Policy Framework Working Group B. Moore
INTERNET-DRAFT L. Rafalow
Category: Standards Track IBM
Y. Ramberg
Y. Snir
J. Strassner
A. Westerinen
Cisco Systems
R. Chadha
Telcordia Technologies
M. Brunner
NEC
R. Cohen
Ntear LLC
February, 2001

Policy Core Information Model Extensions

<draft-ietf-policy-pcim-ext-00.txt>
Friday, February 23, 2001, 11:07 AM

Status of this Memo

Copyright Notice

Abstract

This document proposes a number of changes to the Policy Core Information Model (PCIM, RFC 3060). These changes include both extensions of PCIM into areas that it did not previously cover, and changes to the existing PCIM classes and associations. Both sets of changes are done in a way that, to the extent possible, preserves interoperability with

implementations of the original PCIM model.

Table of Contents

1. Introduction

This document (PCIM Extensions, abbreviated here to PCIMe) proposes a number of changes to the Policy Core Information Model (PCIM, RFC 3060 [3]). These changes include both extensions of PCIM into areas that it did not previously cover, and changes to the existing PCIM classes and associations. Both sets of changes are done in a way that, to the extent possible, preserves interoperability with implementations of the original PCIM model.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119, reference [1].

2. Overview of the Changes

2.1. How to Change an Information Model

The Policy Core Information Model is closely aligned with the DMTF's CIM Core Policy model. Since there is no separately documented set of rules for specifying IETF information models such as PCIM, it is reasonable to look to the CIM specifications for guidance on how to modify and extend the model. Among the CIM rules for changing an information model are the following. Note that everything said here about "classes" applies to association classes (including aggregations) as well as to non-association classes.

- o Properties may be added to existing classes.

o Classes, and individual properties, may be marked as DEPRECATED. If there is a replacement feature for the deprecated class or property, it is identified explicitly. Otherwise the notation "No value" is used. In this document, the notation "DEPRECATED FOR <feature-name>" is used to indicate that a feature has been deprecated, and to identify its replacement feature.

- o Classes may be inserted into the inheritance hierarchy above existing classes, and properties from the existing classes may then be "pulled up" into the new classes. The net effect is that the existing classes have exactly the same properties they had before, but the properties are inherited rather than defined explicitly in the classes.
- o New subclasses may be defined below existing classes.

2.2. List of Changes to the Model

The following subsections provide a very brief overview of the changes to PCIM being proposed in PCIMe.

2.2.1. Changes to PolicyRepository

Because of the potential for confusion with the Policy Framework component Policy Repository (from the four-box picture: Policy Management Tool, Policy Repository, PDP, PEP), "PolicyRepository" is a bad name for the PCIM class representing a container of reusable policy elements. Thus the class PolicyRepository is being replaced with the class ReusablePolicyContainer. To accomplish this change, it is necessary to deprecate the PCIM class PolicyRepository and its three associations, and replace them with a new class ReusablePolicyContainer and new associations.

As a separate change, the associations for ReusablePolicyContainer are being broadened, to allow a ReusablePolicyContainer to contain any reusable policy elements. In PCIM, the only associations defined for a PolicyRepository were for it to contain reusable policy conditions and policy actions.

2.2.2. Additional Associations and Additional Reusable Elements

The PolicyRuleInPolicyRule and PolicyGroupInPolicyRule aggregations are being imported from QPIM. These associations make it possible to define larger "chunks" of reusable policy to place in a ReusablePolicyContainer. These aggregations also introduce new semantics representing the contextual implications of having one PolicyRule executing within the scope of another PolicyRule.

2.2.3. Priorities and Decision Strategies

Drawing from both QPIM and ICIM, the Priority property is being deprecated in PolicyRule, and placed instead on the aggregations PolicyRuleInPolicyGroup, PolicyGroupInPolicyGroup, PolicyGroupInPolicyRule, and PolicyRuleInPolicyRule. (This is accomplished by placing the Priority property on the abstract aggregation PolicySetComponent, from which these four aggregations are derived.) The

QPIM rules for resolving relative priorities across nested PolicyGroups and PolicyRules are being incorporated into PCIMe as well. With the removal of the Priority property from PolicyRule, a new modeling dependency is introduced: in order to prioritize a PolicyRule relative to other PolicyRules, the rules must be placed in either a common PolicyGroup or a common PolicyRule.

In the absence of any clear, general criterion for detecting policy conflicts, the PCIM restriction stating that priorities are relevant only in the case of conflicts is being removed. In its place, a PolicyDecisionStrategy property is being added to the PolicyGroup and PolicyRule classes, to allow the policy administrator to select one of two behaviors with respect to rule evaluation: either perform the actions for all PolicyRules whose conditions evaluate to TRUE, or perform the actions only for the highest-priority PolicyRule whose conditions evaluate to TRUE. (Once again this is accomplished by placing the PolicyDecisionStrategy property in an abstract class PolicySet, from which PolicyGroup and PolicyRule are derived.) The QPIM rules for applying decision strategies to a nested set of PolicyGroups and PolicyRules are also being imported.

2.2.4. Policy Roles

The concept of policy roles is added to PolicyGroups (being present already in the PolicyRule class). This is accomplished via a new superclass for both PolicyRules and PolicyGroups - PolicySets. For nested PolicyRules and PolicyGroups, any roles associated with the outer rule or group are automatically "inherited" by the nested one. Additional roles may be added at the level of the nested rule or group.

It was also observed that there was no mechanism in PCIM for assigning roles to resources. For example, while it was possible to associate a PolicyRule with the role "FrameRelay&&WAN", there was no way to indicate which interfaces matched this criterion. A new PolicyRoleCollection class is defined in PCIMe, representing the collection of resources associated with a particular role. The linkage between a PolicyRule or PolicyGroup and a set of resources is then represented by an instance of PolicyRoleCollection. Equivalent values should be defined in entries in the PolicyRoles property, inherited by PolicyRules and PolicyGroups from PolicySet, and in the PolicyRole property in PolicyRoleCollection.

2.2.5. CompoundPolicyConditions and CompoundPolicyActions

The concept of a CompoundPolicyCondition is also being imported into PCIMe from QPIM, and broadened to include a parallel CompoundPolicyAction. In both cases the idea is to create reusable "chunks" of policy that can exist as named elements in a

ReusablePolicyContainer. The "Compound" classes and their associations incorporate the condition and action semantics that PCIM defined at the PolicyRule level: DNF/CNF for conditions, and ordering for actions.

Compound conditions and actions are defined to work with any component conditions and actions. In other words, while the components may be instances, respectively, of SimplePolicyCondition and SimplePolicyAction (discussed immediately below), they need not be.

2.2.6. Variables and Values

The SimplePolicyCondition / PolicyVariable / PolicyValue structure is being imported into PCIMe from QPIM. A list of PCIMe-level variables is defined, as well as a list of PCIMe-level values. Other variables and values may, if necessary, be defined in submodels of PCIMe.

A corresponding SimplePolicyAction / PolicyVariable / PolicyValue structure is also defined. While the semantics of a SimplePolicyCondition are "variable matches value", a SimplePolicyAction has the semantics "set variable to value".

2.2.7. Packet Filtering

For packet filtering done in the context of a PolicyCondition, a set of PolicyVariables and PolicyValues are defined, corresponding to the fields in an IP packet header plus the most common Layer 2 frame header fields. It is expected that policy conditions that filter on these header fields will be expressed in terms of CompoundPolicyConditions built up from SimplePolicyConditions that use these variables and values. An additional PolicyVariable, PacketDirection, is also defined, to indicate whether a packet being filtered is traveling inbound or outbound on an interface.

For packet filtering in other contexts (specifically, for the packet classifier filters modeled in QDDIM), these variables and values need not be used. Filter classes derived from the CIM FilterEntryBase class hierarchy may still be used in these contexts.

3. The Updated Class and Association Class Hierarchies

The following figure shows the class inheritance hierarchy for PCIMe. Changes from the PCIM hierarchy are noted parenthetically.

```
  ManagedElement (abstract)
     |
     +--Policy (abstract)
     |  |
     |  +---PolicySet (abstract -- new - 4.3)
     |  |   |
     |  |   +---PolicyGroup (moved - 4.3)
     |  |   |
     |  |   +---PolicyRule (moved - 4.3)
     |  |
     |  +---PolicyCondition (abstract)
     |  |   |
     |  |   +---PolicyTimePeriodCondition
     |  |   |
     |  |   +---VendorPolicyCondition
     |  |   |
     |  |   +---SimplePolicyCondition (new - 4.8.1)
     |  |   |
     |  |   +---CompoundPolicyCondition (new - 4.7.1)
     |  |       |
     |  |       +---CompoundFilterCondition (new - 4.9)
     |  |
     |  +---PolicyAction (abstract)
     |  |   |
     |  |   +---VendorPolicyAction
     |  |   |
     |  |   +---SimplePolicyAction (new - 4.8.4)
     |  |   |
     |  |   +---CompoundPolicyAction (new - 4.7.2)
     |  |
     |  +---PolicyVariable (abstract -- new - 4.8.5)
     |  |   |
     |  |   +---PolicyExplicitVariable (new - 4.8.6)
     |  |   |
     |  |   +---PolicyImplicitVariable (abstract -- new - 4.8.7)
     |  |       |
     |  |       +---(subtree of more specific classes -- new - 5.12)
     |  |
     |  +---PolicyValue (abstract -- new - 4.8.10)
     |      |
     |      +---(subtree of more specific classes -- new - 5.14)
     |
     +--Collection (abstract -- newly referenced)
        |
        +--PolicyRoleCollection (new - 4.6.2)
```

(continued on following page)

(continued from previous page)

```
ManagedElement(abstract)
   |
   +--ManagedSystemElement (abstract)
      |
      +--LogicalElement (abstract)
         |
         +--System (abstract)
            |
            +--AdminDomain (abstract)
               |
               +---ReusablePolicyContainer (new - 4.2)
               |
               +---PolicyRepository (deprecated - 4.2)
```

Figure 1.   Class Inheritance Hierarchy for PCIMe

The following figure shows the association class hierarchy for PCIMe. As before, changes from PCIM are noted parenthetically.

```
[unrooted]
   |
   +---PolicyComponent (abstract)
   |   |
   |   +---PolicySetComponent (abstract -- new - 4.3)
   |   |    |
   |   |    +---PolicyGroupInPolicyGroup (moved - 4.3)
   |   |    |
   |   |    +---PolicyRuleInPolicyGroup (moved - 4.3)
   |   |    |
   |   |    +---PolicyGroupInPolicyRule (new - 4.3)
   |   |    |
   |   |    +---PolicyRuleInPolicyRule (new - 4.3)
   |   |
   |   +---CompoundedPolicyCondition (abstract -- new - 4.7.1)
   |   |    |
   |   |    +---PolicyConditionInPolicyRule  (moved - 4.7.1)
   |   |    |
   |   |    +---PolicyConditionInPolicyCondition (new - 4.7.1)
   |   |
   |   +---PolicyRuleValidityPeriod
   |   |
   |   +---CompoundedPolicyAction (abstract -- new - 4.7.2)
   |   |    |
   |   |    +---PolicyActionInPolicyRule  (moved - 4.7.2)
   |   |    |
   |   |    +---PolicyActionInPolicyAction (new - 4.7.2)
   |   |
   |   +---PolicyVariableInSimplePolicyCondition (new - 4.8.2)
   |   |
   |   +---PolicyValueInSimplePolicyCondition (new - 4.8.2)
   |   |
   |   +---PolicyVariableInSimplePolicyAction (new - 4.8.4)
   |   |
   |   +---PolicyValueInSimplePolicyAction (new - 4.8.4)
```

(continued on following page)

(continued from previous page)

```
[unrooted]
   |
   +---Dependency (abstract)
   |   |
   |   +---PolicyInSystem (abstract)
   |   |   |
   |   |   +---PolicyGroupInSystem
   |   |   |
   |   |   +---PolicyRuleInSystem
   |   |   |
   |   |   +---ReusablePolicy (new - 4.2)
   |   |   |
   |   |   +---PolicyConditionInPolicyRepository (deprecated - 4.2)
   |   |   |
   |   |   +---PolicyActionInPolicyRepository (deprecated - 4.2)
   |   |
   |   +---PolicyValueConstraintInVariable (new - 4.8)
   |   |
   |   +---PolicyRoleCollectionInSystem (new - 4.6.2)
   |
   +---Component (abstract)
   |   |
   |   +---SystemComponent
   |       |
   |       +---PolicyContainerInPolicyContainer (new - 4.2)
   |       |
   |       +---PolicyRepositoryInPolicyRepository (deprecated - 4.2)
   |
   +---MemberOfCollection (newly referenced)
       |
       +--- ElementInPolicyRoleCollection (new - 4.6.2)
```

Figure 2. Association Class Inheritance Hierarchy for PCIMe

In addition to these changes that show up at the class and association class level, there are other changes from PCIM involving individual class properties. In some cases new properties are introduced into existing classes, and in other cases existing properties are deprecated (without deprecating the classes that contain them).

4. Areas of Extension to PCIM

The following subsections describe each of the areas for which PCIM extensions are being defined.

4.1. **Scope of Policies:** Domain Policies and Device Policies

Policies vary in level of abstraction, from the business-level expression of service level agreements (SLAs) to the specification of a set of rules that apply to devices in a network. Those latter policies can, themselves, be classified into at least two groups: those policies consumed by a Policy Decision Point (PDP) that specify the rules for an administrative and functional domain, and those policies consumed by a Policy Enforcement Point (PEP) that specify the device-specific rules for a functional domain. The higher-level rules consumed by a PDP may have late binding variables unspecified, or specified by a classification, whereas the device-level rules are likely to have fewer unresolved bindings.

There is a relationship between these levels of policy specification that is out of scope for this standards effort, but that is necessary in the development and deployment of a usable policy-based configuration system. An SLA-level policy transformation to the domain-level policy may be thought of as analogous to a visual builder that takes human input and develops a programmatic rule specification. The relationship between the domain-level policy and the device-level policy may be thought of as analogous to that of a compiler and linkage editor that translates the rules into specific instructions that can be executed on a specific type of platform.

The policy core information model may be used to specify rules at any and all of these levels of abstraction. However, at different levels of abstraction, different mechanisms may be more or less appropriate.

4.2. Reusable Policy Elements

In PCIM, a distinction was drawn between reusable PolicyConditions and PolicyActions and rule-specific ones. The PolicyRepository class was also defined, to serve as a container for these reusable elements. The name "PolicyRepository" has proven to be an unfortunate choice for the class that serves as a container for reusable policy elements. This term is already used in documents like the Policy Framework, to denote the location from which the PEP retrieves all policy specifications, and into which the Policy Management Tool places all policy specifications. Consequently, the PolicyRepository class is being deprecated, in favor of a new class ReusablePolicyContainer.

When a class is deprecated, any associations that refer to it must also be deprecated. So replacements are needed for the two associations PolicyConditionInPolicyRepository and PolicyActionInPolicyRepository, as well as for the aggregation PolicyRepositoryInPolicyRepository. In addition to renaming the PolicyRepository class to ReusablePolicyContainer, however, PCIMe is also broadening the types of policy elements that can be reusable. Consequently, rather than providing one-for-one replacements for the two associations, a single higher-level association ReusablePolicy is defined. This new association allows any policy element (that is, an instance of any subclass of the abstract class Policy) to be placed in a ReusablePolicyContainer.

Summarizing, the following changes in Sections 5 and 6 are the result of this item:

o The class ReusablePolicyContainer is defined.
o PCIM's PolicyRepository class is deprecated.
o The association ReusablePolicy is defined.
o PCIM's PolicyConditionInPolicyRepository association is deprecated.
o PCIM's PolicyActionInPolicyRepository association is deprecated.
o The aggregation PolicyContainerInPolicyContainer is defined.
o PCIM's PolicyRepositoryInPolicyRepository aggregation is deprecated.

4.3. Policy Sets

A "policy" can be thought of as a coherent set of rules to administer, manage, and control access to network resources (PolTerm, reference [12]). The structuring of these coherent sets of rules into subsets is enhanced in this document. In section 4.4, we discuss the new options for the nesting of policy rules.

A new abstract class, PolicySet, is introduced to provide an abstraction for a set of rules. It is derived from Policy, and it is inserted into the inheritance hierarchy above both PolicyGroup and PolicyRule. This reflects the additional structure flexibility and semantic capability of both subclasses.

Two properties are defined in PolicySet: PolicyDecisionStrategy and PolicyRoles. PolicyDecisionStrategy is added to PolicySet to define the evaluation relationship between the rules in the policy set. See section 4.5 for more information. PolicyRoles is added to PolicySet to name the retrieval sets. See section 4.6 for more information.

Along with the definition of the PolicySet class, a new abstract aggregation class is defined that will also be discussed in the following sections. PolicySetComponent is defined as a subclass of PolicyComponent; it provides the containment relationship for a PolicySet. PolicyGroupInPolicyGroup and PolicyRuleInPolicyGroup are modified to subclass from PolicySetComponent. PolicyGroupInPolicyRule and PolicyRuleInPolicyRule, discussed in the next section, are also defined as subclasses of PolicySetComponent.

4.4. Nested Policy Rules

As previously discussed, policy is described by a set of policy rules that may be grouped into subsets. In this section we introduce the notion of nested rules, or the ability to define rules within rules. Nested rules are also called sub-rules, and we use both terms in this document interchangeably. Two new aggregations are defined for this purpose: PolicyRuleInPolicyRule and PolicyGroupInPolicyRule.

4.4.1. Usage Rules for Nested Rules

The relationship between rules and sub-rules is defined as follows:

- The parent rule's condition clause is a pre-condition for evaluation of all nested rules. If the parent rule's condition clause evaluates to FALSE, all sub-rules SHALL be skipped and their condition clauses SHALL NOT be evaluated.

- If the parent rule's condition evaluates to TRUE, the set of sub-rules SHALL BE executed according to the decision strategy and priorities as discussed in Section 4.5.
- If the parent rule's condition evaluates to TRUE, the parent rule's set of actions is executed BEFORE the evaluation and execution of the sub-rules. The parent rule's actions are not to be confused with default actions. A default action is one that is to be executed only if none of the more specific sub-rules are executed. If a default action needs to be specified, it needs to be defined as an action that is part of a catchall sub-rule associated with the parent rule. The association linking the default action(s) in this special sub-rule should have the lowest priority relative to all other sub-rule associations:

```
if precondition then parent rule's action
      if condA then actA
      if condB then ActB
      if True then default action
```

  Default actions have meaning when FirstMatching decision strategies are in effect (see section 4.5).

- Policy rules have an implicit context in which they are executed. For example, the context of a policy rule could be all packets running on an interface or set of interfaces on which the rule is applied. Similarly, a parent rule provides a context to all of its sub-rules. The context of the sub-rules is the restriction of the context of the parent rule to the set of cases that match the parent rule's condition clause.

4.4.2. Motivation

The motivation for introducing nested rules includes enhancing the definition of Policy, defining and reusing context hierarchies, optimizing how a rule is evaluated, and providing finer-grained control over condition evaluation.

Rule nesting enhances Policy readability, expressiveness and reusability. The ability to nest policy rules and form sub-rules is important for manageability and scalability, as it enables complex policy rules to be constructed from multiple simpler policy rules. These enhancements ease the policy management tools' task, allowing policy rules to be expressed in a way closer to how humans think.

Sub-rules enable the policy designer to define a hierarchy of rules. This hierarchy has the property that sub-rules can be scoped by their parent rules. This scoping, or context of evaluation and execution, is a powerful tool in enabling the policy designer to obtain the fine-grained control needed to appropriately manage resources for certain applications. The example in the following section demonstrates that expressing relative bandwidth allocation rules can be done very naturally using a hierarchical rule structure.

Rule nesting can be used to optimize the way policy rules are evaluated and executed. Once the parent rule's condition clause is evaluated to FALSE, all sub-rules are skipped, optimizing the number of lookups required. Note that this is not the prime reason for rule nesting, but rather a side benefit. Optimization of rule execution can be done in the PDP or in the PEP by dedicated code. This is similar to the relation between a high level programming language like C and machine code. An optimizer can create a more efficient machine code than any optimization done by the programmer within the source code. Nevertheless, if the PEP or PDP does not do optimization, the administrator writing the policy can optimize the policy rules for execution using rule nesting.

In a model where condition evaluation may have side effects, nesting rules allow control of condition evaluation, as sub-rule conditions SHALL NOT be evaluated if the condition of the parent rule evaluates to FALSE.

Nested rules are not designed for policy repository retrieval optimization. It is assumed that all rules and groups that are assigned to a role are retrieved by the PDP or PEP from the policy repository and enforced. Optimizing the number of rules retrieved should be done by clever selection of roles.

4.4.3. Usage Example

This section provides a usage example that aims to clarify the motivation for the definition of rule nesting and the use of the relative context. Consider the following example, where a set of rules is used to specify the minimal bandwidth allocations on an interface. The policy reads:

> On any interface on which these rules apply, allocate at least 30% of the interface bandwidth to UDP flows, and at least 40% of the interface bandwidth to TCP flows.

This single rule is translated to a set of two rules:

```
If (IP protocol is UDP) THEN Set MinBW to 30%  (1)
If (IP protocol is TCP) THEN Set MinBW to 40%  (2)
```

Now, let's add some sub-rules to further differentiate how bandwidth should be allocated to specific UDP and TCP applications (indentation indicates rule nesting):

```
If (IP protocol is UDP) THEN Set MinBW to 30%     (1)
      If (protocol is TFTP) THEN Set MinBW to 10%   (1a)
      If (protocol is NFS) THEN Set MinBW to 40%    (1b)
If (IP protocol is TCP) THEN Set MinBW to 40%     (2)
```

```
If (protocol is HTTP) THEN Set MinBW to 20%   (2a)
If (protocol is FTP) THEN Set MinBW to 30%    (2b)
```

This means that for UDP flows, TFTP should be allocated 10% of the bandwidth while NFS should be allocated 40%. For TCP flows, HTTP should be allocated 20% of the bandwidth while FTP should be allocated 30%.

The context of each of the two high-level rules (those marked (1) and (2) above) is all flows running on an interface. The two sub-rules of the UDP rule, marked (1a) and (1b) above specify a more granular context: within UDP flows, TFTP should be allocated 10% of the bandwidth while NFS should be allocated 40%. The context of these sub-rules is therefore UDP flows only. Similar functionality applies for the hierarchy of rules treating TCP flows.

A context hierarchy enhances reusability. The rules that divide bandwidth between TFTP and NFS can be re-used and associated to rules that allocate different percentages of the bandwidth for different interfaces (or even for the same interface, but under different conditions) for UDP.

4.5. Priorities and Decision Strategies

A "decision strategy" is used to specify the evaluation method for the policies in a PolicySet. Two decision strategies are defined: "FirstMatching" and "AllMatching." The FirstMatching strategy is used to cause the evaluation of the rules in a set such that the actions of only the first rule that matches are enforced on a given examination of the PolicySet. The AllMatching strategy is used to cause the evaluation of all rules in a set; for all of the rules that match, the actions are enforced. (Strawman: Implementations MUST support the FirstMatching decision strategy; implementations MAY support the AllMatching decision strategy.)

As previously discussed, the PolicySet subclasses are PolicyGroup and PolicyRule, and either subclass may contain PolicySets of either subclass. Loops, including the degenerate case of a PolicySet that contains itself, are not allowed when PolicySets contain other PolicySets. The containment relationship is specified using the PolicySetComponent subclasses: PolicyGroupInPolicyGroup, PolicyRuleInPolicyGroup, PolicyGroupInPolicyRule and PolicyRuleInPolicyRule.

The order of evaluation within a PolicySet is established by the Priority property of the PolicySetComponent aggregation. Instances of the subclasses of PolicySetComponent specify the relative priority of the contained policy groups and rules within the containing group or rule. The use of PCIM's PolicyRule.Priority property is deprecated in favor of this new property. The separation of the priority property from the rule has two advantages. First, it generalizes the concept of priority, so it can be used for both groups and rules; and, second, it places the priority on the relationship between the parent policy set and the subordinate policy group or rule. The assignment of a priority value, then, becomes much easier in that the value is used only in relationship to other priorities in the same set.

Together, the PolicySet.PolicyDecisionStrategy and PolicySetComponent.Priority determine the processing for the rules contained in a PolicySet. As before, the larger priority value represents the higher priority. Unlike the earlier definition, PolicySetComponent.Priority MUST have a unique value when compared with others defined for the aggregating PolicySet. Thus, the evaluation of rules within a set is deterministically specified.

For a FirstMatching decision strategy, the order of evaluation, then, is high to low priority. The first rule (i.e., the one with the highest priority) in the set that evaluates to True, is the only rule whose actions are enforced for a particular evaluation pass through the PolicySet.

For an AllMatching decision strategy, the order of evaluation is also from high priority to low priority; however, all of the matching rules are executed. Although higher priority rules are evaluated first, lower priority rules may get the "last word." So, for example, if two rules both evaluate to True, and the higher priority rule sets the DSCP to 3 and the lower priority rule sets the DSCP to 4, the lower priority rule will be evaluated later and, therefore, will "win," in this example, setting the DSCP to 4. Thus, conflicts between rules are resolved by this evaluation order.

4.5.1. Structuring Decision Strategies

When policy sets are nested, as shown in Figure 3, the decision strategies may be nested arbitrarily. In this example, the evaluation order for the nested rules is 1A, 1B1, 1X2, 1B3, 1C, 1C1, 1X2 and 1C3. (Note that PolicyRule 1X2 is included in both PolicyGroup 1B and PolicyRule 1C, but with different priorities.) Of course, the evaluation order is also dependent on which rules, if any, match.

```
PolicyGroup 1: FirstMatching
  |
  +-- Pri=6 -- PolicyRule 1A
  |
  +-- Pri=5 -- PolicyGroup 1B: AllMatching
  |              |
  |              +-- Pri=5 -- PolicyGroup 1B1: AllMatching
  |              |              |
  |              |              +---- etc.
  |              |
  |              +-- Pri=4 -- PolicyRule 1X2
  |              |
  |              +-- Pri=3 -- PolicyRule 1B3: FirstMatching
  |                             |
  |                             +---- etc.
  |
  +-- Pri=4 -- PolicyRule 1C: FirstMatching
                 |
                 +-- Pri=4 -- PolicyRule 1C1
                 |
                 +-- Pri=3 -- PolicyRule 1X2
                 |
                 +-- Pri=2 -- PolicyRule 1C3
```

Figure 3.   Nested PolicySets with Different Decision Strategies

- o Because PolicyGroup 1 has a FirstMatching decision strategy, if the conditions of PolicyRule 1A match, its actions are enforced and the evaluation stops.

- o If it does not match, PolicyGroup 1B is evaluated using an AllMatching strategy. Since PolicyGroup 1B1 also has an AllMatching strategy all of the rules and groups of rules contained in PolicyGroup 1B1 are evaluated and enforced as appropriate. PolicyRule 1X2 and PolicyRule 1B3 are also evaluated and enforced as appropriate. If any of the sub-rules in the subtrees of PolicyGroup 1B evaluate to True, then PolicyRule 1C is not evaluated because the FirstMatching strategy of PolicyGroup 1 has been satisfied.

- o If neither PolicyRule 1A nor PolicyGroup 1B yield a match, then PolicyRule 1C is evaluated. Since it is first matching, rules 1C1, 1X2, and 1C3 are evaluated until the first match, if any.

4.5.2. Deterministic Decisions

As mentioned above, we propose that Priority values are to be unique within a containing PolicySet. Although there are certainly cases where rules need not have a unique priority value (i.e., where evaluation and execution order is not important), it is believed that the flexibility gained by this capability is not sufficiently beneficial to justify the possible variations in implementation behavior and the resulting confusion that might occur.

Therefore, all PolicySetComponent.Priority values MUST be unique among the values in the aggregating PolicySet. Each PolicySet, then, has a deterministic behavior based upon the decision strategy and uniquely defined order of evaluation.

4.5.3. Multiple PolicySet Trees For a Resource

As shown in the example in Figure 3, PolicySet trees are defined by the PolicySet subclass instances and the PolicySetComponent subclass aggregation instances between them. Each PolicySet tree has a defined set of decision strategies and evaluation orders. However, a given resource may have multiple, disjoint PolicySet trees; we need a join algorithm that describes the decision strategy and evaluation order among the top-level (called "unrooted") PolicySet instances. (Note that an unrooted PolicySet instance may only be unrooted in a given context.)

<<Solution under discussion - see Open Issue 9>>

4.6. Policy Roles

A policy role is defined in [12] as "an administratively specified characteristic of a managed element (for example, an interface). It is a selector for policy rules and PRovisioning Classes (PRCs), to determine the applicability of the rule/PRC to a particular managed element."

In PCIMe, PolicyRoles is defined as a property of PolicySet, which is inherited by both PolicyRules and PolicyGroups. In this draft, we also add PolicyRole as the identifying name of a collection of resources (PolicyRoleCollection), where each element in the collection has the specified role characteristic.

4.6.1. Comparison of Roles in PCIM with Roles in snmpconf

In the Configuration Management with SNMP (snmpconf) working group's Policy Based Management MIB [13], policy rules are of the form

```
if <policyFilter> then <policyAction>
```

where <policyFilter> is a set of conditions that are used to determine whether or not the policy applies to an object instance. The policy filter can perform comparison operations on SNMP variables already defined in MIBS (e.g., "ifType == ethernet").

The policy management MIB defined in [13] defines a Role table that enables one to associate Roles with elements, where roles have the same semantics as in PCIM. Then, since the policyFilter in a policy allows one to define conditions based on the comparison of the values of SNMP variables, one can filter elements based on their roles as defined in the Role group.

This approach differs from that adopted in PCIM in the following ways. First, in PCIM, a set of role(s) is associated with a policy rule as the values of the PolicyRoles property of a policy rule. The semantics of role(s) are then expected to be implemented by the PDP (i.e. policies are applied to the elements with the appropriate roles). In [draft-ietf-snmpconf-pm-04], however, no special processing is required for realizing the semantics of roles; roles are treated just as any other SNMP variables and comparisons of role values can be included in the policy filter of a policy rule.

Secondly, in PCIM, there is no formally defined way of associating a role with an object instance, whereas in [13] this is done via the use of the Role tables (pmRoleESTable and pmRoleSETable). The Role tables associate Role values with elements.

4.6.2. Addition of PolicyRoleCollection to PCIMe

In order to remedy the latter shortcoming in PCIM (i.e. the lack of a way of associating a role with an object instance), we define a new class PolicyRoleCollection that subclasses from the CIM Collection class. Resources that share a common role belong to a PolicyRoleCollection instance. Membership in this collection is indicated using the aggregation ElementInPolicyRoleCollection. The resource's role is specified in the PolicyRole property of the PolicyRoleCollection class.

A PolicyRoleCollection always exists in the context of a system. As was done in PCIM for PolicyRules and PolicyGroups, this is captured by an association, PolicyRoleCollectionInSystem. Remember that in PCIM, a System is a base class for describing network devices and administrative domains.

When associating a PolicyRoleCollection with a System, this should be done consistently with the system that scopes the policy rules/groups that are applied to the resources in that collection. A PolicyRoleCollection is associated with the same system as the applicable PolicyRules and/or PolicyGroups, or to a System higher in the tree formed by the SystemComponent association. When a PEP belongs to multiple Systems (i.e., AdminDomains), and scoping by a single domain is impractical, two alternatives exist. One is to arbitrarily limit domain membership to one System/AdminDomain. The other option is to define a more global AdminDomain that simply includes the others, and/or that spans the business or enterprise.

As an example, suppose that there are 20 traffic trunks in a network, and that an administrator would like to assign three of them to provide "gold" service. Also, the administrator has defined several policy rules which specify how the "gold" service is delivered. For these rules, the PolicyRoles property (inherited from PolicySet) is set to "Gold Service".

In order to associate three traffic trunks with "gold" service, an instance of the PolicyRoleCollection class is created and its PolicyRole property is also set to "Gold Service". Following this, the administrator associates three traffic trunks with the new instance of PolicyRoleCollection via the ElementInPolicyRoleCollection aggregation. This enables a PDP to determine that the "Gold Service" policy rules apply to the three aggregated traffic trunks.

Note that roles are used to optimize policy retrieval. It is not mandatory to implement roles or, if they have been implemented, to group elements in a PolicyRoleCollection. However, if roles are used, then either the collection approach should be implemented, or elements should be capable of reporting their "pre-programmed" roles (as is done in COPS).

4.6.3. Roles for PolicyGroups

In PCIM, role(s) are only associated with policy rules. However, it may be desirable to associate role(s) with groups of policy rules. For example, a network administrator may want to define a group of rules that apply only to Ethernet interfaces. A policy group can be defined with a role-combination="Ethernet", and all the relevant policy rules can be placed in this policy group. (Note that in PCIMe, role(s) are made available to PolicyGroups as well as to PolicyRules by moving PCIM's PolicyRoles property up from PolicyRule to the new abstract class PolicySet. The property is then inherited by both PolicyGroup and PolicyRule.) Then every policy rule in this policy group implicitly inherits this role-combination from the containing policy group. A similar implicit inheritance applies to nested policy groups.

Note that there is no explicit copying of role(s) from container to contained entity. Obviously, this implicit inheritance of role(s) leads to the possibility of defining inconsistent role(s) (as explained in the example below); the handling of such inconsistencies is beyond the scope of PCIMe.

As an example, suppose that there is a PolicyGroup PG1 that contains three PolicyRules, PR1, PR2, and PR3. Assume that PG1 has the roles "Ethernet" and "Fast". Also, assume that the contained policy rules have the role(s) shown below:

```
+-------------------------------+
| PolicyGroup PG1               |
| PolicyRoles = Ethernet, Fast  |
+-------------------------------+
           |
           |        +------------------------+
           |        | PolicyRule PR1         |
           |--------| PolicyRoles = Ethernet |
           |        +------------------------+
           |
           |        +--------------------------+
           |        | PolicyRule PR2           |
           |--------| PolicyRoles = <undefined>|
           |        +--------------------------+
           |
           |        +------------------------+
           |        | PolicyRule PR3         |
           |--------| PolicyRoles = Slow     |
                    +------------------------+
```

Figure 4. Inheritance of Roles

In this example, the PolicyRoles property value for PR1 is consistent with the value in PG1, and in fact, did not need to be redefined. The value of PolicyRoles for PR2 is undefined. Its roles are implicitly inherited from PG1. Lastly, the value of PolicyRoles for PR3 is "Slow". This appears to be in conflict with the role, "Fast," defined in PG1. However, whether these roles are actually in conflict is not clear. In one scenario, the policy administrator may have wanted only "Fast"-"Ethernet" rules in the policy group. In another scenario, the administrator may be indicating that PR3 applies to all "Ethernet" interfaces regardless of whether they are "Fast" or "Slow." Only in the former scenario (only "Fast"-"Ethernet" rules in the policy group) is there a role conflict.

Note that it is possible to override implicitly inherited roles via appropriate conditions on a PolicyRule. For example, suppose that PR3 above had defined the following conditions:

  (interface is not "Fast") and (interface is "Slow")

This results in unambiguous semantics for PR3.

4.7. Compound Policy Conditions and Compound Policy Actions

Compound policy conditions and compound policy actions are introduced to provide additional reusable "chunks" of policy.

4.7.1. Compound Policy Conditions

A CompoundPolicyCondition is a PolicyCondition representing a Boolean combination of simpler conditions. The conditions being combined may be SimplePolicyConditions (discussed below in section 4.7), but the utility of reusable combinations of policy conditions is not necessarily limited to the case where the component conditions are simple ones.

The PCIM extensions to introduce compound policy conditions are relatively straightforward. Since the purpose of the extension is to apply the DNF / CNF logic from PCIM's PolicyConditionInPolicyRule aggregation to a compound condition that aggregates simpler conditions, the following changes are required:

- o Create a new aggregation PolicyConditionInPolicyCondition, with the same GroupNumber and ConditionNegated properties as PolicyConditionInPolicyRule. The cleanest way to do this is to move the properties up to a new abstract aggregation superclass CompoundedPolicyCondition, from which the existing aggregation PolicyConditionInPolicyRule and a new aggregation PolicyConditionInPolicyCondition are derived. For now there is no need to re-document the properties themselves, since they are already documented in PCIM as part of the definition of the PolicyConditionInPolicyRule aggregation.
- o It is also necessary to define a concrete subclass CompoundPolicyCondition of PolicyCondition, to introduce the ConditionListType property. This property has the same function, and works in exactly the same way, as the corresponding property currently defined in PCIM for the PolicyRule class.

The class and property definitions for representing compound policy conditions are below, in Section 5.

4.7.2. Compound Policy Actions

A compound action is a convenient construct to represent a sequence of actions to be applied as a single atomic action within a policy rule. In many cases, actions are related to each other and should be looked upon as sub-actions of one "logical" action. An example of such a logical action is "shape & mark" (i.e., shape a certain stream to a set of predefined bandwidth characteristics and then mark these packets with a certain DSCP value). This logical action is actually composed of two different QoS actions, which should be performed in a well-defined order and as a complete set.

The CompoundPolicyAction construct allows one to create a logical relationship between a number of actions, and to define the activation logic associated with this logical action.

The CompoundPolicyAction construct allows the reusability of these complex actions, by storing them in a ReusablePolicyContainer and reusing them in different policy rules. Note that a compound action may also be aggregated by another compound action.

As was the case with CompoundPolicyCondition, the PCIM extensions to introduce compound policy actions are relatively straightforward. This time the goal is to apply the property ActionOrder from PCIM's PolicyActionInPolicyRule aggregation to a compound action that aggregates simpler actions. The following changes are required:

- o Create a new aggregation PolicyActionInPolicyAction, with the same ActionOrder property as PolicyActionInPolicyRule. The cleanest way to do this is to move the property up to a new abstract aggregation superclass CompoundedPolicyAction, from which the existing aggregation PolicyActionInPolicyRule and a new aggregation PolicyActionInPolicyAction are derived. For now there is no need to re-document the ActionOrder property itself, since it is already documented in PCIM as part of the definition of the PolicyActionInPolicyRule aggregation.
- o It is also necessary to define a concrete subclass CompoundPolicyAction of PolicyAction, to introduce the SequencedActions property. This property has the same function, and works in exactly the same way, as the corresponding property currently defined in PCIM for the PolicyRule class.
- o Finally, a new property ExecutionStrategy is needed for both the PCIM class PolicyRule and the new class CompoundPolicyAction. This property allows the policy administrator to specify how the PEP should behave in the case where there are multiple actions aggregated by a PolicyRule or by a CompoundPolicyAction.

The class and property definitions for representing compound policy actions are below, in Section 5.

Compound actions allow the definition of logically complex policy rules and action behavior. The following example illustrates two advantages of using compound actions.

A QoS policy domain may include a rule that defines the following behavior:

```
  If (CONDITION) Then Do:
    "Shape traffic to <X> and Set DSCP to EF (high priority traffic);
     if canÆt shape than Set DSCP to BE (best effort)."
```

This rule can be realized by defining two CompoundPolicyAction instances, A and B. Two sub-actions are grouped into CompoundPolicyAction A:

```
     Shape traffic to <X>
     Mark to EF (DSCP).
```

The ExecutionStrategy property of CompoundPolicyAction A would be defined as "Mandatory Do all". This means that if shaping or marking cannot both be done, then nothing should be done.

A second action, CompoundPolicyAction B, would hold the Mark to BE sub-action.

CompoundPolicyAction A and CompoundPolicyAction B would be aggregated into the policy rule using the PolicyActionInPolicyRule aggregation. The CompoundPolicyAction A will be ordered for execution before the CompoundPolicyAction B. The PolicyRule's ExecutionStrategy property would be set to "Do until success". In this way, CompoundPolicyAction A will be enforced on all PEPs that support shaping, while CompoundPolicyAction B will be enforced otherwise.

4.8. Variables and Values

The following subsections introduce several related concepts, including PolicyVariables and PolicyValues (and their numerous subclasses), SimplePolicyConditions, and SimplePolicyActions.

4.8.1. Simple Policy Conditions

The SimplePolicyCondition class models elementary Boolean conditional expressions of the form: "If (<variable> MATCH <value>)". The "If" clause and the "MATCH" are implied in the formal notation. The relationship is always 'MATCH' and is interpreted based on the variable and the value. Section 4.8.3 explains the semantics of the operator and how to extend them. Arbitrarily complex Boolean expressions can be formed by chaining together any number of simple conditions using relational operators. Individual simple conditions can be negated as well. Arbitrarily complex Boolean expressions are modeled by the class CompoundPolicyCondition (described in section 4.7.1).

For example, the expression "If SourcePort == 80" can be modeled by a simple condition. In this example, 'SourcePort' is a variable, '==' is the relational operator denoting the equality relationship (which is generalized by PCIMe to a "match" relationship), and '80' is an integer value. The complete interpretation of a simple condition depends on the binding of the variable. Section 4.8.5 describes variables and their binding rules.

The SimplePolicyCondition class refines the basic structure of the PolicyCondition class defined in PCIM by using the pair <variable> and <value> to form the condition. Note that the operator between the variable and the value is always implied in PCIMe: it is not a part of the formal notation.

The variable specifies the attribute of an object that should be matched when evaluating the condition. For example, for a QoS derivation, this object could represent the flow that is being conditioned. A set of predefined variables that cover network attributes that are commonly used for filtering is introduced here in PCIMe to encourage interoperability.

This list covers layer 3 IP attributes such as IP network addresses, protocols and ports, as well as a set of layer 2 attributes (e.g., MAC addresses).

The PCIMe defines a single operator, "match", as explained in section 4.8.3.

The bound variable is matched against a value to produce the Boolean result. For example, in the condition "If the source IP address of the flow belongs to the 10.1.x.x subnet", a source IP address variable is matched against a 10.1.x.x subnet value. The operator specifies the type of relation between the variable and the value evaluated in the condition.

4.8.2. Using Simple Policy Conditions

Simple conditions can be used in policy rules directly, or as building blocks for creating compound policy conditions.

Simple condition composition MUST enforce the following data-type conformance rule: The ValueTypes property of the variable must be compatible with the type of the value class used. The simplest (and friendliest, from a user point-of-view) is to equate the type of the value class with the name of the class. By ensuring that the ValueTypes property of the variable matches the name of the value class used, we know that the variable and value instance values are compatible with each other.

Composing a simple condition requires that an instance of the class SimplePolicyCondition be created, and that instances of the variable and value classes that it uses also exist. Note that the variable and/or value instances may already exist as reusable objects in an appropriate ReusablePolicyContainer.

Two aggregations are used in order to create the pair <variable>, <value>. The aggregation PolicyVariableInSimplePolicyCondition relates a SimplePolicyCondition to a single variable instance. Similarly, the aggregation PolicyValueInSimplePolicyCondition relates a SimplePolicyCondition to a single value instance. Both aggregations are defined in this document.

Figure 5 depicts a SimplePolicyCondition with its associated variable and value.

```
                              +-----------------------+
                              | SimplePolicyCondition |
                              +-----------------------+
                                    *         @
                                    *         @
              +------------------+  *         @  +---------------+
              | (PolicyVariable) |***        @@@| (PolicyValue) |
              +------------------+               +---------------+
                 #            #
                 #    ooo     #
                 #            #
   +---------------+        +---------------+
   | (PolicyValue) |  ooo   | (PolicyValue) |
   +---------------+        +---------------+

   Aggregation Legend:
     ****  PolicyVariableInSimplePolicyCondition
     @@@@  PolicyValueInSimplePolicyCondition
     ####  PolicyValueConstraintInVariable
```

Figure 5. SimplePolicyCondition

Note: The class names in parenthesis denote subclasses. The named classes in the figure are abstract and cannot, therefore, be instantiated.

4.8.3. The Simple Condition Operator

A simple condition models an elementary Boolean expression conditional clause of the form "If variable MATCHes value". However, the formal notation of the SimplePolicyCondition, together with its associations, models only a pair, {variable, value}. The "If" term and the "MATCH" operator are not directly modeled -- they are implied.

The implied MATCH operator carries an overloaded semantics. For example, in the simple condition "If DestinationPort MATCH '80'" the interpretation of the MATCH operator is equality (the 'equal' operator). Clearly, a different interpretation is needed in the following cases:

- o "If DestinationPort MATCH {'80', '8080'}" -- operator is 'IS SET MEMBER'

- o "If DestinationPort MATCH {'1 to 255'}" -- operator is 'IN INTEGER RANGE'

- o "If SourceIPAddress MATCH 'MyCompany.com'" -- operator is 'IP ADDRESS AS RESOLVED BY DNS'

The examples above illustrate the implicit, context dependant nature of the interpretation of the MATCH operator. The interpretation depends on the actual variable and value instances in the simple condition. PCIMe does not contain text to explicitly detail the possible interpretations of MATCH operations. The interpretation is always derived from the value instance associated with the simple condition. Text accompanying the value class definition SHOULD be used as a guideline for interpreting the semantics of the MATCH relationship.

The PolicyValueConstraintInVariable association specifies additional constraints on the possible values that can be matched with a variable within a simple condition. Using this association a source or destination port can be constrained to be matched against integer values in the range 0-65535. A source or destination IP address can be constrained to be matched against a specified list of IPv4 address values, etc. In order to check whether a value X can be used with a variable A constrained by value Y, the following conformance test should be made. If all events for which the SimplePolicyCondition (A match X) evaluates to TRUE also evaluate to TRUE for the SimplePolicyCondition (A match Y), than X conforms to the constraint Y. If multiple values Y1, Y2, ..., Yn constrain a variable, then the conformance test involves checking against the condition (A match Y1) OR (A match Y2) OR ... OR (A match Yn).

4.8.4. SimplePolicyActions

The SimplePolicyAction class models the elementary set operation. "SET <variable> TO <value>". The set operator MUST overwrite an old value of the variable.

For example, the action "set DSCP to EF" can be modeled by a simple action. In this example, 'DSCP' is an implicit variable referring to the IP packet header DSCP field. 'EF' is an integer or bit string value (6 bits). The complete interpretation of a simple action depends on the binding of the variable. Section [4.8.4] describes variables and their binding rules for conditions.

The SimplePolicyAction class refines the basic structure of the PolicyAction class defined in PCIM, by specifying the contents of the action using the <variable> <value> pair to form the action. The variable specifies the attribute of an object that has passed the condition by evaluating to true. This means the binding of the variable is delayed until the condition evaluates to true for one or more objects. The value of the object's attribute is set to <value>.

SimplePolicyActions can be used in policy rules directly, or as building blocks for creating CompoundPolicyActions.

SimplePolicyAction execution MUST enforce the following data type conformance and translation rule: The ValueTypes property of the variable must be compatible with the type of the value class used. The following table shows the compatibility and transformation rules. 'ND' means the transformation is not defined.

```
+---------------------------------------------------------------------+
|variable  |              value type                                  |
|type      |                                                          |
+---------------------------------------------------------------------+
|          |String |Integer|BitString| IPv4Addr | IPv6Addr |MACAddr|
+---------------------------------------------------------------------+
| String   |   X   |to text|  [0|1]  | A.B.C.D  | dotted   | X:X.. |
+---------------------------------------------------------------------+
| Integer  |"atoi" |   X   |BinaryVal| 32bit int|    ND    |   ND  |
+---------------------------------------------------------------------+
| BitString|convert|convert|    X    |     ND   |    ND    |   ND  |
+---------------------------------------------------------------------+
| IPv4Addr |convert|convert|   ND    |     X    |    ND    |   ND
+---------------------------------------------------------------------+
| IPv6Addr |convert|   ND  |   ND    | v4 format|     X    |   ND  |
+---------------------------------------------------------------------+
| MACAddr  |   ND  |   ND  |   ND    |    ND    |    ND    |    X  |
+---------------------------------------------------------------------+
```

Composing a simple action requires that an instance of the class SimplePolicyAction be created, and that instances of the variable and value classes that it uses also exist. Note that the variable and/or value instances may already exist as reusable objects in an appropriate ReusablePolicyContainer.

Two aggregations are used in order to create the pair <variable> <value>. The aggregation PolicyVariableInSimplePolicyAction relates a SimplePolicyAction to a single variable instance. Similarly, the aggregation PolicyValueInSimplePolicyAction relates a SimplePolicyAction to a single value instance. Both aggregations are defined in this document.

Figure 6 depicts a SimplePolicyAction with its associated variable and value.

```
                                +------------------------+
                                | SimplePolicyAction     |
                                |                        |
                                +------------------------+
                                      *         @
                                      *         @
                +-------------------+ *         @  +-----------------+
                | (PolicyVariable) |***         @@@| (PolicyValue) |
                +-------------------+              +-----------------+
                   #            #
                   #    ooo     #
                   #            #
     +-----------------+       +-----------------+
     | (PolicyValue) |  ooo    | (PolicyValue) |
     +-----------------+       +-----------------+

     Aggregation Legend:
       ****  PolicyVariableInSimplePolicyAction
       @@@@  PolicyValueInSimplePolicyAction
       ####  PolicyValueConstraintInVariable
```

Figure 6.   SimplePolicyAction

4.8.5. Policy Variables

A variable generically represents information that changes (or "varies"), and that is set or evaluated by software. In policy, conditions and actions can abstract information as "policy variables" to be evaluated in logical expressions, or set by actions.

PCIMe defines two types of PolicyVariables, a PolicyImplicitVariable and a PolicyExplicitVariable. The semantic difference between these classes is based on modeling context. Explicit variables are bound to exact model constructs, while implicit variables are defined and evaluated outside of a model, in a more subjective context. For example, one can imagine a PolicyCondition testing for a CIM ManagedSystemElement's Status property set to "Error." The Status property is an explicitly defined PolicyVariable (i.e., it is defined in the context of the CIM Schema and evaluated in the context of a specific instance). On the other hand, network packets are not explicitly modeled or instantiated, since there is no perceived value (at this time) in managing at the packet level. Therefore, a PolicyCondition can make no explicit reference to a model construct that represents a network packet's source address. In this case, an implicit PolicyVariable is defined to allow evaluation of a packet's source address.

4.8.6. Explicitly Bound Policy Variables

Explicitly bound policy variables indicate the class and property names of the model construct to be evaluated or set. The CIM Schema defines and constrains "appropriate" values for the variable (i.e., model property) using data types and other information such as class/property qualifiers.

A PolicyExplicitVariable is "explicit" because its model semantics are exactly defined. It is NOT explicit due to an exact binding to a particular object. If PolicyExplicitVariable is only tied to instances (either via association or by a object identification property in the class itself), then we are forcing element-specific rules. On the other hand, if we only specify the object's model context (class and property name), but leave the binding to the policy framework (for example, using policy roles), then greater flexibility results for either general or element-specific rules.

For example, an element-specific rule is obtained by a condition (variable/operator/value triplet) that defines, for example, CIM LogicalDevice DeviceID="12345". Alternately, if a PolicyRule's PolicyRoles is "edge device" and your condition (variable/operator/value triplet) is Status="Error", then a general rule results for all edge devices in error.

Refer to Section 5.10 for the formal definition of the class PolicyExplicitVariable.

4.8.7. Implicitly Bound Policy Variables

Implicitly bound policy variables define the data type and semantics of a variable. This determines how the variable is bound to a value in a condition clause. Further instructions are provided for specifying data type and/or value constraints for implicitly bound variables.

Implicitly bound variables can be interpreted by different sub-models to mean different things, depending on the particular context in which they are used. For example, an implicitly bound variable named "SourceIP" may be interpreted by a QoS policy information model to denote the source address field in the IP header of a packet if a device is configured to select certain packets for particular treatment. The same variable may be bound to the sender address delivered by a RSVP PATH message for a decision by a policy server. It is incumbent upon the particular domain-specific information model to provide full and unambiguous interpretation details (binding rules, type and value constraints) for the implicitly bound variables it uses.

PCIMe introduces an abstract class, PolicyImplicitVariable, to model implicitly bound variables. This class is derived from the abstract class PolicyVariable also defined in PCIMe. Each of the implicitly bound variables introduced by PCIMe (and those that are introduced by domain-specific sub-models) MUST be derived from the PolicyImplicitVariable class. The rationale for using this mechanism for modeling is explained below in Section 4.8.9.

A domain-specific policy information model that extends PCIMe may define additional implicitly bound variables either by deriving them directly from the class PolicyImplicitVariable, or by further refining an existing variable class such as SourcePort. When refining a class such as SourcePort, existing binding rules, type or value constraints may be narrowed.

4.8.8. Structure and Usage of Pre-Defined Variables

A class derived from PolicyImplicitVariable to model a particular implicitly bound variable SHOULD be constructed so that its name depicts the meaning of the variable. For example, a class defined to model the source port of a TCP/UDP flow SHOULD be named 'SourcePort'.

PCIMe defines one association and one general-purpose mechanism that together characterize each of the implicitly bound variables that it introduces:

1. The PolicyValueConstraintInVariable association defines the set of value classes that could be matched to this variable.

2. The list of constraints on the values that the PolicyVariable can hold (i.e., values that the variable must match) are defined by the appropriate properties of an associated PolicyValue class.

In the example presented above, a PolicyImplicitVariable represents the SourcePort of incoming traffic. The ValueTypes property of an instance of this class will hold the class name PolicyIntegerValue. This by itself constrains the data type of the SourcePort instance to be an integer. However, we can further constrain the particular values that the SourcePort variable can hold by entering valid ranges in the IntegerList property of the PolicyIntegerValue instance (0 - 65535 in this document).

The combination of the VariableName and the PolicyValueConstraintInVariable association provide a consistent and extensible set of metadata that define the semantics of variables that are used to form policy conditions. Since the PolicyValueConstraintInVariable association points to another class, any of the properties in the PolicyValue class can be used to constrain values that the PolicyImplicitVariable can hold. For example:

- o The ValueTypes property can be used to ensure that only proper classes are used in the expression. For example, the SourcePort variable will not be allowed to ever be of type PolicyIPv4AddrValue, since source ports have different semantics than IP addresses and may not be matched. However, integer value types are allowed as the property ValueTypes holds the string "PolicyIntegerValue", which is the class name for integer values.

- o The PolicyValueConstraintInVariable association also ensures that variable-specific semantics are enforced (e.g., the SourcePort variable may include a constraint association to a value object defining a specific integer range that should be matched).

4.8.9. Rationale for Modeling Implicit Variables as Classes

An implicitly bound variable can be modeled in one of several ways, including a single class with an enumerator for each individual implicitly bound variable and an abstract class extended for each individual variable. The reasons for using a class inheritance mechanism for specifying individual implicitly bound variables are these:

1. It is easy to extend. A domain-specific information model can easily extend the PolicyImplicitVariable class or its subclasses to define domain-specific and context-specific variables. For example, a domain-specific QoS policy information model may introduce an implicitly bound variable class to model applications by deriving a qosApplicationVariable class from the PolicyImplicitVariable abstract class.

2. Introduction of a single structural class for implicitly bound variables would have to include an enumerator property that contains all possible individual implicitly bound variables. This means that a domain-specific information model wishing to introduce an implicitly bound variable must extend the enumerator itself. This results in multiple definitions of the same class, differing in the values available in the enumerator class. One definition, in this document, would include the common implicitly bound variables' names, while a second definition, in the domain-specific information model document, may include additional values ('qosApplicationVariable' in the example above). It wouldnÆt even be obvious to the application developer that multiple class definitions existed. It would be harder still for the application developer to actually find the correct class to use.

3. In addition, an enumerator-based definition would require each additional value to be registered with IANA to ascertain adherence to standards. This would make the process cumbersome.

4. A possible argument against the inheritance mechanism would cite the fact that this approach results in an explosion of class definitions compared to an enumerator class, which only introduces a single class. While, by itself, this is not a strike against the approach, it may be argued that data models implemented, which are mapped to this information model, may be more difficult to optimize for applications. This argument is rejected on the grounds that application optimization is of lesser value for an information model than clarity and ease of extension. In addition, it is hard to claim that the inheritance model places an absolute burden on the optimization. For example, a data model may still use enumeration to denote instances of pre-defined

variables and claim PCIMe compliance, as long as the data moel can be mapped correctly to the definitions specified in this document. Furthermore, the very nature of implicitly bound variables is to be interpreted in context. This means that unless an additional variable is required by a sub-model (in which case both approaches result in some overhead), there's an upper limit on the class explosion. After all, once properly documented, no need exists for a sub-model to add a class definition. The implementation needs only to cite and use the PCIMe variable, but impose the documented context-dependent semantics.

4.8.10. Policy Values

The abstract class PolicyValue is used for modeling values and constants used in policy conditions. Different value types are derived from this class, to represent the various attributes required. Extensions of the abstract class PolicyValue, defined in this document, provide a list of values for representing basic network attributes. Values can be used to represent constants as named values. Named values can be kept in a reusable policy container to be reused by multiple conditions. Examples of constants include well-known ports, well-known protocols, server addresses, and other similar concepts.

The PolicyValue subclasses define three basic types of values: scalars, ranges and sets. For example, a well-known port number could be defined using the PolicyIntegerValue class, defining a single value (80 for HTTP), a range (80-88), or a set (80, 82, 8080) of ports, respectively. For details, please see the class definition for each value type in Section 5.12 of this document.

PCIMe defines the following subclasses of the abstract class PolicyValue:

Classes for general use:

- PolicyStringValue,
- PolicyIntegerValue,
- PolicyBitStringValue
- PolicyBooleanValue.

Classes for layer 3 Network values:

- PolicyIPv4AddrValue,
- PolicyIPv6AddrValue.

Classes for layer 2 Network values:

- PolicyMACAddrValue.

For details, please see the class definition section of each class in Section 5.14 of this document.

4.9. Packet Filtering

In addition to filling in the holes in the overall Policy infrastructure, PCIMe proposes a single mechanism for expressing packet filters in policy conditions. This is being done in response to concerns that even though the initial "wave" of submodels derived from PCIM were all filtering on

IP packets, each was doing it in a slightly different way. PCIMe proposes a common way to express IP packet filters. The following figure illustrates how packet-filtering conditions are expressed in PCIMe.

```
                 +---------------------------------+
                 | CompoundFilterCondition         |
                 |   - IsMirrored   boolean        |
                 |   - ConditionListType (DNF|CNF) |
                 +---------------------------------+
                  +               +               +
                  +               +               +
                  +               +               +
              SimplePC        SimplePC        SimplePC
              *      @        *      @        *      @
              *      @        *      @        *      @
              *      @        *      @        *      @
  FlowDirection    "In"     SrcIP  <addr1>  DstIP  <addr2>

Aggregation Legend:
  ++++  PolicyConditionInPolicyCondition
  ****  PolicyVariableInSimplePolicyCondition
  @@@@  PolicyValueInSimplePolicyCondition
```

Figure 7.   Packet Filtering in Policy Conditions

In Figure 7, each SimplePolicyCondition represents a single field to be filtered on: Source IP address, Destination IP address, Source port, etc. An additional SimplePolicyCondition indicates the direction that a packet is traveling on an interface: inbound or outbound. Because of the FlowDirection condition, care must be taken in aggregating a set of SimplePolicyConditions into a CompoundFilterCondition. Otherwise, the resulting CompoundPolicyCondition may match all inbound packets, or all outbound packets, when this is probably not what was intended.

Individual SimplePolicyConditions may be negated when they are aggregated by a CompoundFilterCondition.

CompoundFilterCondition is a subclass of CompoundPolicyCondition. It introduces one additional property, the Boolean property IsMirrored. The purpose of this property is to allow a single CompoundFilterCondition to match packets traveling in both directions on a higher-level connection such as a TCP session. When this property is TRUE, additional packets match a filter, beyond those that would ordinarily match it. An example will illustrate how this property works.

Suppose we have a CompoundFilterCondition that aggregates the following three filters, which are ANDed together:

- o   FlowDirection = "In"

- o Source IP = 9.1.1.1
- o Source Port = 80

Regardless of whether IsMirrored is TRUE or FALSE, inbound packets will match this CompoundFilterCondition if their Source IP address = 9.1.1.1 and their Source port = 80. If IsMirrored is TRUE, however, an outbound packet will also match the CompoundFilterCondition if its Destination IP address = 9.1.1.1 and its Destination port = 80.

IsMirrored "flips" the following Source/Destination packet header fields:

- FlowDirection "In" / FlowDirection "Out"
- Source IP address / Destination IP address
- Source port / Destination port
- Source MAC address / Destination MAC address
- Source [layer-2] SAP / Destination [layer-2] SAP.

5. Class Definitions

The following definitions supplement those in PCIM itself. PCIM definitions that are not DEPRECATED here are still current parts of the overall Policy Core Information Model.

5.1. The Abstract Class "PolicySet"

PolicySet is an abstract class that may group policies into a structured set of policies.

```
NAME             PolicySet
DESCRIPTION      An abstract class that represents a set of policies
                 that form a coherent set.  The set of contained
                 policies has a common decision strategy and a common
                 set of policy roles.  Subclasses include PolicyGroup
                 and PolicyRule.
DERIVED FROM     Policy
ABSTRACT         TRUE
PROPERTIES       PolicyDecisionStrategy
                 PolicyRoles
```

The PolicyDecisionStrategy property specifies the evaluation method for policy groups and rules contained within the policy set.

```
NAME             PolicyDecisionStrategy
DESCRIPTION      The evaluation method used for policies contained in
                 the PolicySet.  FirstMatching enforces the actions of
                 the first rule that evaluates to TRUE; AllMatching
                 enforces the actions of all rules that evaluate to
                 TRUE.
SYNTAX           uint16
VALUES           1 [FirstMatching], 2 [AllMatching]
DEFAULT VALUE    1 [FirstMatching]
```

The definition of PolicyRoles is unchanged from PCIM. It is, however, moved from the class Policy up to the superclass PolicySet.

5.2. Updates to PCIM's Class "PolicyGroup"

The PolicyGroup class is modified to be derived from PolicySet.

```
NAME             PolicyGroup
DESCRIPTION      A container for a set of related PolicyRules and
                 PolicyGroups.
DERIVED FROM     PolicySet
ABSTRACT         FALSE
PROPERTIES       (none)
```

5.3. Updates to PCIM's Class "PolicyRule"

The PolicyRule class is modified to be derived from PolicySet, and to deprecate the use of Priority in the rule. PolicyRoles is now inherited from the parent class PolicySet. Finally, a new property ExecutionStrategy is introduced, paralleling the property of the same name in the class CompoundPolicyAction.

```
NAME             PolicyRule
DESCRIPTION      The central class for representing the "If Condition
                 then Action" semantics associated with a policy rule.
DERIVED FROM     PolicySet
ABSTRACT         FALSE
PROPERTIES       Enabled
                 ConditionListType
                 RuleUsage
                 Priority DEPRECATED FOR PolicySetComponent.Priority
                 Mandatory
                 SequencedActions
                 ExecutionStrategy
```

The property ExecutionStrategy defines the execution strategy to be used upon the sequenced actions aggregated by this PolicyRule. (An equivalent ExecutionStrategy property is also defined for the CompoundPolicyAction class, to provide the same indication for the sequenced actions aggregated by a CompoundPolicyAction.) This draft defines four execution strategies:

```
Mandatory Do all û execute ALL actions that are part of the modeled
                   set.  If one or more of the actions cannot be
                   executed, none of the actions should be executed.
Do until success û execute actions according to predefined order, until
                   successful execution of a single action.
Do All -           execute ALL actions which are part of the modeled
                   set, according to their predefined order. Continue
                   doing this, even if one or more of the actions
                   fails.
```

Do until Failure - execute actions according to predefined order, until the first failure in execution of a single sub-action.

The property definition is as follows:

```
NAME             ExecutionStrategy
DESCRIPTION      An enumeration indicating how to interpret the action
                 ordering for the actions aggregated by this
                 PolicyRule.
SYNTAX           uint16 (ENUM, {1=Mandatory Do All, 2=Do Until Success,
                 3=Do All, 4=Do Until Failure} )
DEFAULT VALUE    Do All (3)
```

5.4. The Class "SimplePolicyCondition"

A simple policy condition is composed of an ordered triplet:

<Variable> MATCH <Value>

No formal modeling of the MATCH operator is provided. The 'match' relationship is implied. Such simple conditions are evaluated by answering the question:

Does <variable> match <value>?

The 'match' relationship is to be interpreted by analyzing the variable and value instances associated with the simple condition.

Simple conditions are building blocks for more complex Boolean Conditions, modeled by the CompoundPolicyCondition class.

The SimplePolicyCondition class is derived from the PolicyCondition class defined in PCIM.

A variable and a value must be associated with a simple condition to make it a meaningful condition, using, respectively, the aggregations PolicyVariableInSimplePolicyCondition and PolicyValueInSimplePolicyCondition.

The class definition is as follows:

```
NAME             SimplePolicyCondition
DERIVED FROM     PolicyCondition
ABSTRACT         False
PROPERTIES       (none)
```

5.5. The Class "CompoundPolicyCondition"

This class represents a compound policy condition, formed by aggregation of simpler policy conditions.

```
NAME             CompoundPolicyCondition
```

DESCRIPTION   A subclass of PolicyCondition that introduces the ConditionListType property, used for assigning DNF / CNF semantics to subordinate policy conditions.

DERIVED FROM   PolicyCondition

```
   ABSTRACT          FALSE
   PROPERTIES        ConditionListType
```

The ConditionListType property is used to specify whether the list of policy conditions associated with this compound policy condition is in disjunctive normal form (DNF) or conjunctive normal form (CNF). If this property is not present, the list type defaults to DNF. The property definition is as follows:

```
   NAME              ConditionListType
   DESCRIPTION       Indicates whether the list of policy conditions
                     associated with this policy rule is in disjunctive
                     normal form (DNF) or conjunctive normal form (CNF).
   SYNTAX            uint16
   VALUES            DNF(1), CNF(2)
   DEFAULT VALUE     DNF(1)
```

5.6. The Class "CompoundFilterCondition"

This subclass of CompoundPolicyCondition introduces one additional property, the boolean IsMirrored. This property turns on or off the "flipping" of corresponding source and destination fields in a filter specification.

```
   NAME              CompoundFilterCondition
   DESCRIPTION       A subclass of CompoundPolicyCondition that introduces
                     the IsMirrored property.
   DERIVED FROM      CompoundPolicyCondition
   ABSTRACT          FALSE
   PROPERTIES        IsMirrored
```

The IsMirrored property indicates whether packets that "mirror" a compound filter condition should be treated as matching the filter. The property definition is as follows:

```
   NAME              IsMirrored
   DESCRIPTION       Indicates whether packets that mirror the specified
                     filter are to be treated as matching the filter.
   SYNTAX            boolean
   DEFAULT VALUE     FALSE
```

5.7. The Class "SimplePolicyAction"

The SimplePolicyAction class models the elementary set operation. "SET <variable> TO <value>". The set operator MUST overwrite an old value of the variable.

Two aggregations are used in order to create the pair <variable> <value>.

The aggregation PolicyVariableInSimplePolicyAction relates a SimplePolicyAction to a single variable instance. Similarly, the aggregation PolicyValueInSimplePolicyAction relates a SimplePolicyAction to a single value instance. Both aggregations are defined in this document.

```
NAME             SimplePolicyAction
DESCRIPTION      A subclass of PolicyAction that introduces the notion
                 of "SET variable TO value".
DERIVED FROM     PolicyAction
ABSTRACT         FALSE
PROPERTIES       (none)
```

5.8. The Class "CompoundPolicyAction"

The CompoundPolicyAction class is used to represent an expression consisting of an ordered sequence of action terms. Each action term is represented as a subclass of the PolicyAction class, defined in [PCIM]. Compound actions are constructed by associating dependent action terms together using the PolicyActionInPolicyAction aggregation.

The class definition is as follows:

```
NAME             CompoundPolicyAction
DESCRIPTION      A class for representing sequenced action terms.  Each
                 action term is defined to be a subclass of the
                 PolicyAction class.
DERIVED FROM     PolicyAction
ABSTRACT         FALSE
PROPERTIES       SequencedActions
                 ExecutionStrategy
```

This is a concrete class, and is therefore directly instantiable.

The Property SequencedActions is identical to the SequencedActions property defined in PCIM for the class PolicyRule.

The property ExecutionStrategy defines the execution strategy to be used upon the sequenced actions associated with this compound action. (An equivalent ExecutionStrategy property is also defined for the PolicyRule class, to provide the same indication for the sequenced actions associated with a PolicyRule.) This draft defines four execution strategies:

```
Mandatory Do all û execute ALL actions that are part of the modeled
                   set.  If one or more of the sub-actions cannot be
                   executed, none of the actions should be executed.
Do until success û execute actions according to predefined order, until
                   successful execution of a single sub-action.
Do All -           execute ALL actions which are part of the modeled
                   set, according to their predefined order. Continue
                   doing this, even if one or more of the sub-actions
```

fails.

Do until Failure - execute actions according to predefined order, until the first failure in execution of a single sub-action.

The property definition is as follows:

```
NAME             ExecutionStrategy
DESCRIPTION      An enumeration indicating how to interpret the action
                 ordering for the actions aggregated by this
                 CompoundPolicyAction.
SYNTAX           uint16 (ENUM, {1=Mandatory Do All, 2=Do Until Success,
                 3=Do All, 4=Do Until Failure} )
DEFAULT VALUE    Do All (3)
```

5.9. The Abstract Class "PolicyVariable"

Variables are used for building individual conditions. The variable specifies the property of a flow or an event that should be matched when evaluating the condition. However, not every combination of a variable and a value creates a meaningful condition. For example, a source IP address variable can not be matched against a value that specifies a port number. A given variable selects the set of matchable value types.

A variable can have constraints that limit the set of values within a particular value type that can be matched against it in a condition. For example, a source-port variable limits the set of values to represent integers to the range of 0-65535. Integers outside this range cannot be matched to the source-port variable, even though they are of the correct data type. Constraints for a given variable are indicated through the PolicyValueConstraintInVariable association.

The PolicyVariable is an abstract class. Implicit and explicit context variable classes are defined as sub classes of the PolicyVariable class. A set of implicit variables is defined in this document as well.

The class definition is as follows:

```
NAME             PolicyVariable
DERIVED FROM     Policy
ABSTRACT         TRUE
PROPERTIES       (none)
```

5.10. The Class "PolicyExplicitVariable"

Explicitly defined policy variables are evaluated within the context of the CIM Schema and its modeling constructs. The PolicyExplicitVariable class indicates the exact model property to be evaluated or manipulated.

The class definition is as follows:

```
NAME             PolicyExplicitVariable
DERIVED FROM     PolicyVariable
```

```
ABSTRACT          False
PROPERTIES        ModelClass, ModelProperty
```

5.10.1. The Single-Valued Property "ModelClass"

This property is a string specifying the class name whose property is evaluated or set as a PolicyVariable. The property is defined as follows:

```
NAME              ModelClass
SYNTAX            String
```

5.10.2. The Single-Valued Property ModelProperty

This property is a string specifying the property name, within the ModelClass, which is evaluated or set as a PolicyVariable. The property is defined as follows:

```
NAME              ModelProperty
SYNTAX            String
```

5.11. The Abstract Class "PolicyImplicitVariable"

Implicitly defined policy variables are evaluated outside of the context of the CIM Schema and its modeling constructs. Subclasses specify the data type and semantics of the PolicyVariables.

Interpretation and evaluation of a PolicyImplicitVariable can vary, depending on the particular context in which it is used. For example, a "SourceIP" address may denote the source address field of an IP packet header, or the sender address delivered by an RSVP PATH message.

The class definition is as follows:

```
NAME              PolicyImplicitVariable
DERIVED FROM      PolicyVariable
ABSTRACT          True
PROPERTIES        ValueTypes[ ]
```

5.11.1. The Multi-Valued Property "ValueTypes"

This property is a set of strings specifying an unordered list of possible value/data types that can be used in simple conditions and actions, with this variable. The value types are specified by their class names (subclasses of PolicyValue such as PolicyStringValue). The list of class names enables an application to search on a specific name, as well as to ensure that the data type of the variable is of the correct type.

The list of default ValueTypes for each subclass of PolicyImplicitVariable is specified within that variable's definition.

The property is defined as follows:

```
NAME              ValueTypes
SYNTAX            String
```

5.12. Subclasses of "PolicyImplicitVariable" Specified in PCIMe

The following subclasses of PolicyImplicitVariable are defined in PCIMe.

5.12.1. The Class "PolicySourceIPVariable"

```
NAME              PolicySourceIPVariable
DESCRIPTION       The source IP address.

                  ALLOWED VALUE TYPES:
                    - PolicyIPv4AddrValue
                    - PolicyIPv6AddrValue

DERIVED FROM      PolicyImplicitVariable
ABSTRACT          FALSE
PROPERTIES        (none)
```

5.12.2. The Class "PolicyDestinationIPVariable"

```
NAME              PolicyDestinationIPVariable
DESCRIPTION       The destination IP address.

                  ALLOWED VALUE TYPES:
                    - PolicyIPv4AddrValue
                    - PolicyIPv6AddrValue

DERIVED FROM      PolicyImplicitVariable
ABSTRACT          FALSE
PROPERTIES        (none)
```

5.12.3. The Class "PolicySourcePortVariable"

```
NAME              PolicySourcePortVariable
DESCRIPTION       Ports are defined as the abstraction that transport
                  protocols use to distinguish among multiple
                  destinations within a given host computer.  For TCP
                  and UDP flows, the PolicySourcePortVariable is
                  logically bound to the source port field.

                  ALLOWED VALUE TYPES:
                    - PolicyIntegerValue
                    - PolicyBitStringValue

DERIVED FROM      PolicyImplicitVariable
```

ABSTRACT	FALSE
PROPERTIES	(none)

5.12.4. The Class "PolicyDestinationPortVariable"

```
NAME              PolicyDestinationPortVariable
DESCRIPTION       Ports are defined as the abstraction that transport
                  protocols use to distinguish among multiple
                  destinations within a given host computer.  For TCP
                  and UDP flows, the PolicyDestinationPortVariable is
                  logically bound to the destination port field.

                  ALLOWED VALUE TYPES:
                    - PolicyIntegerValue
                    - PolicyBitStringValue

DERIVED FROM      PolicyImplicitVariable
ABSTRACT          FALSE
PROPERTIES        (none)
```

5.12.5. The Class "PolicyIPProtocolVariable"

```
NAME              PolicyIPProtocolVariable
DESCRIPTION       The IP protocol number.

                  ALLOWED VALUE TYPES:
                    - PolicyIntegerValue
                    - PolicyBitStringValue

DERIVED FROM      PolicyImplicitVariable
ABSTRACT          FALSE
PROPERTIES        (none)
```

5.12.6. The Class "PolicyIPVersionVariable"

```
NAME              PolicyIPVersionVariable
DESCRIPTION       The IP version number.  The well-known values are 4
                  and 6.

                  ALLOWED VALUE TYPES:
                    - PolicyIntegerValue
                    - PolicyBitStringValue

DERIVED FROM      PolicyImplicitVariable
ABSTRACT          FALSE
PROPERTIES        (none)
```

5.12.7. The Class "PolicyIPToSVariable"

```
NAME              PolicyIPToSVariable
DESCRIPTION       The IP TOS octet.
```

ALLOWED VALUE TYPES:
  - PolicyIntegerValue
  - PolicyBitStringValue

```
DERIVED FROM     PolicyImplicitVariable
ABSTRACT         FALSE
PROPERTIES       (none)
```

5.12.8. The Class "PolicyDSCPVariable"

```
NAME             PolicyDSCPVariable
DESCRIPTION      The 6 bit Differentiated Service Code Point.

                 ALLOWED VALUE TYPES:
                   - PolicyIntegerValue
                   - PolicyBitStringValue

DERIVED FROM     PolicyImplicitVariable
ABSTRACT         FALSE
PROPERTIES       (none)
```

5.12.9. The Class "PolicySourceMACVariable"

```
NAME             PolicySourceMACVariable
DESCRIPTION      The source MAC address.

                 ALLOWED VALUE TYPES:
                   - PolicyMACAddrValue

DERIVED FROM     PolicyImplicitVariable
ABSTRACT         FALSE
PROPERTIES       (none)
```

5.12.10. The Class "PolicyDestinationMACVariable"

```
NAME             PolicyDestinationMACVariable
DESCRIPTION      The destination MAC address.

                 ALLOWED VALUE TYPES:
                   - PolicyMACAddrValue

DERIVED FROM     PolicyImplicitVariable
ABSTRACT         FALSE
PROPERTIES       (none)
```

5.12.11. The Class "PolicyVLANVariable"

```
NAME             PolicyVLANVariable
DESCRIPTION      The virtual Bridged Local Area Network Identifier, a
                 12-bit field as defined in the IEEE 802.1q standard.

                 ALLOWED VALUE TYPES:
```

- PolicyIntegerValue
- PolicyBitStringValue

```
    DERIVED FROM     PolicyImplicitVariable
    ABSTRACT         FALSE
    PROPERTIES       (none)
```

5.12.12. The Class "PolicyCoSVariable"

```
    NAME             PolicyCoSVariable
    DESCRIPTION      Class of Service, a 3-bit field, used in the layer 2
                     header to select the forwarding treatment.  Bound to
                     the IEEE 802.1q user-priority field.

                     ALLOWED VALUE TYPES:
                       - PolicyIntegerValue
                       - PolicyBitStringValue

    DERIVED FROM     PolicyImplicitVariable
    ABSTRACT         FALSE
    PROPERTIES       (none)
```

5.12.13. The Class "PolicyEthertypeVariable"

```
    NAME             PolicyEthertypeVariable
    DESCRIPTION      The Ethertype protocol number of Ethernet frames.

                     ALLOWED VALUE TYPES:
                       - PolicyIntegerValue
                       - PolicyBitStringValue

    DERIVED FROM     PolicyImplicitVariable
    ABSTRACT         FALSE
    PROPERTIES       (none)
```

5.12.14. The Class "PolicySourceSAPVariable"

```
    NAME             PolicySourceSAPVariable
    DESCRIPTION      The Source SAP number.

                     ALLOWED VALUE TYPES:
                       - PolicyIntegerValue
                       - PolicyBitStringValue

    DERIVED FROM     PolicyImplicitVariable
    ABSTRACT         FALSE
    PROPERTIES       (none)
```

5.12.15. The Class "PolicyDestinationSAPVariable"

```
    NAME             PolicyDestinationSAPVariable
    DESCRIPTION      The Destination SAP number.
```

ALLOWED VALUE TYPES:
  - PolicyIntegerValue

```
                      - PolicyBitStringValue

    DERIVED FROM      PolicyImplicitVariable
    ABSTRACT          FALSE
    PROPERTIES        (none)
```

5.12.16. The Class "PolicySNAPVariable"

```
    NAME              PolicySNAPVariable
    DESCRIPTION       The protocol number over a SNAP SAP encapsulation.

                      ALLOWED VALUE TYPES:
                        - PolicyIntegerValue
                        - PolicyBitStringValue

    DERIVED FROM      PolicyImplicitVariable
    ABSTRACT          FALSE
    PROPERTIES        (none)
```

5.12.17. The Class "PolicyFlowDirectionVariable"

```
    NAME              PolicyFlowDirectionVariable
    DESCRIPTION       The direction of a flow relative to a network element.
                      Direction may be "IN" and/or "OUT".

                      ALLOWED VALUE TYPES:
                        - PolicyStringValue

    DERIVED FROM      PolicyImplicitVariable
    ABSTRACT          FALSE
    PROPERTIES        (none)
```

To match on both inbound and outbound flows, the associated PolicyStringValue object has two entries in its StringList property: "IN" and "OUT".

5.13. The Abstract Class "PolicyValue"

This is an abstract class that serves as the base class for all subclasses that are used to define value objects in the PCIMe. It is used for defining values and constants used in policy conditions. The class definition is as follows:

```
    NAME              PolicyValue
    DERIVED FROM      Policy
    ABSTRACT          True
    PROPERTIES        (none)
```

5.14. Subclasses of "PolicyValue" Specified in PCIMe

The following subsections contain the PolicyValue subclasses defined in PCIMe. Additional subclasses may be defined in models derived from PCIMe.

5.14.1. The Class "PolicyIPv4AddrValue"

This class is used to provide a list of IPv4Addresses, hostnames and address range values to be matched against in a policy condition. The class definition is as follows:

```
NAME             PolicyIPv4AddrValue
DERIVED FROM     PolicyValue
ABSTRACT         False
PROPERTIES       IPv4AddrList[ ]
```

The IPv4AddrList property provides an unordered list of strings, each specifying a single IPv4 address, a hostname, or a range of IPv4 addresses, according to the ABNF definition [8] of an IPv4 address, as specified below:

```
IPv4address = 1*3DIGIT "." 1*3DIGIT "." 1*3DIGIT "." 1*3DIGIT
IPv4prefix  = IPv4address "/" 1*2DIGIT
IPv4range = IPv4address"-"IPv4address
IPv4maskedaddress = IPv4address","IPv4address
Hostname (as defined in [9])
```

In the above definition, each string entry is either:

1. A single Ipv4address in dot notation, as defined above. Example: 121.1.1.2

2. An IPv4prefix address range, as defined above, specified by an address and a prefix length, separated by "/". Example: 2.3.128.0/15

3. An IPv4range address range defined above, specified by a starting address in dot notation and an ending address in dot notation, separated by "-". The range includes all addresses between the range's starting and ending addresses, including these two addresses. Example: 1.1.22.1-1.1.22.5

4. An IPv4maskedaddress address range, as defined above, specified by an address and mask. The address and mask are represented in dot notation, separated by a comma ",". The masked address appears before the comma, and the mask appears after the comma. Example: 2.3.128.0,255.255.248.0.

5. A single Hostname. The Hostname format follows the guidelines and restrictions specified in [9]. Example: www.bigcompany.com.

The property definition is as follows:

```
NAME              IPv4AddrList
SYNTAX            String
FORMAT            IPv4address | IPv4prefix | IPv4range |
                  IPv4maskedaddress | hostname
```

5.14.2. The Class "PolicyIPv6AddrValue

This class is used to define a list of IPv6 addresses, hostnames, and address range values. The class definition is as follows:

```
NAME              PolicyIPv6AddrValue
DERIVED FROM      PolicyValue
ABSTRACT          False
PROPERTIES        IPv6AddrList[ ]
```

The property IPv6AddrList provides an unordered list of strings, each specifying an IPv6 address, a hostname, or a range of IPv6 addresses. IPv6 address format definition uses the standard address format defined in [10]. The ABNF definition [8] as specified in [10] is:

```
IPv6address = hexpart [ ":" IPv4address ]
IPv4address = 1*3DIGIT "." 1*3DIGIT "." 1*3DIGIT "." 1*3DIGIT
IPv6prefix  = hexpart "/" 1*2DIGIT
hexpart = hexseq | hexseq "::" [ hexseq ] | "::" [ hexseq ]
hexseq  = hex4 *( ":" hex4)
hex4    = 1*4HEXDIG
IPv6range = IPv6address"-"IPv6address
IPv6maskedaddress = IPv6address","IPv6address
Hostname (as defines in [NAMES])
```

Each string entry is either:

1. A single IPv6address as defined above.

2. A single Hostname. Hostname format follows guidelines and restrictions specified in [9].

3. An IPv6range address range, specified by a starting address in dot notation and an ending address in dot notation, separated by "-". The range includes all addresses between the range's starting and ending addresses, including these two addresses.

4. An IPv4maskedaddress address range defined above specified by an address and mask. The address and mask are represented in dot notation separated by a comma ",".

5. A single IPv6prefix as defined above.

5.14.3. The Class "PolicyMACAddrValue"

This class is used to define a list of MAC addresses and MAC address range values. The class definition is as follows:

```
NAME             PolicyMACAddrValue
DERIVED FROM     PolicyValue
ABSTRACT         False
PROPERTIES       MACAddrList[ ]
```

The property MACAddrList provides an unordered list of strings, each specifying a MAC address or a range of MAC addresses. The 802 MAC address canonical format is used. The ABNF definition [8] is:

```
MACaddress  = 1*4HEXDIG ":" 1*4HEXDIG ":" 1*4HEXDIG
MACmaskedaddress = MACaddress","MACaddress
```

Each string entry is either:

1. A single MAC address. Example: 0000:00A5:0000

2. A MACmaskedaddress address range defined specified by an address and mask. The mask specifies the relevant bits in the address. Example: 0000:00A5:0000,FFFF:FFFF:0000 defines a range of MAC addresses in which the first four octets are equal to 0000:00A5.

The property definition is as follows:

```
NAME             MACAddrList
SYNTAX           String
FORMAT           MACaddress | MACmaskedaddress
```

5.14.4. The Class "PolicyStringValue"

This class is used to represent a single string value, or a set of string values. Each value can have wildcards. The class definition is as follows:

```
NAME             PolicyStringValue
DERIVED FROM     PolicyValue
ABSTRACT         False
PROPERTIES       StringList[ ]
```

The property StringList provides an unordered list of strings, each representing a single string with wildcards. The asterisk character "*" is used as a wildcard, and represents an arbitrary substring replacement. For example, the value "abc*def" matches the string "abcxyzdef", and the value "abc*def*" matches the string "abcxxxdefyyyzzz". The syntax definition is identical to the substring assertion syntax defined in [11]. If the asterisk character is required as part of the string value itself, it MUST be quoted as described in section 4.3 of [11].

The property definition is as follows:

```
NAME                 StringList
SYNTAX               String
```

5.14.5. The Class "PolicyBitStringValue"

This class is used to represent a single bit string value, or a set of bit string values. The class definition is as follows:

```
NAME             PolicyBitStringValue
DERIVED FROM     PolicyValue
ABSTRACT         False
PROPERTIES       BitStringList[ ]
```

The property BitStringList provides an unordered list of strings, each representing a single bit string or a set of bit strings. The number of bits specified SHOULD equal the number of bits of the expected variable. For example, for a one-octet variable, 8 bits should be specified. If the variable does not have a fixed length, the bit string should be matched against the variable's most significant bit string. The formal definition of a bit string is:

```
binary-digit = "0" / "1"
bitString = 1*binary-digit
maskedBitString = bitString","bitString
```

Each string entry is either:

1. A single bit string. Example: 00111010

2. A range of bit strings specified using a bit string and a bit mask. The bit string and mask fields have the same number of bits specified. The mask bit string specifies the significant bits in the bit string value. For example, 110110, 100110 and 110111 would match the maskedBitString 100110,101110 but 100100 would not.

The property definition is as follows:

```
NAME             BitStringList
SYNTAX           String
FORMAT           bitString | maskedBitString
```

5.14.6. The Class "PolicyIntegerValue"

This class provides a list of integer and integer range values. Integers of arbitrary sizes can be represented. The class definition is as follows:

```
NAME              PolicyIntegerValue
DERIVED FROM      PolicyValue
```

```
ABSTRACT          False
PROPERTIES        IntegerList[ ]
```

The property IntegerList provides an unordered list of integers and integer range values, represented as strings. The format of this property takes one of the following forms:

1. An integer value.

2. A range of integers. The range is specified by a starting integer and an ending integer, separated by '-'. The starting integer MUST be less than or equal to the ending integer. The range includes all integers between the starting and ending integers, including these two integers. Care must be taken in reading integer ranges involving negative integers, since the unary minus and the range indicator are the same character '-'.

To represent a range of integers that is not bounded, the reserved words -INFINITY and/or INFINITY can be used in place of the starting and ending integers.

The ABNF definition [8] is:

```
integer = [-]1*DIGIT | "INFINITY" | "-INFINITY"
integerrange = integer"-"integer
```

Using ranges, the operators greater-than, greater-than-or-equal-to, less-than, and less-than-or-equal-to can be expressed. For example, "X is-greater-than 5" (where X is an integer) can be translated to "X matches 6-INFINITY". This enables the match condition semantics of the operator for the SimplePolicyCondition class to be kept simple (i.e., just the value "match").

The property definition is as follows:

```
NAME              IntegerList
SYNTAX            String
FORMAT            integer | integerrange
```

5.14.7. The Class "PolicyBooleanValue"

This class is used to represent a Boolean (TRUE/FALSE) value. The class definition is as follows:

```
NAME              PolicyBooleanValue
DERIVED FROM      PolicyValue
ABSTRACT          False
PROPERTIES        BooleanValue
```

The property definition is as follows:

```
NAME              BooleanValue
```

```
   SYNTAX           boolean
```

5.15. The Class "PolicyRoleCollection"

This class represents a collection of managed elements that share a common role. The PolicyRoleCollection always exists in the context of a system, specified using the PolicyRoleCollectionInSystem association. The value of the PolicyRole property in this class specifies the role, and can be matched with the value(s) in the PolicyRoles array in PolicyRules and PolicyGroups. ManagedElements that share the role defined in this collection are aggregated into the collection via the association ElementInPolicyRoleCollection.

```
   NAME             PolicyRoleCollection
   DESCRIPTION      A subclass of the CIM Collection class used to group
                    together managed elements that share a role.
   DERIVED FROM     Collection
   ABSTRACT         FALSE
   PROPERTIES       PolicyRole
```

5.15.1. The Single-Valued Property "PolicyRole"

This property represents the role associated with a PolicyRoleCollection. The property definition is as follows:

```
   NAME             PolicyRole
   DESCRIPTION      A string representing the role associated with a
                    PolicyRoleCollection.
   SYNTAX           string
```

5.16. The Class "ReusablePolicyContainer"

The new class ReusablePolicyContainer is defined as follows:

```
   NAME             ReusablePolicyContainer
   DESCRIPTION      A class representing an administratively defined
                    container for reusable policy-related information.
                    This class does not introduce any additional
                    properties beyond those in its superclass AdminDomain.
                    It does, however, participate in a number of unique
                    associations.
   DERIVED FROM     AdminDomain
   ABSTRACT         FALSE
   PROPERTIES       (none)
```

5.17. Deprecation of PCIM's Class "PolicyRepository"

The class definition of PolicyRepository (from PCIM) is updated as follows, with an indication that the class has been deprecated. Note that when an element of the model is deprecated, its replacement element is identified explicitly.

```
NAME             PolicyRepository
DEPRECATED FOR   ReusablePolicyContainer
DESCRIPTION      A class representing an administratively defined
                 container for reusable policy-related information.
                 This class does not introduce any additional
                 properties beyond those in its superclass AdminDomain.
                 It does, however, participate in a number of unique
                 associations.
DERIVED FROM     AdminDomain
ABSTRACT         FALSE
PROPERTIES       (none)
```

6. Association and Aggregation Definitions

The following definitions supplement those in PCIM itself. PCIM definitions that are not DEPRECATED here are still current parts of the overall Policy Core Information Model.

6.1. The Abstract Aggregation "PolicySetComponent"

PolicySetComponent is a new abstract aggregation class that collects instances of PolicySet subclasses (PolicyGroups and PolicyRules) into coherent sets of policies.

```
NAME             PolicySetComponent
DESCRIPTION      An abstract class representing the components of a
                 policy set that have the same decision strategy, and
                 are prioritized within the set.
DERIVED FROM     PolicyComponent
ABSTRACT         TRUE
PROPERTIES       GroupComponent[ref PolicySet[0..n]]
                 PartComponent[ref PolicySet[0..n]]
                 Priority
```

The definition of the Priority property is unchanged from its previous definition in [PCIM].

```
NAME             Priority
DESCRIPTION      A non-negative integer for prioritizing this PolicySet
                 component relative to other components of the same
                 PolicySet.  A larger value indicates a higher
                 priority.
SYNTAX           uint16
DEFAULT VALUE    0
```

6.2. Update to PCIM's Aggregation "PolicyGroupInPolicyGroup"

The PolicyGroupInPolicyGroup aggregation class is modified to be derived from PolicySetComponent.

```
NAME              PolicyGroupInPolicyGroup
DESCRIPTION       A class representing the aggregation of PolicyGroups
                  by a higher-level PolicyGroup.
DERIVED FROM      PolicySetComponent
ABSTRACT          FALSE
PROPERTIES        GroupComponent[ref PolicyGroup[0..n]]
                  PartComponent[ref PolicyGroup[0..n]]
```

6.3. Update to PCIM's Aggregation "PolicyRuleInPolicyGroup"

The PolicyRuleInPolicyGroup aggregation class is modified to be derived from PolicySetComponent.

```
NAME              PolicyRuleInPolicyGroup
DESCRIPTION       A class representing the aggregation of PolicyRules by
                  a PolicyGroup.
DERIVED FROM      PolicySetComponent
ABSTRACT          FALSE
PROPERTIES        GroupComponent[ref PolicyGroup[0..n]]
                  PartComponent[ref PolicyRule[0..n]]
```

6.4. The Aggregation "PolicyGroupInPolicyRule"

A policy rule may aggregate one or more policy groups, via the PolicyGroupInPolicyRule aggregation. Grouping of policy groups and their subclasses into a policy rule is for administrative convenience, scalability and manageability, as it enables more complex policies to be constructed from multiple simpler policies.

Policy rules do not have to contain policy groups. In addition, a policy group may also be used by itself, without belonging to a policy rule, and policy rules may be individually aggregated by other policy rules by the PolicyRuleInPolicyRule aggregation. Note that it is assumed that this aggregation is used to form directed acyclic graphs and NOT ring structures.

The class definition for this aggregation is as follows:

```
NAME              PolicyGroupInPolicyRule
DERIVED FROM      PolicySetComponent
ABSTRACT          False
PROPERTIES        GroupComponent[ref PolicyRule[0..n]]
                  PartComponent[ref PolicyGroup[0..n]]
```

The reference property "GroupComponent" is inherited from PolicySetComponent, and overridden to become an object reference to a PolicyRule that contains one or more PolicyGroups. Note that for any single instance of the aggregation class PolicyGroupInPolicyRule, this property (like all reference properties) is single-valued. The [0..n] cardinality indicates that there may be 0, 1 or more than one PolicyRules that contain any given PolicyGroup.

The reference property "PartComponent" is inherited from PolicySetComponent, and overridden to become an object reference to a PolicyGroup contained by one or more PolicyRules. Note that for any single instance of the aggregation class PolicyGroupInPolicyRule, this property (like all reference properties) is single-valued. The [0..n] cardinality indicates that a given PolicyRule may contain 0, 1, or more than one PolicyGroup.

6.5. The Aggregation "PolicyRuleInPolicyRule"

A policy rule may aggregate one or more policy rules, via the PolicyRuleInPolicyRule aggregation. The ability to nest policy rules and form sub-rules is important for manageability and scalability, as it enables complex policy rules to be constructed from multiple simpler policy rules.

A policy rule does not have to contain sub-rules. A policy rule may contain a group of sub-rules using the PolicyGroupInPolicyRule aggregation. Note that it is assumed that this aggregation is used to form directed a-cyclic graphs and NOT ring structures.

The class definition for this aggregation is as follows:

```
NAME              PolicyRuleInPolicyRule
DERIVED FROM      PolicySetComponent
ABSTRACT          False
PROPERTIES        GroupComponent[ref PolicyRule[0..n]]
                  PartComponent[ref PolicyRule[0..n]]
```

The reference property "GroupComponent" is inherited from PolicySetComponent, and overridden to become an object reference to a PolicyRule that contains one or more PolicyRules. Each contained PolicyRule can be conceptualized as a sub-rule of the containing PolicyRule. This nesting can be done to any desired level. However, the deeper the nesting, the more complex the results of the decisions taken by the nested rules.

Note that for any single instance of the aggregation class PolicyRuleInPolicyRule, this property is single-valued. The [0..n] cardinality indicates that there may be 0, 1 or more than one PolicyRules that contain any given PolicyRule.

The reference property "PartComponent" is inherited from PolicySetComponent, and overridden to become an object reference to a PolicyRule contained by a PolicyRule. Note that for any single instance of the aggregation class PolicyRuleInPolicyRule, this property is single-valued. The [0..n] cardinality indicates that a given PolicyRule may contain 0, 1, or more than one other PolicyRules.

6.6. The Abstract Aggregation "CompoundedPolicyCondition"

NAME CompoundedPolicyCondition

```
DESCRIPTION      A class representing the aggregation of
                 PolicyConditions by an aggregating instance.
DERIVED FROM     PolicyComponent
ABSTRACT         TRUE
PROPERTIES       PartComponent[ref PolicyCondition[0..n]]
                 GroupNumber
                 ConditionNegated
```

6.7. Update to PCIM's Aggregation "PolicyConditionInPolicyRule"

The PCIM aggregation "PolicyConditionInPolicyRule" is updated, to make it a subclass of the new abstract aggregation CompoundedPolicyCondition. The properties GroupNumber and ConditionNegated are now inherited, rather than specified explicitly as they were in PCIM.

```
NAME             PolicyConditionInPolicyRule
DESCRIPTION      A class representing the aggregation of
                 PolicyConditions by a PolicyRule.
DERIVED FROM     CompoundedPolicyCondition
ABSTRACT         FALSE
PROPERTIES       GroupComponent[ref PolicyRule[0..n]]
```

6.8. The Aggregation "PolicyConditionInPolicyCondition"

A second subclass of CompoundedPolicyCondition is defined, representing the compounding of policy conditions into a higher-level policy condition.

```
NAME             PolicyConditionInPolicyCondition
DESCRIPTION      A class representing the aggregation of
                 PolicyConditions by another PolicyCondition.
DERIVED FROM     CompoundedPolicyCondition
ABSTRACT         FALSE
PROPERTIES       GroupComponent[ref PolicyCondition[0..n]]
```

6.9. The Abstract Aggregation "CompoundedPolicyAction"

```
NAME             CompoundedPolicyAction
DESCRIPTION      A class representing the aggregation of PolicyActions
                 by an aggregating instance.
DERIVED FROM     PolicyComponent
ABSTRACT         TRUE
PROPERTIES       PartComponent[ref PolicyAction[0..n]]
                 ActionOrder
```

6.10. Update to PCIM's Aggregation "PolicyActionInPolicyRule"

The PCIM aggregation "PolicyActionInPolicyRule" is updated, to make it a subclass of the new abstract aggregation CompoundedPolicyAction. The property ActionOrder is now inherited, rather than specified explicitly as it was in PCIM.

```
NAME              PolicyActionInPolicyRule
DESCRIPTION       A class representing the aggregation of PolicyActions
                  by a PolicyRule.
DERIVED FROM      CompoundedPolicyAction
ABSTRACT          FALSE
PROPERTIES        GroupComponent[ref PolicyRule[0..n]]
```

6.11. The Aggregation "PolicyActionInPolicyAction"

A second subclass of CompoundedPolicyAction is defined, representing the compounding of policy actions into a higher-level policy action.

```
NAME              PolicyActionInPolicyAction
DESCRIPTION       A class representing the aggregation of PolicyActions
                  by another PolicyAction.
DERIVED FROM      CompoundedPolicyAction
ABSTRACT          FALSE
PROPERTIES        GroupComponent[ref PolicyAction[0..n]]
```

6.12. The Aggregation "PolicyVariableInSimplePolicyCondition"

A simple policy condition is represented as an ordered triplet {variable, operator, value}. This aggregation provides the linkage between a SimplePolicyCondition instance and a single PolicyVariable. The aggregation PolicyValueInSimplePolicyCondition links the SimplePolicyCondition to a single PolicyValue. The Operator property of SimplePolicyCondition represents the third element of the triplet, the operator.

The class definition for this aggregation is as follows:

```
NAME              PolicyVariableInSimplePolicyCondition
DERIVED FROM      PolicyComponent
ABSTRACT          False
PROPERTIES        GroupComponent[ref SimplePolicyCondition[0..n]]
                  PartComponent[ref PolicyVariable[1..1] ]
```

The reference property "GroupComponent" is inherited from PolicyComponent, and overridden to become an object reference to a SimplePolicyCondition that contains exactly one PolicyVariable. Note that for any single instance of the aggregation class PolicyVariableInSimplePolicyCondition, this property is single-valued. The [0..n] cardinality indicates that there may be 0, 1, or more SimplePolicyCondition objects that contain any given policy variable object.

The reference property "PartComponent" is inherited from PolicyComponent, and overridden to become an object reference to a PolicyVariable that is defined within the scope of a SimplePolicyCondition. Note that for any single instance of the association class PolicyVariableInSimplePolicyCondition, this property (like all reference properties) is single-valued. The [1..1] cardinality indicates that a SimplePolicyCondition must have exactly one policy variable defined within its scope in order to be meaningful.

6.13. The Aggregation "PolicyValueInSimplePolicyCondition"

A simple policy condition is represented as an ordered triplet {variable, operator, value}. This aggregation provides the linkage between a SimplePolicyCondition instance and a single PolicyValue. The aggregation PolicyVariableInSimplePolicyCondition links the SimplePolicyCondition to a single PolicyVariable. The Operator property of SimplePolicyCondition represents the third element of the triplet, the operator.

The class definition for this aggregation is as follows:

```
NAME              PolicyValueInSimplePolicyCondition
DERIVED FROM      PolicyComponent
ABSTRACT          False
PROPERTIES        GroupComponent[ref SimplePolicyCondition[0..n]]
                  PartComponent[ref PolicyValue[1..1] ]
```

The reference property "GroupComponent" is inherited from PolicyComponent, and overridden to become an object reference to a SimplePolicyCondition that contains exactly one PolicyValue. Note that for any single instance of the aggregation class PolicyValueInSimplePolicyCondition, this property is single-valued. The [0..n] cardinality indicates that there may be 0, 1, or more SimplePolicyCondition objects that contain any given policy value object.

The reference property "PartComponent" is inherited from PolicyComponent, and overridden to become an object reference to a PolicyValue that is defined within the scope of a SimplePolicyCondition. Note that for any single instance of the association class PolicyValueInSimplePolicyCondition, this property (like all reference properties) is single-valued. The [1..1] cardinality indicates that a SimplePolicyCondition must have exactly one policy value defined within its scope in order to be meaningful.

6.14. The Aggregation "PolicyVariableInSimplePolicyAction"

A simple policy action is represented as a pair {variable, value}. This aggregation provides the linkage between a SimplePolicyAction instance and a single PolicyVariable. The aggregation PolicyValueInSimplePolicyAction links the SimplePolicyAction to a single PolicyValue.

The class definition for this aggregation is as follows:

```
NAME            PolicyVariableInSimplePolicyAction
DERIVED FROM    PolicyComponent
ABSTRACT        False
PROPERTIES      GroupComponent[ref SimplePolicyAction[0..n]]
                PartComponent[ref PolicyVariable[1..1] ]
```

The reference property "GroupComponent" is inherited from PolicyComponent, and overridden to become an object reference to a SimplePolicyAction that contains exactly one PolicyVariable. Note that for any single instance of the aggregation class PolicyVariableInSimplePolicyAction, this property is single-valued. The [0..n] cardinality indicates that there may be 0, 1, or more SimplePolicyAction objects that contain any given policy variable object.

The reference property "PartComponent" is inherited from PolicyComponent, and overridden to become an object reference to a PolicyVariable that is defined within the scope of a SimplePolicyAction. Note that for any single instance of the association class PolicyVariableInSimplePolicyAction, this property (like all reference properties) is single-valued. The [1..1] cardinality indicates that a SimplePolicyAction must have exactly one policy variable defined within its scope in order to be meaningful.

6.15. The Aggregation "PolicyValueInSimplePolicyAction"

A simple policy action is represented as a pair {variable, value}. This aggregation provides the linkage between a SimplePolicyAction instance and a single PolicyValue. The aggregation PolicyVariableInSimplePolicyAction links the SimplePolicyAction to a single PolicyVariable.

The class definition for this aggregation is as follows:

```
NAME            PolicyValueInSimplePolicyAction
DERIVED FROM    PolicyComponent
ABSTRACT        False
PROPERTIES      GroupComponent[ref SimplePolicyAction[0..n]]
                PartComponent[ref PolicyValue[1..1] ]
```

The reference property "GroupComponent" is inherited from PolicyComponent, and overridden to become an object reference to a SimplePolicyAction that contains exactly one PolicyValue. Note that for any single instance of the aggregation class PolicyValueInSimplePolicyAction, this property is single-valued. The [0..n] cardinality indicates that there may be 0, 1, or more SimplePolicyAction objects that contain any given policy value object.

The reference property "PartComponent" is inherited from PolicyComponent, and overridden to become an object reference to a PolicyValue that is defined within the scope of a SimplePolicyAction. Note that for any single instance of the association class PolicyValueInSimplePolicyAction, this property (like all reference properties) is single-valued. The [1..1] cardinality indicates that a SimplePolicyAction must have exactly one policy value defined within its scope in order to be meaningful.

6.16. The Association "ReusablePolicy"

The association ReusablePolicy makes it possible to include any subclass of the abstract class "Policy" in a ReusablePolicyContainer.

```
NAME             ReusablePolicy
DESCRIPTION      A class representing the inclusion of a reusable
                 policy element in a ReusablePolicyContainer.  Reusable
                 elements may be PolicyGroups, PolicyRules,
                 PolicyConditions, PolicyActions, PolicyVariables,
                 PolicyValues, or instances of any other subclasses of
                 the abstract class Policy.
DERIVED FROM     PolicyInSystem
ABSTRACT         FALSE
PROPERTIES       Antecedent[ref ReusablePolicyContainer[0..1]]
```

6.17. Deprecate PCIM's "PolicyConditionInPolicyRepository"

```
NAME             PolicyConditionInPolicyRepository
DEPRECATED FOR   ReusablePolicy
DESCRIPTION      A class representing the inclusion of a reusable
                 PolicyCondition in a PolicyRepository.
DERIVED FROM     PolicyInSystem
ABSTRACT         FALSE
PROPERTIES       Antecedent[ref PolicyRepository[0..1]]
                 Dependent[ref PolicyCondition[0..n]]
```

6.18. Deprecate PCIM's "PolicyActionInPolicyRepository"

```
NAME             PolicyActionInPolicyRepository
DEPRECATED FOR   ReusablePolicy
DESCRIPTION      A class representing the inclusion of a reusable
                 PolicyAction in a PolicyRepository.
DERIVED FROM     PolicyInSystem
ABSTRACT         FALSE
PROPERTIES       Antecedent[ref PolicyRepository[0..1]]
                 Dependent[ref PolicyAction[0..n]]
```

6.19. The Association PolicyValueConstraintInVariable

This association links a PolicyValue object to a PolicyVariable object, modeling specific value constraints. Using this association, a variable (instance) may be constrained to be bound-to/assigned only a set of allowed values. For example, modeling an enumerated source port variable, one creates an instance of the PolicySourcePortVariable class and associates it with the set of values (integers) representing the allowed enumeration, using appropriate number of instances of the PolicyValueConstraintInVariable association.

Note that a single variable instance may be constrained by any number of values and a single value may be used to constrain any number of variables. These relationships are manifested by the n-to-m cardinality of the association.

The class definition for the association is as follows:

```
NAME              PolicyValueConstraintInVariable
DESCRIPTION       A class representing the association of a constraints
                  object to a variable object.
DERIVED FROM      Dependency
ABSTRACT          FALSE
PROPERTIES        Antecedent [ref PolicyVariable[0..n]]
                  Dependent [ref PolicyValue [0..n]]
```

The reference property Antecedent is inherited from Dependency. Its type and cardinality are overridden to provide the semantics of a variable optionally having value constraints. The [0..n] cardinality indicates that any number of variables may be constrained by a given value.

The reference property "Dependent" is inherited from Dependency, and overridden to become an object reference to a PolicyValue that is used to constrain the values that a particular PolicyVariable can have. The [0..n] cardinality indicates that a given policy variable may have 0, 1 or more than one PolicyValues defined to model the constraints on the values that the policy variable can take.

6.20. The Aggregation "PolicyContainerInPolicyContainer"

The aggregation PolicyContainerInPolicyContainer provides for nesting of one ReusablePolicyContainer inside another one.

```
NAME              PolicyContainerInPolicyContainer
DESCRIPTION       A class representing the aggregation of
                  ReusablePolicyContainers by a higher-level
                  ReusablePolicyContainer.
DERIVED FROM      SystemComponent
ABSTRACT          FALSE
PROPERTIES        GroupComponent[ref ReusablePolicyContainer [0..n]]
                  PartComponent[ref ReusablePolicyContainer [0..n]]
```

6.21. Deprecate PCIM's "PolicyRepositoryInPolicyRepository"

```
NAME              PolicyRepositoryInPolicyRepository
DEPRECATED FOR    PolicyContainerInPolicyContainer
```

DESCRIPTION	A class representing the aggregation of PolicyRepositories by a higher-level PolicyRepository.
DERIVED FROM	SystemComponent
ABSTRACT	FALSE

```
  PROPERTIES       GroupComponent[ref PolicyRepository[0..n]]
                   PartComponent[ref PolicyRepository[0..n]]
```

6.22. The Aggregation "ElementInPolicyRoleCollection"

The following aggregation is used to associate ManagedElements with a PolicyRoleCollection object that represents a role played by these ManagedElements.

```
  NAME             ElementInPolicyRoleCollection
  DESCRIPTION      A class representing the inclusion of a ManagedElement
                   in a collection, specified as having a given role.
                   All the managed elements in the collection share the
                   same role.
  DERIVED FROM     MemberOfCollection
  ABSTRACT         FALSE
  PROPERTIES       Collection[ref PolicyRoleCollection [0..n]]
                   Member[ref ManagedElement [0..n]]
```

6.22.1. The Weak Association "PolicyRoleCollectionInSystem"

A PolicyRoleCollection is defined within the scope of a System. This association links a PolicyRoleCollection to the System in whose scope it is defined.

When associating a PolicyRoleCollection with a System, this should be done consistently with the system that scopes the policy rules/groups that are applied to the resources in that collection. A PolicyRoleCollection is associated with the same system as the applicable PolicyRules and/or PolicyGroups, or to a System higher in the tree formed by the SystemComponent association.

The class definition for the association is as follows:

```
  NAME             PolicyRoleCollectionInSystem
  DESCRIPTION      A class representing the fact that a
                   PolicyRoleCollection is defined within the scope of a
                   System.
  DERIVED FROM     Dependency
  ABSTRACT         FALSE
  PROPERTIES       Antecedent[ref System[1..1]]
                   Dependent[ref PolicyRoleCollection[weak]]
```

The reference property Antecedent is inherited from Dependency, and overridden to restrict its cardinality to [1..1]. It serves as an object reference to a System that provides a scope for one or more PolicyRoleCollections. Since this is a weak association, the cardinality for this object reference is always 1, that is, a PolicyRoleCollection is always defined within the scope of exactly one System.

The reference property Dependent is inherited from Dependency, and overridden to become an object reference to a PolicyRoleCollection defined within the scope of a System. Note that for any single instance of the association class PolicyRoleCollectionInSystem, this property (like all Reference properties) is single-valued. The [0..n] cardinality indicates that a given System may have 0, 1, or more than one PolicyRoleCollections defined within its scope.

7. Intellectual Property

8. Acknowledgements

The starting point for this document was PCIM itself [3], and the first three submodels derived from it [5], [6], [7]. The authors of these documents created the extensions to PCIM, and asked the questions about PCIM, that are reflected in PCIMe.

9. Security Considerations

The Policy Core Information Model (PCIM) [3] describes the general security considerations related to the general core policy model. The extensions defined in this document do not introduce any additional considerations related to security.

10. References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement

Levels", BCP 14, RFC 2119, March 1997.

[2] Hovey, R., and S. Bradner, "The Organizations Involved in the IETF Standards Process", BCP 11, RFC 2028, October 1996.

[3] Strassner, J., and E. Ellesson, B. Moore, A. Westerinen, "Policy Core Information Model -- Version 1 Specification", RFC 3060, February 2001.

[4] Distributed Management Task Force, Inc., "DMTF Technologies: CIM Standards û CIM Schema: Version 2.5", available via links on the following DMTF web page: http://www.dmtf.org/spec/cim_schema_v25.html.

[5] Snir, Y., and Y. Ramberg, J. Strassner, R. Cohen, "Policy Framework QoS Information Model", work in progress, draft-ietf-policy-qos-info-model-02.txt, November 2000.

[6] Jason, J., and L. Rafalow, E. Vyncke, "IPsec Configuration Policy Model", work in progress, draft-ietf-ipsp-config-policy-model-02.txt, March 2001.

[7] Chadha, R., and M. Brunner, M. Yoshida, J. Quittek, G. Mykoniatis, A. Poylisher, R. Vaidyanathan, A. Kind, F. Reichmeyer, "Policy Framework MPLS Information Model for QoS and TE", work in progress, draft-chadha-policy-mpls-te-01.txt, December 2000.

[8] Crocker, D., and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

[9] P. Mockapetris, "DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION", RFC1035, November 1987.

[10] R. Hinden, S. Deering, "IP Version 6 Addressing Architecture", RFC2373, July 1998.

[11] M. Wahl, A. Coulbeck, "Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions", RFC 2252.

[12] A. Westerinen, et al., "Policy Terminology", <draft-ietf-policy-terminology-01.txt>, November 2000.

[13] S. Waldbusser, and J. Saperia, T. Hongal, "Policy Based Management MIB", <draft-ietf-snmpconf-pm-04.txt>, November 2000.

11. Authors' Addresses

Bob Moore
IBM Corporation, BRQA/502
4205 S. Miami Blvd.
Research Triangle Park, NC 27709
Phone: +1 919-254-4436
Fax: +1 919-254-6243
E-mail: remoore@us.ibm.com

Lee Rafalow
IBM Corporation, BRQA/502

4205 S. Miami Blvd.
Research Triangle Park, NC 27709
Phone: +1 919-254-4455
Fax: +1 919-254-6243
E-mail: rafalow@us.ibm.com

Yoram Ramberg
Cisco Systems
4 Maskit Street
Herzliya Pituach, Israel 46766
Phone: +972-9-970-0081
Fax: +972-9-970-0219
E-mail: yramberg@cisco.com

Yoram Snir
Cisco Systems
4 Maskit Street
Herzliya Pituach, Israel 46766
Phone: +972-9-970-0085
Fax: +972-9-970-0366
E-mail: ysnir@cisco.com

John Strassner
Cisco Systems
Building 20
725 Alder Drive
Milpitas, CA 95035
Phone: +1-408-527-1069
Fax: +1-408-527-2477
E-mail: johns@cisco.com

Andrea Westerinen
Cisco Systems
Building 20
725 Alder Drive
Milpitas, CA 95035
Phone: +1-408-853-8294
Fax: +1-408-527-6351
E-mail: andreaw@cisco.com

Ritu Chadha
Telcordia Technologies
MCC 1J-218R
445 South Street
Morristown NJ 07960.
Phone: +1-973-829-4869
Fax: +1-973-829-5889
E-mail: chadha@research.telcordia.com

Marcus Brunner
NEC Europe Ltd.
C&C Research Laboratories

Adenauerplatz 6
D-69115 Heidelberg, Germany
Phone: +49 (0)6221 9051129
Fax: +49 (0)6221 9051155
E-mail: brunner@ccrle.nec.de

Ron Cohen
Ntear LLC
Phone:
Fax:
E-mail: ronc@ntear.com

12. Full Copyright Statement

13. Appendix A: Open Issues

The PCIMe authors do not all agree with everything included in the -00 draft of the document. Input is solicited from the working group as a whole on the following open issues:

1. Unrestricted use of DNF/CNF for CompoundPolicyConditions.
   Alternative: for the conditions aggregated by a

CompoundPolicyCondition, allow only ANDing, with negation of individual conditions. Note that this is sufficient to build multi-field packet filters from single-field SimplePolicyConditions.

2. For a PolicyVariable in a SimplePolicyCondition, restrict the set of possible values both via associated PolicyValue objects (tied in with the PolicyValueConstraintInVariable association) and via the ValueTypes property in the PolicyVariable class. Alternative: restrict values only via associated PolicyValue objects.

3. Transactional semantics, including rollback, for the ExecutionStrategy property in PolicyRule and in CompoundPolicyAction. Alternative: have only 'Do until success' and 'Do all'.

4. Stating that CompoundFilterConditions are the preferred way to do packet filtering in a PolicyCondition. Alternative: make CompoundFilterConditions and FilterEntries available to submodels, with no stated (or implied) preference.

5. Prohibiting equal values for Priority within a PolicySet. Alternative: allow equal values, with resulting indeterminacy in PEP behavior.

6. Modeling a SimplePolicyAction with just a related PolicyVariable and PolicyValue -- the "set" or "apply" operation is implicit. Alternative: include an Operation property in SimplePolicyAction, similar to the Operation property in SimplePolicyCondition.

7. Representation of PolicyValues: should values like IPv4 addresses be represented only as strings (as in LDAP), or natively (e.g., an IPv4 address would be a four-octet field) with mappings to other representations such as strings?

8. The nesting of rules and groups within rules introduces significant change and complexity in the model. This nesting introduces program state (procedural language) into the model (heretofore a declarative model) as well as implicit hierarchical contexts on which the rules operate. These require a much more sophisticated rule-evaluation engine than in the past.

   Alternative: Maintain the declarative model, by prohibiting program state in rule evaluation (i.e., no rules within rules).

9. Need to specify a join algorithm for disjoint rule sets.

10. Clarify PolicyImplicitVariables.

11. Clarify PolicyExplicitVariables.