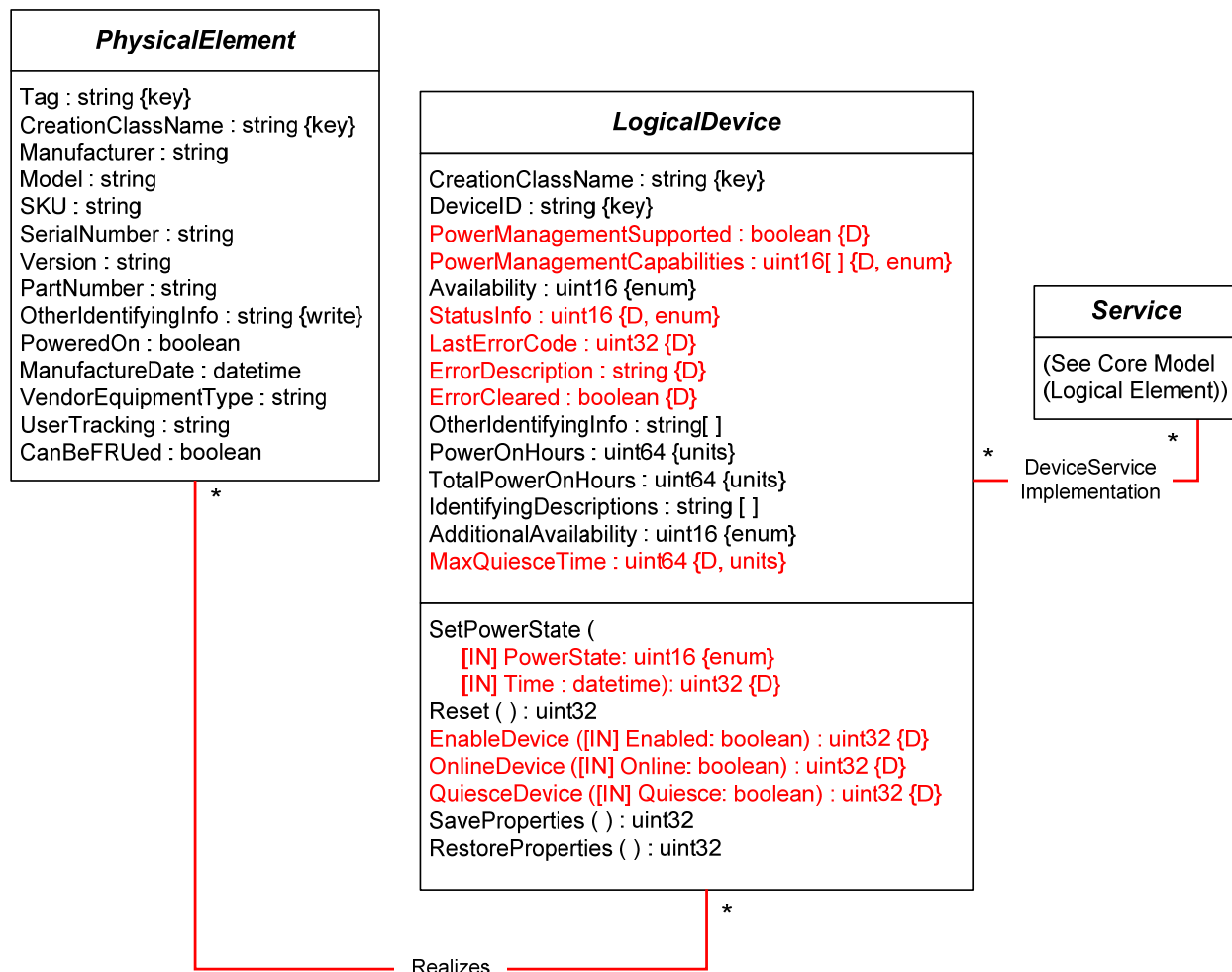


Figure 4 – Example of CIM Associations

### 11.3 InCollection

Checks whether an object is a member of a collection

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

Shall return TRUE if the value returned by <expression> appears in <collection>. The type of the <expression> shall be 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))

## 1459 11.8 AllInCollection

1460 Checks whether all objects in a collection have a given property

1461 **allInCollection**(*<expression>* *<op>* *<collection>*)

1462 where *<op>* shall be either a Boolean or relational operator. If *<op>* is a Boolean operator,  
1463 *<expression>* shall be Boolean. If *<op>* is relational, *<expression>* shall be of a type compatible with  
1464 the operator. The operation shall return TRUE if for every object *<oj>* in *<collection>*, *<expression>*  
1465 *<op>* *<oj>* returns TRUE.

1466 **Example:**

1467 anyInCollection(240 > collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice,  
1468 CIM\_LogicalDevice, TotalPowerOnHours, TotalPowerOnHours > 5))

## 1469 11.9 ApplyToCollection

1470 Applies an arithmetic operation to each element in a collection and shall return a numeric collection

1471 **applyToCollection**(*<expression>* *<op>* *<collection>*)

1472 where *<op>* shall be a binary numeric operator and *<expression>* is a numeric expression. The  
1473 operation shall return a collection similar to *<collection>* but in which every object *<oj>* in  
1474 *<collection>* is replaced by the result of the expression *<expression>* *<op>* *<oj>*.

1475 **Example:**

1476 applyToCollection(1024 + collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice,  
1477 CIM\_LogicalDevice, TotalPowerOnHours, TotalPowerOnHours > 5))

## 1478 11.10 Sum

1479 Shall return the sum of a collection of numeric CIM data elements. The *<collection>* shall be a collection  
1480 of numeric values.

1481 **sum**(*<collection>*)

1482 **Example:**

1483 sum(collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice, CIM\_LogicalDevice,  
1484 TotalPowerOnHours, TotalPowerOnHours > 5))

## 1485 11.11 MaxInCollection

1486 Shall return the maximum object from a collection of totally ordered CIM data objects

1487 **maxInCollection**(*<collection>*)

1488 **Example:**

1489 maxInCollection(collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice,  
1490 CIM\_LogicalDevice, TotalPowerOnHours, true))

1491 Strings are ordered lexicographically based on UTF-8.

## 1492 11.12 MinInCollection

1493 Shall return the smallest object from a collection of totally ordered CIM data objects

1494 **minInCollection**(<collection>)

1495 **Example:**

1496 minInCollection(collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice,  
1497 CIM\_LogicalDevice, TotalPowerOnHours, true))

## 1498 11.13 AvgInCollection, MedianInCollection, sdInCollection

1499 Shall return the average/median/standard deviation in a double from a collection of numeric CIM data  
1500 objects. The <collection> shall be a collection of numeric values.

1501 **avgInCollection**(<collection>) / **medianInCollection**(<collection>) / **sdInCollection**(<collection>)

1502 **Example:**

1503 avgInCollection(collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice,  
1504 CIM\_LogicalDevice, TotalPowerOnHours, true))

## 1505 11.14 CollectionSize

1506 Shall return the size of a collection in a UINT32

1507 **collectionSize**(<collection>)

1508 **Example:**

1509 collectionSize(collect(PhysicalElement.Self , Realizes, PhysicalElement, LogicalDevice,  
1510 CIM\_LogicalDevice, TotalPowerOnHours, true))

## 1511 12 Policy Example

1512 The following example shows a policy that is invoked when a file system is 85 percent full. The policy  
1513 expands the storage pool by 25 percent.

1514 **Import** CIM\_X\_XX\_XXXX::CIM\_LocalFileSystem;

1515 Strategy Execute\_All\_Applicable;

1516 **Policy** {

1517 **Declaration** { */\* Macros to traverse HostedService associations to get \*/*

1518 */\* to FileSystemConfigurationService for ModifyFile and \*/*

1519 */\* StorageConfigurationService for CreateOrModify... \*/*

1520 computer\_system = collect(Self, CIM\_HostedFileSystem,  
1521 PartComponent, GroupComponent, null, true)[0];

1522 storage\_config\_service =  
1523 collect(computer\_system, CIM\_HostedService, Antecedent,  
1524 Dependent, CIM\_StorageConfigurationService,  
1525 true)[0];

1526 logical\_disk = collect(Self, CIM\_ResidesOnExtent,  
1527 Dependent, Antecedent, null, true)[0];

1528 storage\_pool = collect(logical\_disk, CIM\_AllocatedFromStoragePool,

```

1529             Dependent, Antecedent, null, true)[0];
1530     fs_goal = collect(Self, CIM_ElementSettingData, ManagedElement,
1531         SettingData, CIM_FileSystemSetting true)[0];
1532 }
1533 Condition {
1534     (AvailableSpace / FileSystemSize) < 0.25
1535 }
1536 Decision {
1537     storage_conf_service.CreateOrModifyElementFromStoragePool("LogicalDisk",
1538         /* ElementType Volume */
1539         8, job, fs_goal,
1540         1.25 * FileSystemSize,
1541         storage_pool,
1542         logical_disk)
1543     // CreateOrModifyElementFromStoragePool defined in pp. 1024
1544     // of SMI-S 1.1.0 SNIA Standard document
1545     // ElementType value list can be found in pp. 1116
1546     | DoSomethingOnFailure()
1547 }
1548 }:1;

```

### 1549 13 CIM-SPL Grammar

1550 In the following grammar, non-terminal symbols are represented by sequences of uppercase letters in  
 1551 boldface. Alternatives in the production rules are represented by "|", except for the use of "||" in Boolean  
 1552 expressions and "|||" and "|" in action blocks. All blanks but one are ignored in the rules. Blanks do not  
 1553 appear in any of the terminal symbols.

```

1554 CIMPOLICY →
1555     Import IMPORTSTATEMENT ;
1556     Strategy STRATEGYSTATEMENT ;
1557     DECLARATIONSTATEMENT
1558     POLICYSTATEMENTS
1559 IMPORTSTATEMENT →
1560     MOFFILENAME :: CLASSNAME OPTIONALBOOLEANCONDITION
1561 MOFFILENAME →
1562     cim_vMAJOR_MINOR_RELEASE_TYPE_FILENAME
1563 MAJOR →
1564     DECIMALNUMBER
1565 MINOR →
1566     DECIMALNUMBER
1567 RELEASE →
1568     DECIMALNUMBER
1569 TYPE →
1570     preliminary | final
1571 FILENAME →

```

```

1572     <a MOF file name without the extension>
1573 CLASSNAME →
1574     <the name of a CIM class name defined in the MOF file>
1575 OPTIONALBOOLEANCONDITION →
1576     <> | SIMPLEBOOLEANEXPRESSION
1577 STRATEGYSTATEMENT →
1578     Execute_All_Applicable | Execute_First_Applicable
1579 DECLARATIONSTATEMENT →
1580     <> | Declaration { DECLARATIONS }
1581 DECLARATIONS →
1582     CONSTANTDECLARATION MACRODECLARATION
1583 CONSTANTDECLARATION →
1584     <> | NAME = EXPRESSION ; CONSTANTDECLARATION
1585 MACRODECLARATION →
1586     <> | Macro { MACRO } MACRODECLARATION
1587 MACRO →
1588     Name = NAME ; type = CIMTYPE ; ARGUMENTS procedure = EXPRESSION
1589 ARGUMENTS →
1590     <> | argument = NAME : CIMTYPE MOREARGUMENTS ;
1591 MOREARGUMENTS →
1592     <> | , NAME : CIMTYPE MOREARGUMENTS
1593 POLICYSTATEMENTS →
1594     POLICY ; MOREPOLICYSTATEMENTS | POLICYGROUP ; MOREPOLICYSTATEMENTS
1595 MOREPOLICYSTATEMENTS →
1596     <> | POLICYSTATEMENTS
1597 POLICY →
1598     Policy { DECLARATIONSTATEMENT CONDITIONSTATEMENT DECISION } : PRIORITY
1599 CONDITIONSTATEMENT →
1600     <> | Condition { BOOLEANEXPRESSION }
1601 DECISION →
1602     Decision { ACTIONBLOCK }
1603 PRIORITY →
1604     DECIMALNUMBER
1605 EXPRESSION →
1606     BOOLEANEXPRESSION | ARITHMETICEXPRESSION | STRINGEXPRESSION |
1607     DATETIMEEXPRESSION
1608 BOOLEANEXPRESSION →
1609     TRUE | FALSE | IDENTIFIER | FUNCTIONCALL |
1610     BOOLEANEXPRESSION BOOLEANOPERATOR BOOLEANEXPRESSION |
1611     ARITHMETICEXPRESSION RELATIONALOPERATOR ARITHMETICEXPRESSION |
1612     STRINGEXPRESSION RELATIONALOPERATOR STRINGEXPRESSION |
1613     BOOLEANEXPRESSION EQOPERATOR BOOLEANEXPRESSION |
1614     ( BOOLEANEXPRESSION ) | ! ( BOOLEANEXPRESSION )
1615 BOOLEANOPERATOR →

```

```

1616      && | || | ^
1617  RELATIONALOPERATOR →
1618      EQOPERATOR | >= | <= | > | <
1619  EQOPEATOR →
1620      == | !=
1621  ARITHMETICEXPRESSION →
1622      NUMBER | IDENTIFIER | FUNCTIONCALL |
1623      ARITHMETICEXPRESSION * ARITHMETICEXPRESSION |
1624      ARITHMETICEXPRESSION / ARITHMETICEXPRESSION |
1625      ARITHMETICEXPRESSION + ARITHMETICEXPRESSION |
1626      ARITHMETICEXPRESSION - ARITHMETICEXPRESSION |
1627      ( ARITHMETICEXPRESSION )
1628  NUMBER →
1629      UNSIGNINTEGER | INTEGER | REAL
1630  UNSIGNINTEGER →
1631      0UDecimalNUMBER
1632  INTEGER →
1633      SIGN DECIMALNUMBER
1634  REAL →
1635      INTEGER DECIMAL EXP
1636  DECIMAL →
1637      <> | .DECIMALNUMBER
1638  EXP →
1639      <> | EINTEGER
1640  SIGN →
1641      <> | + | -
1642  DECIMALNUMBER →
1643      DIGIT MOREDIGITS
1644  DIGIT →
1645      0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
1646  MOREDIGITS →
1647      <> | DIGIT MOREDIGITS
1648  STRINGEXPRESSION →
1649      <any sequence of Unicode characters in between '>' |
1650      IDENTIFIER | FUNCTIONCALL
1651  DATETIMEEXPRESSION →
1652      DATETIME | IDENTIFIER | FUNTIONCALL
1653  DATETIME →
1654      DIGIT DIGIT DIGIT DIGIT-DIGIT DIGIT-DIGIT DIGIT T
1655      DIGIT DIGIT:DIGIT DIGIT:DIGIT DIGIT TZ=<javaTimezoneID>
1656  IDENTIFIER →
1657      Self | SIMPLEIDENTIFIER | COMPLEXIDENTIFIER
1658  SIMPLEIDENTIFIER →
1659      NAME INDEX | NAME.NAME INDEX

```

1660 **NAME** →  
 1661     < any sequence of letters, numbers, and "\_" that starts with a letter>  
 1662 **INDEX** →  
 1663     <> | [ **ARITHMETICEXPRESSION** ]  
 1664 **COMPLEXIDENTIFIER** →  
 1665     **FUNCTIONCALL**.**SIMPLEIDENTIFIER**  
 1666 **FUNCTIONCALL** →  
 1667     **NAME** ( **PARAMETERS** ) |  
 1668     collect( **PARAMETERS** )[ **ARITHMETICEXPRESSION** ]. **SIMPLEIDENTIFIER**  
 1669 **PARAMETERS** →  
 1670     <> | **EXPRESSION** **MOREPARAMETERS**  
 1671 **MOREPARAMETERS** →  
 1672     <> | , **PARAMETERS**  
 1673 **ACTIONBLOCK** →  
 1674     **IDENTIFIER** ( **ACTIONARGS** ) **COMP** |  
 1675     **ACTIONBLOCK** -> **ACTIONBLOCK** |  
 1676     **ACTIONBLOCK** && **ACTIONBLOCK**  
 1677     **ACTIONBLOCK** || **ACTIONBLOCK**  
 1678     **ACTIONBLOCK** | **ACTIONBLOCK** |  
 1679     ( **ACTIONBLOCK** )  
 1680 **ACTIONARGS** →  
 1681     <> | **EXPRESSIONLIST**  
 1682 **EXPRESIONLIST** →  
 1683     **EXPRESSION** | **EXPRESSION**, **EXPRESSIONLIST**  
 1684 **COMP** →  
 1685     <> | **COP** **INTEGER**  
 1686 **COP** →  
 1687     == | != | <= | < | > | >=  
 1688 **POLICYGROUP** →  
 1689     Policygroup:ASSONAME { **CIMPOLICY** }:**PRIORITY**  
 1690 **ASSONAME** →  
 1691     <> | **NAME** ( **NAME** , **NAME** )  
 1692 **CIMTYPE** →  
 1693     **SHORT** | **USHORT** | **INTEGER** | **LINTEGER** |  
 1694     **REAL** | **LREAL** | **STRING** | **BOOL** | **CALENDAR**  
 1695 **SIMPLEBOOLEANEXPRESSION** →  
 1696     <a **BOOLEANEXPRESSION** where all the identifiers are limited to **NAMES**>  
 1697

**ANNEX A**  
(informative)**Change Log**

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

1703

## Bibliography

- 1704 DMTF DSP0107, *CIM Event Model White Paper 2.1*,  
1705 <http://www.dmtf.org/standards/documents/CIM/DSP0107.pdf>
- 1706 DMTF DSP0108, *CIM Policy Model White Paper 2.7*,  
1707 <http://www.dmtf.org/standards/documents/CIM/DSP0108.pdf>
- 1708