SUPA   Y. Cheng
Internet-Draft   China Unicom
Intended status: Informational   D. Liu
Expires: February 6, 2017   Alibaba Group
                                                               B. Fu
                                                        China Telecom
                                                             D. Zhang
                                                           Freelancer
                                                          N. Vadrevu
                                               VN Telecom Consultancy
                                                      August 5, 2016

                          Applicability of SUPA
                    draft-cheng-supa-applicability-00.txt

Abstract

   SUPA will define a generic policy model, an imperative ECA (Event
   Condition Action) policy information model and a declarative (intent-
   based) policy information model which is the extension of the generic
   model, and a set of policy data models which will make use of the
   common concepts defined in the generic model. This memo will explore
   some typical use cases and demonstrate the applicability of SUPA
   policy models.

Status of This Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at http://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on February 6, 2017.

Copyright Notice

Table of Contents

1. Introduction

   One of the ways for network service automation is using network
   management and operation software applications. The applications may
   not be able to directly communicate with each network element; a
   hierarchical and extensible framework should be considered to hide
   the protocol specific and/or vendor specific details, high level
   network and service abstraction, and standardized programming API
   will be necessary.

SUPA will define policy generic models and data models, for service management and operation applications. [I-D.strassner-supa-generic-policy-info-model] defines a common set of concepts for various data models which may use different languages, protocols, and repositories.

Three generic models are defined in [I-D.strassner-supa-generic-policy-info-model]: Generic Policy Model, Eca Policy Rule Model, Logic Statement Model. The ECA information model is intended for dynamic service automation; while the Logic Statement Model is intended for expressing high requirements without being involved in network details.

Data models can be defined by developers / operators or by any third party, as long as they follow the common concepts defined in SUPA generic model. [I-D.chen-supa-eca-data-model] defines a policy data model of Event-Condition-Action (ECA), which is an example.

The generic data models will be used for domain or service specific data model. And there is no interoperability requirement for domain specific data models. The interoperability is guaranteed at the generic data model level via the common concepts.

2. Terminology

   DC Data Center

   PCE Path Computation Element

   SES Switched Ethernet services

   SP Service Provider

   SUPA Simplified Use of Policy Abstractions

   VM Virtual Machine

   VPC Virtual Private Cloud

3. Framework

```
+-----------------------------------------------------------------+
|                      Service Management                          |
|                                                                 |
|          +----------------------------------+                   |
|          |       Generic Policy Model       |                   |
|          +----+-----------------------+----+                    |
|               D                       R                         |
|               D                       R                         |
|               \ /                     \ /                        |
|  +---------------------------+ +------------------------------+  |
|  | Generic Policy Data Model | | Service Management Data Model |  |
|  +---------------------------+ +------------+----------------+  |
|              / \                           / \                  |
|               |                             |                   |
|               |                             |                   |
+---------------+-----------------------------+-------------------+
                |                             |
                |       NETCONF/RESTCONF      |
          +----+-----------------------+----+
               C                       C
               C                       C
               \ /                     \ /
  +---------------+----------+   +-------+-------------------+
  | Network Manager/Controller|  | Network Manager/Controller|
  | +--------------------+    |  | +---------------------+   |
  | |  Network Resource  |    |  | |  Network Resource   |   |
  | |    Data Model      |    |  | |    Data Model       |   |
  | +--------------------+    |  | +---------------------+   |
  +---+---+---+--------------+   +-----+---+---+-------------+
     / \ / \ / \                    / \ / \ / \
      C   C   C                      C   C   C
      C   C   C                      C   C   C
      C   C   C                      C   C   C
      \ / \ / \ /                    \ / \ / \ /
      NE1 NE2 NEn                    NE1 NE2 NEn
```
        Figure 1: Use of SUPA Models

   C: Communications

   D: Derived from

   R: References (i.e., the generic model is used by the system to
   instantiate the data model).

   As shown in Figure 1, SUPA will define generic policy models, which
   are independent of services and use cases. Policy data models can be
   derived from the generic models. The data model will define high
   level, maybe network-wide policies. Policy data model will be used

in conjunction with service data models to generate configurations
for network elements. The service data model is use case specific
and will be developed by operators or third parties, which is out the
scope of SUPA.

The service management applications will send SUPA data models to the
service management system, where policy making and automated policy
enforcement will be performed, and the data models will be mapped to
configuration of network elements. Configuration of network elements
is vendor specific, using various protocols, such Netconf, Restconf,
etc.

SUPA also make use of information collected from network elements.
The information may include warning or fault event, load status,
traffic statistics, etc, which can be used to adjust network
configurations. This kind of automation is done through ECA data
models.

3.1. Network Manager/Controller
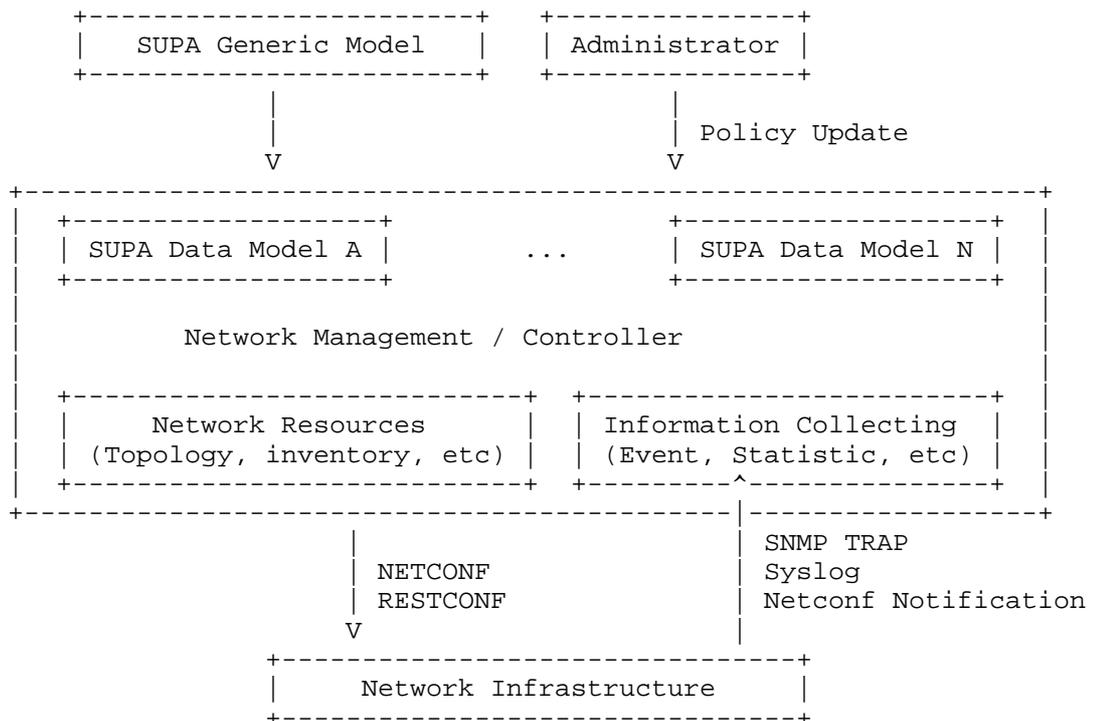
```
       +------------------------+  +---------------+
       |   SUPA Generic Model   |  | Administrator |
       +------------------------+  +---------------+
                   |                   |
                   |                   | Policy Update
                   V                   V
   +----------------------------------------------------------------+
   |  +-------------------+              +-------------------+       |
   |  | SUPA Data Model A |    ...       | SUPA Data Model N |       |
   |  +-------------------+              +-------------------+       |
   |                                                                |
   |          Network Management / Controller                       |
   |                                                                |
   |  +--------------------------+  +------------------------+      |
   |  |     Network Resources    |  | Information Collecting  |     |
   |  | (Topology, inventory, etc) |  | (Event, Statistic, etc) |   |
   |  +--------------------------+  +--------^---------------+      |
   +-----------------------------------------|----------------------+
                       |                     | SNMP TRAP
                       | NETCONF             | Syslog
                       | RESTCONF            | Netconf Notification
                       V                     |
            +-------------------------------+
            |     Network Infrastructure    |
            +-------------------------------+
```
        Figure 1: Network Manager / Controller

The internal details of the network manager / controller may be out
of the scope of SUPA, but explaining how it works may help people to
understand and implement SUPA.

Network administrator can send service deployment and management
request to network manager / controller via SUPA data models. The
data models will be converted into network elements configuration
snippets. The configuration change may be performed instantly, or
later triggered by events. The network manager / controller has the
intelligence to decide which network devices should be configured,
and what the configuration will be, which is derived from the actions
specific in the data models explicitly or implicitly.

Network management related resources and information are stored in
the network manager/controller, which contains the network topology
(physical and virtual interconnection of network elements, etc),
inventory (database of network elements, ports, device type,
capabilities, etc.), protocol specific information, etc.

SUPA will make use of the existing work of other IETF WGs and other
SDOs, such as if the topology data model is already defined in
another IETF WG, SUAP will reference it rather than trying to define
it again.

The network manager / controller will find out the list of network
devices which should be configured for a specific demand or service.

For example, there is a configuration request:

All edge routers shall have SSH disabled.

An edge router is a router with connection to network(s) outside of
the current network domain. The controller will query the topology
database and find out all the routers with the attribute of "device-
role == edge", or the controller may use more complicated algorithms
to find out if a router is an edge route, which is implementation
specific.

Similarly, another example is, the controller can make use of PCE
engine to plan the links between DCs, and make sure the links are
disjoint for better availability in case of failure. The PCE engine
will be used in conjunction with the topology database to find out
possible disjoint links.

The network manager / controller will also have other information,
such as protocol specific information, traffic with TCP destination
port 22 is SNMP traffic.

The network manager / controller also collect information from the
network device, such events, logs, statistics, etc. The information
may come from SNMP TRAP, Syslog, NETCONF notification, and other
sources such as vendor specific protocols or extensions. The
collected information may be used in conjunction with SUPA ECA data
models for dynamic configuration change. An example use of the
information is, if the load on a link between two DC exceeds a
threshold, and there are multiple disjoint links between the two DCs,
traffic steering will be triggered.

Event: link_load > threshold

Condition: there are disjoint links

Action: perform traffic steering

Some of the events are already standardized, such SNMP TRAP and
NETCONF notification; some are implementation specific.

SUPA data models explicitly or implicitly specify network actions,
and the actions may be expanded into more detail actions if
necessary, and finally converted into protocol specific, vendor
specific network element configuration snippets.

In the previous example shown below again:

All edge routers shall have SSH disabled.

The action in this case is "disable SSH traffic", the network manager
/ controller should converted this action into configuration "disable
traffic on TCP port 22" in the IP stack, or an ACL rule which will
drop traffic with TCP destination port 22.

The network manager / controller can support various types of
southbound interface, such as NETCONF, RESTCONF, SNMP, OpenFLow, etc,
which make it possible to support devices from different vendors.
This is implementation specific and out of the scope of SUPA.

4. Use Cases of SUPA

4.1. Use Case 1: SES

4.1.1. Scenario
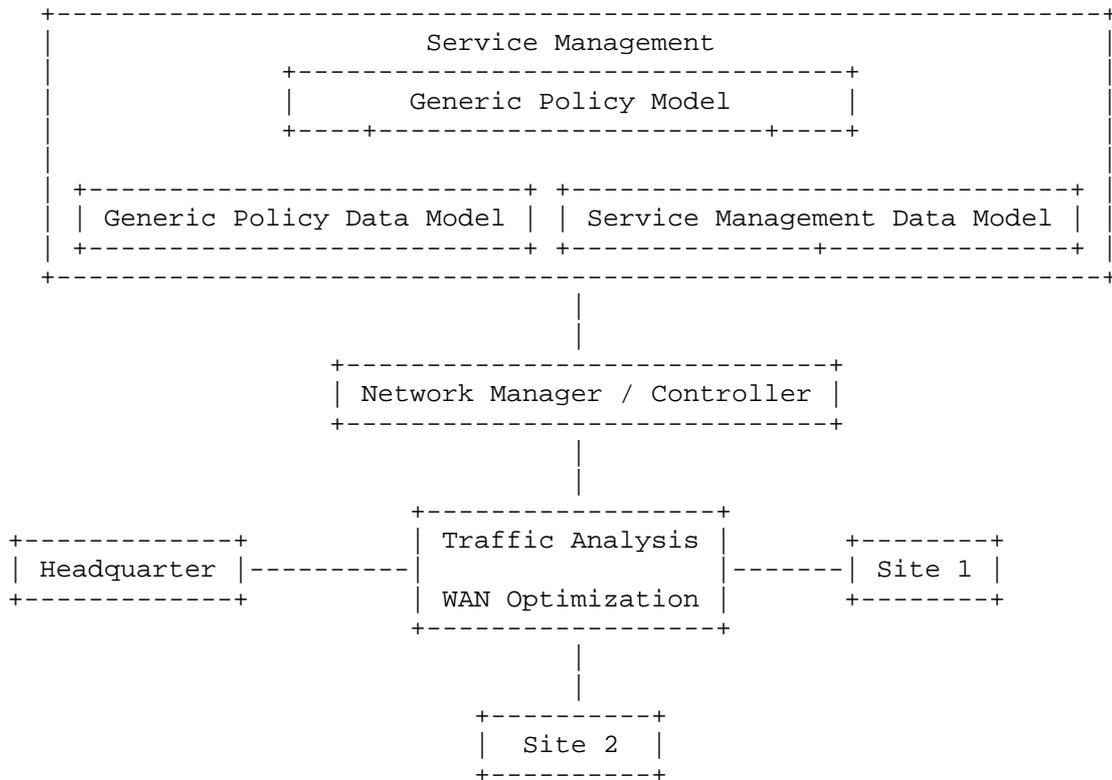
```
+------------------------------------------------------------------+
|                      Service Management                          |
|         +----------------------------------+                     |
|         |       Generic Policy Model       |                     |
|         +----+-----------------------+----+                      |
|                                                                  |
| +-------------------------+ +------------------------------+ |
| | Generic Policy Data Model | | Service Management Data Model | |
| +-------------------------+ +--------------+---------------+ |
+------------------------------------------------------------------+
                                 |
                                 |
                 +------------------------------+
                 | Network Manager / Controller |
                 +------------------------------+
                                 |
                                 |
                       +------------------+
+-------------+        | Traffic Analysis |        +--------+
| Headquarter |--------|                  |-------| Site 1 |
+-------------+        | WAN Optimization |        +--------+
                       +------------------+
                                 |
                                 |
                          +----------+
                          |  Site 2  |
                          +----------+
```
                Figure 3: Switched Ethernet Service

Switched Ethernet services (SES) to Small and Medium Businesses
business is a growing business segment of the service provider. As
the Enterprise's applications grow in demands in terms of the
bandwidth and richness of applications, WAN optimization is needed to
improve the service quality. SUPA policy data models can be used for
maximizing the WAN performance by analyzing the traffic and
performing application management and acceleration tools for the
network.

In the use case below, Service Manager (SM) is used for service and
policy definition and Network Manager (Controller) is used for
network topology maintenance and mapping data models to detail
network configurations.

While speed and bandwidth are at the forefront of the WAN
Optimization there need to be tools in place to detect, diagnose,
remedy and report application performance to ensure the SLAs for a
customer are enforced.

The service is modeled in terms of what kind of service (Ethernet, VLAN), bandwidth (10Mbps- 10 Gbps), service package (platinum, gold, silver) etc.

Policy models are based on an Event condition action like:

1. Bandwidth usage alarm triggers data caching

2. Latency alarm triggers reduction of re transmission

3. WAN outage at a specific site can trigger geographic redundancy (provided the service is setup for GR)

The above are 3 of the primitives (Event condition action - ECA) on which the run time operations could be based on. When the service model is comprehensively designed with more possibilities (variables), more policy models could be implemented

4.1.2. Generic Policy Models

Requirements and configurations derived from above application scenarios can be described by service data model and policy data models as below:

Service data model can be used to describe attributes for the SES, including service package type (Platinum, gold etc), bandwidth bought by the subscriber (100Mbps, 10Gbps), connection name -copper/ GigE, latency, etc.

Policy data model describes a condition when the link capacity reaches 90%, Service prioritization and WAN optimization need to be enforced based on the customers service package. Event is the link utilization and condition is the usage and action is the WAN optimization. The actions could trigger multiple actions like data compression, protocol acceleration (like streaming gets priority) which are beyond the scope of SUPA.

ECA Policy:

Event: link_load > 90%

Condition: acceleration for service available

Action: data compression; protocol acceleration

It is assumed that the network management/controller module has the network topology and monitors the load on links in the topology.

When translating and processing the SUPA data model, the link
information, including link attributes and load, will be provided by
the network management/controller. If the load on a specific link
exceeds a threshold, the network manager/controller will trigger
actions specified in the model.

The actual actions may be vendor specific, network management/
controller specific or device specific. The actions will be mapped
into configuration for network devices. The network management/
controller also need to figure out the set of network devices which
need to be configured based on network topology together with some
other information, such as service specific information. This is the
internal functions of network management/controller, which is out of
the scope of SUPA.

4.1.3. Programmatic approach - SUPA modeling

The advantage of the programmatic approach can be maximized by
defining as many SUPA ECA models as possible in a top down approach.

In this use case, since this is a switched service, point to point
traffic can be identified (by IP Address and port number) and
segmented and whole bandwidth can be utilized by many applications
simultaneously. Examples are: Print jobs, backups etc..

The benefit of the SUPA is in creating many policies upfront. As the
operations grow in complexity SUPA can expand an existing policy by
adding more variables. This is how reusable policies can be
developed upfront and configuration and maintenance operations can be
dealt by modeling and programmatic approach.

Logic Statement Model can also be called as declarative or intent
model. This type of model will describe the service intention
without specifying low level details, such protocol level or network
device level detail, but just the service requirements itself.

4.1.4. SUPA Data Model for SES Use Case

The following model segment is based on [I-D.chen-supa-eca-data-
model].

In the model, the event can be expressed using some standardized
names, such as the SNMP TRAP (linkDown, linkup, Failure, etc), or
"link-load > 90%".

The condition(s) can be expressed using script, such as Python script
hasAcceleration("ses") or Python script hasDisjointLinks(DC1, DC2).
The script is supposed to be interpreted by a script tool and there

are various script tools, the implementer can use any one as they like, either an existing one like Python or a new one. The script itself is out the scope of SUPA; a simple value will be return by the script tool. Some complex combination of conditions can be expressed using script which will give more flexibility.

When handling the condition script, the script tool will be called to process the script. In this case, the script will communicate with service management system and/or the tenant database to find out if any optimization is available for this service or tenant.

Script can also be used for actions.

An example of the script using Python is:

```
service-name="ses"

// input: service-name, type: string

// output: enhancement, type: string or None if no enhance

def queryEnhanceinCapability(service-name):

  for i in range(len(capability-models)):

    if getServiceName(capability-models[i]) == service-name:

      return getEnhance(capability-models[i])

return None


// input: service-name, type: string

// output: True/False, type: boolean

def hasAcceleration(service-name):

  if queryEnhanceinCapability(service-name) == None:

    return False

  else:

    return True
```

The capability data models are supposed to contain the following:

```
<capability-data-model>

  ...

  <services>

    <service>

      <service-name>ses</service-name>

      <service-enhance>compression</service-enhance >

    </service>

    <service>

      ...

    </service>

  <services>

</capability-data-model>
```

The SUPA XML example is shown below:

```
<supa-policy>
  <supa-policy-name>ses-policy</supa-policy-name>
  <supa-policy-priority>0</supa-policy-priority>
  <supa-policy-validity-period>
    <start>00-00-0000</start>
    <end>00-00-0000</end>
  </supa-policy-validity-period>

  <supa-policy-target>
    <profileType>domain</profileType>
    <asDomainName>operatorA-domain1</asDomainName>
    <businessTypeName>ses</businessTypeName>
    <instance>
      <instanceName>
        // detail to be provided by controller
        <flow-filter>
          <src-ip-addr>10.1.1.0/24</src-ip-addr>
          <dst-ip-addr>20.1.1.0/24</dst-ip-addr>
```

```
          </flow-filter>
          <flow-filter>
            ...... // more filters
          </flow-filter>
        </instanceName>
      </instance>
    </supa-policy-target>

    <supa-policy-atomic>
      <supa-ECA-policy-rule>
        <policy-rule-deploy-status>
          ...... // to be provided by controller
        </policy-rule-deploy-status>
        <policy-rule-exec-status>
          ...... // to be provided by controller
        </policy-rule-exec-status>
        <supa-ECA-component>
          <supa-policy-events>
            <has-policy-events>YES</has-policy-events>
          </supa-policy-events>
          <supa-policy-conditions>
            <has-policy-conditions>YES</has-policy-conditions>
            <conjunctive-type>and</conjunctive-type>
          </supa-policy-conditions>
          <supa-policy-actions>
            <action-execution>YES</action-execution>

        </supa-policy-actions>
        </supa-ECA-component>
      </supa-ECA-policy-rule>
    </supa-policy-atomic>

    <supa-policy-statement>
      <event-list>
        <event-name>
          <eventType>entity</eventType>
          // entity or script or boolean
          <entity>"link-load > 90%"</entity>
        </event-name>
      </event-list>

      <condition-list>
        <condition-linkThreshold>
          <conditionType>script</conditionType>
          // entity or script or boolean
          <supa-script>
            <supa-script-content>hasAcceleration(ses)</supa-script-
content>
```

```
                  <supa-script-type>Python</supa-script-type>
                   // Python or Perl or any other script
                </supa-script>
              </condition-linkThreshold>
           </condition-list>

           <action-list>
             <actionName>data compression</actionName>
             <actionName>protocol acceleration</actionName>
           </action-list>
         </supa-policy-statement>
       </supa-policy>
```

The data model can be augmented according to developers' need. The
developers can add vendor specific events, conditions and actions via
"augment" Yang function in [RFC6020], as suggested in [I-D.chen-
supa-eca-data-model].

An example of of augmented model is shown below:

```
// ------- yang model snippet start -------
   augment "/supa:supa-policy/supa:supa-policy-statement/supa:event-
   list" {
     leaf my-event{
       description "customized event";
     type bool;

     }
   }

   augment "/supa:supa-policy/supa:supa-policy-
   statement/supa:condition-list" {
     container my-condition{
       description "The bandwidth threshold, unit is Mbps";
     type uint32;
     }
   }

   augment "/supa:supa-policy/supa:supa-policy-statement/supa:action-
   list" {
     container my-action-drop{
     description "drop packets";
     type string;
     }
   }
```

```
// ------- yang model snippet end -------


// ------ xml model snippet start -------

     // assume the above augmentation is in a name space "mymodel"
<supa-policy>
  ......  // others

  <supa-policy-statement>
    <event-list>
      <event-name>
        ......  // other events
      </event-name>
      <mymodel:my-event>
        true
      </mymodel:my-event>   // added event
    </event-list>

    <condition-list>
      <condition-linkThreshold>
        ...... // other conditions
      </condition-linkThreshold>
      <mymodel:my-condition>
        32
      </mymodel:my-condition>   // added condition
    </condition-list>

    <action-list>
      <actionName>

        ...... // other actions
      </actionName>
      <mymodel:my-action-drop>
        drop
      </mymodel:my-action-drop> // added action

    </action-list>
  </supa-policy-statement>
</supa-policy>

// ------ xml model snippet end ------
```

4.2. Use Case 2: VPC

4.2.1. Generic

   In practice, a public cloud operator can virtualize the cloud
   resources into multiple isolated virtualized private clouds and
   provide them to different tenants. Such a Virtualized Private Cloud
   is referred to as a VPC. In a typical VPC provided by, e.g., Alibaba
   or Amazon, through a control portal, tenants can establish and manage
   their VPC networks easily, for instance, deploying or removing
   virtualized network devices (e.g., virtualized routers and
   virtualized switches), adjusting the topologies of VPC networks,
   specifying packet forwarding policies, and deploying or removing
   virtual services (e.g., load balancers, firewalls, databases, DNS,
   etc.). The network functionalities that the tenant can access are
   virtualized and actually could be performed by the VMs located on the
   servers connected through physical or overlay networks. Note that
   the servers may be located in different data centers which are
   geographically distributed.

   The manipulation of the virtualized VPC network may also affect the
   configuration of physical networks. For instance, when a tenant
   cloud networks and specify the policies to steer the traffics through
   different VPNs in different conditions. Note that the VPCs that the
   tenant may be located in different geographic regions and the VPNs to
   those VPCs may need to be generated at run time. newly deploys two
   VMs in the VPC which are located in different DCs, the VPC control
   mechanism may have to generate a VPN between two DCs for the internal
   VPC communication. Therefore, the control mechanism for a VPC should
   be able to adjust the underlying network when a tenant changes the
   network or service deployment of the virtual VPC network.

   In addition, a VPC, often provides other value added services (e.g.,
   database Services, DNS) for VMs in certain VPCs. The VMs and the
   value added services could be located in different DCs, or even
   provided by different vendors. VPNs are configured for the VPCs to
   provide connection to the internal services in a tenant's own DC or
   organization. The access of such services should be controlled. For
   instance, the VMs in a VPC can access the database services only when
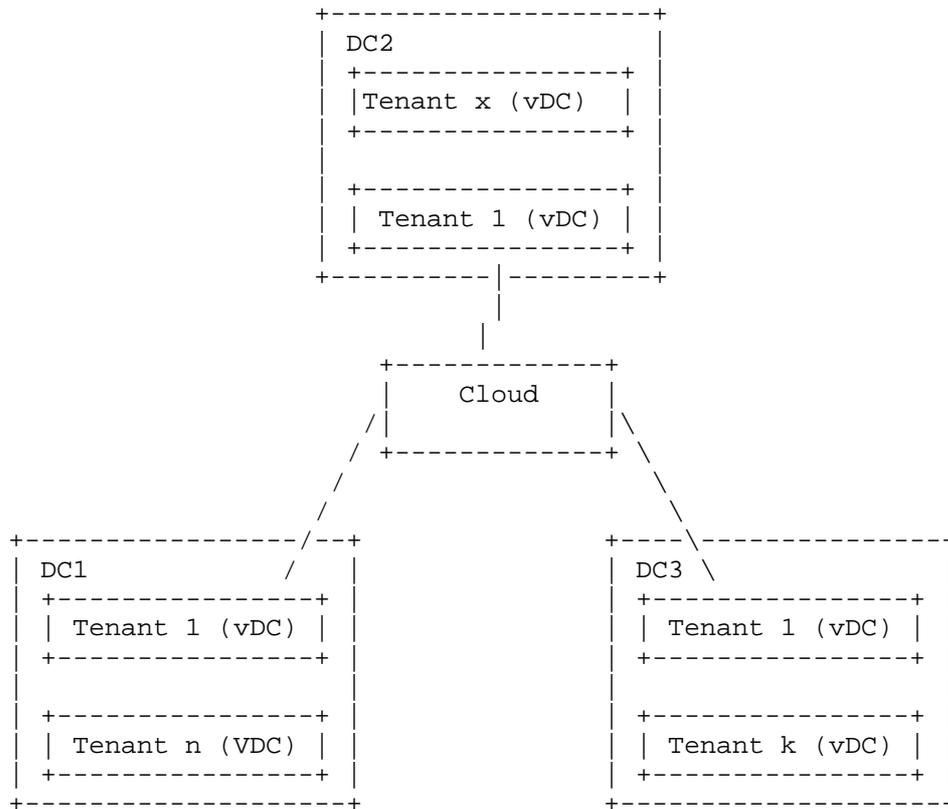   the tenant has deployed a database within its VPC through the control
   portal.

   In many cases, a tenant may need to specify how the VPCs are
   connected to its enterprise cloud networks. For instance, a tenant
   wants to deploy multiple VPNs to connect the VPC with its private
   cloud networks and specify the policies to steer the traffics through
   different VPNs in different conditions. Note that the VPCs that the

tenant may be located in different geographic regions and the VPNs to
those VPCs may need to be generated at run time.

In addition, a VPC, often provides other value added services (e.g.,
database Services, DNS) for VMs in certain VPCs. The VMs and the
value added services could be located in different DCs, or even
provided by different vendors. VPNs are configured for the VPCs to
provide connection to the internal services in tenant's own DC or
organization, and to create and manage VPNs to internal services.
The access of VMs to data resources should be controlled. For
instance, the VMs in a VPC can access a database service only when
the tenant has deployed a database service into its VPC through the
control portal.

4.2.2. Example 1

```
                     +--------------------+
                     | DC2                |
                     | +----------------+ |
                     | |Tenant x (vDC)  | |
                     | +----------------+ |
                     |                    |
                     | +----------------+ |
                     | | Tenant 1 (vDC) | |
                     | +----------------+ |
                     +----------|---------+
                                |
                                |
                        +-------------+
                        |   Cloud     |
                       /|             |\
                      / +-------------+ \
                     /                   \
                    /                     \
    +-----------------/--+         +----\---------------+
    | DC1           /   |         | DC3 \              |
    | +----------------+ |         | +----------------+ |
    | | Tenant 1 (vDC) | |         | | Tenant 1 (vDC) | |
    | +----------------+ |         | +----------------+ |
    |                    |         |                    |
    | +----------------+ |         | +----------------+ |
    | | Tenant n (VDC) | |         | | Tenant k (vDC) | |
    | +----------------+ |         | +----------------+ |
    +--------------------+         +--------------------+
```
        Figure 4: Resource Inter-connection for a VPC Tenant

When a cloud / DC operator signs a contract with customers, resource information such as network bandwidth, storage size, number of CPU, memory size, etc, will be specified.

But in deployment, the resources may be located in multiple distributed data centers, and tunnels will be created to connect these resources, which makes it look like one seamless entity - a virtual DC. There could be quite a number of tunnels, and the tunnels are dynamic, either for the reason of load balancing purpose or VM migration, or other reasons. This will make it difficult to configure the service statically or manually, service automation is very necessary.

The service management system will have a repository of available resources, including the topology. And also the management system will have the customer specific information (location, SLA, agreed resources, etc).

The administrator can send the service requirement to the management system by a high level data model, which can further be mapped to low level detail data models, then finally mapped to configurations of network devices.

Target: Provide VPC service to customer A with specified resources and function (storage, computing, DNS, etc)

Declarative policy:

1. Allocate the required services on DCs according to a user's profile

2. Services located in multiple distributed DCs must be interconnected via VPNs

3. The VPNs associated to the services provided for a user must match the user's profile in terms of latency, speed and bandwidth

4.2.3. Example 2
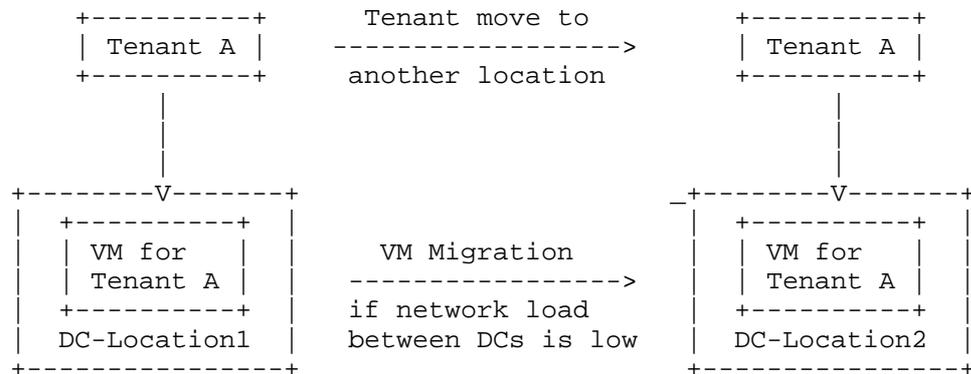
```
            +----------+     Tenant move to    +----------+
            | Tenant A |    ----------------->  | Tenant A |
            +----------+      another location  +----------+
                 |                                   |
                 |                                   |
                 |                                   |
       +--------V-------+                  _+--------V-------+
       | +----------+   |                   | +----------+   |
       | | VM for   |   |    VM Migration   | | VM for   |   |
       | | Tenant A |   |   --------------->| | Tenant A |   |
       | +----------+   |    if network load| +----------+   |
       | DC-Location1   |   between DCs is low| DC-Location2  |
       +----------------+                    +----------------+
           Figure 5: VM Migration if Tenant Move
```

As shown in the above figure, when a VPC tenant move from one
location to another, where it is near to another DC, and the network
load between the new DC and the previous DC is low, the tenant's VM
should be migrated to the new DC in order for better user experience.
After the VM is moved to the new DC, the network related to the VM
must be updated accordingly.

        Target: Perform VM migration when user location changed and the
        network load between the DCs is low.

        ECA Policy:

        Event: a VPC user's location is changed (near to another DC).

        Condition: network_load(DC_old, DC_new) < threshold.

        Action:

        1. Migrate the VM to the new data center (DC_new).

        2. Update the VPNs connecting the user's services.

    In the above model it is assumed that the network management/
    controller has the network topology, including attributes of the
    links, such as bandwidth. The network management/controller also
    monitors the real-time load on the links in the network topology.

    The user's location can be identified by the user's IP address. When
    a user login, the network management/controller will check the user's

IP address against an IP address database, such as the IP address assignments by IANA.

The network management/controller also maintain a mapping of DCs and IP address segments, say, a DC should serve users in a near location which can be identified by IP address segments. Though this is not always the case, sometimes the geographical distribution of network resource will also need to be considered besides the location (IP address). But, anyway, a mapping of DC and the IP address it should serve should be maintained.

If the controller detects a location change and a new DC is possible for the user, and the network load between the new DC and the old DC is low, then VM migration will be triggered and related network configuration will be performed.

4.3. Use Case 3: Traffic Manipulation cross DCs

DCs usually have multiple external links, either to other DCs or to the internet. Because of the dynamic nature of network traffic, the load on a link may vary at different times of a day, e.g. link mainly carries enterprise traffic may have a high load in the working hours but less traffic in the night. Some events may also impact the load of links, such as one link is physically damaged and the load in it will go to another link.

In order to make full use of the bandwidth of the links, dynamic traffic steering is necessary for SLA meanwhile with full use of network resource.
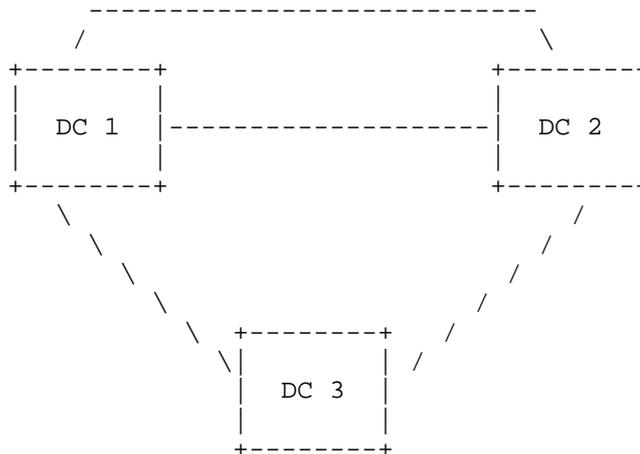
```
                   _____
                  /                            \
        +--------+                              +--------+
        |        |                              |        |
        |  DC 1  |------------------------------|  DC 2  |
        |        |                              |        |
        +--------+                              +--------+
            \  \                               /
             \   \                           /
              \    \                       /
               \     \   +--------+      /
                \     \|  |        | /
                   |  DC 3  |
                   |        |
                   +--------+
```
                Figure 6: Multiple Disjoint Links Between DCs

   Target: a DC has multiple external links. When the load on a link is
   over a threshold, perform traffic steering for a better bandwidth
   resource usage

   ECA Policy:

   Event: load on a DC link exceeds threshold.

   Condition: multiple disjoint links between DCs.

   Action: steer some traffic to link with low load.

   In the above case, it is assumed that the network management/
   controller has the network topology, including attributes of the
   links, such as bandwidth. The network management/controller also
   monitors the real-time load on the links in the network topology.
   The network topology also contains the connections between network
   devices. The network management/controller will be able to figure
   out if there are multiple disjoint links between two DCs. The
   algorithm for finding out disjoint links is out of the scope of SUPA.

   When the network management/controller detects the load on a link
   exceeds a threshold, it can check if there are multiple disjoint
   links, and if yes, it will then further perform necessary actions as
   pre-specified.

4.4. Use Case 4: Virtual SP

   Virtual network operators usually do not build all networks,
   including access network, metro network, and backbone network, by
   themselves. Instead, they rent network from other operators. For
   instance, a virtual operator may not have the access network,
   traffics of broadband network subscribers will go through an access
   network rent from another operators, and then be directed to the
   virtual operators network from the BNG via tunnels. In some another
   case, a virtual operator may not have the backbone network, the
   network islands and DCs will be connected by tunnels.

   In above cases, virtual network operators may have to face an issue.
   That is, they have no control over the tunnels and cannot decide the
   exact path that a tunnel should go through. In some scenarios, if a
   tunnel goes through the border of two network operators, or the
   tunnel goes through an area where network load is too high, the SLA
   may become a problem. Due to cost issue, virtual network operators
   cannot buy service from other operators with critical SLA. This
   problem will be even more serious to the a virtual network operator
   who runs its business in a large geographical region.

A possible solution for such a virtual network operator is to rent or put some routers in network operators' DCs, and then configure tunnels between the routers and perform traffic steering. In this way, virtual network operators can have control over the tunnels, pin down the path. When a problem is detected, such as QoS of a tunnel is below a threshold, virtual network operator can perform "network wide" optimization, reconfigure the tunnels and/or perform traffic steering.
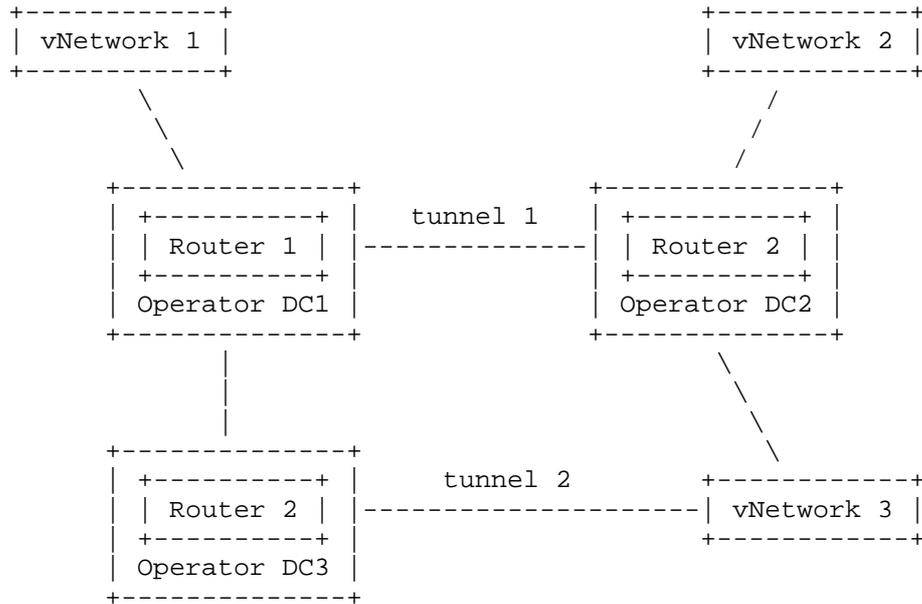
```
        +------------+                          +------------+
        | vNetwork 1 |                          | vNetwork 2 |
        +------------+                          +------------+
               \                                    /
                \                                  /
                 \                                /
        +--------------+              +--------------+
        | +----------+ |  tunnel 1    | +----------+ |
        | | Router 1 | |--------------| | Router 2 | |
        | +----------+ |              | +----------+ |
        | Operator DC1 |              | Operator DC2 |
        +--------------+              +--------------+
               |                             \
               |                              \
               |                               \
        +--------------+                        \
        | +----------+ |    tunnel 2      +------------+
        | | Router 2 | |------------------| vNetwork 3 |
        | +----------+ |                  +------------+
        | Operator DC3 |
        +--------------+
```
          Figure 7: Segment Tunnels for Virtual Network Operator

Assume the route of a direct tunnel built between virtual operator's networks (e.g. vNetwork1-to-vNetwork3) is out of control. For instance, the route may go through network node with problems, or the route may go across the border of different operators where QoS cannot be guaranteed.

In this case, the virtual network operator can configure three tunnels rather than one to connect vNetwork1 to vNetwork3: vNetwork1-to-Router1, Router1-to-Router2, Router2-to-vNetwork3.

After the initial network configuration is finished, if any problem is detected in any tunnel, the network management system can perform network wide optimization, taking all the routers into account and working out another set of tunnels if necessary.

        ECA Policy:

        Event: QoS parameters < threshold.

        Condition: multiple disjoint tunnels available.

        Action: Network wide tunnel optimization + traffic steering.

   In this case, the virtual SP can monitor the real-time QoS parameters
   between the virtual networks and the rented routers. If the QoS
   parameters exceed a threshold, and the virtual has deployed multiple
   rented routers which can provide multiple disjoint tunnels, then the
   network management/controller can trigger network wide tunnel
   optimization and/or perform traffic steering.

   When performing the tunnel optimization, the network management/
   controller may terminate the tunnel(s) which go through specific
   network area with problems, and/or build new tunnels, and/or perform
   network wide traffic steering. This will give the operator a lot of
   flexibility in controlling the network.

   The traffic steering may need to be combined with the network
   topology, and dynamically distribute traffic in the whole network.

4.5. Use Case 5: Instant VPN
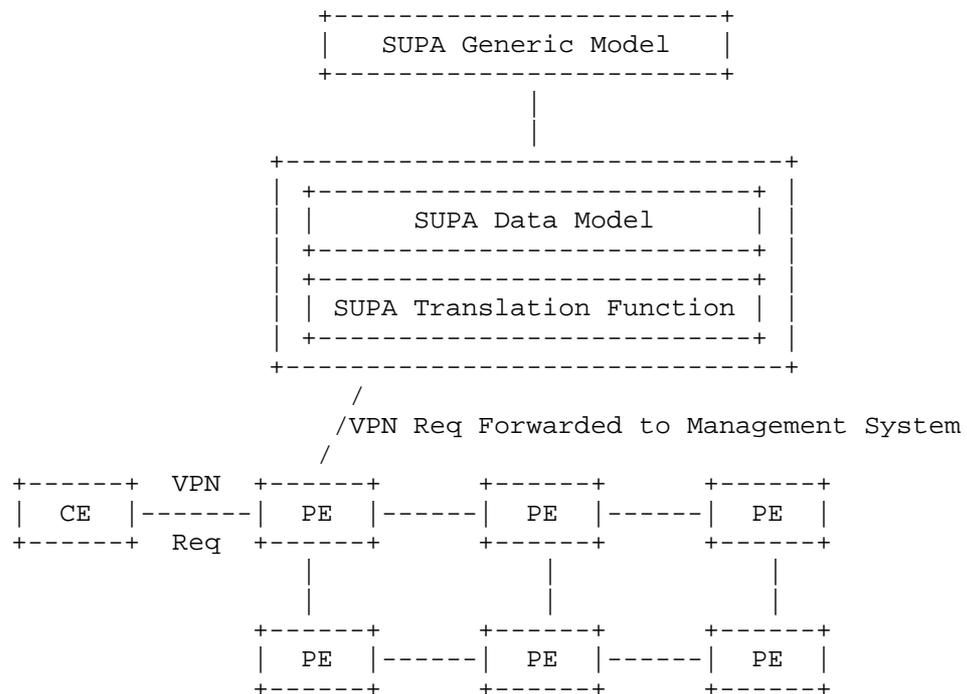
```
                  +-----------------------+
                  |  SUPA Generic Model   |
                  +-----------------------+
                             |
                             |
            +-------------------------------+
            | +---------------------------+ |
            | |    SUPA Data Model        | |
            | +---------------------------+ |
            | +---------------------------+ |
            | | SUPA Translation Function | |
            | +---------------------------+ |
            +-------------------------------+
                /
               /VPN Req Forwarded to Management System
              /
  +------+  VPN +------+      +------+      +------+
  | CE   |------| PE   |------| PE   |------| PE   |
  +------+  Req +------+      +------+      +------+
               |             |             |
               |             |             |
            +------+      +------+      +------+
            | PE   |------| PE   |------| PE   |
            +------+      +------+      +------+
         Figure 8: Instant VPN
```

Traditionally, when an operator needs to deploy VPN services for an
enterprise customer, they will send a service staff to the customer
site and make the wire connection between the CE and PE. The service
staff will also collect the configuration information, e.g.
port/frame/slot of PE, PE ID, etc, and then send the collected
information back to the management system. The management system
will configure the network according to this information as well as
the customer' information (such as bandwidth, SLA, etc). The problem
of this approach is that the service staff needs to collect the
connection information and feedback to the management system, and
MUST make sure the information matches the actual connection. This
process is error prone.

New approach should not count on the physical / geographical
information feedback by the service staff, minimize the operation
procedures. The CE should send authentication (with credentials)
request to the PE, and PE should forward the request to the
management system together with port/frame/slot on which the request
is received, the PE ID etc.

Target: Configure VPN for an enterprise customer to connect its
enterprise network with VPC

ECA Policy:

Event: service management system receive a CE request for VPN
creation (forwarded by PE).

Condition: Authentication and Authorization results are OK.

Action: Configure VPN based on received request, including the user's
grade and physical info (port/slot/frame/route id, etc, from which
the request is received).

4.6. Use Case 6: traffic optimization and Qos assurance on ISP DC

   ISPs usually build DCs at the core network border, DCs have more than
   one uplinks to DC core network; users at MAN can access the core from
   different links too. Different links may have unbalanced load
   because of the nature of network traffic. In order to provide
   service assurance for import tenant, network administrators need to
   schedule the traffic in specific periods. Traditional network
   management is usually complex with a long cycle, so it is difficult
   to meet the request of real-time traffic optimization.
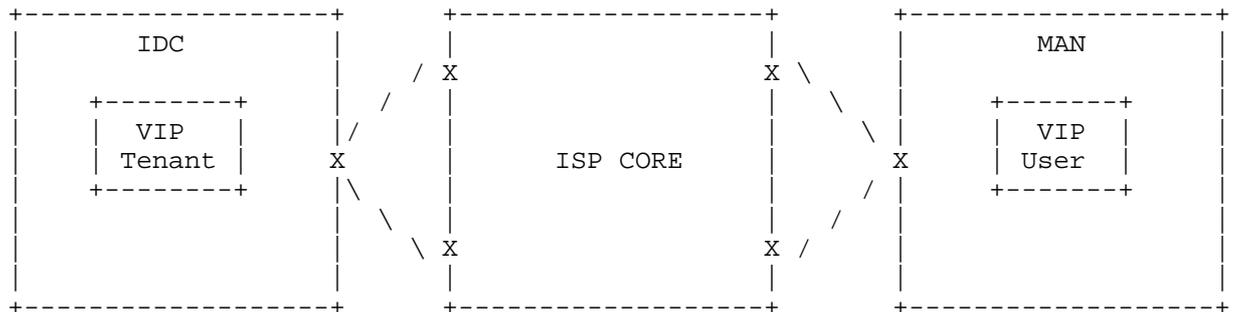
```
+------------------+     +------------------+     +------------------+
|      IDC         |     |                  |     |      MAN         |
|                  |   / X                X \ |   |                  |
|    +--------+    |  /  |                  |  \ |   +-------+        |
|    |  VIP   |    | /   |                  |   \ |   |  VIP  |       |
|    | Tenant |    X     |    ISP CORE      |     X   | User  |      |
|    +--------+    |\    |                  |    / |   +-------+      |
|                  | \   |                  |   /  |                  |
|                  |  \ X                X /  |   |                  |
|                  |    |                  |    |   |                  |
+------------------+     +------------------+     +------------------+
```
        Figure 9: traffic optimization and Qos assurance on ISP DC

   Assume that a network controller or orchestrator can monitor network
   topology, including real-time link utilization and flow information.
   When utilization of a link reaches a certain threshold, specific
   flows should be steered to a low load link according to IP address
   and AS number.

   Large service provider's traffic usually has time characteristics,
   for example, online big sales, network administrator can provide
   bandwidth assurance for important tenant according to their IP
   address.

Target 1: a DC has multiple external links. When the load on a link
is over a threshold, perform traffic steering for a better bandwidth
resource usage.

ECA Policy:

Event: load on a DC link exceeds threshold or a VIP tenant needs
bandwidth assurance.

Condition: DC has multiple external links.

Action: steer VIP's traffic to link with low load in a specific
period.

Target 2: Tenants or users may have critical request on network Qos.
When there are enough bandwidth along the link, perform resource
reservation for VIP's traffic on specific links.

ECA Policy:

Event: Tenants or users have critical network requests.

Condition: Resources along the link are enough for reservation.

Action: perform resource reservation for VIP's traffic on specific
links.

5. IANA Considerations

   This document makes no request of IANA.

   Note to RFC Editor: this section may be removed on publication as an
   RFC.

6. Security Considerations

   Since SUPA models can be used to generate configurations for network
   elements, the management applications which send models to service
   management system must go through authentication and authorization.

   The handling of conflicition of different polcies is out of scope of
   this memo.

7. Acknowledgements

   This document has benefited from reviews, suggestions, comments and
   proposed text provided by the following members, listed in

alphabetical order: Juergen Schoenwaelder, John Strassner, James
Huang.

8. Normative References

   [RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate
               Requirement Levels", BCP 14, RFC 2119,
               DOI 10.17487/RFC2119, March 1997,
               <http://www.rfc-editor.org/info/rfc2119>.

   [RFC6020]   Bjorklund, M., Ed., "YANG - A Data Modeling Language for
               the Network Configuration Protocol (NETCONF)", RFC 6020,
               DOI 10.17487/RFC6020, October 2010,
               <http://www.rfc-editor.org/info/rfc6020>.

Authors' Addresses

   Ying Cheng (Editor)
   China Unicom
   No.21 Financial Street, XiCheng District
   Beijing 100033
   China

   Phone: +86-010-66259394
   Email: chengying10@chinaunicom.cn

   Dapeng Liu
   Alibaba Group
   Beijing 100022
   China

   Email: max.ldp@alibaba-inc.com

   Borui Fu (Editor)
   China Telecom
   Beijing
   China

   Email: fubr@ctbri.com.cn

Dacheng Zhang
Freelancer
Beijing
China

Email: Dacheng.zhang@gmail.com

Narasimha Vadrevu
VN Telecom Consultancy

Email: vadrevun@von20.com

                   Generic Policy Information Model for
              Simplified Use of Policy Abstractions (SUPA)
              draft-ietf-supa-generic-policy-info-model-01

Abstract

   This document defines an information model for representing
   policies using a common extensible framework that is independent
   of language, protocol, repository. It is also independent of the
   level of abstraction of the content and meaning of a policy.

Status of this Memo

Copyright Notice

Table of Contents

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

1. Overview

   This document defines an information model for representing
   policies using a common extensible framework that is independent
   of language, protocol, repository, and the level of abstraction of
   the content and meaning of a policy. This enables a common set of
   concepts defined in this information model to be mapped into
   different representations of policy (e.g., procedural, imperative,
   and declarative). It also enables different data models that use
   different languages, protocols, and repositories to optimize
   their usage. The definition of common policy concepts also
   provides better interoperability by ensuring that each data
   model can share a set of common concepts, independent of its
   level of detail or the language, protocol, and/or repository
   that it is using. It is also independent of the target data
   model that will be generated.

   This version of the information model focuses on defining one
   type of policy rule: the event-condition-action (ECA) policy rule.
   Accordingly, this document defines two sets of model elements:

      1. A framework for defining the concept of policy,
          independent of how policy is represented or used; this is
          called the SUPA Generic Policy Information Model (GPIM)
      2. A framework for defining a policy model that uses the
          event-condition-action (ECA) paradigm; this is called the
          SUPA ECA Policy Rule Information Model (EPRIM), and extends
          concepts from the GPIM.

   The combination of the GPIM and the EPRIM provides an extensible
   framework for defining policy that uses an event-condition-action
   representation that is independent of data repository, data
   definition language, query language, implementation language, and
   protocol.

   The Appendices describe how the structure of the GPIM defines a
   set of generic concepts that enables other types of policies, such
   as declarative (or "intent-based") policies, to be added later.

1.1. Introduction

   Simplified Use of Policy Abstractions (SUPA) defines a technology-
   independent neutral information model for creating high-level,
   possibly network-wide policies as input and producing element
   configurations (either whole or snippets) as output. SUPA addresses
   the needs of operators, end-users, and application developers to
   represent multiple types of ECA policy rules, such as for traffic
   selection and configuration or security. These ECA policy rules may
   vary in the level of abstraction to suit the needs of different
   actors (e.g., end-users vs. administrators) [1], [10].

Different constituencies of users would like to use terminology and
concepts that are familiar to each constituency. Rather than require
multiple software systems to be used for each constituency, a common
information model enables these different concepts and terms to be
mapped to elements in the information model. This facilitiates the
use of a single software system to generate data models for each
language. In the example shown in Figure 1 (which is a simplified
Policy Continuum [10]), each constituency uses different concepts
and terms (according to their skill sets and roles) to formulate
(ECA) policy rules that are useful for their job functions. A
unified information model is one way to build a consensual lexicon
that enables terms from one language to be mapped to terms of
another language. This approach enables the syntax of each language
to be modified appropriate to its user while keeping a common set
of semantics for all languages. This is shown in Figure 1.

```
                        +-------------------+
                        | Information Model |
                        |  and one or more  +<-----------------------+
                        |    Data Models    |                        |
                        +-------------------+                        |
                                                                     |
                        +-------------------+                        |
+---------------+     \| High-level Policies |    \+-------------+   |
| Business User |----|  Without Technical  |----| Language #1 +<----+
+---------------+     /|     Terminology     |    /+-------------+   |
                        +-------------------+                        |
                                                                     |
                        +-------------------+                        |
+---------------+     \| Policies That Use  |    \+-------------+   |
|   Developer   |----| Classes, Attributes,|----| Language #2 +<----+
+---------------+     /| Relationships, ... |    /+-------------+   |
                        +-------------------+                        |
                                                                     |
      ...                     ...                       ...          |
                                                                     |
                        +-------------------+                        |
+---------------+     \| Low-level Policies |    \+-------------+   |
|     Admin     |----| with Technology-   |----| Language #n +<----+
+---------------+     /| Specific Terms in a |    /+-------------+
                        | Specific Language  |
                        +-------------------+
```
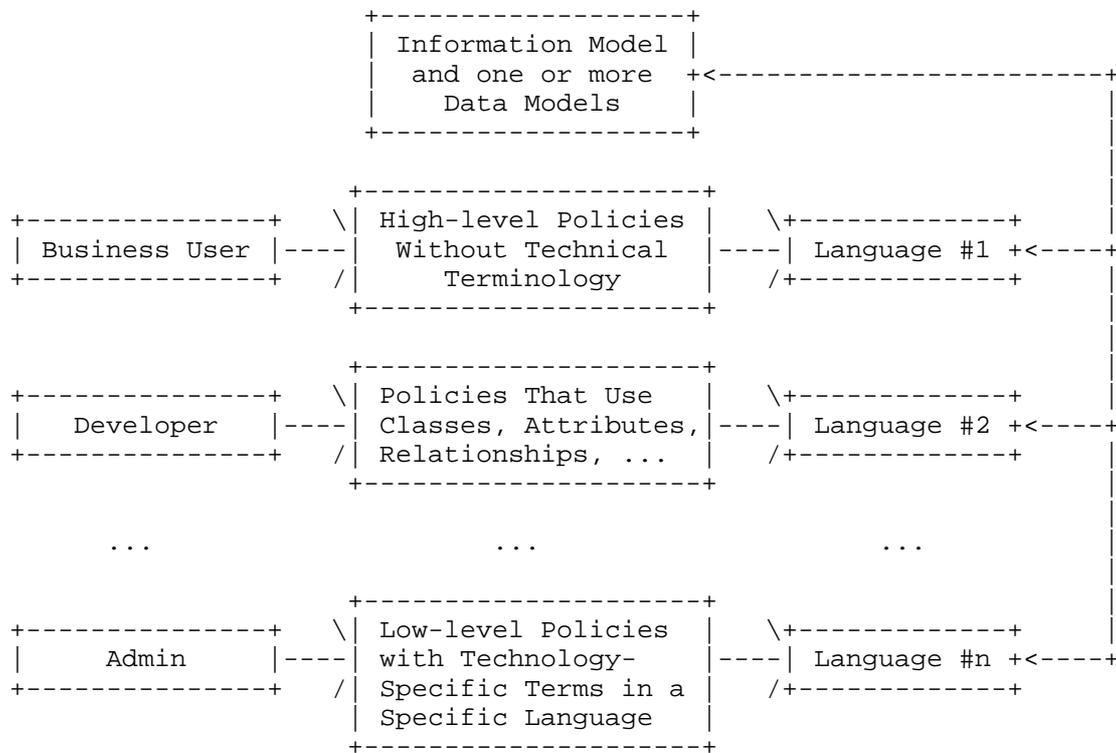
Figure 1. Different Constituencies Need Different Policies

More importantly, an information model defines concepts in a
uniform way, enabling formal mapping processes to be developed to
translate the information model to a set of data models. This
simplifies the process of constructing software to automate the
policy management process. It also simplifies the language
generation process, though that is beyond the scope of this
document.

This common framework takes the form of an information model that is divided into one high-level module and one or more number of lower-level modules. A lower-level module extends the higher-level module into a new domain; each lower-level domain module can itself be extended to model more granular domain-specific (but still technology- and vendor-independent) concepts as necessary.

Conceptually, a set of model elements (e.g., classes, attributes, constraints, and relationships) are used to define the Generic Policy Information Model (GPIM); this module defines a common set of policy concepts that are independent of the type of policy (e.g., imperative, procedural, declarative, or otherwise). Then, any number of additional modules can be derived from the GPIM; each additional module MUST extend the GPIM to define a new type of policy rule by adding to the GPIM. Each additoinal module MUST NOT alter any of the model elements of the GPIM. The use of extensions preserves the interoperability of this approach; if the base GPIM was modified, then this would adversely compromise interoperability.

The SUPA ECA Policy Rule Information Model (EPRIM) extends the GPIM to represent policy rules that use the Event-Condition-Action (ECA) paradigm.

1.2. Changes Since Version -00

There are several changes in this version of this document compared to the previous versions of this document. They are:

     1) Rewrote parts of the Introduction.
     2) Clarified how to extend the GPIM and EPRIM
     3) Redesigned the SUPAPolicyVersionMetadataDef class
     4) Added Fully Qualified Path Names where applicable
     5) Removed Appendices B and C (declarative policies)
     6) Fixed typos

2. Conventions Used in This Document

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 [RFC2119]. In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying [RFC2119] significance.

3. Terminology

   This section defines acronyms, terms, and symbology used in the
   rest of this document.

3.1. Acronyms

   CLI   Command Line Interface
   CRUD   Create, Read, Update, Delete
   CNF   Conjunctive Normal Form
   DNF   Disjunctive Normal Form
   ECA   Event-Condition-Action
   EPRIM   (SUPA) ECA Policy Rule Information Model
   GPIM   (SUPA) Generic Policy Information Model
   OAM&P   Operations, Administration, Management, and Provisioning
   OID   Object IDentifier
   SUPA   Simplified Use of Policy Abstractions
   TMF   TeleManagent Forum (TM Forum)
   UML   Unified Modeling Language
   URI   Uniform Resource Identifier
   YANG   A data definition language for use with NETCONF
   ZOOM   Zero-touch Orchestration, Operations, and Management
               (a TMF project that also works on information models)

3.2. Definitions

   This section defines the terminology that is used in this document.

3.2.1. Core Terminology

   The following subsections define the terms "information model" and
   "data model", as well as "container" and "policy container".

3.2.1.1. Information Model

   An information model is a representation of concepts of interest
   to an environment in a form that is independent of data repository,
   data definition language, query language, implementation language,
   and protocol.

   Note: this definition is more specific than that of [RFC3198], so
   as to focus on the properties of information models. That definition
   was: "An abstraction and representation of the entities in a managed
   environment, their properties, attributes and operations, and the
   way that they relate to each other. It is independent of any
   specific repository, software usage, protocol, or platform."

3.2.1.2. Data Model

   A data model is a representation of concepts of interest to an
   environment in a form that is dependent on data repository, data
   definition language, query language, implementation language, and/or
   protocol (typically, but not necessarily, all five).

   Note: this definition is more specific than that of [RFC3198], so
   as to focus on the properties of data models that are generated
   from information models. That definition was: "A mapping of the
   contents of an information model into a form that is specific to a
   particular type of data store or repository."

3.2.1.3. Class

   A class is a set of objects that exhibit a common set of
   characteristics and behavior.

3.2.1.3.1. Abstract Class

   An abstract class is a class that cannot be directly instantiated.
   It MAY have abstract or concrete subclasses. It is denoted with a
   capital A (for abstract) near the top-left side of the class.

3.2.1.3.2. Concrete Class

   A concrete class is a class that can be directly instantiated. Note
   that classes are either abstract or concrete. In addition, once a
   class has been defined as concrete in the hierarchy, all of its
   subclasses MUST also be concrete. It is denoted with a capital C
   (for concrete) near the top-left side of the class.

3.2.1.4. Container

   A container is an object whose instances may contain zero or more
   additional objects, including container objects. A container
   provides storage, query, and retrieval of its contained objects
   in a well-known, organized way.

3.2.1.5. PolicyContainer

   In this document, a PolicyContainer is a special type of container
   that provides at least the following three functions:

      1. It uses metadata to define how its content is interpreted
      2. It separates the content of the policy from the
          representation of the policy
      3. It provides a convenient control point for OAM&P operations

   The combination of these three functions enables a PolicyContainer
   to define the behavior of how its constituent components will be
   accessed, queried, stored, retrieved, and how they operate.

This document does NOT define a specific data type to implementation a PolicyContainer, as many different types of data types can be used. However, the data type chosen SHOULD NOT allow duplicate members in the PolicyContainer. In addition, order is irrelevant, since priority will override any initial order of the members of this PolicyContainer.

3.2.2. Policy Terminology

The following terms define different policy concepts used in the SUPA Generic Policy Information Model (GPIM). Note that the prefix "SUPA" is used for all classes and relationships defined in this model to ensure name uniqueness. Similarly, the prefix "supa" is defined for all SUPA class attributes.

3.2.2.1. SUPAPolicyObject

A SUPAPolicyObject is the root of the GPIM class hierarchy. It is an abstract class that all classes inherit from, except the SUPAPolicyMetadata class and its subclasses.

3.2.2.2. SUPAPolicy

A SUPAPolicy is, in this version of this document, an ECA policy rule that is a type of PolicyContainer. The PolicyContainer MUST contain an ECA policy rule, SHOULD contain one or more SUPAPolicyMetadata objects, and MAY contain other elements that define the semantics of the policy rule. Policies are generically defined as a means to monitor and control the changing and/or maintaining of the state of one or more managed objects [1]. In this context, "manage" means that one or more of the following six fundamental operations are supported: create, read, write, delete, start, and stop) [16].

3.2.2.3. SUPAPolicyClause

A SUPAPolicyClause is an abstract class. Its subclasses define different types of clauses that are used to create the content for different types of SUPAPolicies.

For example, the SUPABooleanClause subclass models the content of a SUPAPolicy as a Boolean clause, where each Boolean clause is made up of a set of reusable objects. In contrast, a SUPAEncodedClause encodes the entire clause as a set of attributes. All types of SUPAPolicies MUST use one or more SUPAPolicyClauses to construct a SUPAPolicy.

3.2.2.4. SUPAECAPolicyRule

   An Event-Condition-Action (ECA) Policy (SUPAECAPolicyRule) is an
   abstract class that is a type of PolicyContainer. It represents
   a policy rule as a three-tuple, consisting of an event, a
   condition, and an action clause. In an information model, this
   takes the form of three different aggregations, one for each
   clause. Each clause MUST be represented by at least one
   SUPAPolicyClause. Optionally, the SUPAECAPolicyRule MAY contain
   zero or more SUPAPolicySources, zero or more SUPAPolicyTargets,
   and zero or more SUPAPolicyMetadata objects. Note that for this
   version of this document, ECA Policy Rules are the **only**
   types of Policies that are defined.

3.2.2.5. SUPAMetadata

   Metadata is, literally, data about data. SUPAMetadata is an
   abstract class that contains prescriptive and/or descriptive
   information about the object(s) to which it is attached. While
   metadata can be attached to any information model element, this
   document only considers metadata attached to classes and
   relationships.

   When defined in an information model, each instance of the
   SUPAMetadata class MUST have its own aggregation relationship
   with the set of objects that it applies to. However, a data model
   MAY map these definitions to a more efficient form (e.g.,
   flattening the object instances into a single object instance).

3.2.2.6. SUPAPolicyTarget

   SUPAPolicyTarget is an abstract class that defines a set of
   managed objects that may be affected by the actions of a
   SUPAPolicyClause. A SUPAPolicyTarget may use one or more
   mechanisms to identify the set of managed objects that it
   affects; examples include OIDs and URIs.

   When defined in an information model, each instance of the
   SUPAPolicyTarget class MUST have its own aggregation
   relationship with each SUPAPolicy that uses it. However, a
   data model MAY map these definitions to a more efficient form
   (e.g., flattening the SUPAPolicyTarget, SUPAMetadata, and
   SUPAPolicy object instances into a single object instance).

3.2.2.7. SUPAPolicySource

   SUPAPolicySource is an abstract class that defines a set of
   managed objects that authored this SUPAPolicyClause. This is
   required for auditability and authorization policies, as well
   as some forms of deontic and alethic logic.

A SUPAPolicySource may use one or more mechanisms to identify the
set of managed objects that authored it; examples include OIDs and
URIs. Specifically, policy CRUD MUST be subject to authentication
and authorization, and MUST be auditable. Note that the mechanisms
for doing these three operations are currently not included, and
are for further discussion.

When defined in an information model, each instance of the
SUPAPolicySource class MUST have its own aggregation relationship
with each SUPAPolicy that uses it. However, a data model MAY map
these definitions to a more efficient form (e.g., flattening the
SUPAPolicySource, SUPAMetadata, and SUPAPolicy object instances
into a single object instance).

3.2.3. Modeling Terminology

   The following terms define different types of relationships used
   in the information models of the SUPA Generic Policy Information
   Model (GPIM).

3.2.3.1. Inheritance

   Inheritance makes an entity at a lower level of abstraction (e.g.,
   the subclass) a type of an entity at a higher level of abstraction
   (e.g., the superclass). Any attributes and relationships that are
   defined for the superclass are also defined for the subclass.
   However, a subclass does NOT change the characteristics or behavior
   of the attributes or relationships of the superclass that it
   inherits from. Formally, this is called the Liskov Substitution
   Principle [7]. This principle is one of the key characteristics
   that is NOT followed in [4], [6], [RFC3060], and [RFC3460].

   A subclass MAY add new attributes and relationships that refine
   the characteristics and/or behavior of it compared to its
   superclass. A subclass MUST NOT change inherited attributes or
   relationships.

3.2.3.2. Relationship

   A relationship is a generic term that represents how a first set
   of entities interact with a second set of entities. A recursive
   relationship sets the first and second entity to the same entity.
   There are three basic types of relationships, as defined in the
   subsections below: associations, aggregations, and compositions.

   A subclass MUST NOT change the multiplicity (see section 3.2.3.7)
   of a relationship that it inherits. A subclass MUST NOT change any
   attributes of a relation that it inherits that is realized using
   an association class (see section 3.2.3.6).

3.2.3.3. Association

   An association represents a generic dependency between a first
   and a second set of entities. In an information model, an
   association MAY be represented as a class.

3.2.3.4. Aggregation

   An aggregation is a stronger type (i.e., more restricted
   semantically) of association, and represents a whole-part
   dependency between a first and a second set of entities. Three
   objects are defined by an aggregation: the first entity, the
   second entity, and a new third entity that represents the
   combination of the first and second entities.

   The entity owning the aggregation is referred to as the
   "aggregate", and the entity that is aggregated is referred to as
   the "part". In an information model, an aggregation MAY be
   represented as a class.

3.2.3.5. Composition

   A composition is a stronger type (i.e., more restricted
   semantically) of aggregation, and represents a whole-part
   dependency with two important behaviors. First, an instance of the
   part is included in at most one instance of the aggregate at a
   time. Second, any action performed on the composite entity (i.e.,
   the aggregate) is propagated to its constituent part objects.
   For example, if the composite entity is deleted, then all of its
   constituent part entities are also deleted. This is not true of
   aggregations or associations - in both, only the entity being
   deleted is actually removed, and the other entities are unaffected.
   In an information model, a composition MAY be represented as
   a class.

3.2.3.6. Association Class

   A relationship may be implemented as an association class. This is
   used to define the relationship as having its own set of features.
   (Note: in this document, all relationships are implemented as
   association classes for consistency and to simplify implementation.)
   More specifically, if the relationship is implemented as an
   association class, then the attributes of the association class, as
   well as other relationships that the association class participates
   in, may be used to define the semantics of the relationship. If the
   relationship is not implemented as an association class, then no
   additional semantics (beyond those defined by the type of the
   relationship) are expressed by the relationship.

3.2.3.7. Multiplicity

   A specification of the range of allowable cardinalities that a set
   of entities may assume. This is always a pair of ranges, such as
   1 - 1 or 0..n - 2..5.

3.2.3.8. Navigability

   A relationship may restrict one object from accessing the other
   object. This document defines two choices:

      1. Each object is navigable by the other, which is indicated
          by NOT providing any additional symbology, or
      2. An object A can navigate to object B, but object B cannot
          navigate to object A. This is indicated by an open-headed
          arrow pointing to the object that cannot navigate to the
          other object. An example is shown below:

```
      +---------+            3..4 +---------+
      |         | 1..2          \|          |
      | Class A |---------------| Class B  |
      |         |             /|          |
      +--------+                +---------+
```
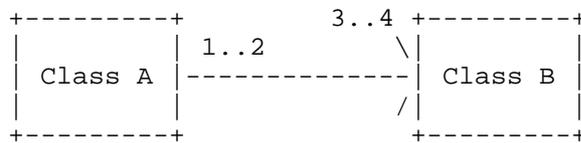
   The above figure shows a navigability restriction. Class A can
   navigate to Class B, but Class B cannot navigate to Class A. This is
   a mandatory association, since none of the multiplicities contain a
   '0'. This association reads as follows:

      Class A depends on 3 to 4 instances of Class B, and
      Class B depends on 1 to 2 instances of Class A.

3.3. Symbology

   The following symbology is used in this document.

3.3.1. Inheritance

   Inheritance: a subclass inherits the attributes and relationships
             of its superclass, as shown below:

```
                   +------------+
                   | Superclass |
                   +------+-----+
                        / \
                         I
                         I
                         I
                   +------+-----+
                   |  Subclass  |
                   +------------+
```
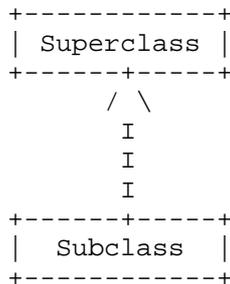
3.3.2. Association

   Association: Class B depends on Class A, as shown below:

```
                                      +---------+     +---------+
    +---------+     +---------+        |         |    \|         |
    | Class A |------| Class B |       | Class A |------| Class B |
    +---------+     +---------+        |         |    /|         |
                                      +---------+     +---------+

          association with no              association with
        navigability restrictions        navigability restrictions
```
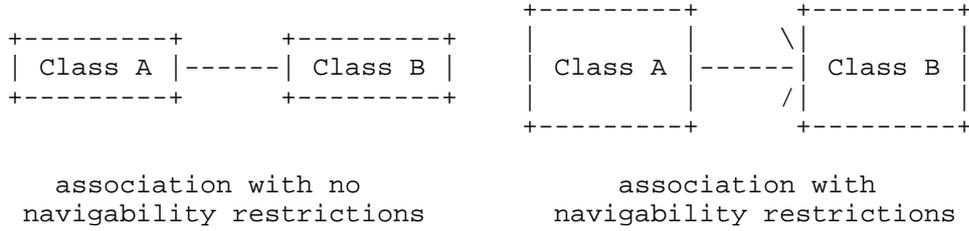
3.3.3. Aggregation

   Aggregation: Class B is the part, Class A is the aggregate,
            as shown below:

```
    +---------+                        +---------+         +---------+
    |         |/ \   +---------+        |         |/ \    \|         |
    | Class A | A ---| Class B |        | Class A | A ------| Class B |
    |         |\ /   +---------+        |         |\ /    /|         |
    +---------+                        +---------+         +---------+

          aggregation with no              aggregation with
        navigability restrictions        navigability restrictions
```
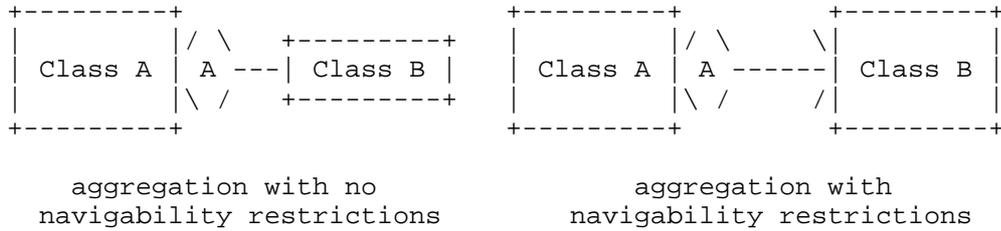
3.3.4. Composition

   Composition: Class B is the part, Class A is the composite,
            as shown below:

```
    +---------+                        +---------+         +---------+
    |         |/ \   +---------+        |         |/ \    \|         |
    | Class A | C ---| Class B |        | Class A | C ------| Class B |
    |         |\ /   +---------+        |         |\ /    /|         |
    +---------+                        +---------+         +---------+

          composition with no              composition with
        navigability restrictions        navigability restrictions
```
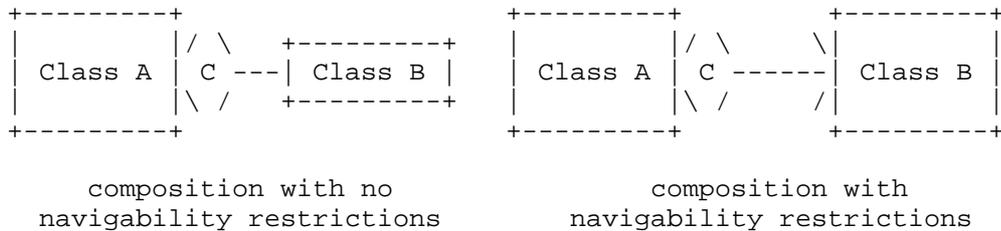
3.3.5. Association Class

   Association Class: Class C is the association class implementing
                    the relationship D between classes A and B

```
            +---------+          +---------+
            | Class A |----+-----| Class B |
            +---------+    ^      +---------+
                          |
                          |
              +----------+----------+
              | Association Class C |
              +---------------------+
```
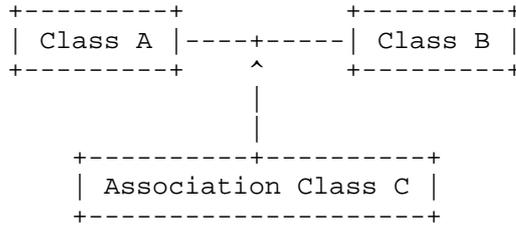
3.3.6. Abstract vs. Concrete Classes

   In UML, abstract classes are denoted with their name in italics.
   For this draft, a capital 'A' will be placed at either the top
   left or right corner of the class to signify that the class is
   abstract. Similarly, a captial 'C' will be placed in the same
   location to represent a concrete class. This is shown below.

```
              A                    C
              +---------+          +---------+
              | Class A |          | Class B |
              +---------+          +---------+

            An Abstract Class    A Concrete Class
```
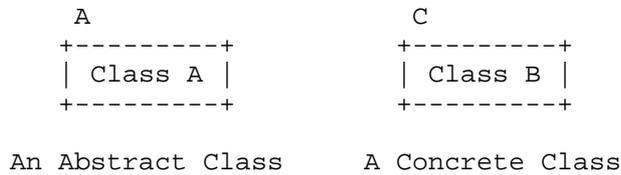
4. Policy Abstraction Architecture

   This section describes the motivation for the policy abstractions
   that are used in SUPA. The following abstractions are provided:

      o The GPIM defines a technology-neutral information model that
        can express the concept of Policy.
           o All classes, except for SUPAPolicyMetadata, inherit from
             SUPAPolicyObject, or one of its subclasses.
           o SUPAPolicyObject and SUPAPolicyMetadata are designed to
             inherit from classes in another model; the GPIM does not
             define an "all-encompassing" model.
      o This version of this document restricts the expression of
        Policy to a set of event-condition-action clauses.
           o Each clause is defined as a Boolean expression, and MAY
             also be defined as a reusable object.
           o Clauses may be combined to form more complex Boolean
             expressions.
      o The purpose of the GPIM is to enable different policies that
        have fundamentally different representations to share common
        model elements. Policy statements, which are implemented as
        instances of the SUPAPolicyClause class, separates the content
        of a Policy from its representation. This is supported by:
           o All policy rules (of which SUPAECAPolicyRule is the
             first example of a concrete class) are derived from
             the SUPAPolicyStructure class.
           o All objects that are components of policy rules are
             derived from the SUPAPolicyComponentStructure class.
           o A SUPAPolicy MUST contain at least one SUPAPolicyClause.
           o A SUPAPolicy MAY specify one or more SUPAPolicyTarget,
             SUPAPolicySource, and SUPAPolicyMetadata objects to
             augment the semantics of the SUPAPolicy
      o A SUPAPolicyClause has two subclasses:
           o A SUPABooleanClause, which is used to build
             SUPAECAPolicyRules from reusable objects.
           o A SUPAEncodedClause, which is used for using
             attributes instead of objects to construct a
             SUPAECAPolicyRule.
      o A SUPAECAPolicyRule defines the set of events and conditions
        that are responsible for executing its actions; it MUST have
        at least one event clause, at least one condition clause, and
        at least one action clause.
           o The action(s) of a SUPAECAPolicyRule are ONLY executed
             if both the event and condition clauses evaluate to TRUE
           o A SUPAPolicyAction MAY invoke another SUPAECAPolicyRule
             (see section 6.13).
      o SUPAMetadata MAY be defined for any SUPAPolicyObject class.
      o SUPAMetadata MAY be prescriptive and/or descriptive in nature.

4.1. Motivation

   The power of policy management is its applicability to many
   different types of systems. There are many different actors that
   can use a policy management system, including end-users, operators,
   application developers, and administrators. Each of these
   constituencies have different concepts and skills, and use
   different terminology. For example, an operator may want to express
   an operational rule that states that only Platinum and Gold users
   can use streaming multimedia applications. As a second example, a
   network administrator may want to define a more concrete policy
   rule that looks at the number of dropped packets and, if that
   number exceeds a programmable threshold, changes the queuing and
   dropping algorithms used.

   SUPA may be used to define other types of policies, such as for
   systems and operations management; an example is: "All routers and
   switches must have password login disabled". See section 3 of [8]
   for additional declarative and ECA policy examples.

   All of the above examples are commonly referred to as "policy
   rules", but they take very different forms, since they are at very
   different levels of abstraction and typically authored by
   different actors. The first was very abstract, and did not contain
   any technology-specific terms, while the second was more concrete,
   and likely used technical terms of a general (e.g., IP address
   range, port numbers) as well as a vendor-specific nature (e.g.,
   specific queuing, dropping, and/or scheduling algorithms
   implemented in a particular device). The third restricted the type
   of login that was permissible for certain types of devices in the
   environment.

   Note that the first two policy rules could directly affect each
   other. For example, Gold and Platinum users might need different
   device configurations to give the proper QoS markings to their
   streaming multimedia traffic. This is very difficult to do if a
   common policy model does not exist, especially if the two policies
   are authored by different actors that use different terminology
   and have different skill sets. More importantly, the users of
   these two policies likely have different job responsibilities.
   They may have no idea of the concepts used in each policy. Yet,
   their policies need to interact in order for the business to
   provide the desired service. This again underscores the need for
   a common policy framework.

   Certain types of policy rules (e.g., ECA) may express actions, or
   other types of operations, that contradict each other. SUPA
   provides a rich object model that can be used to support language
   definitions that can find and resolve such problems.

4.2. SUPA Approach

   The purpose of the SUPA Generic Policy Information Model (GPIM) is
   to define a common framework for expressing policies at different
   levels of abstraction. SUPA uses the GPIM as a common vocabulary
   for representing policy concepts that are independent of language,
   protocol, repository, and level of abstraction. This enables
   different actors to author and use policies at different levels of
   abstraction. This forms a policy continuum [1] [2], where more
   abstract policies can be translated into more concrete policies,
   and vice-versa.

   Most systems define the notion of a policy as a single entity.
   This assumes that all users of policy have the same terminology,
   and use policy at the same level of abstraction. This is rarely,
   if ever, true in modern systems. The policy continuum defines a
   set of views (much like RM-ODP's viewpoints [9]) that are each
   optimized for a user playing a specific role. SUPA defines the
   GPIM as a standard vocabulary and set of concepts that enable
   different actors to use different formulations of policy. This
   corresponds to the different levels in the policy continuum, and
   as such, can make use of previous experience in this area.

   It may be necessary to translate a Policy from a general to a more
   specific form (while keeping the abstraction level the same). For
   example, the declarative policy "Every network attached to a VM
   must be a private network owned by someone in the same group as
   the owner of the VM" may be translated to more formal form (e.g.,
   Datalog (as in OpenStack Congress). It may also be necessary to
   translate a Policy to a different level of abstraction. For
   example, the previous Policy may need to be translated to a form
   that network devices can process directly. This requires a common
   framework for expressing policies that is independent of the level
   of abstraction that a Policy uses.

4.3. SUPA Generic Policy Information Model Overview

   Figure 2 illustrates the approach for representing policy rules
   in SUPA. The top two layers are defined in this document; the
   bottom layer (Data Models) are defined in separate documents.
   Conceptually, the GPIM defines a set of objects that define the
   key elements of a Policy independent of how it is represented or
   its content. As will be shown, there is a significant difference
   between SUPAECAPolicyRules (see Section 6) and other types of
   policies (see Section 7). In principle, other types of SUPAPolicies
   could be defined, but the current charter is restricted to using
   only event-condition-action SUPAPolicies as exemplars.

   Note: the GPIM MAY be used without the EPRIM. However, in order to
   use the EPRIM, the GPIM MUST also be used.

```
           +------------------------------------------------+
           | SUPA Generic Policy Information Model (GPIM)    |
           +---------------------+--------------------------+
                              / \
                               |
                               |
             +-----------------+-------------+
             |                               |
             |                               |
     +-----------+-------------+   +------------+-------------+
     |     SUPAECAPolicyRule    |   | Other Policy Models that |
     |  Information Model (EPRIM)|   | are Derived from the GPIM |
     +-----------+-------------+   +------------+-------------+
              / \                             / \
               |                               |
               |                               |
     +-----------+-----------+     +-----------+------------+
     |     ECAPolicyRule      |     |     Other Types of     |
     |      Data Model        |     |      Data Models       |
     +-----------------------+     +------------------------+
```
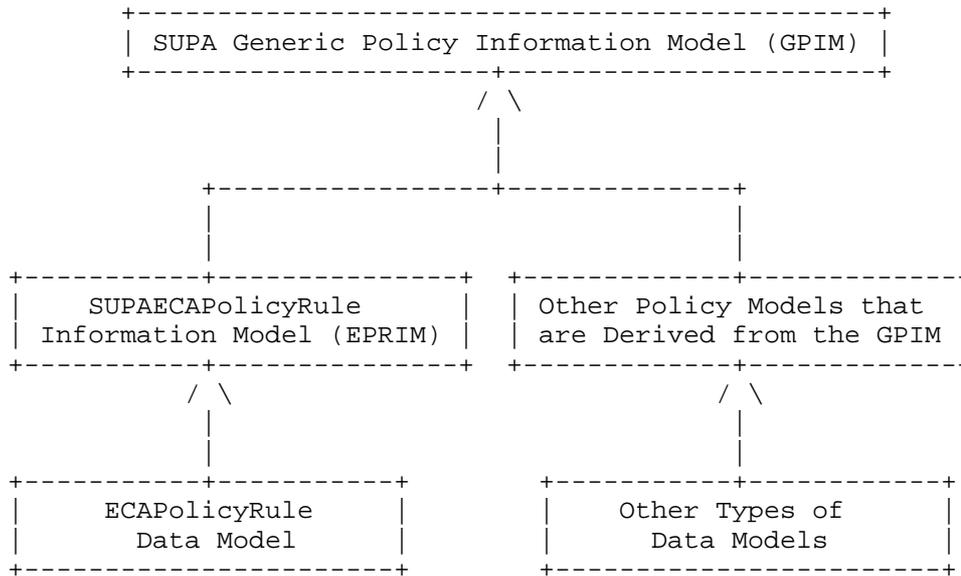
                Figure 2. Overview of SUPA Policy Rule Abstractions

   This draft defines the GPIM and EPRIM. This draft further assumes
   that the SUPA Information Model is made up of either the GPIM or the
   combination of the GPIM and the EPRIM. Extensions to both the GPIM
   and the EPRIM can be made as long as these extensions do not
   conflict with the content and structure defined in the GPIM and
   EPRIM. If the GPIM and EPRIM are part of another information model,
   then they should collectively still define a single information
   model. The GPIM defines the following concepts:

     o A class defining the top of the GPIM class hierarchy, called
       SUPAPolicyObject
     o Four subclasses of SUPAPolicyObject, representing:
       o the top of the Policy hierarchy, called
         SUPAPolicyStructure
       o the top of the Policy component hierarchy, called
         SUPAPolicyComponentStructure
       o PolicySource
       o PolicyTarget

   The SUPAPolicyStructure class is the superclass for all types of
   Policies (e.g., imperative, declarative, and others). This
   document is currently limited to imperative (e.g., ECA) policies.
   However, care has been taken to ensure that the attributes and
   relationships of the SUPAPolicyStructure class are extensible,
   and can be used for more types of policies than just ECA policies.

   This yields the following high-level structure:

```
                        A
                        +-----------------+
                        | SUPAPolicyObject |
                        +--------+---------+
                                / \
                                 I
                                 I
             +---------------+------------------+
             I               I                  I
      A      I               I      A           I
  +-------+-----------+      I   +-----------+----------------+
  | SUPAPolicyStructure |    I   | SUPAPolicyComponentStructure |
  +----+--------------+      I   +-----------+----------------+
     / \      I               I            / \
     I        I     +--------+----+        I
     I        I     I             I        I
     I   C    I     I             I        I
     I  +-------+----------+      I        I
     I  | SUPAPolicySource |      I        I
     I  +-----------------+       I        I
     I                            I        I
     I         +------------+     I        I
     I         I                  I        I
     I   C     I             +-----+-------+
     I  +-------+----------+  I             I
     I  | SUPAPolicyTarget |  I     A       I                I
     I  +-----------------+   +--------+---------+   I
  A  I                        | SUPAPolicyClause |   I
+-----+-----------+           +-----------------+    I
| SUPAECAPolicyRule|                                 I
+-----------------+              A                    I
                       +-------------------+---------+
                       | SUPAPolicyComponentDecorator |
                       +----------------------------+
```

Figure 3. Functional View of the Top-Level GPIM

   Note that all classes except the SUPAPolicySource and the
   SUPAPolicyTarget classes are defined as abstract. This provides
   more freedom for the data modeler in implementing the data model.
   For example, if the data model uses an object-oriented language,
   such as Java, then the above structure enables all of the abstract
   classes to be collapsed into a single concrete class. If this is
   done, attributes as well as relationships are inherited.

4.3.1. SUPAPolicyObject

   A SUPAPolicyObject serves as a single root of the SUPA system
   (i.e., all other classes in the model are subclasses of the
   SUPAPolicyObject class) except for the Metadata objects, which are
   in a separate class hierarchy. This simplifies code generation and
   reusability. It also enables SUPAPolicyMetadata objects to be
   attached to any appropriate subclass of SUPAPolicyObject.

4.3.2. SUPAPolicyStructure

   SUPAPolicyStructure is an abstract superclass that is the base
   class for defining different types of policies (however, in this
   version of this document, only ECA policy rules are modeled). It
   serves as a convenient aggregation point to define atomic (i.e.,
   individual policies that can be used independently) and composite
   (i.e., hierarchies of policies) SUPAPolicies; it also enables
   PolicySources and/or PolicyTargets to be associated with a given
   set of Policies.

   SUPAPolicies are defined as either a stand-alone PolicyContainer
   or a hierarchy of PolicyContainers. A PolicyContainer specifies
   the structure, content, and optionally, source, target, and
   metadata information for a SUPAPolicy. This is implemented by the
   subclasses of SUPAPolicyStructure. For example, the composite
   pattern is used to create two subclasses of the SUPAECAPolicyRule
   class; SUPAECAPolicyRuleAtomic is used for stand-alone policies,
   and SUPAECAPolicyRuleComposite is used to build hierarchies of
   policies.

   This document defines a SUPAPolicy as an ECA Policy Rule, though
   the GPIM enables other types of policies to be defined and used
   with an ECA policy rule. The GPIM model is used in [2] and [5],
   along with extensions that allow [2] and [5] to define multiple
   types of policies that are derived from the GPIM. They also allow
   different combinations of different types of policy rules to be
   used with each other. Most previous work cannot define different
   types of policy rules; please see Appendix A for a comparison to
   previous work.

4.3.3. SUPAPolicyComponentStructure

   SUPAPolicyComponentStructure is an abstract superclass that is the
   base class for defining components of different types of policies.
   SUPAPolicyStructure subclasses define the structure of a policy,
   while SUPAPolicyComponentStructure subclasses define the content
   that is contained in the structure of a policy. For example, a
   SUPAECAPolicyRule is an imperative policy rule, and defines its
   structure; its event, condition, and action clauses are populated
   by SUPAPolicyComponentStructure subclasses. The strength of this
   design is that different types of policies (e.g., imperative and
   declarative policies) can be represented using a common set of
   policy components.

   Please see Appendix for a comparison to previous work.

4.3.4. SUPAPolicyClause

   All policies derived from the GPIM are made up of one or more
   SUPAPolicyClauses, which define the content of the Policy.
   This enables a Policy of one type (e.g., ECA) to invoke Policies
   of the same or different types. SUPAPolicyClause is an abstract
   class, and serves as a convenient aggregation point for assembling
   other objects that make up a SUPAPolicyClause.

   The GPIM defines a single concrete subclass of SUPAPolicyClause,
   called SUPAEncodedClause. This is a generic clause, and can be
   used by any type of Policy in a stand-alone fashion. It can also
   be used in conjunction with other SUPAPolicyClauses. The EPRIM
   also defines a subclass of SUPAPolicyClause; see section 6.7).

   The structure of the GPIM is meant to provide an extensible
   framework for defining different types of policies. This is
   demonstrated by the EPRIM (see section 6) and the LSIM (see the
   Appendices) that each define new subclasses of SUPAPolicyClause
   (i.e., SUPABooleanClause and SUPALogicClause, respectively)
   without defining new classes that have no GPIM superclass.

   A SUPAPolicyClause is defined as an object. Therefore, clauses and
   sets of clauses are objects, which promotes reusability.

4.3.5. SUPAPolicyComponentDecorator

   One of the problems in building a policy model is the tendency to
   have a multitude of classes, and hence object instances, to
   represent different combinations of policy events, conditions, and
   actions. This can lead to class and/or relationship explosion.
   Please see Appendix A for a comparison to previous work.

   SUPAPolicyClauses are constructed using the Decorator Pattern
   [11]. This is a design pattern that enables behavior to be
   selectively added to an individual object, either statically or
   dynamically, without affecting the behavior of other objects from
   the same class. The decorator pattern uses composition, instead of
   inheritance, to avoid class and relationship explosion. The
   decorator pattern also enable new objects to be composed from
   parts or all of existing objects without affecting the existing
   objects.

   This enables the resulting SUPAPolicyClause to be constructed
   completely from objects in the SUPA information model. This
   facilitates the construction of policies at runtime by a machine.
   This is also true of [2] and [5]; however, this is NOT true of
   most other models. Please see Appendix A for a comparison to
   previous work.

SUPAPolicyComponentDecorator defines four types of objects that
can be used to form a SUPAPolicyClause. Each object may be used
with all other objects, if desired. The first three are defined
in the GPIM, with the last defined in the EPRIM. The objects are:

> o SUPAPolicyTerm, which enables a clause to be defined in a
>   canonical {variable, operator, value} form
> o SUPAGenericDecoratedComponent, which enabled a custom object
>   to be defined and then used in a SUPAPolicyClause
> o SUPAPolicyCollection, which enables a collection of objects
>   to be gathered together and associated with all or a portion
>   of a SUPAPolicyClause
> o SUPAECAComponent, which defines Events, Conditions, and
>   Actions as reusable objects

This approach facilitates the machine-driven construction of
policies. Note that this is completely optional; policies do not
have to use these constructs.

4.3.6. SUPAPolicyTarget

A SUPAPolicyTarget is a set of managed entities that a SUPAPolicy
is applied to. A managed entity can only be designated a
SUPAPolicyTarget if it can process actions from a SUPAPolicy.

A managed object may not be in a state that enables management
operations to be performed on it. Furthermore, the policy-based
management system SHOULD ensure that the management entity
performing the management operations has the proper permissions to
perform the management operations. The design of the
SUPAPolicyTarget addresses both of these criteria.

4.3.7. SUPAPolicySource

A SUPAPolicySource is a set of managed entities that authored, or
are otherwise responsible for, this SUPAPolicy. Note that a
SUPAPolicySource does NOT evaluate or execute SUPAPolicies. Its
primary use is for auditability and the implementation of deontic
and/or alethic logic.

4.4. The Design of the GPIM

This section describes the overall design of the GPIM.

The GPIM defines a policy as a type of PolicyContainer. For this
version, only ECA Policy Rules will be described. However, it
should be noted that the mechanism described is applicable to
other types of policies (e.g., declarative) as well.

4.4.1. Structure of Policies

   Recall that a PolicyContainer was defined as a special type of
   container that provides at least the following three functions:

   1. It uses metadata to define how its content is described
       and/or prescribed
   2. It separates the content of the policy from the
       representation of the policy
   3. It provides a convenient control point for OAMP operations.

   The first requirement is provided by the ability for any subclass
   of Policy (the root of the information model) to aggregate one or
   more concrete instances of a SUPAPolicyMetadata class. This is
   explained in detail in section 5.2.2.

   The second requirement is met by representing an ECA Policy as
   having two parts: (1) a rule part and (2) components that make up
   the rule. Since functional and declarative policies are not,
   strictly speaking, "rules", the former is named PolicyStructure,
   while the latter is named PolicyComponentStructure.

   The third requirement is met by the concrete subclasses of
   PolicyStructure. Since they are PolicyContainers, they are made
   up of the SUPAECAPolicyRule, its commponents, and any metadata
   that applies to the PolicyContainer, the SUPAECAPolicyRule, and.or
   any components of the SUPAECAPolicyRule. This provides optional
   low-level control over any part of the SUPAECAPolicyRule.

   The above requirements result in the design shown in Figure 4.

```
 A                         SUPAHasPolicyMetadata   A
+-------------------+/ \                     \+-------------------+
| SUPAPolicyObject + A ---------------------+ SUPAPolicyMetadata |
+--------+--------+\ /                       /+-------------------+
        / \         0..n                 0..n
         I
         I
         +------+-----------------------------------+
               I                                    I
         A     I              A                     I
      +--------+----------+    +-----------------+-----------+
      | SUPAPolicyStructure |  | SUPAPolicyComponentStructure |
      +--------+----------+    +------------+---------------+
            / \                        / \
             I                          I
             I                          I
         (subclasses representing    (subclasses representing
      different types of policies)  different policy components)
```
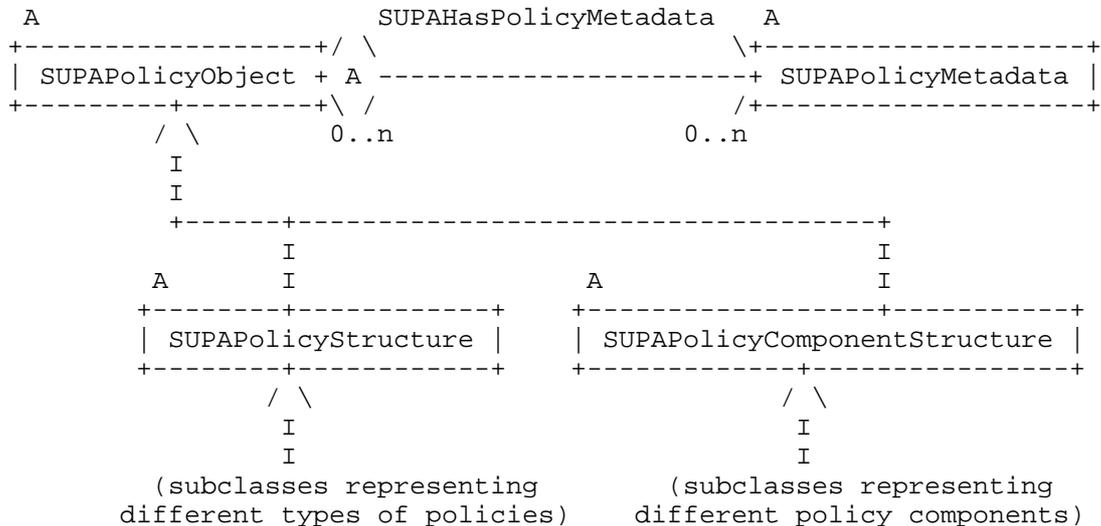
                  Figure 4. Structure of a Policy

Note that aggregation in Figure 4 (named SUPAHasPolicyMetadata)
is realized as an association class, in order to manage which set
of Metadata can be aggregated by which SUPAPolicyObject. The
combination of these three functions enables a PolicyContainer
to define the behavior of how its constituent components will be
accessed, queried, stored, retrieved, and how they operate.

It is often necessary to construct groups of policies. The GPIM
follows [2] and [5], and uses the composite pattern [11] to
implement this functionality, as shown in Figure 5 below. There
are a number of advantages to using the composite pattern over a
simple relationship, as detailed in [11].

Figure 5 shows that SUPAPolicyStructure has a single subclass,
called SUPAECAPolicyRule. Note, however, that other types of
policies, such as declarative policies, can be defined as
subclasses of SUPAPolicyStructure in the future.

```
                          A
                    +---------------------+
                    | SUPAPolicyStructure |
                    +--------+------------+
                            / \
                             I
                             I
            +---------------+----------------+
            I                                I
        C   I                           A    I
  +---------------+---------------+     +-----------+-----------+
  | Future Subclasses to Represent |    |   SUPAECAPolicyRule   |
  |  Represent Different Policies  |    +-----------------------+
  +-------------------------------+
```
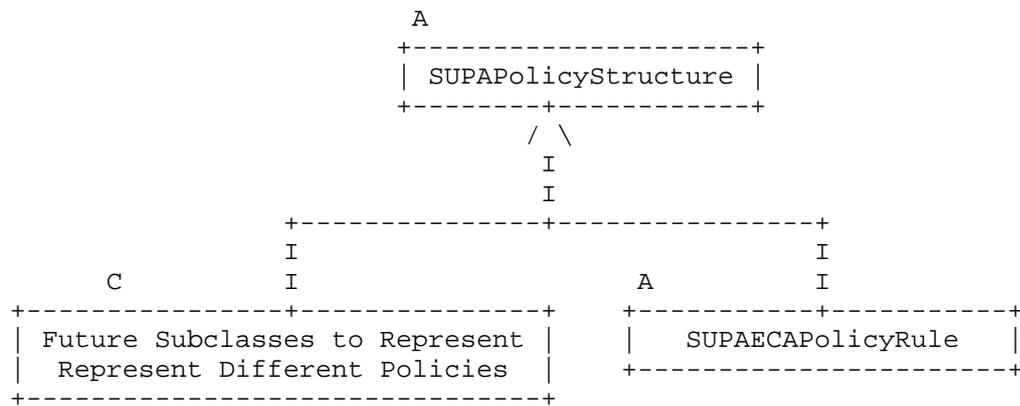
   Figure 5. The Composite Pattern Applied to SUPAPolicyStructure

4.4.2. Representing an ECA Policy Rule

   An ECA policy rule is a 3-tuple, which is made up of an event
   clause, a condition clause, and an action clause. Each of these
   three types of clauses may in turn be made up of a Boolean
   combination of clauses of that type. Each clause may be viewed as
   a predicate, as it provides a TRUE or FALSE output. The canonical
   form of a clause is a 3-tuple of the form "variable operator value",
   and can be made into more complex Boolean expressions. For example,
   the SUPAPolicyClause: "((A AND B) OR NOT (C AND D)) consists of two
   clauses, "(A AND B)" and "(C OR D)", that are combined together
   using the operators OR and NOT.

   A SUPAECAPolicyRule is defined (in the EPRIM) as an abstract
   subclass of SUPAPolicyStructure.

```
    A                                    A
   +-------------------------+          +-----------------+
   |    SUPAPolicyStructure   |          | SUPAPolicyClause |
   +--------+--------+------+          +-------+----+----+
        / \        / \ 0..1          1..n / \   / \
         I          A                  |      I
         I          \ /                |      I
         I           |                 |      I
         I           |  SUPAHasPolicyClause  |      I
         I           +----------------------+   I
      A  I                              A       I
   +------+----------+          +---------+-------+
   | SUPAECAPolicyRule |          | SUPAPolicyClause |
   +-----------------+          +-----------------+
```
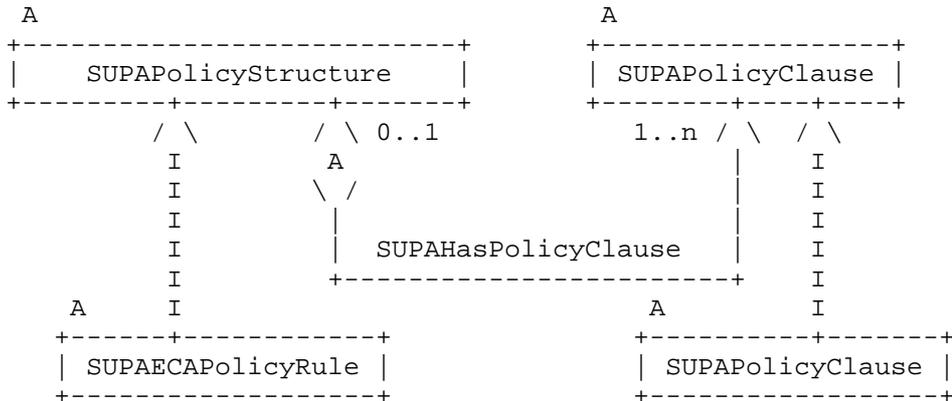
        Figure 6. SUPAECAPolicyRule Aggregating SUPAPolicyClauses

   Note that the aggregation SUPAHasPolicyClause in Figure 6 is
   realized as an association class, in order to manage which set
   of SUPAPolicyClauses can be aggregated by which set of
   SUPAECAPolicyRules. This aggregation is defined at the
   SUPAPolicyStructure level, and not at the lower level of
   SUPAECAPolicyRule, so that non-ECA policies can also use this
   aggregation.

   Since a SUPAECAPolicyRule consists of three SUPAPolicyClauses,
   at least three separate instances of the SUPAHasPolicyClause
   aggregation are instantiated in order to make a complete
   SUPAECAPolicyRule, as shown in Figure 7.

```
     A                                    A
    +------------------+          +-------------------+
    | SUPAECAPolicyRule |          |  SUPAPolicyClause  |
    +--+----+----+------+          +-------+----+----+--+
     / \  / \  / \  1..n          0..n  / \  / \  / \
       A    A    A                       |    |    |
      \ /  \ /  \ /                       |    |    |
       |    |    |                        |    |    |
       |    |    |  SUPAHasPolicyClause #1  |    |    |
       |    |    +----------------------------+   |    |
       |    |                             |    |
       |    |    SUPAHasPolicyClause #2    |    |
       |    +---------------------------------+   |
       |                                  |
       |    SUPAHasPolicyClause #3         |
       +-------------------------------------------+
```
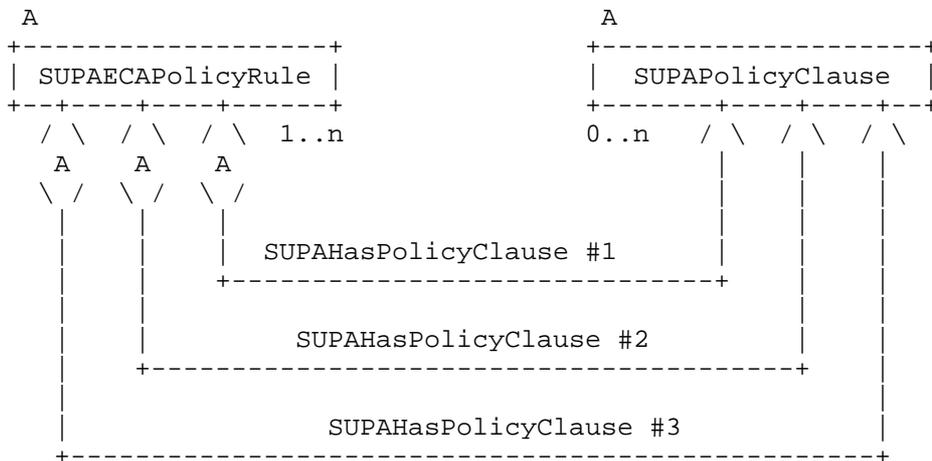
        Figure 7. Instantiating a SUPAECAPolicyRule, part 1

In figure 7, SUPAECAPolicyRule is shown as "owning" these three
aggregations, since it inherits them from its superclass
(SUPAPolicyStructure). The three aggregations represent the
event, condition, and action clauses of a SUPAECAPolicyRule.
Note that each of these clauses MAY consist of one or more
SUPAPolicyClauses. Similarly, each SUPAPolicyClause MAY consist
of one or more predicates. In this way, complex event, condition,
and action clauses, which are combinations of Boolean expressions
that form a logical predicate) are supported, without having to
define additonal objects (as is done in previous work; please
see Appendix A for a comparison to previous work.

The multiplicity of the SUPAHasPolicyClause aggregation is
0..n on the aggregate side and 1..n on the part side. This means
that a particular SUPAECAPolicyRule MUST aggregate at least one
SUPAPolicyClause, and that a given SUPAPolicyClause MAY be
aggregated by zero or more SUPAECAPolicyRule objects.

This cardinality MAY be refined to 3..n for SUPAECAPolicyRules,
since a SUPAECAPolicyRule MUST have at least three separate clauses.
However, since a SUPAPolicyStructure is the owner of this
aggregation (which is inherited by SUPAECAPolicyRule), the
cardinality is defined to be 1..n on the part side because other
types of Policies have different needs. The 0..n cardinality
means that a SUPAPolicyClause may be aggregated by zero or more
SUPAECAPolicyRules. The zero is provided so that SUPAPolicyClauses
can be stored in (for example) a repository before the
SUPAECAPolicyRule is created; the "or more" recognizes the fact
that multiple SUPAECAPolicyRules could aggregate the same
SUPAPolicyClause.

In Figure 7, suppose that SUPAHasPolicyClause#1, #2, and #3
represent the aggregations for the event, condition, and action
clauses, respectively. This means that each of these
SUPAHasPolicyClause aggregations must explicitly identify the
type of clause that it represents.

In looking at Figure 7, there is no difference between any of the
three aggregations, except for the type of clause that the
aggregation represents (i.e., event, condition, or action clause).

Therefore, three different aggregations, each with their own
association class, is not needed. Instead, the GPIM defines a
single aggregation (SUPAHasPolicyClause) that is realized using a
(single) abstract association class (SUPAHasPolicyClauseDetail);
this association class is then subclassed into three concrete
subclasses, one each to represent the semantics for an event,
condition, and action clause.

The policy management system may use any number of different
software mechanisms, such as introspection or reflection, to
determine the nature of the aggregation (i.e., what object types
are being aggregated) in order to select the appropriate subclass
of SUPAHasPolicyClauseDetail. The three subclasses of
SUPAHasPolicyClauseDetail are named SUPAHasPolicyEventDetail,
SUPAHasPolicyConditionDetail, and SUPAHasPolicyActionDetail,
respectively. While Event, Condition, and Action objects are
typically used in ECA policy rules, the design in this document
enables them to be used as policy components of other types of
policies as well. This is shown in Figure 8.

```
    A                                    A
    +-------------------+                +------------------+
    | SUPAECAPolicyRule |                | SUPAPolicyClause |
    +---------+---------+                +----------+-------+
          / \ 1..n                         0..n / \
           A                                     |
           \ /                                   |
            |                                    |
            |         SUPAHasPolicyClause        |
          +-------------+-----------------+
                        ^
                        |
          A             |
          +--------------+-----------+
          | SUPAHasPolicyClauseDetail |
          +--------------+-----------+
                       / \
                        I
                        I
          +---------------+--------------------+
          I               I                    I
    C     I         C     I             C      I
    +--------+-----+ +-------+----------+ +---------+-----+
    |Event subclass| |Condition subclass| |Action subclass|
    +-------------+ +-----------------+ +--------------+
```

          Figure 8. Instantiating a SUPAECAPolicyRule, part 2

4.4.3. Creating SUPA Policy Clauses

   There are two different types of Policy Components. They are a
   SUPAPolicyClause and a SUPAPolicyComponentDecorator. The former
   is used to construct SUPAECAPolicyRules, while the latter is used
   to add behavior to a SUPAPolicyClause. This enables the structure
   and capabilities of the SUPAPolicyClause to be adjusted
   dynamically at runtime.

However, since each SUPAECAPolicyRule can be made up of a variable number of SUPAPolicyComponents, the decorator pattern is used to "wrap" any concrete subclass of SUPAPolicyClause with zero or more concrete subclasses of the PolicyComponentDecorator object. This avoids problems of earlier models that resulted in a proliferation of classes and relationships.

Figure 9 shows these two class subclasses. Note that the decorator pattern [11] is used to enable subclasses of the SUPAPolicyComponentDecorator class to add their attributes and/or behavior to a SUPAPolicyClause (as stated in section 4.3) without affecting the behavior of other objects from the same class. More specifically, concrete subclasses of the (abstract) SUPAPolicyComponentDecorator class can be used to decorate, or "wrap", any of the concrete subclasses of the (abstract) SUPAPolicyClause class.

```
            A
      +-----------------------------+ 1..n
      | SUPAPolicyComponentStructure +---------------------+
      +---------------+-------------+                       |
                      / \                                   |
                      I   SUPAHasPolicyComponentDecorators  |
                      I                                     |
      +--------------+----------+                           |
      I                         I                           |
 A    I           A             I                           |
+---------+-------+  +-------------+---------------+ 0..1    |
|                 |  |                             |/ \      |
| SUPAPolicyClause |  | SUPAPolicyComponentDecorator + A -----+
|                 |  |                             |\ /
+-----------------+  +-----------------------------+
```
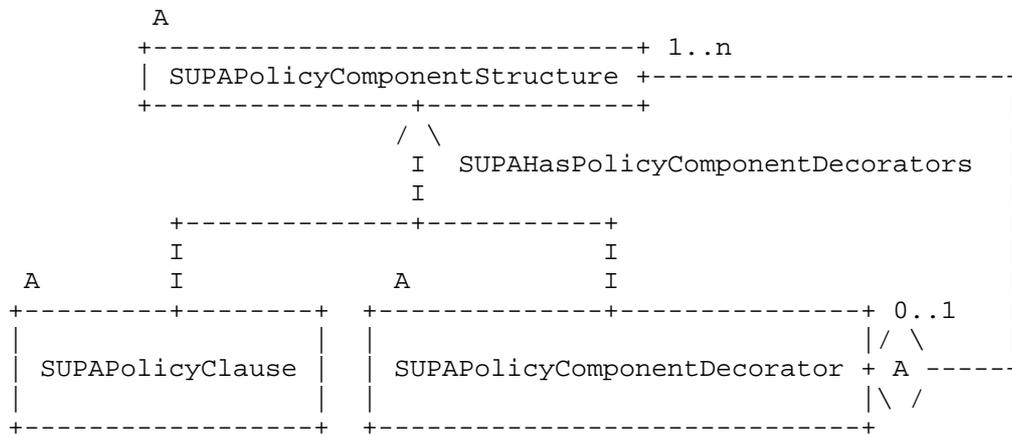
          Figure 9. Subclasses of SUPAPolicyComponentStructure

Instead of using inheritance to statically create new classes to represent new types of objects, the decorator pattern uses composition to dynamically combine attributes and behavior from existing objects into new objects. This is done by defining an interface in SUPAPolicyComponent that all of the subclasses of SUPAPolicyComponent conform to. Since the subclasses are of the same type as SUPAPolicyComponent, they all have the same interface. This allows each concrete SUPAPolicyComponentDecorator subclass to add its attributes and/or behavior to the concrete subclass of SUPAPolicyClause that it is decorating (or "wrapping").

This represents an important design optimization for data models. Note that a single SUPAECAPolicyRule can consist of any number of SUPAPolicyClauses, each of very different types. If inheritance was used, then a subclass AND an aggregation would be required for each separate clause that makes up the policy rule.

   Clearly, continuing to create subclasses is not practical. Worse,
   suppose composite objects are desired (e.g., a new object Foo is
   made up of existing objects Bar and Baz). If all that was needed
   was one attribute of Bar and two of Baz, the developer would still
   have to use the entire Bar and Baz classes. This is wasteful and
   inefficient. In contrast, the decorator pattern enables all, or
   just some, of the attributes and/or behavior of a class to "wrap"
   another class. This is used heavily in many production systems
   (e.g., the java.io package) because the result is only the
   behavior that is required, and no other objects are affected.

   The SUPAPolicyComponentDecorator class hierarchy is used to define
   objects that may be used to construct a SUPAPolicyClause. The
   decorator object can add behavior before, and/or after, it
   delegates to the object that it is decorating. The subclasses of
   SUPAPolicyComponentDecorator provide a very flexible and completely
   dynamic mechanism to:

   1) add or remove behavior to/from an object
   2) ensure that objects are constructed using the minimum amount
      of features and functionality required

   SUPAPolicyComponentDecorator defines four subclasses, as shown in
   Figure 10.

```
                            A
               +------------------------------+
               | SUPAPolicyComponentDecorator |
               +--------------+---------------+
                             / \
                              I
                              I
                              I
          +-----------+------------+--------------+
          I           I            I              I
    A     I           I    C       I              I
+--------+-------+     I   +--------+-----------+  I
| SUPAPolicyTerm |     I   | SUPAPolicyCollection | I
+----------------+     I   +----------------------+ I
   (for defining       I   (for defining sets and/or I
    clauses in         I       groups of objects)    I
  canonical form)      I                             I
                       I                             I
        C              I              A              I
      +----------------+-------------+    +--------+--------+
      | SUPAGenericDecoratedComponent |  | SUPAECAComponent |
      +------------------------------+    +-----------------+
          (for decorating concrete        (for defining reusable
       subclasses of SUPAPolicyClause)    event, condition,
                                          and action objects)
```

        Figure 10. Subclasses of SUPAPolicyComponentDecorator

If a SUPAEncodedClause is being used, then there is no need to use any of the SUPAPolicyComponentDecorator subclasses, since the SUPAEncodedClause already completely defines the content of the SUPAPolicyClause.

However, if a SUPAEncodedClause is NOT being used, then a SUPAPolicyClause will be constructed using one or more types of objects that are each subclasses of SUPAPolicyComponentDecorator.

These four subclasses provide four different ways to construct a SUPAPolicyClause:

   1) SUPAPolicyTerm: as a {variable, operator, value} clause
   2) SUPAEncodedClause: as an encoded object (e.g., to pass YANG
      or CLI code)
   3) SUPAPolicyCollection: as a collection of objects that
      requires further processing in order to be made into a
      SUPAPolicyClause
   4) SUPAECAComponent: subclasses define reusable Event,
      Condition, or Action objects

These four different types of objects can be intermixed. For example, the first and last types can be combined as follows:

```
   Variable == Event.baz                                   (A)
   Condition BETWEEN VALUE1 and VALUE2                     (B)
   (Event.severity == 'Critical' AND
     (SLA.violation == TRUE OR User.class == 'Gold'))      (C)
```

In the above rules, (A) uses Event.baz to refer to an attribute of the Event class; (B) defines two different instances of a Value class, denoted as Value1 and Value2; (C) uses the nomenclature foo.bar, where foo is the name of a class, and bar is the name of an attribute of that class.

4.4.4. Creating SUPAPolicyClauses

   The GPIM defines a single subclass of SUPAPolicyClause, called SUPAEncodedClause. This clause is generic in nature, and MAY be used with any type of policy (ECA or otherwise). The EPRIM defines an ECA-specific subclass of the GPIM, called a SUPABooleanClause, which is intended to be used with just ECA policy rules; however, other uses are also possible.

   Together, the GPIM and EPRIM provide several alternatives to implement a SUPAPolicyClause, enabling the developer to optimize the solution for different constraints:

   1) The SUPAPolicyClause can be encoded using one or more
      SUPAEncodedClauses; a SUPAEncodedClause encodes the
      entire content of its respective event, condition, or
      action clause.
   2) The SUPAPolicyClause can be defined using one or more
      SUPABooleanClauses; each of the three clauses can be
      defined as either a single SUPABooleanClause, or a
      combination of SUPABooleanClauses that are logically
      ANDed, ORed, and/or NOTed.
   3) The above two mechanisms can be combined (e.g., the first
      used to define the event clause, and the second used to
      define the condition and action clauses).

   Figure 11 shows the subclasses of SUPAPolicyClause.

```
                              A
                    +------------------+
                    | SUPAPolicyClause |
                    +--------+---------+
                            / \
                             I
                             I
                             I
          +---------------+------------+
          I                            I
      A   I                    C       I
    +--------+----------+        +----------+--------+
    | SUPABooleanClause |        | SUPAEncodedClause |
    +-------------------+        +-------------------+
```
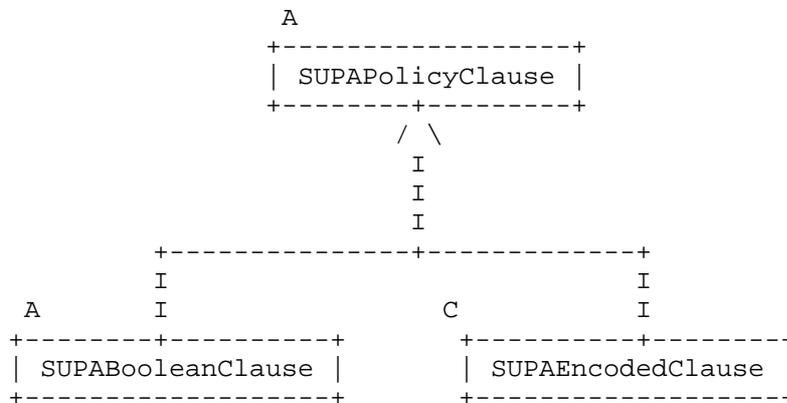
            Figure 11. Subclasses of SUPAPolicyClause

   SUPABooleanClause is defined in the EPRIM, and is used to
   construct Boolean clauses that collectively make up a
   SUPAPolicyClause. It is abstract, so that the composite pattern
   can be applied to it, which enables hierarchies of Boolean
   clauses to be created. SUPAEncodedClause (see section 6.7) is
   used to encode the content of a SUPAPolicyClause as an attribute
   (instead of reusable objects).

4.4.5. SUPAPolicySources

   A SUPAPolicySource is a set of managed entities that authored,
   or are otherwise responsible for, this SUPAPolicy. Note that a
   SUPAPolicySource does NOT evaluate or execute SUPAPolicies. Its
   primary use is for auditability, authorization policies, and
   other applications of deontic and/or alethic logic.

SUPAPolicyStructure defines four relationships. Two of these
(SUPAHasPolicySource and SUPAHasPolicyTarget), which are both
aggregations, relate a SUPAPolicyStructure to a SUPAPolicySource
and a SUPAPolicyTarget, respectively. Since SUPAECAPolicyRule is
a subclass of SUPAPolicyStructure, it (and its subclasses) inherit
both of these aggregations. This enables SUPAPolicySources and/or
SUPAPolicyTargets to be attached to SUPAECAPolicyRules (but NOT to
components of a SUPAPolicy).

Figure 12 shows how SUPAPolicySources and SUPAPolicyTargets are
attached to a SUPAPolicy. Note that both of these aggregations
are defined as optional, since their multiplicity is 0..n - 0..n.
In addition, both of these aggregations are realized as
association classes, in order to be able to control which
SUPAPolicySources and SUPAPolicyTargets are attached to a given
SUPAECAPolicyRule.

```
                    A
                    +------------------+
                    | SUPAPolicyObject |
                    +--------+---------+
                            / \
                             I
                             I
                             I
              +-------------+-----+--------------------+
              I                   I                    I
         A    I              C    I               C    I
     +----------+--------+   +-------+--------+   +--------+-------+
     | SUPAPolicyStructure | |SUPAPolicySource|  |SUPAPolicyTarget|
     +------+-------+------+   +----------+-----+   +----------+-----+
      0..n / \      / \ 0..n       0..n / \            0..n / \
          A        A                     |                   |
          \ /      \ /                    |                   |
           |        |                     |                   |
           |        |                     |                   |
           |    +--------------------+    |                   |
           |        SUPAHasPolicySource   |                   |
           |                              |                   |
           +----------------------------------------------+
                      SUPAHasPolicyTarget
```

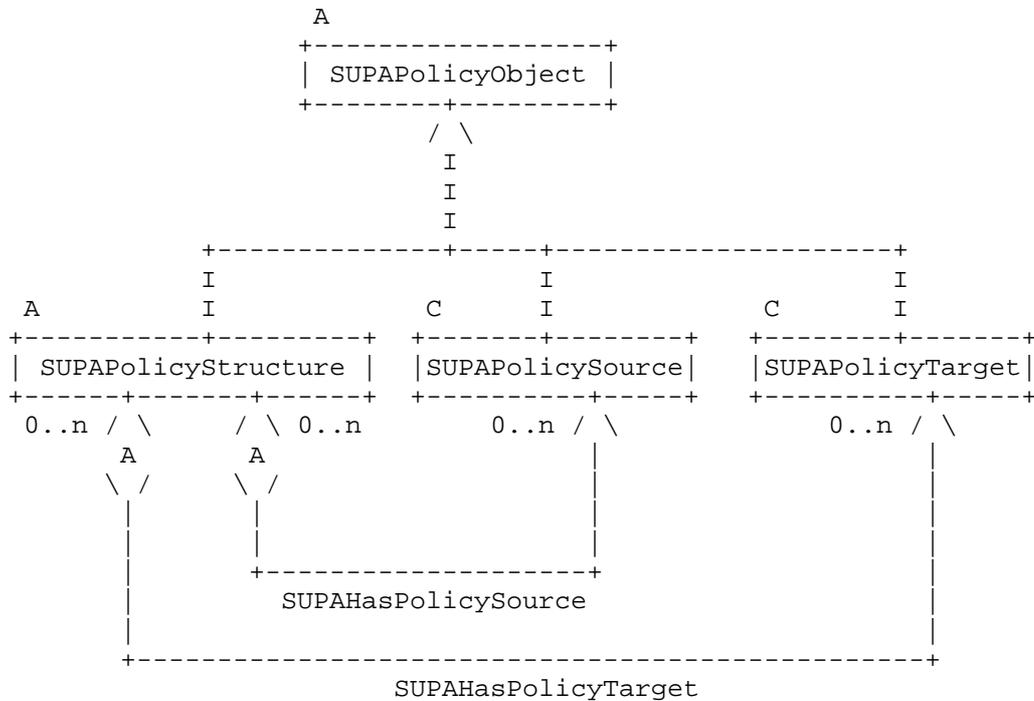   Figure 12. ECAPolicyRules, SUPAPolicySources, and PolicyTargets

A SUPAPolicySource MAY be mapped to a role (e.g., using the
role-object pattern [11]); this indirection makes the system less
fragile, as entities can be transparently added or removed from
the role definition without adversely affecting the definition of
the SUPAPolicy. Note that SUPAPolicyRole is a subclass of
SUPAPolicyMetadata.

4.4.6. SUPAPolicyTargets

   A SUPAPolicyTarget defines the set of managed entities that a
   SUPAPolicy is applied to. This is useful for debugging, as well as
   when the nature of the application requires the set of managed
   entities affected by a Policy to be explicitly identified. This is
   determined by two conditions:

     1) The set of managed entities that are to be affected by the
        SUPAPolicy must all agree to play the role of a
        SUPAPolicyTarget. For example, a managed entity may not be
        in a state that enables SUPAPolicies to be applied to it;
        hence, in this case, it MUST NOT assume the role of a
        SUPAPolicyTarget
     2) A SUPAPolicyTarget must be able to:
        a) process (either directly or with the aid of a proxy)
           SUPAPolicies, or
        b) receive the results of a processed SUPAPolicy and
           apply those results to itself.

   Figure 12 showed how SUPAPolicyTargets are attached to
   SUPAECAPolicyRules.

   A SUPAPolicyTarget MAY be mapped to a role (e.g., using the
   role-object pattern [11]); this indirection makes the system less
   fragile, as entities can be transparently added or removed from
   the role definition without adversely affecting the definition of
   the SUPAPolicy. Note that SUPAPolicyRole is a subclass of
   SUPAPolicyMetadata.

4.4.7. Policy Metadata

   Metadata is, literally, data about data. As such, it can be
   descriptive or prescriptive in nature.

4.4.7.1. Motivation

   There is a tendency in class design to make certain attributes,
   such as description, status, validFor, and so forth, bound to a
   specific class (e.g., [6]). This is bad practice in an information
   model. For example, different classes in different parts of the
   class hierarchy could require the use of any of these attributes;
   if one class is not a subclass of the other, then they must each
   define the same attribute as part of their class structure. This
   makes it difficult to find all instances of the attribute and
   ensure that they are synchronized. Furthermore, context can
   dynamically change the status of an object, so an easy way to
   update the status of one object instance without affecting other
   instances of the same object is required.

Many models, such as [4] and [6], take a simplistic approach of
defining a common attribute high in the hierarchy, and making it
optional. This violates classification theory, and defeats the
purpose of an information model, which is to specify the
differences in characteristics and behavior between classes (as
well as define how different classes are related to each other).
Note that this also violates a number of well-known software
architecture principles, including:

> o the Liskov Substitution Principle [13]
>   (if A is a subclass of B, then objects instantiated from
>   class B may be replaced with objects instantiated from
>   class A WITHOUT ALTERING ANY OF THE PROGRAM SEMANTICS)
> o the Single Responsibility Principle [14]
>   (every class should have responsibility over one, and only
>   one, part of the functionality provided by the program)
> o the Open/Closed Principle (software should be open for
>   extension, but closed for modification) [17]
> o the Interface-Segreation Principle (clients should not be
>   forced to depend on methods that they do not use) [14]
> o the Dependency Inversion Principle (high-level modules should
>   not depend on low-level modules; both should depend on
>   abstractions) [14]

Most models use inheritance, not composition. The former is
simpler, but has some well-known problems. One is called "weak
encapsulaton", meaning that a subclass can use attributes and
methods of a superclass, but if the superclass changes, the
subclass may break. Another is that each time a new object is
required, a new subclass must be created. These problems are
present in [RFC3460], [4], and [6].

Composition is an alternative that provides code that is easier to
use. This means that composition can provide data models that are
more resistant to change and easier to use. By using composition,
we can select just the metadata objects that are needed, instead
of having to rely on statically defined objects. We can even
create new objects from a set of existing objects through
composition. Finally, we can use the decorator pattern to select
just the attributes and behaviors that are required for a given
instance.

In [2] and [5], a separate metadata class hierarchy is defined to
address this problem. This document follows this approach.

4.4.7.2. Design Approach

The goal of the GPIM is to enable metadata to be attached to any
subclass of SUPAPolicyObject that requires it. Since this is a
system intended for policy-based management, it therefore makes
sense to be able to control which metadata is attached to which
policies dynamically (i.e., at runtime).

One solution is to use the Policy Pattern [1], [2], [6], [12].
This pattern was built to work with management systems whose
actions were dependent upon context. The Policy Pattern works as
follows:

    o Context is derived from all applicable system inputs (e.g.,
      OAMP data from network elements, business goals, time of
      day, geo-location, etc.).
    o Context is then used to select a working set of Policies.
    o Policies are then used to define behavior at various
      control points in the system.
    o One simple type of control point is an association class.
      Since the association class represents the semantics of how
      two classes are related to each other, then
        o ECAPolicyRule actions can be used to change the attribute
          values, methods, and relationships of the association
          class
        o This has the affect of changing how the two classes are
          related to each other
    o Finally, as context changes, the working set of policies
      change, enabling the behavior to be adjusted to follow
      changes in context (according to appropriate business goals
      and other factors, of course) in a closed loop manner.

Conceptually, this is accomplished as shown in Figure 13 below.

```
                        Defines
    +----------+        Behavior                  +------------+
    | Policies +---------------+                  | SUPAPolicy |
    +----+-----+ 1..n          |                  +------+-----+
   0..n / \                    |                   / \ 0..n
        |                      |                    A
        |                      |                   \ /
        |               1..n \ /                    |
        |          +----------+-------------+       |
        |          | SUPAPolicyMetadataDetail |     |
        |          +----------+-------------+       |
        |                     |                     |
        | Selects             |                     |
        | Policies            |                     |
        |                     |                     |
        |          +------------------->+
        |                    Applies           |
       / \                   Behavior          |
        A                                       |
   0..n \ /                                 \ / 0..n
    +----+-----+                      +--------+---------+
    | Context  |                      |SUPAPolicyMetadata|
    +----------+                      +------------------+
```
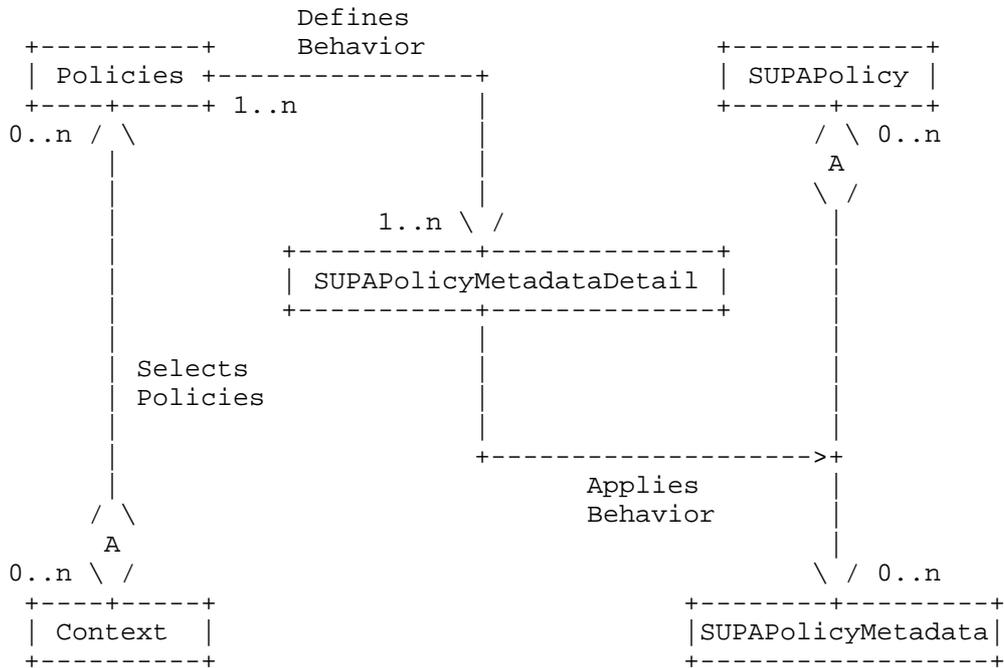
                Figure 13. Context-Aware Policy Rules

4.4.7.2.1. Policies and Actors

   The Policy Continuum ([1] [5] [10] [12]) was defined to associate
   different actors with different policies at different levels of
   business and/or technical specificity. Context-aware policy rules,
   and the Policy Pattern, were defined to realize this association.

   Four important functions related to the lifecycle of policies are
   design, implementation, deployment, and execution. There are many
   different possible definitions of these functions (even for policy
   lifecycle management); however, for the purposes of this document,
   they are defined as follows:

      o Design: The process of defining a software architecture
                      to satisfy user requirements.
      o Development: the process of documenting, programming,
                      testing, and maintaining code and applications
                      as part of a software product
      o Deployment: the process that assembles and transfers
                      completed software artifacts to a state that
                      enables their execution
      o Execution: the process of installing, activating, running,
                      and subsequently deactivating executable
                      software products

   The design process is responsible for producing a software
   architecture. This emphasizes the design, as opposed to the
   programming, of software systems. In contrast to design,
   development emphasizes constructing software artifacts via coding
   and documentation.

   Deployment may be described as the process of releasing software.
   It includes all of the operations required to assemble a completed
   software product. It typically also includes the process of
   preparing a software product for execution (e.g., assembling a set
   of software products into a larger product, determining if the
   consumer site has appropriate resources to install and execute the
   software product, and collecting information on the feasibility of
   using the software product). This contrasts with the execution
   process, which is the set of processes that follow deployment.

   In summary, exemplar states in the policy lifecycle process
   include:

      o Design: determining how the policy-based management
                      system will operate
      o Development: documenting, programming, testing, and
                      maintaining policies and policy components
      o Deployment: assembling the components of a policy-based
                      management system
      o Execution: installing, enabling, running, disabling,
                      and uninstalling policies and policy components

4.4.7.2.2. Deployment vs. Execution of Policies

   One of the primary reasons for separating the deployment and
   execution processes is to differentiate between environments that
   are not ready to execute policies (i.e., deployment) and
   environments that are ready to execute policies (i.e., execution).
   This is an important consideration, since policies that are
   related to the same set of tasks may be deployed in many different
   places (e.g., in a policy system vs. in a network device). In
   addition, each managed entity in the set of SUPAPolicyTargets may
   or may not be in a state that allows SUPAPolicies to be applied to
   it (see section 4.4.6.).

   Hence, this design includes dedicated class attributes for
   getting and setting the deployment and execution status, as well
   as enabling and disabling, SUPAPolicies (see section 5.3.1.).

4.4.7.2.3. Using SUPAMetadata for Policy Deployment and Execution

   One way of encoding deployment and execution status for policies
   and policy components is to attach Metadata objects to affected
   SUPAPolicyStructure and SUPAPolicyComponentStructure objects.
   This provides an extensible and efficient means to describe
   and/or prescribe deployment and/or execution status of a policy
   or a policy component. It is extensible, since classes and
   relationships can be used, as opposed to a set of attributes. It
   is efficient, because the decorator pattern (see section 5.7) is
   used (this enables attributes and/or methods of objects, or the
   entire object, to be used to add characteristics and/or behavior
   to a given object.

   SUPAPolicyMetadata objects (see sections 5.16 - 5.20) may be
   attached to the SUPAECAPolicyRule and/or any of its components
   to define additional semantics of the SUPAECAPolicyRule. For
   example, SUPAAccessMetadataDef (see section 5.19) and/or
   SUPAVersionMetadataDef (see section 5.20) may be attached to
   define the access privileges and version information,
   respectively, of a policy rule and/or its components.

   The SUPAPolicyStructure contains two attributes,
   supaPolDeployStatus and supaPolExecStatus (see sections 5.3.1.3.
   and 5.3.1.4., respectively) that SUPAPolicyMetadata objects can
   use to get and set the deployment and execution status of a
   SUPAPolicy. This allows metadata to be used to alter the
   deployment and/or execution state of a policy (or a set of
   policy components) without having to affect other parts of the
   policy-based management system. The supaPolDeployStatus attribute
   indicates that this SUPAPolicy can or cannot be deployed. If it
   cannot be deployed. Similarly, the supaPolExecStatus attribute
   is used to indicate if a particular SUPAPolicy has executed, is
   currently executing, or is ready to execute, and whether or not
   the execution of that SUPAPolicy had any failures.

The reverse is also true (and hence, forms a closed-loop system controlled by metadata). For example, if the set of deployed SUPAPolicies are SUPAECAPolicyRules, then when the actions of these SUPAECAPolicyRules are executed, the overall context has changed (see section 4.4.7.2). The context manager could then change attribute values (directly or indirectly) in the SUPAPolicyMetadataDetail association class. This class represents the behavior of the SUPAHasPolicyMetadata aggregation, which is used to define which SUPAPolicyMetadata can be attached to which SUPAPolicy objet in this particular context. For example, the access privileges of a policy and/or policy component could be changed dynamically, according to changes in context.

By using the decorator pattern on SUPAPolicyMetadata, any number of SUPAPolicyMetadata objects (or their attributes, etc.) can be wrapped around a concrete subclass of SUPAPolicyMetadata. This is shown in Figure 14 below.

4.4.7.3. Structure of SUPAPolicyMetadata

SUPAPolicyMetadata also uses the decorator pattern to provide an extensible framework for defining metadata to attach to SUPAPolicy subclasses. Its two principal subclasses are SUPAPolicyConcreteMetadata and SUPAPolicyMetadataDecorator. The former is used to define concrete subclasses of SUPAPolicyMetadata that are attached at runtime to SUPAPolicy subclasses, while the latter is used to define concrete objects that represent reusable attributes, methods, and relationships that can be added to subclasses of SUPAPolicyConcreteMetadata.

For example, concepts like identification, access control, and version information are too complex to represent as a single attribute, or even a couple of attributes - they require the generic power of objects to represent their characteristics and behavior. Furthermore, defining concrete classes to represent these concepts in the policy hierarchy is fragile, because:

1. not all objects that use these concepts need all of the information represented by them (e.g., two subclasses of an Identification Object may be Passport and Certificate, but these two objects are rarely used together, and even those contexts that use one of these classes may not need all of the data in that class)
2. defining a class means defining its attributes, methods, and relationships at a particular place in the hierarchy; this means that defining a relationship between a class A and another class B SHOULD only be done if all of the subclasses of B can use the attributes, methods, and relationships of A (e.g., in the above example, defining a relationship between an Identification Object and a superclass of a router class is not appropriate, since routers do not use Passports)

```
   A
  +-----------------+
  | SUPAPolicyObject |
  +--------+---------+
          / \ 0..n
           A
           \ /                                              A
           |                          0..n +--------------------+
           |     SUPAHasPolicyMetadata     \|                    |
           |     +------------+----------------+ SUPAPolicyMetadata |
           |                  ^             /|                    |
           |                  |               +------+------+------+
           C                  |              / \      | 1..n
           +------------+--------------+      I        |
      1..n |            |                |    I        |
  +----------+ SUPAHasPolicyMetadataDetail |  I        |
  |         |            |                |  I        |
  |         +----------------------------+   I        |
  |                                          I        |
  | (gets/sets values of attributes and/or   I        |
  |  methods of the SUPAHasPolicyMetadataDetail I     |
  |  class; this affects which SUPAPolicyMetadata I   |
  |  objects can be attached to which policies I      |
  |  and policy components)                   I        |
  |                                           I        |
  |          C                                I        |
  |          +----------------------------+   I        |
  |          |                            |   I        |
  +----------+ SUPAPolicyConcreteMetadata +IIIIIIIII+  |
  |          |                            |   I        |
  |          +----------------------------+   I        |
  |                                           I        |
  |          A                                I        |
  |          +----------------------------+   I        |
  |          |                            |   I        |
  |          | SUPAPolicyMetadataDecorator +IIIIIIIII+ |
  |          |                            |            |
  |          +-------+-------------+------+            |
  |                 / \             / \ 0..1           |
  |                  I               A                 |
  |                  I               \ /               |
  |                  I               |                 |
       subclasses for adding         |                 |
       behavior to policies    +----------------------+
       and policy components     SUPAHasMetadataDecorator
```
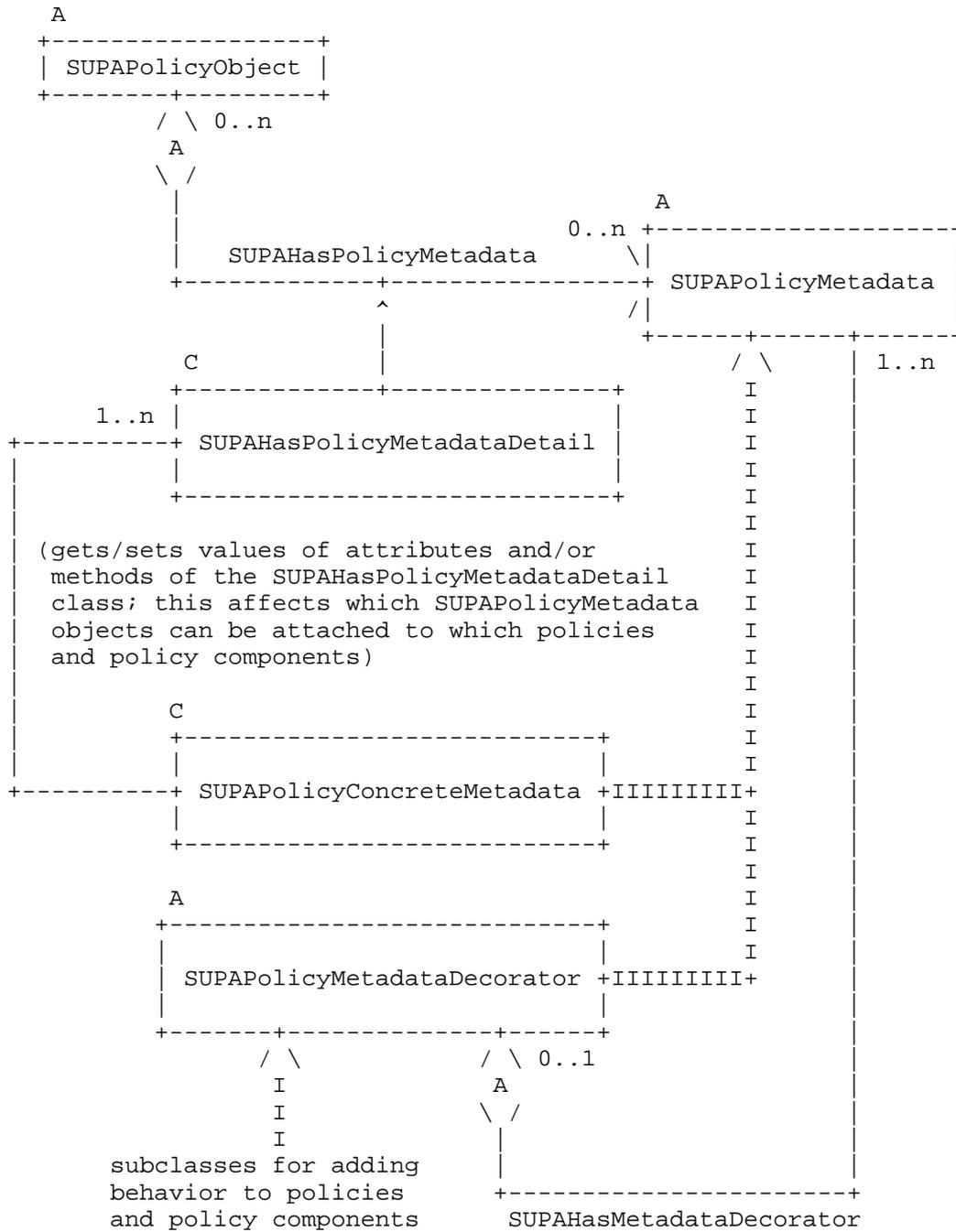
       Figure 14. SUPAPolicyMetadata Subclasses and Relationships

   Since a class encapsulates attributes, methods, and behavior,
   defining the Identification Object in the above example as a type
   of SUPAPolicyMetadata object enables the decorator pattern to be
   used to attach all or part of that object to other objects that
   need it.

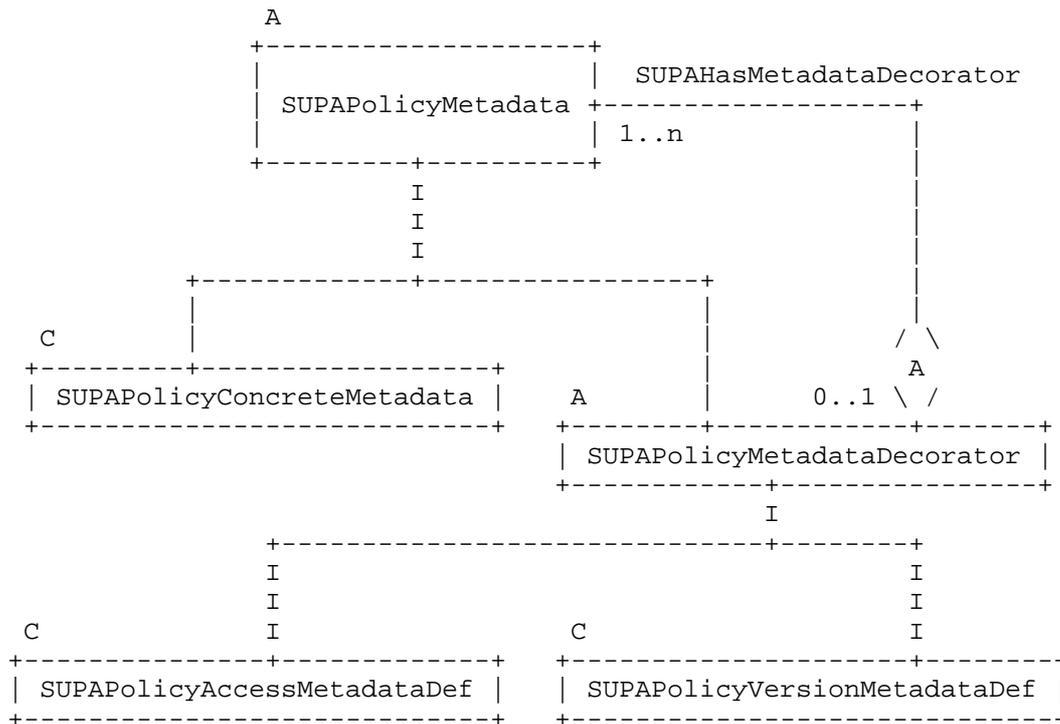   Figure 15 shows a portion of the SUPAPolicyMetadata hierarchy.


```
                   A
          +-------------------+
          |                   |   SUPAHasMetadataDecorator
          | SUPAPolicyMetadata +------------------+
          |                   | 1..n             |
          +---------+---------+                  |
                    I                            |
                    I                            |
                    I                            |
          +-------------+---------------+        |
          |             |               |        |
     C    |             |               |       / \
   +--------+-----------------+         |        A
   | SUPAPolicyConcreteMetadata |     A  |      0..1 \ /
   +--------------------------+   +--------+-----------+-------+
                                  | SUPAPolicyMetadataDecorator |
                                  +-----------+---------------+
                                              I
          +-----------------------------+--------+
          I                                      I
          I                                      I
     C    I                                 C    I
   +--------------+------------+    +-------------------+--------+
   | SUPAPolicyAccessMetadataDef |    | SUPAPolicyVersionMetadataDef |
   +--------------------------+    +---------------------------+
```

       Figure 15. SUPAPolicyMetadata Subclasses and Relationships

   Figure 15 shows a relevant portion of the SUPAPolicyMetadata
   hierarchy. SUPAPolicyConcreteMetadata is a concrete class that
   subclasses of the SUPAPolicyMetadataDecorator class can wrap.
   Two such subclasses, SUPAPolicyAccessMetadataDef and
   SUPAPolicyVersionMetadataDef, are shown in Figure 15. This
   enables access control and version information to be added
   statically (at design time) or dynamically (at runtime) to
   SUPAPolicyConcreteMetadata; this enables metadata-driven systems
   to adjust the behavior of the management system to changes in
   context, business rules, services given to end-users, and other
   similar factors. This is discussed more in sections 5.18 - 5.20.

4.5. Advanced Features

   This section will be completed in the next revision of this
   document.

4.5.1. Policy Grouping

   This section will be completed in the next revision of this
   document.

4.5.2. Policy Rule Nesting

   This section will be completed in the next revision of this
   document.

5. GPIM Model
    This section defines the classes, attributes, and relationships of
    the GPIM.

5.1. Overview
    The overall class hierarchy is shown in Figure 16; section numbers
    are appended after each class.

    (Class of another model that SUPA is integrating into)
          |
        +---SUPAPolicyObject (5.2)
        | |
        | +---SUPAPolicyStructure (5.3)
        | |
        | +---SUPAPolicyComponentStructure (5.4)
        | | |
        | | +---SUPAPolicyClause (5.5)
        | | | |
        | | | +---SUPAEncodedClause (5.6)
        | | |
        | | +---SUPAPolicyComponentDecorator (5.7)
        | | |
        | | +---SUPAPolicyTerm (5.8)
        | | | |
        | | | +---SUPAPolicyVariable (5.9)
        | | | |
        | | | +---SUPAPolicyOperator (5.10)
        | | | |
        | | | +---SUPAPolicyValue (5.11)
        | | |
        | | +---SUPAGenericDecoratedComponent (5.12)
        | | |
        | | +---SUPAPolicyCollection (5.13)
        | |
        | +---SUPAPolicySource (5.14)
        | |
        | +---SUPAPolicyTarget (5.15)
        |
        +---SUPAPolicyMetadata (5.16)
              |
              +---SUPAPolicyConcreteMetadata (5.17)
              |
              +---SUPAPolicyMetadataDecorator (5.18)
                    |
                    +---SUPAPolicyAccessMetadataDef (5.19)
                    |
                    +---SUPAPolicyVersionMetadataDef (5.20)

                 Figure 16: Main Classes of the GPIM

   SUPAPolicy is the root of the SUPA class hierarchy. For
   implementations, it is assumed that SUPAPolicy is subclassed from
   a class from another model.

   Classes, attributes, and relationships that are marked as
   "mandatory" MUST be part of a conformant implementation (i.e., a
   schema MUST contain these entities). This does not mean that these
   entities must be instantiated; rather it means that they must be
   able to be instantiated. Classes, attributes, and relationships
   that are marked as "optional" MAY be part of a conformant
   implementation.

   Unless otherwise stated, all classes (and attributes) defined in
   this section were abstracted from DEN-ng [2], and a version of
   them are in the process of being added to [5]. However, the work
   in [5] has been put on hold, and the names of many of the classes,
   attributes, and relationships are slightly different.

5.2. The Abstract Class "SUPAPolicyObject"

   This is a mandatory abstract class. Figure 17 shows the
   SUPAPolicyObject class, and its four subclasses.

```
 A                     0..n                 0..n  A
+----------------+/ \                      \+-----------------+
|SUPAPolicyObject+ A ----------------------+SUPAPolicyMetadata|
+--------+-------+\ / SUPAHasPolicyMetadata /+-----------------+
      / \
         I
         I
         +---------------+---------------+----------+
         I               I               I          I
  A      I               I               I          I
+--------+-----------+    I               I          I
| SUPAPolicyStructure |   I               I          I
+--------------------+    I               I          I
         A               I               I          I
         +---------------+------------+   I          I
         | SUPAPolicyComponentStructure |  I         I
         +----------------------------+   I          I
                            C          I          I
                   +--------+-------+  I          I
                   | SUPAPolicyTarget |  I         I
                   +----------------+  I          I
                            C          I
                   +---------+-------+
                   | SUPAPolicySource |
                   +-----------------+
```
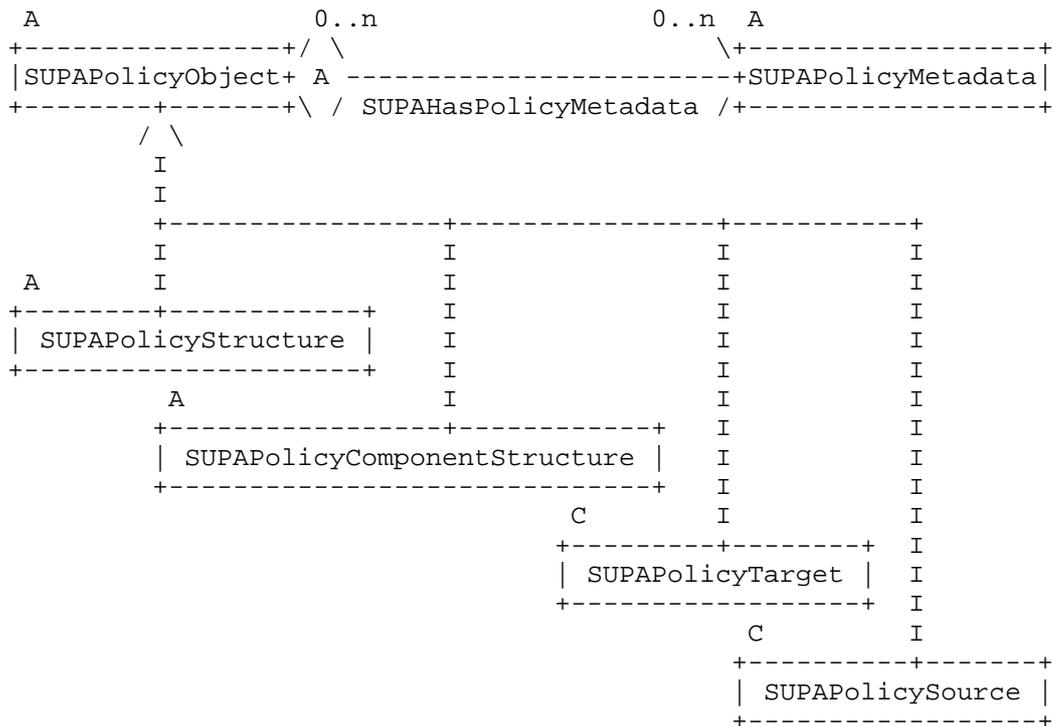
           Figure 17. SUPAPolicyObject and Its Subclasses

   This class is the root of the SUPA class hierarchy. It defines the
   common attributes and relationships that all SUPA subclasses
   inherit.

   A SUPAPolicyObject MAY be qualified by a set of zero or more
   SUPAPolicyMetadata objects. This is provided by the
   SUPAHasPolicyMetadata aggregation (see Section 5.2.2). This
   enables the semantics of the SUPAPolicyObject to be more
   completely specified.

5.2.1. SUPAPolicyObject Attributes

   This section defines the attributes of the SUPAPolicyObject class.
   These attributes are inherited by all subclasses of the GPIM
   except for the SUPAPolicyMetadata class, which is a sibling class.

5.2.1.1. Object Identifiers

   This document defines two class attributes in SUPAPolicyObject,
   called supaPolObjIDContent and supaPolObjIDEncoding, that together
   define a unique object ID. This enables all class instances to be
   uniquely identified.

   One of the goals of SUPA is to be able to generate different data
   models that support different types of protocols and repositories.
   This means that the notion of an object ID must be generic. It is
   inappropriate to use data modeling concepts, such as keys, Globally
   Unique IDentifiers (GUIDs), Universally Unique IDentifiers (UUIDs),
   Fully Qualified Domain Names (FQDNs), Fully Qualified Path Names
   (FQPNs), Uniform Resource Identifiers (URIs), and other similar
   mechanisms, to define the structure of an information model.
   Therefore, a synthetic object ID is defined using these two class
   attributes. This can be used to facilitate mapping to different data
   model object schemes.

   The two attributes work together, with the supaPolObjIDContent
   attribute defining the content of the object ID and the
   supaPolObjIDEncoding attribute defining how to interpret the
   content. These two attributes form a tuple, and together enable
   a machine to understand the syntax and value of an object
   identifier for the object instance of this class.

   Similarly, all SUPA classes are attributes are both uniquely
   named as well as prepended with the prefixes "SUPA" and "supa",
   respectively, to facilitate model integration.

5.2.1.2. The Attribute "supaPolObjIDContent"

   This is a mandatory string attribute that represents part of the
   object identifier of an instance of this class. It defines the
   content of the object identifier. It works with another class
   attribute, called supaPolObjIDEncoding, which defines how to
   interpret this attribute. These two attributes form a tuple,
   and together enable a machine to understand the syntax and value
   of an object identifier for the object instance of this class.
   This is based on the DEN-ng class design [2].

5.2.1.3. The Attribute "supaPolObjIDEncoding"

   This is a mandatory non-zero enumerated integer attribute that
   represents part of the object identifier of an instance of this
   class. It defines the format of the object identifier. It works
   with another class attribute, called supaPolObjIDContent, which
   defines the content of the object ID. These two attributes form
   a tuple, and together enable a machine to understand the syntax
   and value of an object identifier for the object instance of
   this class. The supaPolObjIDEncoding attribute is mapped to the
   following values:

      0: undefined
      1: GUID
      2: UUID
      3: primary key
      4: foreign key
      5: URI
      6: FQDN
      7: FQPN

   The value 0 may be used to initialize the system, or to signal
   that there is a problem with this particular SUPAPolicyObject.

5.2.1.4. The Attribute "supaPolicyDescription"

   This is an optional string attribute that defines a free-form
   textual description of this object.

5.2.1.5. The Attribute "supaPolicyName"

   This is an optional string attribute that defines the name of this
   Policy. This enables any existing generic naming attribute to be
   used for generic naming, while allowing this attribute to be used
   to name Policy entities in a common manner. Note that this is NOT
   the same as the commonName attribute of the Policy class defined
   in [RFC3060], as that attribute is intended to be used with just
   X.500 cn attributes.

5.2.2. SUPAPolicyObject Relationships

   The SUPAPolicyObject class currently defines a single relationship,
   as defined in the subsections below.

5.2.2.1. The Aggregation "SUPAHasPolicyMetadata"

   This is a mandatory aggregation that defines the set of
   SUPAPolicyMetadata that are aggregated by this particular
   SUPAPolicyObject. This aggregation is defined in section
   5.16.2.

5.2.2.2. The Association Class "SUPAHasPolicyMetadataDetail"

   This is a mandatory concrete association class that defines the
   semantics of the SUPAPolicyMetadata aggregation. This enables the
   attributes and relationships of the SUPAPolicyMetadataDetail class
   to be used to constrain which SUPAPolicyMetadata objects can be
   aggregated by this particular SUPAPolicyObject instance. This
   association class is defined in Section 5.16.2.2.

5.3. The Abstract Class "SUPAPolicyStructure"

   This is a mandatory abstract class that is used to represent the
   structure of a SUPAPolicy. This class (and all of its subclasses)
   is a type of PolicyContainer. SUPAPolicyStructure was abstracted
   from DEN-ng [2], and a version of this class is in the process of
   being added to [5]. However, the version in [5] differs
   significantly. First, the class and relationship definitions ared
   different. Second, [5] uses the composite pattern. Neither of
   these are implemented in this document because of optimizations
   done to the SUPA class hierarchy that are NOT present in [5].

   For this release, the only official type of policy that is
   supported is the event-condition-action (ECA) type of policy rule.
   However, the structure of the SUPA hierarchy is defined to
   facilitate adding new types of rules later.

   A SUPAPolicy may take the form of an individual policy or a set
   of policies. This requirement is supported by applying the
   composite pattern to subclasses of the SUPAPolicyStructure class,
   as shown in Figure 5. In this document, this is done for the
   SUPAECAPolicyRule subclass, and results in two subclasses:
   SUPAECAPolicyRuleAtomic (for defining stand-alone policies) and
   SUPAECAPolicyRuleComposite (for defining hierarchies of policies).

   Note that there is no need for a "match strategy attribute" that
   some models [RFC3460], [4], [6] have; this is because the
   SUPAPolicyStructure class is used just for containment. Hence, the
   containers themselves serve as the scoping component for nested
   policies.

5.3.1. SUPAPolicyStructure Attributes

   The following subsections define the attributes of the
   SUPAPolicyStructure class.

   The SUPAPolicyStructure class has a number of attributes that have
   no counterpart in the SUPAPolicyComponentStructure class. This is
   because these attributes are only appropriate at the level of a
   policy rule, not at the level of a policy component.

   Care must be taken in adding attributes to this class,
   because the behavior of future subclasses of this class (e.g.,
   declarative and functional policies) is very different than the
   behavior of SUPAECAPolicyRules.

5.3.1.1. The Attribute "supaPolAdminStatus"

   This is an optional attribute, which is an enumerated non-negative
   integer. It defines the current administrative status of this
   SUPAPolicyClause.

   This attribute can be used to place this particular
   SUPAPolicyStructure object instance into a specific administrative
   state, such as enabled, disabled, or in test. Values include:

      0: Unknown (an error state)
      1: Enabled
      2: Disabled
      3: In Test (i.e., no operational traffic can be passed)

   Value 0 denotes an error that prevents this SUPAPolicyStructure
   from being used. Values 1 and 2 mean that this SUPAPolicyStructure
   is administratively enabled or disabled, respectively. A value of
   3 means that this SUPAPolicyStructure is in a special test mode
   and SHOULD NOT be used as part of an OAM&P policy.

5.3.1.2. The Attribute "supaPolContinuumLevel"

   This is an optional non-negative integer attribute. It defines
   the level of abstraction, or policy continuum level [10], of this
   particular SUPAPolicy. The value assignment of this class is
   dependent on the application; however, it is recommended that
   for consistency with other SUPA attributes, the value of 0 is
   reserved for initialization and/or error conditions.

   By convention, lower values represent more abstract levels of the
   policy continuum. For example, a value of 1 could represent
   business policy, a value of 2 could represent application-specific
   policies, and a value of 3 could represent low=level policies for
   network administrators.

5.3.1.3. The Attribute "supaPolDeployStatus"

   This is an optional enumerated, non-negative integer attribute. The
   purpose of this attribute is to indicate that this SUPAPolicy can
   or cannot be deployed by the policy management system. This
   attribute enables the policy manager to know which SUPAPolicies to
   retrieve, and may be useful for the policy execution system for
   planning the staging of SUPAPolicies. Values include:

      0: undefined
      1: deployed and enabled
      2: deployed and in test
      3: deployed but not enabled
      4: ready to be deployed
      5: cannot be deployed

   If the value of this attribute is 0 or 5, then the policy
   management system SHOULD ignore this SUPAPolicy. Otherwise, the
   policy management MAY use this SUPAPolicy.

5.3.1.4. The Attribute "supaPolExecStatus"

   This is an optional attribute, which is an enumerated,
   non-negative integer. It defines the current execution status
   of this SUPAPolicy. Values include:

      0: undefined
      1: executed and SUCCEEDED (operational mode)
      2: executed and FAILED (operational mode)
      3: currently executing (operational mode)
      4: ready to execute (operational mode)
      5: executed and SUCCEEDED (test mode)
      6: executed and FAILED (test mode)
      7: currently executing (test mode)
      8: ready to execute (test mode)

5.3.1.5. The Attribute "supaPolExecFailStrategy"

   This is an optional non-negative, enumerated integer that defines
   what actions, if any, should be taken by this
   SUPAPolicyStructure object if it fails to execute correctly.

   Note that some systems may not be able to support all options
   specified in this enumeration. If rollback is supported by the
   system, then option 2 may be skipped. Options 3 and 4 can be used
   by systems that do and do not support rollback. Values include:

```
0:  undefined
1:  attempt rollback of all actions taken and stop execution
2:  attempt rollback of only the action that failed and stop
    execution
3:  stop execution but do not rollback any actions
4:  ignore failure and continue execution
```

A value of 0 can be used as an error condition. A value of 1 means
that ALL execution is stopped, rollback of all actions (whether
successful or not) is attempted, and that SUPAPolicies that
otherwise would have been executed are ignored. A value of 2 means
that execution is stopped, and rollback is attempted for ONLY the
SUPAPolicy that failed to execute correctly.

5.3.2. SUPAPolicyStructure Relationships

The SUPAPolicyStructure class owns four relationships, which are
defined in the following subsections.

5.3.2.1. The Aggregation "SUPAHasPolicySource"

This is an optional aggregation, and defines the set of
SUPAPolicySource objects that are attached to this particular
SUPAPolicyStructure object. The semantics of this aggregation
are defined by the SUPAHasPolicySourceDetail association class.
PolicySource objects are used for authorization policies, as well
as to enforce deontic and alethic logic.

The multiplicity of this aggregation is 0..n - 0..n. This means
that it is an optional aggregation; zero or more SUPAPolicySource
objects may be aggregated by this SUPAPolicyStructure object,
and zero or more SUPAPolicyStructure objects may aggregate this
particular SUPAPolicySource object.

5.3.2.2. The Association Class "SUPAHasPolicySourceDetail"

This is an optional concrete association class, and defines the
semantics of the SUPAHasPolicySource aggregation. The attributes and
relationships of this class can be used to define which
SUPAPolicySource objects can be attached to which particular set
of SUPAPolicyStructure objects.

5.3.2.2.1. The Attribute "supaPolSrcIsAuthenticated"

This is an optional Boolean attribute. If the value of this
attribute is true, then this SUPAPolicySource object has been
authenticated by this particular SUPAPolicyStructure object.

5.3.2.2.2. The Attribute "supaPolSrcIsTrusted"

   This is an optional Boolean attribute. If the value of this
   attribute is TRUE, then this particular SUPAPolicySource object
   has been verified to be trusted by this particular
   SUPAPolicyStructure object.

5.3.2.3. The Aggregation "SUPAHasPolicyTarget"

   This is an optional aggregation, and defines the set of
   SUPAPolicyTargets that are attached to this particular
   SUPAPolicyStructure. The semantics of this aggregation is
   defined by the SUPAHasPolicyTargetDetail association class. The
   purpose of this class is to explicitly identify managed objects
   that will be affected by the execution of one or more SUPAPolicies.

   The multiplicity of this aggregation is 0..n - 0..n. This means
   that it is an optional aggregation; zero or more SUPAPolicyTarget
   objects may be aggregated by this SUPAPolicyStructure object,
   and zero or more SUPAPolicyStructure objects may aggregate this
   particular SUPAPolicyTarget object.

5.3.2.4. The Association Class "SUPAHasPolicyTargetDetail"

   This is an optional concrete association class, and defines the
   semantics of the SUPAPolicyTargetOf aggregation. The attributes and
   relationships of this class can be used to define which
   SUPAPolicyTargets can be attached to which particular set of
   SUPAPolicyStructure objects.

5.3.2.4.1. The Attribute "supaPolTgtIsAuthenticated"

   This is an optional Boolean attribute. If the value of this
   attribute is true, then this SUPAPolicyTarget object has been
   authenticated by this particular SUPAPolicyStructure object.

5.3.2.4.2. The Attribute "supaPolTgtIsEnabled"

   This is an optional Boolean attribute. If its value is TRUE, then
   this SUPAPolicyTarget is able to be used as a SUPAPolicyTarget.
   This means that it meets two specific criteria:

      1. it has agreed to play the role of a SUPAPolicyTarget (i.e.,
         it is willing to have SUPAPolicies applied to it, and
      2. it is able to either process (directly or with the aid of
         a proxy) SUPAPolicies or receive the results of a processed
         SUPAPolicy and apply those results to itself.

5.3.2.5. The Association "SUPAHasPolExecFailTakeAction"

   This is an optional association that defines which, if any, actions
   should be taken if this SUPAPolicyStructure object instance fails
   to execute correctly. The semantics of this association are defined
   in the SUPAHasPolExecFailTakeActionDetail association class.

   For a given SUPAPolicyStructure object A, this association defines
   a set of policy action objects B to execute if (and only if) the
   SUPAPolicyStructure object A failed to execute correctly. The
   multiplicity of this association is defined as 0..n on the owner
   (A) side and 1..n on the part (B) side. This means that this
   association is optional; if it is instantiated, then at least one
   SUPAPolicyStructure MUST be instantiated by this
   SUPAPolicyStructure object. Similarly, one or more
   SUPAPolicyStructure objects may be associated with this given
   SUPAPolicyStructure object.

5.3.2.6. The Association Class "SUPAHasPolExecFailTakeActionDetail"

   This is an optional concrete class that defines the semantics for
   the SUPAHasPolExecFailTakeAction association. The attributes and/or
   relationships of this association class can be used to determine
   which policy action objects are executed in response to a failure
   of the SUPAPolicyStructure object instance that owns this
   association. The association relates the policy actions from one
   SUPAPolicyStructure B to be executed if a SUPAPolicyStructure A
   fails to execute properly. Figure 18 illustrates this approach.

```
     A
    +-------------------------+ 0..n
    |                         +--------------------------------+
    |                         |    SUPAHasPolExecFailTakeAction |
    |    SUPAPolicyStructure   |/                               |
    |                         + -------------+-----------------+
    |                         |\             ^
    +-------------------------+ 1..n         |
                                             |
                   C                         |
                  +------------------+-----------------+
                  | SUPAHasPolExecFailTakeActionDetail |
                  +----------------------------------+
```
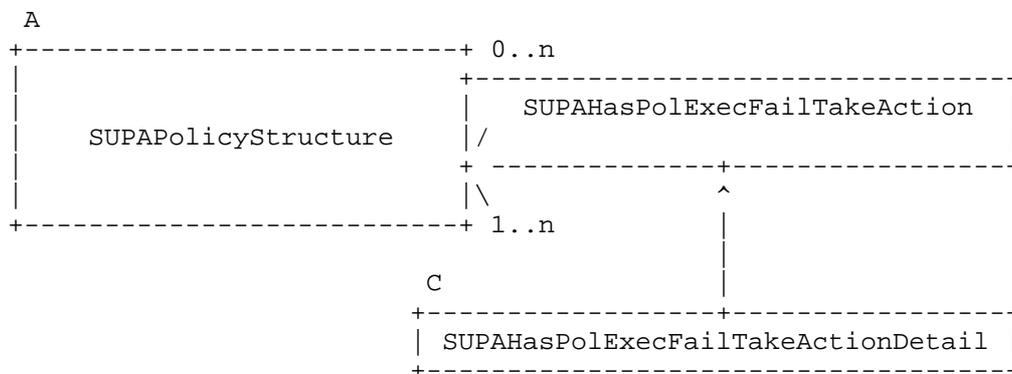
           Figure 18. SUPAHasPolExecFailTakeAction Association

5.3.2.6.1. The Attribute "supaPolExecFailTakeActionEncoding"

   This is an optional enumerated, non-negative integer attribute
   that defines how to find the set of SUPAPolicyActions contained
   in each element of the supaPolExecFailTakeActionName class
   attribute. Values include:

```
            0:   undefined
            1:   String
            2:   GUID
            3:   UUID
            4:   URI
            5:   FQDN
            6:   FQPN
```

5.3.2.6.2. The Attribute "supaPolExecFailTakeActionName[1..n]"

   This is an optional array of string attributes that identifies the
   set of policy actions to take if the SUPAPolicyStructure object
   that owns this association failed to execute properly. The
   interpretation of this string attribute is defined by the
   supaPolExecFailTakeActionEncoding class attribute. The association
   defines the SUPAPolicyStructure that contains the set of policy
   actions to execute, and this attribute defines which of these
   actions are to be executed. Note that there is no need to execute
   a SUPAPolicy, since the event and failure have already occurred.
   Note: [1..n] means that this is a multi-valued property that has
   at least one (and possibly more) attributes.

5.3.2.7. The Aggregation "SUPAHasPolicyClause"

   This is an optional aggregation that defines the set of
   SUPAPolicyClauses that are aggregated by this particular
   SUPAPolicyStructure instance. The semantics of this
   aggregation are defined by the SUPAHasPolicyClauseDetail
   association class.

   Every SUPAPolicyStructure object instance MUST aggregate at
   least one SUPAPolicyClause object instance. However, the
   converse is NOT true. For example, a SUPAPolicyClause could be
   instantiated and then stored for later use in a policy repository.
   Furthermore, the same SUPAPolicyClause could be used by zero or
   more SUPAPolicyStructure object instances at a given time. Thus,
   the multiplicity of this aggregation is defined as 0..1 on the
   aggregate (i.e., the SUPAPolicyStructure side) and 1..n on the
   part (i.e., the SUPAPolicyClause side). This means that at
   least one SUPAPolicyClause MUST be aggregated by this
   SUPAPolicyStructure object. Similarly, a SUPAPolicyClause may
   be aggregated by this particular SUPAPolicyStructure object.

5.3.2.8. The Association Class "SUPAHasPolicyClauseDetail"

   This is an optional abstract association class, and defines the
   semantics of the SUPAHasPolicyClause aggregation. The attributes
   and/or relationships of this association class can be used to
   determine which SUPAPolicyClauses are aggregated by which
   SUPAPolicyStructure objects.

   Attributes will be added to this class at a later time.

5.4. The Abstract Class "SUPAPolicyComponentStructure"

   This is a mandatory abstract class that is the superclass of all
   objects that represent different types of components of a
   SUPAPolicy. Different types of policies have different types of
   structural components. However, all of these are used in at least
   one type of policy. This class represents a convenient control
   point for defining characteristics and behavior that are common
   to objects that serve as components of a policy.

   Note that there are significant differences between the definition
   of this class, and its attributes, and the definition of the
   corresponding class (and its attributes) in [5].

5.4.1. SUPAPolicyComponentStructure Attributes

   No attributes are currently defined for the
   SUPAPolicyComponentStructure class.

5.4.2. SUPAPolicyComponentStructure Relationships

   SUPAPolicyComponentStructure participates in a single relationship,
   SUPAHasDecoratedPolicyComponent, as defined in section 5.7.3.

5.5. The Abstract Class "SUPAPolicyClause"

   This is a mandatory abstract class that separates the
   representation of a SUPAPolicy from its implementation.
   SUPAPolicyClause was abstracted from DEN-ng [2]. This abstraction
   is missing in [RFC3060], [RFC3460], [4], and [6]. This class is
   called PolicyStatement in [5], but the class and relationship
   definitions differ significantly from the corresponding designs
   in this document.

   A SUPAPolicyClause contains an individual or group of related
   functions that are used to define the content of a policy. More
   specifically, since the number and type of functions that make up
   a SUPAPolicyClause can vary, the decorator pattern is used, so
   that the contents of a SUPAPolicyClause can be adjusted
   dynamically at runtime without affecting other objects.

   This document defines two different types of policy clauses:
   SUPAEncodedClause (which is generic, and can be used by any
   type of policy), and SUPABooleanClause (which is also generic,
   but is typically used by SUPAECAPolicyRule objects).

   SUPAPolicyClauses are objects in their own right, which
   facilitates their reuse. SUPAPolicyClauses can aggregate a set
   of any of the subclasses of SUPAPolicyComponentDecorator, which
   was shown in Figure 10. These four subclasses provide four
   different ways to construct a SUPAPolicyClause:

1) SUPAPolicyTerm, which enables constructing a {variable, operator, value} expression for building SUPAPolicyClauses
2) SUPAEncodedClause, which enables policy clauses to be formed as an encoded object (e.g., to pass YANG or CLI code)
3) SUPAPolicyCollection, which defines a collection of objects that requires further processing by the policy management system in order to be made into a SUPAPolicyClause
4) SUPAECAComponent, which enables policy clauses to be formed using (reusable) Event, Condition, and/or Action objects

SUPAPolicyClauses are aggregated by a SUPAPolicyStructure object, which enables all types of SUPAPolicies to uniformly be made up of one or more SUPAPolicyClauses.

5.5.1. SUPAPolicyClause Attributes

This section defines the attributes of the SUPAPolicyClause class, which are inherited by all SUPAPolicyClause subclasses.

5.5.1.1. The Attribute "supaPolClauseExecStatus"

This is an optional enumerated non-negative integer attribute. It defines whether this SUPAPolicyClause is currently in use and, if so, what its execution status is. This attribute can also be used to place this particular SUPAPolicyClause into a specific execution state, such as enabled (values 1-4), in test (value 5) or disabled (value 6). Values include:

    0: Unknown (an error state)
    1: Completed (i.e., successfully executed, but now idle)
    2: Working (i.e., in use and no errors reported)
    3: Not Working (i.e., in use, but errors have been reported)
    4: Available (i.e., could be used, but currently isn't)
    5: In Test (i.e., cannot be used as part of an OAM&P policy)
    6: Disabled (i.e., not available for use)

Value 0 denotes an error that prevents this SUPAPolicyClause from being used. Value 1 means that this SUPAPolicyClause has successfully finished execution, and is now idle and available. Value 2 means that this SUPAPolicyClause is in use; in addition, this SUPAPolicyClause is working correctly. Value 3 is the same as value 2, except that this SUPAPolicyClause is not working correctly. Value 4 means that this SUPAPolicyClause is available, but not currently in use. Value 5 means that this SUPAPolicyClause is in a special test state. A test state signifies that it SHOULD NOT be used to evaluate OAM&P policies. A value of 6 means that this SUPAPolicyClause is unavailable for use.

5.5.2. SUPAPolicyClause Relationships

   SUPAPolicyClause participates in a single relationship,
   SUPAHasPolicyClause, as defined in section 5.3.2.7. Note that
   SUPAPolicyClause uses the decorator pattern to "wrap" this object
   with instances of the (concrete) subclasses of the
   SUPAPolicyComponentDecorator object.

5.6. The Concrete Class "SUPAEncodedClause"

   This is a mandatory concrete class that refines the behavior of a
   SUPAPolicyClause.

   This class defines a generalized extension mechanism for
   representing SUPAPolicyClauses that have not been modeled
   with other SUPAPolicy objects. Rather, the contents of the policy
   clause are directly encoded into the attributes of the
   SUPAEncodedClause. Hence, SUPAEncodedClause objects are reusable
   at the object level, whereas SUPABooleanClause clauses are reusable
   at the individual Boolean expression level.

   This class uses two of its attributes (supaEncodedClauseContent and
   supaEncodedClauseEncoding) for defining the content and type of
   encoding used in a given SUPAPolicyClause. The benefit of a
   SUPAEncodedClause is that it enables direct encoding of the text of
   the SUPAPolicyClause, without having the "overhead" of using other
   objects. However, note that while this method is efficient, it
   does not reuse other SUPAPolicy objects.

5.6.1. SUPAEncodedClause Attributes

   This section defines the attributes of the SUPAEncodedClause class.

5.6.1.1. The Attribute "supaEncodedClauseContent"

   This is a mandatory string attribute, and defines the content of
   this clause. It works with another class attribute, called
   supaEncodedClauseEncoding, which defines how to interpret the
   value of this attribute (e.g., as a string or reference). These
   two attributes form a tuple, and together enable a machine to
   understand the syntax and value of this object instance.

5.6.1.2. The Attribute "supaEncodedClauseEncoding"

   This is a mandatory non-negative integer attribute, and defines
   how to interpret the value of this encoded clause. It works with
   another class attribute (supaEncodedClauseContent), which defines
   the content of the encoded clause. These two attributes form a
   tuple, and together enable a machine to understand the syntax and
   value of the encoded clause for the object instance of this class.
   Values include:

```
0:  undefined
1:  String
2:  GUID
3:  UUID
4:  URI
5:  FQDN
6:  FQPN
```

5.6.1.3. The Attribute "supaEncodedClauseLanguage"

   This is mandatory non-negative integer attribute, and defines
   the type of language used in this encoded clause. Values include:

```
0:  undefined
1:  Text
2:  YANG
3:  XML
4:  CLI
5:  TL1
```

5.6.1.4. The Attribute "supaEncodedClauseLang[0..n]"

   This is an optional array of string attribute that contains
   descriptive information about the type of language used in the
   supaEncodedClauseLanguage class attribute. Text is in comma
   separated value (i.e., vendorName, vendorVersion) format.

5.6.1.5. The Attribute "supaEncodedClauseResponse"

   This is an optional Boolean attribute that emulates a Boolean
   response of this clause, so that it may be combined with other
   subclasses of the SUPAPolicyClause that provide a status as to
   their correctness and/or evaluation state. This enables this
   object to be used to construct more complex Boolean clauses.

5.6.2. SUPAEncodedClause Relationships

   SUPAPolicyClause participates in a single inherited relationship,
   SUPAHasPolicyClause, as defined in section 5.3.2.7.

5.7. The Abstract Class "SUPAPolicyComponentDecorator"

   This is a mandatory class, and is used to implement the decorator
   pattern. The decorator pattern enables all or part of one or more
   objects to "wrap" another concrete object. This means that any
   any concrete subclass of SUPAPolicyClause is wrapped by any
   concrete subclass of SUPAPolicyComponentDecorator, as shown in
   Figure 19 below.

```
          A
        +------------------------------+
        |                              | | 1..n
        |  SUPAPolicyComponentStructure +--------+
        |                              |        | | used to wrap
        +------------------------------+        | concrete
                      / \                        | subclasses of
                       I                         | PolicyClause
                       I                         |
        +--------------+-------------+    / \
              I                       I    A
        A     I              A        I    \ / 0..1
    +----------+---------+   +-------------+-----+---------+
    |  SUPAPolicyClause  |   | SUPAPolicyComponentDecorator |
    +----------+---------+   +-------------+---------------+
              I                        I
              I                        I
            / \                      / \
        Concrete Subclasses,      Concrete Subclasses
     (e.g., SUPAEncodedClause)  (e.g., SUPAPolicyCollection)
       (object being wrapped)      (wrapping object(s))
```
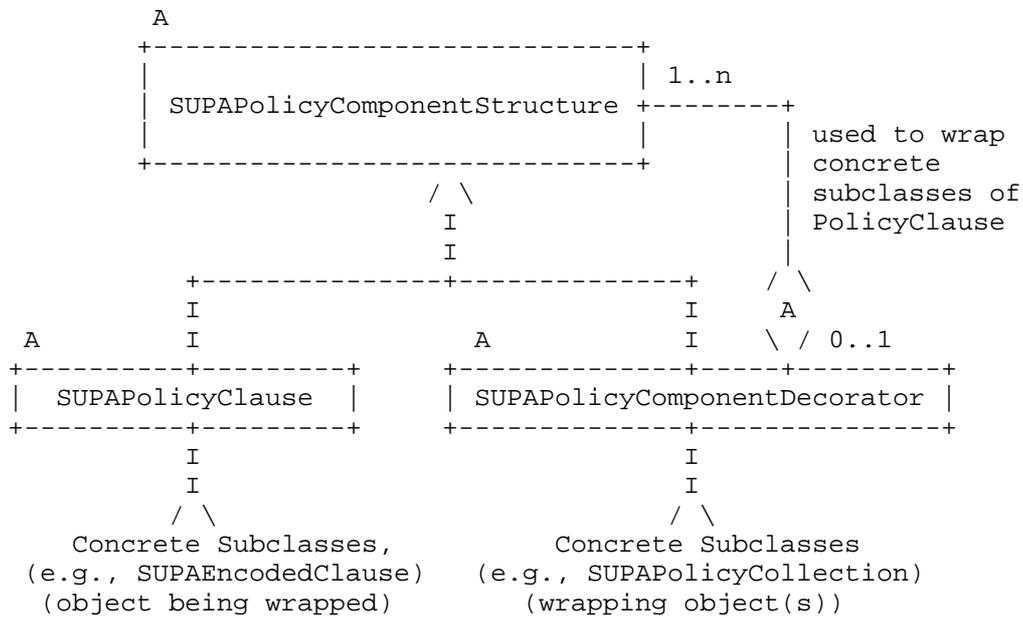
          Figure 19. The PolicyComponent Decorator Pattern

5.7.1. The Decorator Pattern

   Each SUPAPolicyComponentDecorator object HAS_A (i.e., wraps) a
   concrete instance of the SUPAPolicyClause object. This means that
   the SUPAPolicyComponentDecorator object has an instance variable
   that holds a reference to a SUPAPolicyClause object. Since the
   SUPAPolicyComponentDecorator object has the same interface as the
   SUPAPolicyClause object, the SUPAPolicyComponentDecorator object
   (and all of its subclasses) are transparent to clients of the
   SUPAPolicyClause object (and its subclasses). This means that
   SUPAPolicyComponentDecorator object instances can add attributes
   and/or methods to those of the concrete instance of the chosen
   subclass of SUPAPolicyClause.

   Figure 20 shows how this is done for methods. 20a shows the
   initial object to be wrapped; 20b shows SUPAPolicyCollection
   wrapping SUPAEncodedClause; 20c shows SUPAGenericDecoratedComponent
   wrapping SUPAPolicyCollection.
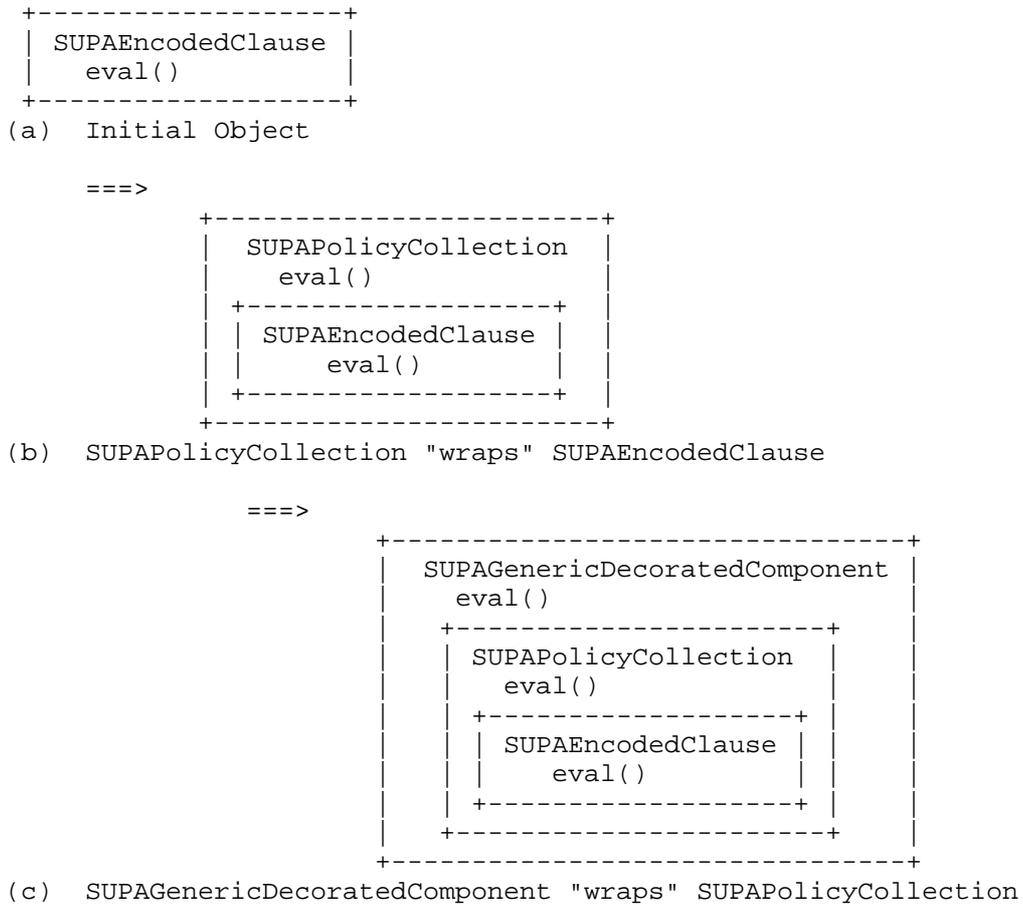
```
   +------------------+
   | SUPAEncodedClause |
   |     eval()        |
   +------------------+
(a)  Initial Object

     ===>
              +-----------------------+
              |  SUPAPolicyCollection  |
              |     eval()             |
              | +------------------+  |
              | | SUPAEncodedClause |  |
              | |     eval()        |  |
              | +------------------+  |
              +-----------------------+
     (b)  SUPAPolicyCollection "wraps" SUPAEncodedClause


            ===>
                   +-------------------------------+
                   |  SUPAGenericDecoratedComponent |
                   |     eval()                     |
                   | +----------------------+      |
                   | |  SUPAPolicyCollection  |      |
                   | |     eval()             |      |
                   | | +------------------+  |      |
                   | | | SUPAEncodedClause |  |      |
                   | | |     eval()        |  |      |
                   | | +------------------+  |      |
                   | +----------------------+      |
                   +-------------------------------+
   (c)  SUPAGenericDecoratedComponent "wraps" SUPAPolicyCollection

        Figure 20.  Conceptual Depiction of eval() Decorated Method
```

When eval() is called in the outermost object
(SUPAGenericDecoratedComponent), it delegates to the eval() method
of SUPAPolicyCollection, which in turn delegates to the eval()
method of SUPAEncodedClause. This method executes and returns the
results to SUPAPolicyCollection, which executes and returns the
results to SUPAGenericDecoratedComponent, which executes and returns
the final result.

5.7.2. SUPAPolicyComponentDecorator Attributes

Currently, there are two attributes defined for this class, which
are described in the following subsections. Both attributes are
used by subclasses to constrain the behavior of that subclass;
they do **not** affect the relationship between the concrete
subclass of SUPAPolicyComponentDecorator that is wrapping the
concrete subclass of SUPAPolicyClause.

This is different than the use of similar attributes defined in
the SUPAHasDecoratedPolicyComponentDetail association class (which
are used to constrain the relationship between the concrete
subclass of SUPAPolicyClause and the concrete subclass of the
SUPAPolicyComponentDecorator object that is wrapping it).
Note that [2] does not define any attributes for this class.

5.7.2.1. The Attribute "supaPolCompConstraintEncoding"

This is a mandatory non-negative enumerated integer that defines
how to interpret each string in the supaPolCompConstraint class
attribute. Values include:

    0: undefined
    1: OCL 2.4
    2: OCL 2.x
    3: OCL 1.x
    4: QVT 1.2 - Relations Language
    5: QVT 1.2 - Operational language
    6: Alloy
    7: English text

Enumerations 1-3 are dedicated to OCL (with OCL 2.4 being the
latest version as of this writing). QVT defines a set of languages
(the two most powerful and useful are defined by enumerations 4
and 5). Alloy is a language for describing constraints, and uses a
SAT solver to guarantee correctness. Note that enumeration 7
(English text) is not recommended (since it is informal, and hence,
not verifiable), but included for completeness.

5.7.2.2. The Attribute "supaAPolCompConstraint[0..n]"

This is a mandatory array of string attributes. Each attribute
specifies a constraint to be applied using the encoding defined in
the supaPolCompConstraintEncoding class attribute. This provides
a more rigorous and flexible treatment of constraints than is
possible in [RFC3460].

Note: [0..n] means that this is a multi-valued property that may
have zero or more attributes.

5.7.3. SUPAPolicyComponentDecorator Relationships

One relationship is currently defined for this class, which is
described in the following subsection.

5.7.3.1. The Aggregation "SUPAHasDecoratedPolicyComponent"

   This is a mandatory aggregation, and is part of a decorator
   pattern. It is used to enable a concrete instance of a
   SUPAPolicyComponentDecorator to dynamically add behavior to a
   specific type of SUPAPolicyClause object. The semantics of this
   aggregation are defined by the
   SUPAHasDecoratedPolicyComponentDetail association class.

5.7.3.2. The Association Class
          "SUPAHasDecoratedPolicyComponentDetail"

   This is a mandatory concrete association class, and defines the
   semantics of the SUPAHasDecoratedPolicyComponent aggregation. The
   purpose of this class is to use the Decorator pattern to determine
   which SUPAPolicyComponentDecorator object instances, if any, are
   required to augment the functionality of the concrete subclass of
   SUPAPolicyClause that is being used.

   Currently, there are two attributes defined for this class, which
   are described in the following subsections. Both attributes are
   used in this association class to constrain the **relationship**
   between the concrete subclass of SUPAPolicyComponentDecorator that
   is wrapping the concrete subclass of SUPAPolicyClause. Note that
   class attributes of SUPAPolicyComponentDecorator (see section
   5.9.2) only affect that specific subclass.

5.7.3.2.1. The Attribute "supaDecoratedConstraintEncoding"

   This is a mandatory non-negative enumerated integer that defines
   how to interpret each string in the supaDecoratedConstraint class
   attribute. Values include:

     0: undefined
     1: OCL 2.4
     2: OCL 2.x
     3: OCL 1.x
     4: QVT 1.2 - Relations Language
     5: QVT 1.2 - Operational language
     6: Alloy
     7: English text

   Enumerations 1-3 are dedicated to OCL (with OCL 2.4 being the
   latest version as of this writing). QVT defines a set of languages
   (the two most powerful and useful are defined by enumerations 4
   and 5). Alloy is a language for describing constraints, and uses a
   SAT solver to guarantee correctness. Note that enumeration 7
   (English text) is not recommended (since it is informal, and hence,
   not verifiable), but included for completeness.

5.7.3.2.2. The Attribute "supaDecoratedConstraint[0..n]"

   This is a mandatory array of string attributes. Its purpose is to
   collect a set of constraints to be applied to a decorated object.
   The interpretation of each constraint in the array is defined in
   the supaDecoratedConstraintsEncoding class attribute.
   Note: [0..n] means that this is a multi-valued property that may
   have zero or more attributes.

5.7.4. Illustration of Constraints in the Decorator Pattern

   The following example will illustrate how the different constraints
   defined in sections 5.7.2 (class attribute constraints) and section
   5.7.3 (relationship constraints) can be used.

   Figure 21 builds a simple SUPAPolicyClause that has both types
   of relationships.

```
   A                                    A
   +-----------------+          0..1 +------------------------------+
   |                 | 1..n     / \|                              |
   |  SUPAPolicyClause +------+-----  A +SUPAPolicyComponentDecorator|
   |                 |      ^    \ /|                              |
   +--------+--------+      |      +----------+----------------+
            I             |                 I
            I             |                 I
       C    I             |            C    I
    +--------+--------+     |       +---------+----------+
    |SUPAEncodedClause|     |       |SUPAPolicyCollection|
    +----------------+     |       +-------------------+
                          |
         C                |
       +----------------+------------------+
       |SUPAHasDecoratedPolicyComponentDetail|
       +----------------------------------+
```
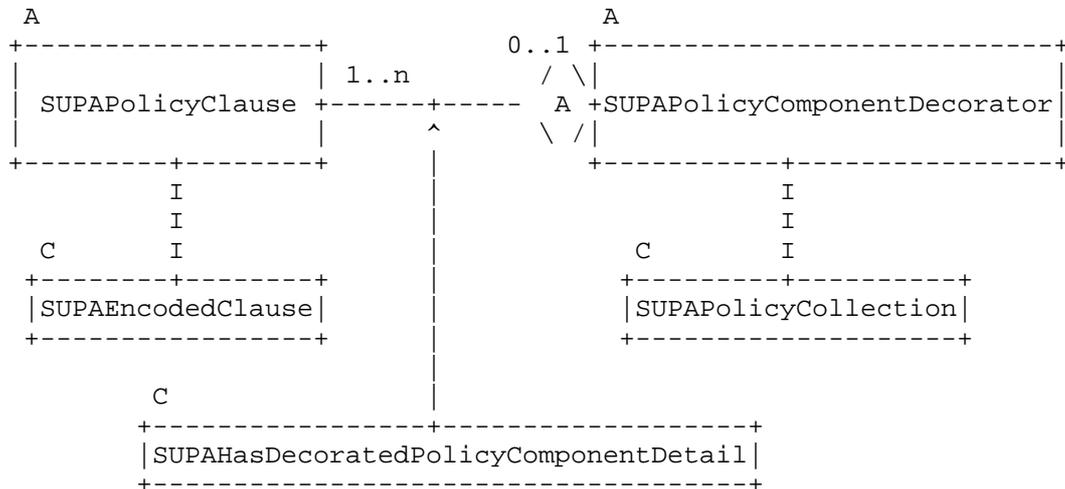
              Figure 21. Constraints in the Decorator Pattern

   Figure 21 says that a SUPAPolicyClause, realized as a
   SUPAEncodedClause, is wrapped by a SUPAPolicyCollection object.
   The attributes in the SUPAPolicyComponentDecorator object are used
   to constrain the attributes in the SUPAPolicyCollection object,
   while the attributes in the SUPAHasDecoratedPolicyComponentDetail
   object are used to contrain the behavior of the aggregation
   (SUPAHasDecoratedPolicyComponent). For example, the attributes in
   the SUPAPolicyComponentDecorator object could restrict the data
   type and range of the components in the SUPAPolicyCollection, while
   the attributes in the SUPAHasDecoratedPolicyComponentDetail object
   could restrict which SUPAPolicyCollection objects are allowed to be
   used with which SUPAEncodedClauses.

5.8. The Abstract Class "SUPAPolicyTerm"

   This is a mandatory abstract class that is the parent of
   SUPAPolicy objects that can be used to define a standard way to
   test or set the value of a variable. It does this by defining a
   3-tuple, in the form {variable, operator, value}, where each
   element of the 3-tuple is defined by a concrete subclass of the
   appropriate type (i.e., SUPAPolicyVariable, SUPAPolicyOperator,
   and SUPAPolicyValue classes, respectively). For example, a
   generic test or set of the value of a variable is expressed as:

       {variable, operator, value}.

   For event and condition clauses, this is typically as written above
   (e.g., does variable = value); for action clauses, it is typically
   written as <operator> <variable> <value> (e.g., SET var to 1). A
   class diagram is shown in Figure 22.

```
               A
               +----------------+
               | SUPAPolicyTerm |
               +--------+-------+
                       / \
                        I
                        I
                        I
        +----------------+---+------------------+
        I                    I                  I
        I                    I                  I
   C    I               C    I             C    I
   +--------+---------+  +--------+---------+  +-------+-------+
   |SUPAPolicyVariable|  |SUPAPolicyOperator|  |SUPAPolicyValue|
   +----------------+  +----------------+  +--------------+
```
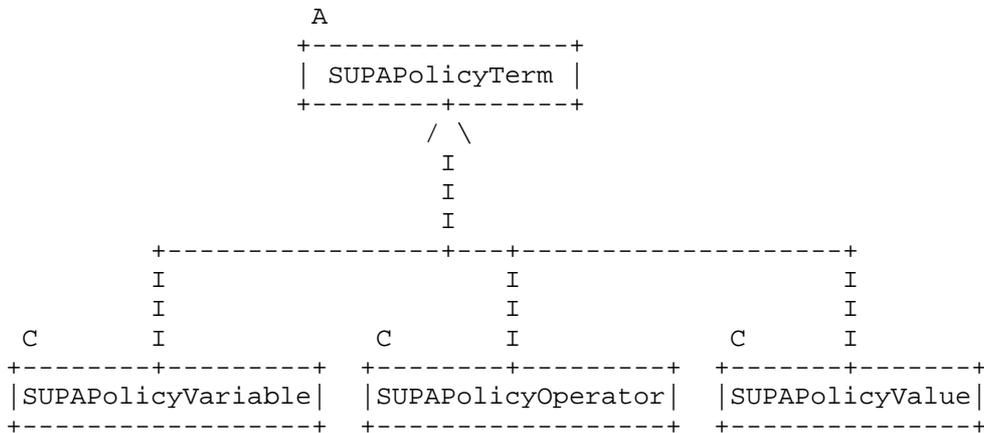
            Figure 22. SUPAPolicyTerm Class Hierarchy

   Note that generic test and set expressions do not have to only use
   objects that are subclasses of SUPAPolicyTerm. For example, the
   polVendorDecoratedContent attribute of the
   SUPAGenericDecoratedComponent could be used as the variable (or the
   value) term of a get or set expression.

   Hence, the utility of the subclasses of SUPAPolicyTerm is in the
   ability of its subclasses to define a generic framework for
   implementing get and set expressions. This is in contrast to
   previous designs (e.g., [RFC3460] and [6]), which depended on
   defining a broad set of subclasses of PolicyVariable and
   PolicyValue. (Note that [4] does not have this generic capability).

5.8.1. SUPAPolicyTerm Attributes

   Currently, SUPAPolicyTerm defines a single attribute, as described
   in the following subsection. Constraints on the subclasses of
   SUPAPolicyTerm can be applied in two different ways:

      1. use SUPAPolicyComponentDecorator attributes to constrain
         just that individual subclass, and/or
      2. use SUPAHasDecoratedPolicyComponentDetail association class
         attributes to constrain the relationship between the concrete
         subclass of SUPAPolicyClause and the concrete subclass of
         the SUPAPolicyTerm class

5.8.1.1. The Attribute "supaPolTermIsNegated"

   This is a mandatory Boolean attribute. If the value of this
   attribute is true, then this particular SUPAPolicyTerm subclass
   (which represents a term) is negated; otherwise, it is not.

5.8.2. SUPAPolicyTerm Relationships

   Currently, no dedicated relationships are defined for the
   SUPAPolicyTerm class (as there are in [RFC3460] and [6]) that
   aggregate policy variable and policy value objects into a policy
   rule). This is:

      1) to enable the subclasses of SUPAPolicyTerm to be used by
         other SUPAPolicyComponentDecorator objects, and
      2) because the decorator pattern replaces how such relationships
         were used in [RFC3460] and [6].

   SUPAPolicyTerm, and its subclasses, inherit the
   SUPAHasDecoratedPolicyComponent aggregation, which was defined in
   section 5.7.3.

5.9. The Concrete Class "SUPAPolicyVariable"

   This is a mandatory concrete class that defines information that
   forms a part of a SUPAPolicyClause. It specifies a concept or
   attribute that represents a variable, which should be compared to
   a value, as specifed in this SUPAPolicyClause. If it is used in
   a SUPAECAPolicyRule, then its value MAY be able to be changed at
   any time, including run-time, via use of the decorator pattern.
   This is not possible in previous designs ([RFC3460, [4], and [6]).

   The value of a SUPAPolicyVariable is typically compared to the
   value of a SUPAPolicyValue using the type of operator defined in
   a SUPAPolicyOperator. However, other objects may be used instead
   of a SUPAPolicyValue object, and other operators may be defined
   in addition to those defined in the SUPAPolicyOperator class.

SUPAPolicyVariables are used to abstract the representation of a
SUPAPolicyRule from its implementation. Some SUPAPolicyVariables
are restricted in the values and/or the data type that they may
be assigned. For example, port numbers cannot be negative, and
they cannot be floating-point numbers. These and other constraints
may be defined in two different ways:

   1. use SUPAPolicyComponentDecorator attributes to constrain
      just that individual subclass, and/or
   2. use SUPAHasDecoratedPolicyComponentDetail association class
      attributes to constrain the relationship between the concrete
      subclass of SUPAPolicyClause and the concrete subclass of
      the SUPAPolicyVariable class

Please refer to the examples in section 7, which show how to
restrict the value, data type, range, and other semantics of the
SUPAPolicyVariable when used in a SUPAPolicyClause.

5.9.1. Problems with the RFC3460 Version of PolicyValue

Please see Appendix A for a detailed comparison.

5.9.2. SUPAPolicyVariable Attributes

SUPAPolicyVariable defines one attribute, as described below.

5.9.2.1. The Attribute "supaPolVarName"

This is an optional string attribute that contains the name of
this SUPAPolicyVariable. This variable name forms part of the
{variable, operator, value} canonical form of a SUPAPolicyClause.

5.9.3. SUPAPolicyVariable Relationships

Currently, no relationships are defined for the SUPAPolicyVariable
class (note that the decorator pattern obviates the need for
relationships such as those in [RFC3460] and [6]).
SUPAPolicyVariable, and its subclasses, inherit the
SUPAHasDecoratedPolicyComponent aggregation, which was defined in
section 5.7.3.

5.10. The Concrete Class "SUPAPolicyOperator"

This is a mandatory concrete class for modeling different types of
operators that are used in a SUPAPolicyClause.

The restriction of the type of operator used in a SUPAPolicyClause
restricts the semantics that can be expressed in that
SUPAPolicyClause. It is typically used with SUPAPolicyVariables
and SUPAPolicyValue to form a SUPAPolicyClause.

5.10.1. Problems with the RFC3460 Version

   Please see Appendix A for a detailed comparison.

5.10.2. SUPAPolicyOperator Attributes

   Currently, SUPAPolicyOperator defines a single generic attribute,
   as described below.

5.10.2.1. The Attribute "supaPolOpType"

   This is a mandatory non-negative enumerated integer that specifies
   the various types of operators that are allowed to be used in this
   particular SUPAPolicyClause. Values include:

      0: Unknown
      1: Greater than
      2: Greater than or equal to
      3: Less than
      4: Less than or equal to
      5: Equal to
      6: Not equal to
      7: IN
      8: NOT IN
      9: SET
     10: CLEAR
     11: BETWEEN (inclusive)

   Note that 0 is an unacceptable value. Its purpose is to support
   dynamically building a SUPAPolicyClause by enabling the
   application to set the value of this attribute to a standard
   default value if the real value is not yet known.

   Additional operators may be defined in future work. For example,
   if SUPAPolicyVariables and SUPAPolicyValues are expanded to/from
   include structured objects, then "deep" versions of operators
   1-6 could also be defined. In this case, values 1-6 will be
   edited to explicitly indicate that they perform "shallow"
   comparison operations.

5.10.3. SUPAPolicyOperator Relationships

   Currently, no relationships are defined for the SUPAPolicyOperator
   class (note that the decorator pattern obviates the need for
   relationships such as those in [6]). SUPAPolicyOperator, and its
   subclasses, inherit the SUPAHasDecoratedPolicyComponent
   aggregation, which was defined in section 5.7.3.

   Please refer to the examples in section 7, which show how to
   restrict the value, data type, range, and other semantics of the
   SUPAPolicyOperator when used in a SUPAPolicyClause.

5.11. The Concrete Class "SUPAPolicyValue"

   The SUPAPolicyValue class is a mandatory concrete class for
   modeling different types of values and constants that occur in a
   SUPAPolicyClause.

   SUPAPolicyValues are used to abstract the representation of a
   SUPAPolicyRule from its implementation. Therefore, the design of
   SUPAPolicyValues depends on two important factors. First, just as
   with SUPAPolicyVariables (see Section 5.11), some types of
   SUPAPolicyValues are restricted in the values and/or the data
   type that they may be assigned. Second, there is a high likelihood
   that specific applications will need to use their own variables
   that have specific meaning to a particular application.

   In general, there are two ways to apply constraints to an object
   instance of a SUPAPolicyValue:

      1. use SUPAPolicyComponentDecorator attributes to constrain
         just that individual subclass, and/or
      2. use SUPAHasDecoratedPolicyComponentDetail association class
         attributes to constrain the relationship between the concrete
         subclass of SUPAPolicyClause and the concrete subclass of
         the SUPAPolicyValue class

   The value of a SUPAPolicyValue is typically compared to the value
   of a SUPAPolicyVariable using the type of operator defined in
   a SUPAPolicyOperator. However, other objects may be used instead
   of a SUPAPolicyVariable object, and other operators may be defined
   in addition to those defined in the SUPAPolicyOperator class.

5.11.1. Problems with the RFC3460 Version of PolicyValue

   Please see Appendix A for a detailed comparison.

5.11.2. SUPAPolicyValue Attributes

   Currently, SUPAPolicyValue defines two generic attributes, as
   described below.

5.11.2.1. The Attribute "supaPolValContent[0..n]"

   This is a mandatory attribute that defines an array of strings.
   The array contains the value(s) of this SUPAPolicyValue object
   instance. Its data type is defined by the supaPolValEncoding
   class attribute.
   Note: [0..n] means that this is a multi-valued property that has
   zero or more attributes.

5.11.2.2. The Attribute "supaPolValEncoding"

    This is a mandatory string attribute that contains the data type
    of the SUPAPolicyValue object instance. Its value is defined by
    the supaPolValContent class attribute. Values include:

         0:   Undefined
         1:   String
         2:   Integer
         3:   Boolean
         4:   Floating Point
         5:   DateTime
         6:   GUID
         7:   UUID
         8:   URI
         9:   FQDN
        10:   FQPN
        11:   NULL

    A string is a sequence of zero or more characters. An Integer is
    a whole number (e.g., it has no fractional part). A Boolean
    represents the values TRUE and FALSE. A floating point number may
    contain fractional values, as well as an exponent. A DateTime
    represents a value that has a date and/or a time component (as in
    the Java or Python libraries). A NULL explicitly models the lack
    of a value.

5.11.3. SUPAPolicyValue Relationships

    Currently, no relationships are defined for the SUPAPolicyValue
    class (note that the decorator pattern obviates the need for
    relationships such as those in [6]). SUPAPolicyValue, and its
    subclasses, inherit the SUPAHasDecoratedPolicyComponent
    aggregation, which was defined in section 5.7.3.

    Please refer to the examples in section 7, which show how to
    restrict the value, data type, range, and other semantics of the
    SUPAPolicyValue when used in a SUPAPolicyClause.

5.12. The Concrete Class "SUPAGenericDecoratedComponent"

    A SUPAGenericDecoratedComponent enables a custom, vendor-specific
    object to be defined and used in a SUPAPolicyClause. This class
    was derived from [2], but is not present in [RFC3460], [4], [5],
    or [6].

    This should not be confused with the SUPAEncodedClause class. The
    SUPAGenericDecoratedComponent class represents a single, atomic,
    vendor-specific object that defines a **portion** of a
    SUPAPolicyClause, whereas a SUPAEncodedClause, which may or
    may not be vendor-specific, represents an **entire**
    SUPAPolicyClause.

5.12.1. SUPAGenericDecoratedComponent Attributes

   Currently, SUPAGenericDecoratedComponent defines two generic
   attributes, as described below.

5.12.1.1. The Attribute "supaVendorDecoratedCompContent[0..n]"

   This is a mandatory attribute that defines an array of strings.
   This array contains the value(s) of the
   SUPAGenericDecoratedComponent object instance. Its data type is
   defined by the supaVendorDecoratedEncoding class attribute.
   Note: [0..n] means that this is a multi-valued property that has
   zero or more attributes.

5.12.1.2. The Attribute "supaVendorDecoratedCompEncoding"

   This is a mandatory integer attribute that defines the format of
   the supaVendorDecoratedContent class attribute. Values include:

       0:   undefined
       1:   String
       2:   Integer
       3:   Boolean
       4:   Floating Point
       5:   DateTime
       6:   GUID
       7:   UUID
       8:   URI
       9:   FQDN
      10:   FQPN
      11:   NULL

   A string is a sequence of zero or more characters. An Integer is
   a whole number (e.g., it has no fractional part). A Boolean
   represents the values TRUE and FALSE. A floating point number may
   contain fractional values, as well as an exponent. A DateTime
   represents a value that has a date and/or a time component (as in
   the Java or Python libraries). A NULL explicitly models the lack
   of a value.

5.12.2. SUPAGenericDecoratedComponent Relationships

   Currently, no relationships are defiend for the
   SUPAGenericDecoratedComponent class (note that the decorator
   pattern obviates the need for relationships such as those in [6]).
   SUPAGenericDecoratedComponent participates in a single relationship,
   SUPAHasDecoratedPolicyComponent, as defined in section 5.7.3.

5.13. The Concrete Class "SUPAPolicyCollection"

   A SUPAPolicyCollection is an optional concrete class that enables
   a collection (e.g., set, bag, or other, more complex, collections
   of elements) of **arbitrary objects** to be defined and used as
   part of a SUPAPolicyClause. This class was derived from [2], but
   is not present in [RFC3460], [4], [5], or [6].

5.13.1. Motivation

   One of the problems with ECA policy rules is when a set of events
   or conditions needs to be tested. For example, if a set of events
   is received, the policy system may need to wait for patterns of
   events to emerge (e.g., any number of Events of type A, followed
   by either one event of type B or two events of type Event C).
   Similarly, a set of conditions, testing the value of an attribute,
   may need to be performed. Both of these represent behavior
   similar to a set of if-then-else statements or a switch statement.

   It is typically not desirable for the policy system to represent
   each choice in such conditions as its own policy clause (i.e., a
   3-tuple), as this creates object explosion and poor performance.
   Furthermore, in these cases, it is often required to have a set of
   complex logic to be executed, where the logic varies according to
   the particular event or condition that was selected. It is much
   too complex to represent this using separate objects, especially
   when the logic is application- and/or vendor-specific.

   However, recall that one of the goals of this document was to
   facilitate the machine-driven construction of policies. Therefore,
   a solution to this problem is needed.

5.13.2. Solution

   Therefore, this document defines the concept of a collection of
   entities, called a SUPAPolicyCollection. Conceptually, the items
   to be collected (e.g., events or conditions) are aggregated in
   one or more SUPAPolicyCollection objects of the appropriate type.
   Another optional SUPAPolicyCollection object could be used to
   aggregate logic blocks (including SUPAPolicies) to execute.
   Once finished, all appropriate SUPAPolicyCollection objects are
   sent to an external system for evaluation.

   The computation(s) represented by the SUPAPolicyCollection may be
   part of a larger SUPAPolicyClause, since SUPAPolicyCollection is a
   subclass of SUPAPolicyComponentDecorator, and can be used to
   decorate a SUPAPolicyClause. Therefore, the external system is
   responsible for providing a Boolean TRUE or FALSE return value, so
   that the policy system can use that value to represent the
   computation of the function(s) performed in the
   SUPAPolicyCollection in a Boolean clause.

5.13.3. SUPAPolicyCollection Attributes

   Currently, SUPAGenericDecoratedComponent defines five attributes,
   as described below.

5.13.3.1. The Attribute "supaPolCollectionContent[0..n]"

   This is an optional attribute that defines an array of strings.
   Each string in the array identifies a domain-suitable identifier of
   an object that is collected by this SUPAPolicyCollection instance.
   Note: [0..n] means that this is a multi-valued property that has
   zero or more attributes.

5.13.3.2. The Attribute "supaPolCollectionEncoding"

   This is an optional non-negative enumerated integer that defines
   the data type of the content of this collection instance. Values
   include:

      0: undefined
      1: by regex (regular expression)
      2: by URI

   For example, if the value of this attribute is 1, then each of
   the strings in the supaPolCollectionContent attribute represent
   a regex that contains all or part of a string to match the class
   name of the object that is to be collected by this instance of
   a SUPAPolicyCollection class.

5.13.3.3. The Attribute "supaPolCollectionFunction"

   This is an optional non-negative enumerated integer that defines
   the function of this collection instance. Values include:

      0: undefined
      1: event collection
      2: condition collection
      3: action collection
      4: logic collection

   Values 1-3 define a collection of objects that are to be used to
   populate the event, condition, or action clauses, respectively, of
   a SUPAECAPolicyRule. A value of 4 indicates that this collection
   contains objects that define logic for processing a SUPAPolicy.

5.13.3.4. The Attribute "supaPolCollectionIsOrdered"

   This is an optional Boolean attribute. If the value of this
   attribute is TRUE, then all elements in this instance of this
   SUPAPolicyCollection are ordered.

5.13.3.5. The Attribute "supaPolCollectionType"

   This is an optional non-negative enumerated integer that defines
   the type of collection that this instance is. Values include:

     0: undefined
     1: set
     2: bag (e.g., multi-set)
     3: dictionary (e.g., associative array)

   A set is an unordered collection of elements that MUST NOT have
   duplicates. A bag is an unordered collection of elements; it MAY
   also have duplicates. A dictionary is a table that associates a
   key with a value.

   Sets have a number of important functions, including:

     o membership: returns TRUE if the element being tested is
                     in the set, and FALSE otherwise
     o subset: returns TRUE if all elements in the first set
                     are also in the second set
     o union: returns all elements from both sets with no
                     duplicates
     o intersection: returns all elements that are in both sets
                     with no duplicates
     o difference: returns all elements in the first set that
                     are not in the second set

   Bags have a number of important functions in addition to the
   functions defined for sets (note that while the above set of
   functions for a set and a bag are the same, a bag is a different
   data type than a set):

     o multiplicity: returns the number of occurrences of an
                      element in the bag
     o count: returns the number of all items, including
                      duplicates
     o countDistinct: returns the number of items, where all
                      duplicates are ignored

   A dictionary is an unordered set of key:value pairs, where each
   key is unique within a given dictionary. The combination of a
   key and a value is called an item. The format of an item is
   defined as one element (the key) followed by a colon followed
   by a second element (the value). Each item in a set of items is
   separated by a comma. Keys MUST NOT be NULL; values MAY be NULL.

   An example of a dictionary is {20:"FTP", 21:"FTP", 22: "SSH"}.
   An example of a null dictionary is simply {}.

5.13.4. SUPAPolicyCollection Relationships

   Currently, no relationships are defined for the
   SUPAGenericDecoratedComponent class (note that the decorator
   pattern obviates the need for relationships such as those in [6]).
   SUPAPolicyCollection participates in a single relationship,
   SUPAHasDecoratedPolicyComponent, as defined in section 5.7.3.

5.14. The Concrete Class "SUPAPolicySource"

   This is an optional class that defines a set of managed entities
   that authored, or are otherwise responsible for, this
   SUPAPolicyRule. Note that a SUPAPolicySource does NOT
   evaluate or execute SUPAPolicies. Its primary use is for
   auditability and the implementation of deontic and/or alethic logic.
   A class diagram is shown in Figure 12.

   A SUPAPolicySource SHOULD be mapped to a role or set of roles
   (e.g., using the role-object pattern [11]). This enables
   role-based access control to be used to restrict which entities
   can author a given policy. Note that Role is a type of
   SUPAPolicyMetadata.

5.14.1. SUPAPolicySource Attributes

   Currently, no attributes are defined for this class.

5.14.2. SUPAPolicySource Relationships

   SUPAPolicySource participates in a single relationship,
   SUPAHasPolicySource, as defined in section 5.3.2.1.
   SUPAPolicySource, and its subclasses, inherit the
   SUPAHasDecoratedPolicyComponent aggregation, which was
   defined in section 5.7.3.

5.15. The Concrete Class "SUPAPolicyTarget"

   This is an optional class that defines a set of managed entities
   that a SUPAPolicy is applied to. Figure 12 shows a class diagram
   of the SUPAPolicyTarget.

   A managed object must satisfy two conditions in order to be defined
   as a SUPAPolicyTarget. First, the set of managed entities that are
   to be affected by the SUPAPolicy must all agree to play the role of
   a SUPAPolicyTarget. In general, a managed entity may or may not be
   in a state that enables SUPAPolicies to be applied to it to change
   its state; hence, a negotiation process may need to occur to enable
   the SUPAPolicyTarget to signal when it is willing to have
   SUPAPolicies applied to it. Second, a SUPAPolicyTarget must be able
   to process (directly or with the aid of a proxy) SUPAPolicies.

If a proposed SUPAPolicyTarget meets both of these conditions, it
SHOULD set its supaPolicyTargetEnabled Boolean attribute to a
value of TRUE.

A SUPAPolicyTarget SHOULD be mapped to a role (e.g., using the
role-object pattern). This enables role-based access control to
be used to restrict which entities can author a given policy.
Note that Role is a type of SUPAPolicyMetadata.

5.15.1. SUPAPolicyTarget Attributes

Currently, no attributes are defined for the SUPAPolicyTarget
class.

5.15.2. SUPAPolicyTarget Relationships

SUPAPolicyTarget participates in a single relationship,
SUPAHasPolicyTarget, as defined in section 5.3.2.3.

5.16. The Abstract Class "SUPAPolicyMetadata"

Metadata is information that describes and/or prescribes
characteristics and behavior of another object that is **not**
an inherent, distinguishing characteristic or behavior of that
object (otherwise, it would be an integral part of that object).

For example, a socialSecurityNumber attribute should not be part
of a generic Person class. First, most countries in the world do
not know what a social security number is, much less use them.
Second, a person is not created with a social security number;
rather, a social security number is used to track people for
administering social benefits, though it is also used as a form
of identification.

Continuing the example, a better way to add this capability to a
model would be to have a generic Identification class, then
define a SocialSecurityNumber subclass, populate it as necessary,
and then define a composition between a Person and it (this is a
composition because social security numbers are not reused).

Since social security numbers are given to US citizens, permanent
residents, and temporary working residents, and because it is
also used to administer benefits, the composition is realized
as an association class to define how it is being used.

An example of descriptive metadata for network elements would be
documentation about best current usage practices (this could also
be in the form of a reference). An example of prescriptive
metadata for network elements would be the definition of a time
period during which specific types of operations are allowable.

This is an optional class that defines the top of a hierarchy of model elements that are used to define different types of metadata that can be applied to policy and policy component objects. This enables common metadata to be defined as objects and then reused when the metadata are applicable. One way to control whether SUPAPolicyMetadata objects are reused is by using the attributes of the SUPAHasPolicyMetadataDetail association class.

It is recommended that this class, along with its SUPAPolicyConcreteMetadata and SUPAPolicyMetadataDecorator subclasses, be used as part of a conformant implementation. It is defined to be optional, since metadata is not strictly required. However, metadata can help specify and describe SUPAPolicyObject entities, and can also be used to drive dynamic behavior.

5.16.1. SUPAPolicyMetadata Attributes

This section defines the attributes of the SUPAPolicyMetadata class.

5.16.1.1. The Attribute "supaPolMetadataDescription"

This is an optional string attribute that defines a free-form textual description of this metadata object.

5.16.1.2. The Attribute "supaPolMetadataIDContent"

This is a mandatory string attribute that represents part of the object identifier of an instance of this class. It defines the content of the object identifier. It works with another class attribute, called supaPolMetadataIDEncoding, which defines how to interpret this attribute. These two attributes form a tuple, and together enable a machine to understand the syntax and value of an object identifier for the object instance of this class.

5.16.1.3. The Attribute "supaPolMetadataIDEncoding"

This is an optional non-zero enumerated integer attribute that represents part of the object identifier of an instance of this class. It defines the format of the object identifier. It works with another class attribute, called supaPolMetadataIDContent, which defines the content of the object ID.

These two attributes form a tuple, and together enable a machine to understand the syntax and value of an object identifier for the object instance of this class. The supaPolMetadataIDEncoding attribute is mapped to the following values:

```
     0:  undefined
     1:  GUID
     2:  UUID
     3:  URI
     4:  FQDN
     5:  FQPN
```

5.16.1.4. The Attribute "supaPolMetadataName"

   This is an optional string attribute that defines the name of this
   SUPAPolicyMetadata object.

5.16.2. SUPAPolicyMetadata Relationships

   SUPAPolicyMetadata participates in a single aggregation, which is
   defined in the following subsections.

5.16.2.1. The Aggregation "SUPAHasPolicyMetadata"

   This is a mandatory aggregation that defines the set of
   SUPAPolicyMetadata that are aggregated by this particular
   SUPAPolicyObject. It is recommended that this aggregation be used
   as part of a conformant implementation.

   The multiplicity of this relationship is defined as 0..n on the
   aggregate (SUPAPolicyObject) side, and 0..n on the part
   (SUPAPolicyMetadata) side. This means that this relationship is
   optional. The semantics of this aggregation are implemented using
   the SUPAHasPolicyMetadataDetail association class.

5.16.2.2. The Abstract Class "SUPAHasPolicyMetadataDetail"

   This is a mandatory concrete association class, and defines the
   semantics of the SUPAHasPolicyMetadata aggregation. Its purpose is
   to determine which SUPAPolicyMetadata object instances should be
   attached to which particular object instances of the
   SUPAPolicyObject class. This is done by using the attributes and
   relationships of the SUPAPolicyMetadataDetail class to constrain
   which SUPAPolicyMetadata objects can be aggregated by which
   particular SUPAPolicyObject instances. It is recommended that this
   association class be used as part of a conformant implementation.

5.16.2.2.1. The Attribute "supaPolMetadataIsApplicable"

   This is an optional Boolean attribute. If the value of this
   attribute is TRUE, then the SUPAPolicyMetadata object(s) of this
   particular SUPAHasPolicyMetadata aggregation SHOULD be aggregated
   by this particular SUPAPolicyObject.

5.16.2.2.2. The Attribute "supaPolMetadataConstraintEncoding"

   This is an optional non-negative enumerated integer that defines
   how to interpret each string in the supaPolMetadataConstraint
   class attribute. Values include:

      0: undefined
      1: OCL 2.4
      2: OCL 2.x
      3: OCL 1.x
      4: QVT 1.2 - Relations Language
      5: QVT 1.2 - Operational language
      6: Alloy
      7: English text

   Enumerations 1-3 are dedicated to OCL (with OCL 2.4 being the
   latest version as of this writing). QVT defines a set of languages
   (the two most powerful and useful are defined by enumerations 4
   and 5). Alloy is a language for describing constraints, and uses a
   SAT solver to guarantee correctness. Note that enumeration 7
   (English text) is not recommended (since it is informal, and hence,
   not verifiable), but included for completeness.

   If this class is instantiated, then this attribute SHOULD also be
   instantiated, and should be part of a conformant implementation.

5.16.2.2.3. The Attribute "supaPolMetadataConstraint[0..n]"

   This is an optional array of string attributes. Each attribute
   specifies a constraint to be applied using the format identified
   by the value of the supaPolMetadataPolicyConstraintEncoding class
   attribute. This provides a more rigorous and flexible treatment of
   constraints than is possible in [RFC3460].

   If this class is instantiated, then this attribute SHOULD also be
   instantiated, and should be part of a conformant implementation.
   Note: [0..n] means that this is a multi-valued property that has
   zero or more attributes.

5.17. The Concrete Class "SUPAPolicyConcreteMetadata"

   This is an optional concrete class. It defines an object that will
   be wrapped by concrete instances of the SUPAPolicyMetadataDecorator
   class. It can be viewed as a "carrier" for metadata that will be
   attached to a subclass of SUPAPolicyObject. Since the decorator
   pattern is used, any number of concrete subclasses of the
   SUPAPolicyMetadataDecorator class can wrap an instance of the
   SUPAPolicyConcreteMetadata class.

   It is recommended that this class be used as part of a conformant
   implementation.

5.17.1. SUPAPolicyConcreteMetadata Attributes

   Currently, two attributes are defined for the
   SUPAPolicyConcreteMetadata class, and are described in the
   following subsections.

5.17.1.1. The Attribute "supaPolMDValidPeriodEnd"

   This is an optional attribute. Its data type should be able to
   express a date and a time. This attribute defines the ending
   date and time that this Metadata object is valid for.

5.17.1.2. The Attribute "supaPolMDValidPeriodStart"

   This is an optional attribute. Its data type should be able to
   express a date and a time. This attribute defines the starting
   date and time that this Metadata object is valid for.

5.17.2. SUPAPolicyConcreteMetadata Relationships

   This class inherits the relationships of the SUPAPolicyMetadata
   class; see section 5.16.2. It can also be used by subclasses of
   the SUPAPolicyMetadataDecorator class, and hence, can participate
   in the SUPAHasMetadataDecorator aggregation; see section 5.18.2.

5.18. The Abstract Class "SUPAPolicyMetadataDecorator"

   This is an optional class, and is used to implement the decorator
   pattern (see section 5.7.1.) for metadata objects. This pattern
   enables all or part of one or more SUPAPolicyMetadataDecorator
   subclasses to "wrap" a SUPAPolicyConcreteMetadata object instance.

   It is recommended that this class be used as part of a conformant
   implementation.

5.18.1. SUPAPolicyMetadataDecorator Attributes

   Currently, no attributes are defined for the
   SUPAPolicyMetadataDecorator class.

5.18.2. SUPAPolicyMetadataDecorator Relationships

   This class inherits the relationships of the SUPAPolicyMetadata
   class; see section 5.16.2. It also defines a single aggregation,
   SUPAHasMetadataDecorator, which is used to implement the decorator
   pattern, as described in the following subsections.

5.18.2.1. The Aggregation "SUPAHasMetadataDecorator"

   This is an optional aggregation, and is part of a decorator
   pattern. It is used to enable a concrete instance of a
   SUPAPolicyMetadataDecorator to dynamically add behavior to a
   SUPAPolicyConcreteMetadata object instance. The semantics of this
   aggregation are defined by the SUPAHasMetadataDecoratorDetail
   association class.

   It is recommended that this aggregation be part of a conformant
   implementation.

   The multiplicity of this aggregation is 0..1 on the aggregate
   (SUPAPolicyMetadataDecorator) side and 1..n on the part
   (SUPAPolicyMetadata) side. This means that if this aggregation is
   defined, then at least one SUPAPolicyMetadata object (e.g., a
   concrete subclass of SUPAPolicyMetadataDecorator) must also be
   instantiated and wrapped by this SUPAPolicyConcreteMetadata object
   instance. The semantics of this aggregation are defined by the
   SUPAHasMetadataDecoratorDetail association class.

5.18.2.2. The Association Class "SUPAHasMetadataDecoratorDetail"

   This is an optional concrete association class, and defines the
   semantics of the SUPAHasMetadataDecorator aggregation. The purpose
   of this class is to use the Decorator pattern to determine
   which SUPAPolicyMetadataDecorator object instances, if any, are
   required to augment the functionality of the
   SUPAPolicyConcreteMetadata object instance that is being used.

   It is recommended that this association class be part of a
   conformant implementation.

   Attributes for this association class will be defined in a future
   version of this document.

5.19. The Concrete Class "SUPAPolicyAccessMetadataDef"

   This is an optional concrete class that defines access control
   information, in the form of metadata, that can be added to a
   SUPAPolicyObject. This is done using the SUPAHasPolicyMetadata
   aggregation (see section 5.2.2.). This enables all or part of a
   standardized description and/or specification of access control
   for a given SUPAPolicyObject to be easily changed at runtime by
   wrapping an object instance of the SUPAPolicyConcreteMetadata
   class (or its subclass) with all or part of this object, and then
   adorning the SUPAPolicyObject with the SUPAPolicyConcreteMetadata
   object instance.

5.19.1. SUPAPolicyAccessMetadataDef Attributes

   Currently, the SUPAPolicyAccessMetadataDef class defines three
   attributes; these are described in the following subsections.

5.19.1.1. The Attribute "supaPolAccessPrivilegeDef"

   This is an optional non-negative enumerated integer attribute. It
   specifies the access privileges that external Applications have
   when interacting with a specific SUPAPolicyObject that is adorned
   with an instance of this SUPAPolicyAccessMetadataDef object. This
   enables the management system to control, in a consistent manner,
   the set of operations that external Applications have for
   SUPAPolicies and components of SUPAPolicies. Values include:

      0: undefined
      1: read only (for all policy components)
      2: read and write (for all policy components)
      3: privileges are specified by an external MAC model
      4: privileges are specified by an external DAC model
      5: privileges are specified by an external RBAC model
      6: privileges are specified by an external ABAC model
      7: privileges are specified by an external custom model

   Values 1 and 2 apply to ALL SUPAPolicyObject instances that are
   adorned with this SUPAPolicyConcreteMetadata object instance;
   these two settings are "all-or-nothing" settings, and are included
   for ease of use.

   Values 3-7 indicate that a formal external access control model is
   used. The name of this model, and its location, are specified in
   two other class attributes, called supaPolAccessPrivilegeModelName
   and supaPolAccessPrivilegeModelRef. MAC, DAC, RBAC, and ABAC
   (values 3-6 stand for Mandatory Access Control, Discretionary
   Access Control, Role-Based Access Control, and Attribute-Based
   Access Control, respectively. A value of 7 indicates that a formal
   external model that is not MAC, DAC, RBAC, or ABAC is used.

5.19.1.2. The Attribute "supaPolAccessPrivilegeModelName"

   This is an optional string attribute that contains the name of
   the access control model being used. If the value of the
   supaPolAccessPrivilegeDef is 0-2, then the value of this attribute
   is not applicable. Otherwise, the text in this class attribute
   should be interpreted according to the value of the
   supaPolAccessPrivilegeModelRef class attribute.

5.19.1.3. The Attribute "supaPolAccessPrivilegeModelRef"

   This is an optional non-negative enumerated integer attribute
   that defines the data type of the supaPolAccessPrivilegeModelName
   attribute. If the value of the supaPolAccessPrivilegeDef class
   attribute is 0-2, then the value of this attribute is not
   applicable. Otherwise, the value of this class attribute defines
   how to interpret the text in the supaPolAccessPrivilegeModelRef
   class attribute. Values include:

      0: Undefined
      1: URI
      2: GUID
      3: UUID
      4: FQDN
      5: FQPN

5.20. The Concrete Class "SUPAPolicyVersionMetadataDef"

   This is an optional concrete class that defines versioning
   information, in the form of metadata, that can be added to a
   SUPAPolicyObject. This enables all or part of a standardized
   description and/or specification of version information for a
   given SUPAPolicyObject to be easily changed at runtime by
   wrapping an object instance of the SUPAPolicyConcreteMetadata
   class (or its subclass) with all or part of this object.

5.20.1. SUPAPolicyVersionMetadataDef Attributes

   Version information is defined in a generic format based on the
   Semantic Versioning Specification [18] as follows:

      <major>.<minor>.<patch>[<pre-release>][<build-metadata>]

   where the first three components (major, minor, and patch) MUST be
   present, and the latter two components (pre-release and build-
   metadata) MAY be present. A version number MUST take the form
   <major>.<minor>.<patch>, where <major>, <minor>, and <patch> are
   each non-negative integers that MUST NOT contain leading zeros.
   In addition, the value of each of these three elements MUST
   increase numerically.

   In this approach:

      o supaVersionMajor denotes a new release; this number MUST be
        incremented when either changes are introduced that are not
        backwards-compatible, and/or new functionality not previously
        present is introduced

      o supaVersionMinor denotes a minor release; this number MUST be
        incremented when new features and/or bug fixes to a major
        release that are backwards-compatible are introduced, and/or
        if any features are marked as deprecated
      o supaVersionPatch denotes a version that consists ONLY of bug
        fixes, and MUST be incremented when these bug fixes are Not
        backwards-compatible

   When multiple versions exist, the following rules define their
   precedence:

   1. Precedence MUST be calculated by separating the version into
       major, minor, patch, and pre-release identifiers, in that
       order. Note that build-metadata is NOT used to calculated
       precedence.
   2. Precedence is determined by the first difference when
       comparing each of these identifiers, from left to right,
       as follows:
         a. Major, minor, and patch versions are always compared
             numerically (e.g., 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1)
         b. When major, minor, and patch are equal, a pre-release
             version has LOWER precedence than a normal version
             (e.g., 1.0.0-alpha < 1.0.0)
         c. Precedence for two pre-release versions with the same
             major, minor, and patch version MUST be determined by
             comparing each dot separated identifier from left to
             right until a difference is found as follows:
             - identifiers consisting only of digits are compared
               numerically and identifiers with letters and/or
               hyphens are compared lexically in ASCII sort order
             - Numeric identifiers always have lower precedence
               than non-numeric identifiers
             - A larger set of pre-release fields has a higher
               precedence than a smaller set, if all of the
               preceding identifiers are equal
     3. Example: 1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-alpha-beta
                 < 1.0.0-beta < 1.0.0-beta.2 < 1.0.0-rc.1 < 1.0.0.

   Currently, the SUPAPolicyVersionMetadataDef class defines five
   attributes; these are described in the following subsections.

5.20.1.1. The Attribute "supaVersionMajor"

   This is a mandatory string attribute, and contains a string
   representation of an integer that is greater than or equal to zero.
   It indicates that a significant increase in functionality is
   present in this version. It MAY also indicate that this version has
   changes that are NOT backwards-compatible; this MAY be denoted in
   the supaVersionBuildMetadata class attribute.

The special string "0.1.0" is for initial development that MUST
NOT be considered stable. Improvements to this initial version,
before they are released to the public, are denoted by incrementing
the minor and patch version numbers.

The major version X (i.e., X.y.z, where X > 0) MUST be incremented
if any backwards incompatible changes are introduced. It MAY
include minor and patch level changes. The minor and patch version
numbers MUST be reset to 0 when the major version number is
incremented.

5.20.1.2. The Attribute "supaVersionMinor"

This is a mandatory string attribute, and contains a string
representation of an integer that is greater than or equal to zero.
It indicates that this release contains a set of features and/or
bug fixes that MUST be backwards-compatible.

The minor version Y (i.e., x.Y.z, where x > 0) MUST be incremented
if new, backwards-compatible changes are introduced. It MUST be
incremented if any features are marked as deprecated. It MAY be
incremented if new functionality or improvements are introduced.
It MAY include patch level changes. The patch version number MUST
be reset to 0 when the minor version number is incremented.

5.20.1.3. The Attribute "supaVersionPatch"

This is a mandatory string attribute, and contains a string
representation of an integer that is greater than or equal to zero.
It indicates that this version contains ONLY bug fixes.

The patch version Z (i.e., x.y.Z, where x > 0) MUST be incremented
if new, backwards-compatible changes are introduced. A bug fix is
defined as an internal change that fixes incorrect behavior.

5.20.1.4. The Attribute "supaVersionPreRelease"

This is an optional string attribute, and contains a string
defining the pre-release version.

A pre-release version MAY be denoted by appending a hyphen and a
series of dot-separated identifiers immediately following the patch
version. Identifiers MUST comprise only ASCII alphanumerics and a
hyphen. Identifiers MUST NOT be empty. Numeric identifiers MUST NOT
include leading zeroes. Pre-release versions have a lower
precedence than the associated normal version. A pre-release version
indicates that the version is unstable and might not satisfy the
intended compatibility requirements as denoted by its associated
normal version. Examples include: 1.0.0-alpha,
1.0.0-alpha.1, 1.0.0-0.3.7, and 1.0.0-x.7.z.92.

5.20.1.5. The Attribute "supaVersionBuildMetadata"

   This is an optional string attribute, and contains a string defining
   the build metadata. Build metadata MAY be denoted by appending a
   plus sign and a series of dot-separated identifiers immediately
   following the patch or pre-release version. Identifiers MUST be
   made up of only ASCII alphanumerics and a hyphen. Identifiers
   MUST NOT be empty. Build metadata SHOULD be ignored when determining
   version precedence. Examples include: 1.0.0.-alpha+1,
   1.0.0+20130313144700, and 1.0.0-beta+exp.sha.5114f85.

6.   SUPA ECAPolicyRule Information Model

   This section defines the classes, attributes, and relationships
   of the SUPA ECAPolicyRule Information Model (EPRIM). Unless
   otherwise stated, all classes (and attributes) defined in this
   section were abstracted from DEN-ng [2], and a version of them are
   in the process of being added to [5].

6.1. Overview

   Conceptually, the EPRIM is a set of subclasses that specialize the
   concepts defined in the GPIM for representing the components of a
   Policy that uses ECA semantics. This is shown in Figure 23 (only
   new EPRIM subclasses and their GPIM superclasses are shown).

```
   (Class of another model that SUPA is integrating into)
      |
     +---SUPAPolicyObject (5.2)
            |
           +---SUPAPolicyStructure (5.3)
           |      |
           |     +---SUPAECAPolicyRule (6.4)
           |            |
           |           +---SUPAECAPolicyRuleAtomic (6.5)
           |            |
           |           +---SUPAECAPolicyRuleComposite (6.6)
           |
           +---SUPAPolicyComponentStructure (5.6)
                  |
                 +---SUPAPolicyClause (5.7)
                 |      |
                 |     +---SUPABooleanClause (6.7)
                 |            |
                 |           +---SUPAECAPolicyRuleAtomic (6.8)
                 |            |
                 |           +---SUPAECAPolicyRuleComposite (6.9)
                 |
                 +---SUPAPolicyComponentDecorator (5.9)
                        |
                       +---SUPAECAComponent(6.10)
                              |
                             +---SUPAPolicyEvent (6.11)
                              |
                             +---SUPAPolicyCondition (6.12)
                              |
                             +---SUPAPolicyAction (6.13)
```

                   Figure 23. The EPRIM Class Hierarchy

   Specifically, the EPRIM specializes the SUPAPolicyStructure class
   class to create a SUPAECAPolicyRule (see sections 6.4 - 6.6); it
   also specializes two subclasses of the SUPAPolicyComponentStructure
   class to create two new sets of policy components. These two
   SUPAPolicyComponentStructure subclasses are:

      o a new subclass of SUPAPolicyClause, called SUPABooleanClause
        (see sections 6.7 - 6.9), is defined for constructing Boolean
        clauses that are specific to the needs of ECA Policy Rules
      o a new subclass of SUPAPolicyComponentDecorator, called
        SUPAECAComponent (see sections 6.10 - 6.13), is defined for
        constructing reusable objects that represent Events,
        Conditions, and Actions

The EPRIM provides new functionality, based on the GPIM, by
extending the GPIM to define new classes and relationships. The
EPRIM does NOT define new classes that are not inherited from
existing GPIM classes. This ensures that the semantics of the GPIM
are not changed, even though new functionality (for ECA Policy
Rules and components) are being defined.

The overall strategy for refining the GPIM is as follows:

   o SUPAECAPolicyRule is defined as a subclass of the GPIM
     SUPAPolicyStructure class
   o A SUPAECAPolicyRule has event, condition, and action clauses
      o Conceptually, this can be viewed as three aggregations
        between the SUPAECAPolicyRule and each clause
      o Each aggregation uses an instance of a concrete subclass of
        SUPAPolicyClause; this can be a SUPABooleanClause
        (making it ECA-specific), a SUPAEncodedClause (making it
        generic in nature), or a new subclass of SUPAPolicyClause
      o Concrete subclasses of SUPAPolicyClause may be decorated
        with zero or more concrete subclasses of the
        SUPAPolicyComponentDecorator class
   o An optional set of GPIM SUPAPolicySource objects can be
     defined to represent the authoring of a SUPAECAPolicyRule
   o An optional set of GPIM SUPAPolicyTarget objects can be
     defined to represent the set of managed entities that will be
     affected by this SUPAECAPolicyRule
   o An optional set of SUPAPolicyMetadata can be defined for any
     of the objects that make up a SUPAECAPolicyRule, including
     any of its components

6.2. Constructing a SUPAECAPolicyRule

   There are several different ways to construct a SUPAECAPolicyRule;
   they differ in which set of components are used to define the
   content of the SUPAECAPolicyRule, and whether each component is
   decorated or not. The following are some examples of creating a
   SUPAECAPolicyRule:

   o Define three types of SUPABooleanClauses, one each for the
     event, condition, and action clauses that make up a
     SUPAECAPolicyRule
      o For one or more of the above clauses, associate an
        appropriate set of SUPAPolicyEvent, SUPAPolicyCondition, or
        SUPAPolicyAction objects, and complete the clause using an
        appropriate SUPAPolicyOperator and a corresponding
        SUPAPolicyValue or SUPAPolicyVariable
      o Note that compound Boolean clauses may be formed using
        one or more SUPABooleanClauseComposite objects with one or
        more SUPABooleanClauseAtomic objects

o Define a SUPAPolicyCollection component, which is used to aggregate a set of objects appropriate for a clause, and complete the clause using an appropriate SUPAPolicyOperator and a corresponding SUPAPolicyValue or SUPAPolicyVariable

o Create a new concrete subclass of SUPAPolicyComponentStructure (i.e., a sibling class of SUPAPolicyComponentDecorator and SUPAPolicyClause), and use this new subclass in a concrete subclass of SUPABooleanClause; note that this approach enables the new concrete subclass of SUPAPolicyComponentStructure to optionally be decorated as well

o Create a new subclass of SUPAPolicyComponentDecorator (e.g., a sibling of SUPAECAComponent) that provides ECA-specific functionality, and use that to decorate a SUPAPolicyClause

o Create a new concrete subclass of SUPAPolicyStructure that provides ECA-specific functionality, and define all or part of its content by aggregating a set of SUPAPolicyClauses

6.3. Working With SUPAECAPolicyRules

   A SUPAECAPolicyRule is a type of SUPAPolicy. It is a tuple that MUST have three clauses, defined as follows:

   o The event clause defines a Boolean expression that, if TRUE, triggers the evaluation of its condition clause (if the event clause is not TRUE, then no further action for this policy rule takes place).

   o The condition clause defines a Boolean expression that, if TRUE, enables the actions in the action clause to be executed (if the condition clause is not TRUE, then no further action for this policy rule takes place).

   o The action clause contains a set of actions (note that an action MAY invoke another SUPAECAPolicyRule; see section 6.13).

   Each of the above clauses can be a simple Boolean expression (of the form {variable operator value}, or a compound Boolean expression consisting of Boolean combinations of clauses. Compound Boolean expressions SHOULD be in CNF or DNF.

   Note that each of the above three clauses MAY have a set of SUPAPolicyMetadata objects that can constrain, or otherwise affect, how that clause is treated. For example, a set of SUPAPolicyMetadata MAY affect whether none, some, or all actions are executed, and what happens if an action fails.

   Each of the three clauses can be constructed from either a SUPAEncodedClause or a SUPABooleanClause. The advantage of using SUPAEncodedClauses is simplicity, as the content of the clause is encoded directly into the attributes of the SUPAEncodedClause. The advantage of using SUPABooleanClauses is reusability, since each term in each clause is potentially a reusable object.

   Since a SUPABooleanClause is a subclass of a SUPAPolicyClause
   (see Section 6.7), it can be decorated by one or more concrete
   subclasses of SUPAPolicyComponentDecorator. Therefore, a
   SUPAECAPolicyRule can be built entirely from objects defined in
   the GPIM and EPRIM, which facilitates the construction of
   SUPAPolicies by a machine.

   The relation between a SUPAECAPolicyRule and a SUPAPolicyClause
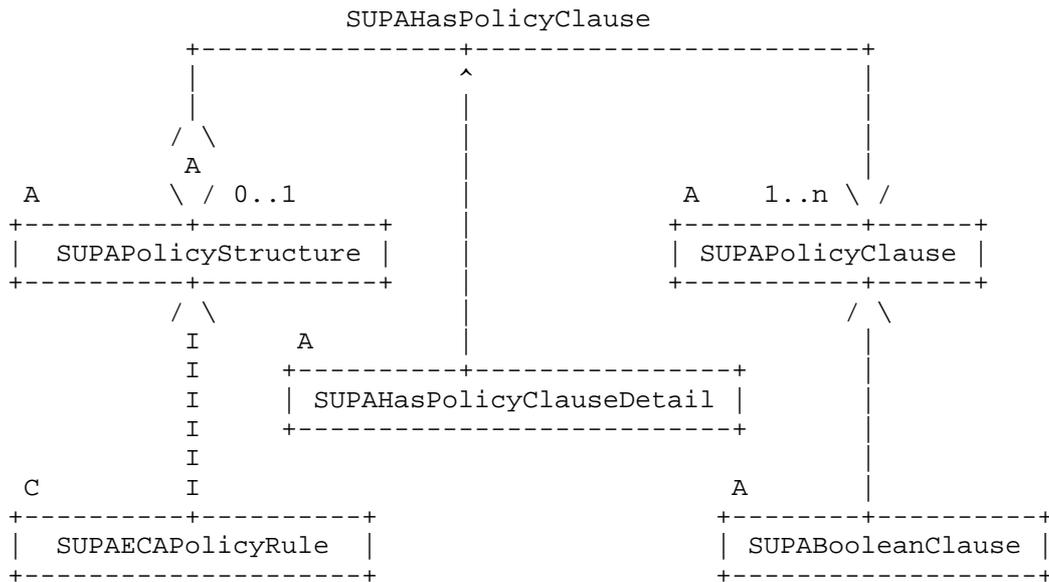   is shown in Figure 24, and is explained in further detail in
   Section 6.4.

```
                         SUPAHasPolicyClause
               +---------------+----------------------+
               |               ^                      |
               |               |                      |
              / \              |                      |
               A               |                      |
     A         \ / 0..1        |          A    1..n \ /
    +---------+-----------+     |         +----------+------+
    |   SUPAPolicyStructure |   |         |  SUPAPolicyClause |
    +---------+-----------+     |         +----------+------+
            / \                 |                 / \
             I       A          |                  |
             I     +---------+---------------+      |
             I     | SUPAHasPolicyClauseDetail |    |
             I     +-------------------------+      |
             I                                      |
     C       I                          A           |
    +---------+----------+              +--------+----------+
    |   SUPAECAPolicyRule  |            |  SUPABooleanClause |
    +-------------------+              +------------------+
```

              Figure 24. SUPAECAPolicyRule Clauses

   The SUPAHasPolicyClause aggregation is implemented using the
   SUPAHasPolicyClauseDetail association class. These were
   described in sections 5.4.2.1 and 5.4.2.2, respectively.

6.4. The Abstract Class "SUPAECAPolicyRule"

   This is a mandatory abstract class, which is a PolicyContainer
   that aggregates PolicyEvents, PolicyConditions, PolicyActions into
   a type of policy rule known as an Event-Condition-Action (ECA)
   policy rule. As previously explained, this has the following
   semantics:

      IF the event clause evaluates to TRUE
         IF the condition clause evaluates to TRUE
            THEN execute actions in the action clause
         ENDIF
      ENDIF

The event clause, condition clause, and action clause collectively form a three-tuple. Each clause MUST be defined by at least one SUPAPolicyClause (which MAY be decorated with other elements, as described in section 5.7).

Each of the three types of clauses is a 3-tuple of the form:

    {variable operator value}

Each of the three clauses MAY be combined with additional clauses using any combination of logical AND, OR, and NOT operators; this forms a "compound" Boolean clause. For example, if A, B, and C are three attributes in an event, then a valid event clause could be:

    (A AND B) OR C

Note that the above expression is in DNF; the equivalent CNF form is ((A OR C) AND (B OR C)). In either case, the output of all three clauses is either TRUE or FALSE; this facilitates combining and chaining SUPAECAPolicyRules.

An action clause MAY invoke a new SUPAECAPolicyRule; see section 6.13 for more details.

An ECAPolicyRule MAY be optionally augmented with PolicySources and/or PolicyTargets (see sections 5.16 and 5.17, respectively). In addition, all objects that make up a SUPAECAPolicyRule MAY have SUPAPolicyMetadata (see section 5.16) attached to them to further describe and/or specify behavior.

When defined in an information model, each of the event, condition, and action clauses MUST be represented as an aggregation between a SUPAECAPolicyRule (the aggregate) and a set of event, condition, or action objects (the components). However, a data model MAY map these definitions to a more efficient form (e.g., by flattening these three types of object instances, along with their respective aggregations, into a single object instance).

The composite pattern [3] is applied to the SUPAECAPolicyRule class, enabling its (concrete) subclasses to be used as either a stand-alone policy rule or as a hierarchy of policy rules. SUPAECAPolicyRuleComposite and SUPAECAPolicyRuleAtomic both inherit from SUPAECAPolicyRule. This means that they are both a type of SUPAECAPolicyRule. Hence, the HasSUPAECAPolicyRule aggregation enables a particular SUPAECAPolicyRuleComposite object to aggregate both SUPAECAPolicyRuleComposite as well as SUPAECAPolicyRuleAtomic objects. In contrast, a SUPAECAPolicyRuleAtomic can NOT aggregate either a SUPAECAPolicyRuleComposite or a SUPAECAPolicyRuleAtomic. SUPAECAPolicyRuleAtomic and SUPAECAPolicyRuleComposite are defined in sections 6.5 and 6.6, respectively. This is shown in Figure 25.

```
                          A
               1..n +------------------+
                 \|                    |
       +-------------- + SUPAECAPolicyRule |
       |            /|                    |
       |          +--------+----------+
       |                   / \
       | SUPAHasECAPolicyRule   I
       |                        I
       |                        I
       |                        I
       |      +--------------+------------+
       |      I                          I
      / \     I                          I
       A      I                          I
   C  \ / 0..1 I              C          I
  +-----+---------+----------+    +-----------+-----------+
  |SUPAECAPolicyRuleComposite|    |SUPAECAPolicyRuleAtomic|
  +------------------------+    +-----------------------+
```

Figure 25.   The Composite Pattern Applied to a SUPAECAPolicyRule

Note that the HasSUPAECAPolicyRule aggregation is defined by the
HasSUPAECAPolicyRuleDetail association class; both are defined
in sections 6.6.2 and 6.6.3, respectively.

6.4.1.   SUPAECAPolicyRule Attributes

   Currently, the SUPAECAPolicyRule defines two attributes, as
   described in the following subsections.

6.4.1.1.   The Attribute "supaECAPolicyRulePriority"

   This is a mandatory non-negative integer attribute that defines
   the priority of this particular SUPAECAPolicyRule. A larger value
   indicates a higher priority. A default value of 0 MAY be assigned.

   Priority is used primarily for 2 reasons: (1) to resolve conflicts
   among policy actions (e.g., given a set of conflicting actions,
   which one will execute) and (2) to define the execution order of
   policy actions (e.g., when one action may depend on the output of
   one or more previous actions).

6.4.1.2.   The Attribute "supaECAPolicyRuleStatus"

   This is an optional non-negative enumerated integer whose value
   defines the current status of this policy rule. Values include:

```
0:  In development, not ready to be deployed
1:  Ready to be deployed
2:  Deployed but not enabled
3:  Deployed and enabled, but not executed
4:  Executed without errors
5:  Executed with errors
6:  Aborted during execution
```

6.4.2. SUPAECAPolicyRule Relationships

   Currently, the SUPAECAPolicyRule does not define any
   relationships. It inherits all four relationships defined by
   the SUPAPolicyStructure class (see section 5.3.2.).

6.5. The Concrete Class "SUPAECAPolicyRuleAtomic"

   This is a mandatory concrete class. This class is a type of
   PolicyContainer, and represents a SUPAECAPolicyRule that can
   operate as a single, stand-alone, manageable object. Put another
   way, a SUPAECAPolicyRuleAtomic object can NOT be modeled as a set
   of hierarchical SUPAECAPolicyRule objects; if this is required,
   then a SUPAECAPolicyRuleComposite object should be used instead.

6.5.1. SUPAECAPolicyRuleAtomic Attributes

   Currently, the SUPAECAPolicyRuleAtomic class does not define any
   attributes.

6.5.2. SUPAECAPolicyRuleAtomic Relationships

   Currently, the SUPAECAPolicyRuleAtomic class does not define any
   relationships. It inherits all four relationships defined by the
   SUPAPolicyStructure class (see section 5.3.2.).

6.6. The Concrete Class "SUPAECAPolicyRuleComposite"

   This is a mandatory concrete class. This class is a type of
   PolicyContainer, and represents a SUPAECAPolicyRule as a hierarchy
   of SUPAPolicy objects, where the hierarchy contains instances of a
   SUPAECAPolicyRuleAtomic and/or SUPAECAPolicyRuleComposite objects.
   Each of the SUPAPolicy objects, including the outermost
   SUPAECAPolicyRuleComposite object, are separately manageable. More
   importantly, each SUPAECAPolicyRuleComposite object represents an
   aggregated object that is itself manageable.

6.6.1. SUPAECAPolicyRuleComposite Attributes

   Currently, the SUPAECAPolicyRuleComposite defines one attribute,
   as described in the following subsection.

6.6.1.1. The Attribute "supaECAEvalStrategy"

   This is a mandatory, non-zero, integer attribute that enumerates
   a set of allowable alternatives that define how the set of
   SUPAECAPolicyRule object instances in a SUPAECAPolicyRuleComposite
   object are evaluated. It is assumed that the event and condition
   clauses of the SUPAECAPolicyRules have evaluated to TRUE (e.g., the
   event has occurred and the conditions were met). Values include:

      0: undefined
      1: execute the first SUPAECAPolicyRule in the
          SUPAECAPolicyRuleComposite and then terminate
      2: execute only the highest priority SUPAECAPolicyRule(s) in
          the SUPAECAPolicyRuleComposite and then terminate
      3: execute all SUPAECAPolicyRules in prioritized order (if
          any) regardless of whether their SUPAPolicyActions
          succeed or fail
      4: execute all SUPAECAPolicyRules in prioritized order (if
          any) until at least one SUPAPolicyAction in a
          SUPAECAPolicyRule fails, and then terminate

   If the value of supaECAEvalStrategy is 3 or 4, then all
   SUPAECAPolicyRules that have a priority will be executed first
   (starting with the SUPAECAPolicyRule(s) that have the highest
   priority, and descending); all SUPAECAPolicyRule(s) that do not
   have a priority are then executed (in any order).

   Assume that the actions in a given SUPAECAPolicyRuleComposite
   are defined as follows

      SUPAECAPolicyRule A, priority 0
      SUPAECAPolicyRule B, priority 10
      SUPAECAPolicyRule C, priority 5
      SUPAECAPolicyRule D, priority 10
      SUPAECAPolicyRule E, priority 2

   Then, if the supaECAEvalStrategy attribute value equals:

      0: an error is issued
      1: only SUPAECAPolicyRule A is executed
      2: only SUPAECAPolicyRules B and D are executed
      3: all SUPAECAPolicyRules are executed, regardless of any
         failures in their SUPAPolicyActions
      4: all SUPAECAPolicyRules are executed until a failure is
         detected, and then execution for all SUPAECAPolicyRules
         terminate

6.6.2. SUPAECAPolicyRuleComposite Relationships

   Currently, the SUPAECAPolicyRuleComposite defines a single
   aggregation between it and SUPAECAPolicyRule, as described below.

6.6.2.1. The Aggregation "SUPAHasECAPolicyRule"

   This is an optional aggregation that implements the composite
   pattern. The multiplicity of this aggregation is 0..1 on the
   aggregate (SUPAECAPolicyRuleComposite) side and 1..n on the part
   (SUPAECAPolicyRule) side. This means that if this aggregation
   is defined, then at least one SUPAECAPolicyRule object (which may
   be either an instance of a SUPAECAPolicyRuleAtomic or a
   SUPAECAPolicyRuleComposite class) must also be instantiated and
   aggregated by this particular SUPAECAPolicyRuleComposite object.
   The semantics of this aggregation are defined by the
   SUPAHasECAPolicyRuleDetail association class.

6.6.3. The Association Class "SUPAHasECAPolicyRuleDetail"

   This is an optional concrete association class, and defines the
   semantics of the SUPHasECAPolicyRule aggregation. This enables the
   attributes and relationships of the SUPAHasECAPolicyRuleDetail
   class to be used to constrain which SUPHasECAPolicyRule objects
   can be aggregated by this particular SUPAECAPolicyRuleComposite
   object instance.

6.6.3.1. The Attribute "supaECAPolicyIsDefault"

   This is an optional Boolean attribute. If the value of this
   attribute is true, then this SUPAECAPolicyRule is a default
   policy, and will be executed if no other SUPAECAPolicyRule
   in the SUPAECAPolicyRuleComposite container has been executed.
   This is a convenient way for error handling, though care should
   be taken to ensure that only one default policy rule is defined
   per SUPAECAPolicyRuleComposite container.

6.7. The Abstract Class "SUPABooleanClause"

   A SUPABooleanClause specializes a SUPAPolicyClause, and defines
   a Boolean expression consisting of a standard structure in the
   form of a SUPAPolicyVariable, a SUPAPolicyOperator, and a
   SUPAPolicyValue. For example, this enables the following Boolean
   clause to be defined:

      Foo >= Baz

   where 'Foo' is a PolicyVariable, '>=' is a PolicyOperator, and
   'Baz' is a PolicyValue.

   Note that in this approach, the SUPAPolicyVariable and
   SUPAPolicyValue terms are defined as an appropriate subclass of
   the SUPAPolicyComponentDecorator class; it is assumed that the
   SUPAPolicyOperator is an instance of the SUPAPolicyOperator class.

   This enables the EPRIM, in conjunction with the GPIM, to be used
   as a reusable class library. This encourages interoperability,
   since each element of the clause is itself an object defined by
   the SUPA object hierarchy.

   An entire SUPABooleanClause may be negated by setting the
   supaBoolClauseIsNegated class attribute of the SUPABooleanClause
   class to TRUE. Individual terms of a Boolean clause can be negated
   by using the supaTermIsNegated Boolean attribute in the
   SUPAPolicyTerm class (see section 5.10).

   A PolicyClause is in Conjunctive Normal Form (CNF) if it is a
   sequence of logically ANDed terms, where each term is a sequence
   of logically ORed terms.

   A PolicyClause is in Disjunctive Normal Form (DNF) if it is a
   sequence of logically ORed terms, where each term is a sequence
   of logically ANDed terms.

   The construction of more complex clauses, which consist of a set
   of simple clauses in CNF or DNF (as shown in the above example),
   is provided by using the composite pattern [3] to construct two
   concrete subclasses of the abstract SUPABooleanClause class. These
   are called SUPABooleanClauseAtomic and SUPABooleanClauseComposite,
   and are defined in sections 6.8 and 6.9, respectively. This
   enables instances of either a SUPABooleanClauseAtomic and/or a
   SUPABooleanClauseComposite to be aggregated into a
   SUPABooleanClauseComposite object.

6.7.1. SUPABooleanClause Attributes

   The SUPABooleanClause class currently defines one attribute,
   which are defined in the following subsections.

6.7.1.1. The Attribute "supaBoolClauseIsNegated"

   This is a mandatory Boolean attribute. If the value of this
   attribute is TRUE, then this (entire) SUPABooleanClause is
   negated. Note that the supaPolTermIsNegated class attribute of
   the SUPAPolicyTerm class is used to negate a single term.

6.7.2. SUPABooleanClause Relationships

   Currently, no relationships are defined for the SUPABooleanClause
   class. It inherits the relationships of SUPAPolicyClause (see
   section 5.5.).

6.8. The Concrete Class "SUPABooleanClauseAtomic"

   This is a mandatory concrete class that represents a
   SUPABooleanClause that can operate as a single, stand-alone,
   manageable object. A SUPABooleanClauseAtomic object can NOT be
   modeled as a set of hierarchical clauses; if this functionality is
   required, then a SUPABooleanClauseComposite object must be used.
   Examples of Boolean clauses that could be contained in a
   SUPABooleanClauseAtomic include P, NOT P, and (P OR Q), where P
   and Q are literals (e.g., a variable name that can be either true
   or false, or a formula that evaluates to a literal). Examples of
   Boolean clauses that are NOT in CNF are NOT(P AND Q),
   (P AND Q) OR R, and P AND (Q OR (R AND S)); their CNF equivalent
   forms are NOT P AND NOT Q, (P AND R) OR (Q AND R), and
   P AND (Q OR S) AND (Q OR S), respectively.

6.8.1. SUPABooleanClauseAtomic Attributes

   No attributes are currently defined for the
   SUPABooleanClauseAtomic class.

6.8.2. SUPABooleanClauseAtomic Relationships

   Currently, no relationships are defined for the
   SUPABooleanClauseAtomic class. It inherits the relationships of
   SUPAPolicyClause (see section 5.5.).

6.9. The Concrete Class "SUPABooleanClauseComposite"

   This is a mandatory concrete class that represents a
   SUPABooleanClause that can operate as a hierarchy of PolicyClause
   objects, where the hierarchy contains instances of
   SUPABooleanClauseAtomic and/or SUPABooleanClauseComposite
   objects. Each of the SUPABooleanClauseAtomic and
   SUPABooleanClauseComposite objects, including the outermost
   SUPABooleanClauseComposite object, are separately manageable.

   More importantly, each SUPAECAPolicyRuleComposite object
   represents an aggregated object that is itself manageable.
   Examples of Boolean clauses that could be contained in a
   SUPABooleanClauseAtomic include ((P OR Q) AND R), and
   ((NOT P OR Q) AND (R OR NOT S) AND T), where P, Q, R, S, and
   T are literals.

6.9.1. SUPABooleanClauseComposite Attributes

   Two attributes are currently defined for the
   SUPABooleanClauseComposite class, and are described in the
   following subsections.

6.9.1.1. The Attribute "supaBoolClauseBindValue"

   This is a mandatory non-zero integer attribute, and defines the
   order in which terms bind to a clause. For example, the Boolean
   expression "((A AND B) OR (C AND NOT (D OR E)))" has the following
   binding order: terms A and B have a bind value of 1; term C has a
   binding value of 2, and terms D and E have a binding value of 3.

6.9.1.2. The Attribute "supaBoolClauseIsCNF"

   This is an optional Boolean attribute. If the value of this
   attribute is TRUE, then this SUPABooleanClauseComposite is in CNF
   form. Otherwise, it is in DNF form.

6.9.2. SUPABooleanClauseComposite Relationships

   Currently, the SUPABooleanClauseComposite class defined a single
   aggregation, which is described in the subsections below.

6.9.2.1. The Aggregation "SUPAHasBooleanClause"

   This is a mandatory aggregation that defines the set of
   SUPABooleanClause objects that are aggregated by this
   SUPABooleanClauseComposite object.

   The multiplicity of this relationship is 0..1 on the aggregate
   (SUPABooleanClauseComposite) side, and 1..n on the part
   (SUPABooleanClause) side. This means that one or more
   SUPABooleanClauses are aggregated and used to define this
   SUPABooleanClauseComposite object. The 0..1 cardinality on the
   SUPABooleanClauseComposite side is necessary to enable
   SUPABooleanClauses to exist (e.g., in a PolicyRepository) before
   they are used by a SUPABooleanClauseComposite. The semantics of
   this aggregation is defined by the SUPAHasBooleanClauseDetail
   association class.

6.9.3. The Association Class "SUPAHasBooleanClauseDetail"

   This is a mandatory concrete association class that defines the
   semantics of the SUPAHasBooleanClause aggregation. This enables the
   attributes and relationships of the SUPAHasBooleanClauseDetail
   class to be used to constrain which SUPABooleanClause objects
   can be aggregated by this particular SUPABooleanClauseComposite
   object instance.

6.9.3.1. SUPAHasBooleanClauseDetail Attributes

   The SUPAHasBooleanClauseDetail class currently does not define
   any attributes.

6.10. The Abstract Class "SUPAECAComponent"

   This is a mandatory abstract class that defines three concrete
   subclasses, one each to represent the concepts of reusable events,
   conditions, and actions. They are called SUPAPolicyEvent,
   SUPAPolicyCondition, and SUPAPolicyAction, respectively.

   SUPAECAComponents provide two different ways to construct
   SUPAPolicyClauses. The first is for the SUPAECAComponent to be
   used as either a SUPAPolicyVariable or a SUPAPolicyValue, and the
   second is for the SUPAECAComponent to contain the entire clause
   text.

   For example, suppose it is desired to define a policy condition
   clause with the text 'queueDepth > 10'. The two approaches could
   satisfy this as follows:

   Approach #1 (canonical form):
      SUPAPolicyCondition.supaPolicyConditionData contains the text
        'queueDepth'
      SUPAPolicyOperator.supaPolOpType is set to '1' (greater than)
      SUPAPolicyValue.supaPolValContent is set to '10'

   Approach #2 (SUPAECAComponent represents the entire clause):
      SUPAPolicyCondition.supaPolicyConditionData contains the text
        'queueDepth > 10'

   The class attribute supaECACompIsTerm, defined in subsection
   6.10.1.1, is used to identify which of these two approaches is
   used by an object instance of this class.

6.10.1. SUPAECAComponent Attributes

   A single attribute is currently defined for this class, and is
   described in the following subsection.

6.10.1.1. The Attribute supaECACompIsTerm

   This is an optional Boolean attribute. If the value of this
   attribute is TRUE, then this SUPAECAComponent is used as the value
   of a SUPAPolicyTerm to construct a SUPAPolicyClause (this is
   approach #1 in section 6.10 above). If the value of this attribute
   is FALSE, then this SUPAECAComponent contains the text of the
   entire corresponding SUPAPolicyClause (this is approach #2 in
   section 6.10 above).

6.10.2. SUPAECAComponent Relationships

   No relationships are currently defined for this class.

6.11. The Concrete Class "SUPAPolicyEvent"

   This is a mandatory concrete class that represents the concept of
   an Event that is applicable to a policy management system. Such
   an Event is defined as anything of importance to the management
   system (e.g., a change in the system being managed and/or its
   environment) occuring on a time-axis (as defined in [19]).
   SUPAPolicyEvents can be used as part of a SUPAPolicyClause; this
   is done by specifying the attribute name and value of an Event in
   the supaPolicyEventData attribute of the SUPAPolicyEvent. This
   enables event attributes to be used as part of a SUPAPolicyClause.

   Note: this class does NOT model the "raw" occurrences of Events.
   Rather, it represents the concept of an Event object whose class
   attributes describe pertinent attributes that can trigger the
   evaluation of a SUPAECAPolicyRule.

6.11.1. SUPAPolicyEvent Attributes

   Currently, five attributes are defined for the SUPAPolicyEvent
   class, which are described in the following subsections.

6.11.1.1. The Attribute "supaPolicyEventIsPreProcessed"

   This is an optional Boolean attribute. If the value of this
   attribute is TRUE, then this SUPAPolicyEvent has been pre-
   processed by an external entity, such as an Event Service Bus,
   before it was received by the Policy Management System.

6.11.1.2. The Attribute "supaPolicyEventIsSynthetic"

   This is an optional Boolean attribute. If the value of this
   attribute is TRUE, then this SUPAPolicyEvent has been produced by
   the Policy Management System. If the value of this attribute is
   FALSE, then this SUPAPolicyEvent has been produced by an entity
   in the system being managed.

6.11.1.3. The Attribute "supaPolicyEventTopic[0..n]"

   This is a mandatory array of string attributes, and contains the
   subject that this PolicyEvent describes.
   Note: [0..n] means that this is a multi-valued property that has
   zero or more attributes.

6.11.1.4. The Attribute "supaPolicyEventEncoding"

   This is a mandatory non-zero enumerated integer attribute, and
   defines how to interpret the supaPolicyEventData class attribute.
   These two attributes form a tuple, and together enable a machine
   to understand the syntax and value of the data carried by the
   object instance of this class. Values include:

```
     0:  Undefined
     1:  String
     2:  Integer
     3:  Boolean
     4:  Floating Point
     5:  DateTime
```

6.11.1.5. The Attribute "supaPolicyEventData[1..n]"

   This is a mandatory attribute that defines an array of strings.
   Each string in the array represents an attribute name and value
   of an Event object. The format of each string is defined as
   name:value. The 'name' part is the name of the SUPAPolicyEvent
   attribute, and the 'value' part is the value of that attribute.
   Note: [1..n] means that this is a multi-valued property that has
   at least one (and possibly more) attributes. For example, if
   this value of this attribute is:

       {(startTime:0800), (endTime:1700), (date:2016-05-11),
        (timeZone:-08:00)}

   then this attribute contains four properties, called startTime,
   endTime, date, and timeZone whose values are 0800, 1700, May 11
   2016, and Pacific Standard Time, respectively.

   Note that the supaPolicyEventEncoding class attribute defines how
   to interpret the value portion of this attribute.

   This attribute works with another class attribute, called
   supaPolicyEventEncoding, which defines how to interpret this
   attribute. These two attributes form a tuple, and together enable
   a machine to understand the syntax and value of the data carried
   by the object instance of this class.

6.11.2. SUPAPolicyEvent Relationships

   No relationships are currently defined for this class. It inherits
   the relationships defined by the SUPAPolicyComponentDecorator (see
   section 5.7.3.).

6.12. The Concrete Class "SUPAPolicyCondition"

   This is a mandatory concrete class that represents the concept of
   an Condition that will determine whether or not the set of Actions
   in the SUPAECAPolicyRule to which it belongs are executed or not.
   SUPAPolicyConditions can be used as part of a SUPAPolicyClause
   (e.g., var = SUPAPolicyCondition.supaPolicyConditionData) or as a
   stand-alone SUPAPolicyClause (e.g., the supaPolicyConditionData
   attribute contains text that defines the entire condition clause).

6.12.1. SUPAPolicyCondition Attributes

   Currently, two attributes are defined for the SUPAPolicyCondition
   class, which are described in the following subsections.

6.12.1.1. The Attribute "supaPolicyConditionData[1..n]"

   This is a mandatory array of string attributes that contains the
   content of this SUPAPolicyCondition object.
   Note: [1..n] means that this is a multi-valued property that has
   at least one (and possibly more) attributes.

   This attribute works with another class attribute, called
   supaPolicyConditionEncoding, which defines how to interpret this
   attribute. These two attributes form a tuple, and together enable
   a machine to understand the syntax and value of the data carried
   by the object instance of this class.

6.12.1.2. The Attribute "supaPolicyConditionEncoding"

   This is a mandatory non-zero enumerated integer attribute, and
   defines the data type of the supaPolicyConditionData attribute.
   These two attributes form a tuple, and together enable a machine
   to understand the syntax and value of the content of this
   SUPAPolicyCondition object. Values include:

      0: undefined
      1: String
      2: OCL 2.x
      3: OCL 1.x
      4: QVT 1.2 - Relations Language
      5: QVT 1.2 - Operational language
      6: Alloy
      7: English text

   Enumerations 1-3 are dedicated to OCL (with OCL 2.4 being the
   latest version as of this writing). QVT defines a set of languages
   (the two most powerful and useful are defined by enumerations 4
   and 5). Alloy is a language for describing constraints, and uses a
   SAT solver to guarantee correctness. Note that enumeration 7
   (English text) is not recommended (since it is informal, and hence,
   not verifiable), but included for completeness.

6.12.2. SUPAPolicyEvent Relationships

   No relationships are currently defined for this class. It inherits
   the relationships defined by the SUPAPolicyComponentDecorator (see
   section 5.7.3.).

6.13. The Concrete Class "SUPAPolicyAction"

   This is a mandatory concrete class that represents the concept of
   an Action, which is a part of a SUPAECAPolicyRule, which may be
   executed when both the event and the condition clauses of its
   owning SUPAECAPolicyRule evaluate to true. The execution of this
   action is determined by its SUPAECAPolicyRule container, and any
   applicable SUPAPolicyMetadata objects. SUPAPolicyActions can be
   used in three different ways:

      o as part of a SUPAPolicyClause (e.g., var =
        SUPAPolicyAction.supaPolicyActionData)
      o as a stand-alone SUPAPolicyClause (e.g., the
        supaPolicyActionData attribute contains text that defines
        the entire action clause)
      o to invoke a SUPAECAPolicyRule

   In the third case, the execution semantics SHOULD be to suspend
   the current execution of the set of SUPAPolicyActions that are
   executing, transfer execution control to the invoked
   SUPAECAPolicyRule, and resume the execution of the original set
   of SUPAPolicyActions when the invoked SUPAECAPolicyRule has
   finished execution.

6.13.1. SUPAPolicyAction Attributes

   Currently, two attributes are defined for the SUPAPolicyCondition
   class, which are described in the following subsections.

6.13.1.1. The Attribute "supaPolicyActionData[1..n]"

   This is a mandatory array of string attributes that contains the
   content of this SUPAPolicyAction object. This attribute works with
   another class attribute, called supaPolicyActionEncoding, which
   defines how to interpret this attribute. These two attributes form
   a tuple, and together enable a machine to understand the syntax
   and value of the data carried by the object instance of this class.
   Note: [1..n] means that this is a multi-valued property that has
   at least one (and possibly more) attributes.

   Since this attribute could represent a term in a SUPAPolicyClause
   (e.g., var = SUPAPolicyAction.supaPolicyActionData), a complete
   SUPAPolicyClause (e.g., the supaPolicyActionData attribute
   contains text that defines the entire action clause), or the
   name of a SUPAECAPolicyRule to invoke, each element in the string
   array is prepended with one of the following strings:

      o 't:' (or 'term:'), to denote a term in a SUPAPolicyClause
      o 'c:' (or 'clause:'), to denote an entire SUPAPolicyClause
      o 'r:' (or 'rule:'), to invoke a SUPAECAPolicyRule

6.13.1.2. The Attribute "supaPolicyActionEncoding"

   This is a mandatory non-zero enumerated integer attribute, and
   defines the data type of the supaPolicyActionData attribute. This
   attribute works with another class attribute, called
   supaPolicyActionData, which contains the content of the action.
   These two attributes form a tuple, and together enable a machine
   to understand the syntax and value of the content of this
   SUPAPolicyAction object. Values include:

       0: undefined
       1: GUID
       2: UUID
       3: URI
       4: FQDN
       5: String
       6: OCL 2.x
       7: OCL 1.x
       8: QVT 1.2 - Relations Language
       9: QVT 1.2 - Operational language
      10: Alloy

6.13.2. SUPAPolicyAction Relationships

   No relationships are currently defined for this class. It inherits
   the relationships defined by the SUPAPolicyComponentDecorator (see
   section 5.7.3.).

   Enumerations 1-4 are used to provide a reference to an action
   object. Enumerations 5-10 are used to express the action to
   perform as a string.

7. Examples

   This will be defined in the next version of this document.

8. Security Considerations

   This will be defined in the next version of this document.

9. IANA Considerations

   This document has no actions for IANA.

10. Contributors

   The following people contributed to creating this document, and are
   listed in alphabetical order:

      Jason Coleman

11. Acknowledgments

   This document has benefited from reviews, suggestions, comments
   and proposed text provided by the following members, listed in
   alphabetical order: Andy Bierman, Bert Wijnen, Bob Natale, Dave
   Hood, Fred Feisullin, Georgios Karagiannis, Liu (Will) Shucheng,
   Marie-Jose Montpetit.

12. References

   This section defines normative and informative references for this
   document.

12.1. Normative References

   [RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate
                Requirement Levels", BCP 14, RFC 2119, March 1997.

   [RFC6020]   Bjorklund, M., "YANG - A Data Modeling Language for
                the Network Configuration Protocol (NETCONF)",
                RFC 6020, October 2010.

   [RFC6991]   Schoenwaelder, J., "Common YANG Data Types", RFC 6991,
                July 2013.

12.2. Informative References

   [RFC3060]   Moore, B., Ellesson, E., Strassner, J., Westerinen,
                A., "Policy Core Information Model -- Version 1
                Specification", RFC 3060, February 2001

   [RFC3198]   Westerinen, A., Schnizlein, J., Strassner, J.,
                Scherling, M., Quinn, B., Herzog, S., Huynh, A.,
                Carlson, M., Perry, J., Waldbusser, S., "Terminology
                for Policy-Based Management", RFC 3198, November, 2001

   [RFC3460]   Moore, B., ed., "Policy Core Information Model (PCIM)
                Extensions, RFC 3460, January 2003

   [1]   Strassner, J., "Policy-Based Network Management",
                Morgan Kaufman, ISBN 978-1558608597, Sep 2003

   [2]   Strassner, J., ed., "The DEN-ng Information Model",
                add stable URI

   [3]   Riehle, D., "Composite Design Patterns", Proceedings
                of the 1997 Conference on Object-Oriented Programming
                Systems, Languages and Applications (OOPSLA '97).
                ACM Press, 1997, Page 218-228

   [4]   DMTF, CIM Schema, v2.44,
                http://dmtf.org/standards/cim/cim_schema_v2440

   [5]   Strassner, J., ed., "ZOOM Policy Architecture and
                Information Model Snapshot", TR235, part of the
                TM Forum ZOOM project, October 26, 2014

   [6]   TM Forum, "Information Framework (SID), GB922 and
                associated Addenda, v14.5,
                https://www.tmforum.org/information-framework-sid/

   [7]   Liskov, B.H., Wing, J.M., "A Behavioral Notion of
                subtyping", ACM Transactions on Programming languages
                and Systems 16 (6): 1811 - 1841, 1994

   [8]   Klyus, M., Strassner, J., Liu, W., Karagiannis, G.,
                Bi, J., "SUPA Value Proposition",
                draft-klyus-supa-value-proposition-00, March 21, 2016

   [9]   ISO/IEC 10746-3 (also ITU-T Rec X.903), "Reference
                Model Open Distributed Processing Architecture",
                April 20, 2010

   [10]   Davy, S., Jennings, B., Strassner, J., "The Policy
                Continuum - A Formal Model", Proc. of the 2nd Intl.
                IEEE Workshop on Modeling Autonomic Communication
                Environments (MACE), Multicon Lecture Notes, No. 6,
                Multicon, Berlin, 2007, pages 65-78

   [11]   Gamma, E., Helm, R., Johnson, R., Vlissides, J.,
                "Design Patterns - Elements of Reusable Object-Oriented
                Software", Addison-Wesley, 1994, ISBN 0-201-63361-2

   [12]   Strassner, J., de Souza, J.N., Raymer, D., Samudrala,
                S., Davy, S., Barrett, K., "The Design of a Novel
                Context-Aware Policy Model to Support Machine-Based
                Learning and Reasoning", Journal of Cluster Computing,
                Vol 12, Issue 1, pages 17-43, March, 2009

   [13]   Liskov, B.H., Wing, J.M., "A Behavioral Notion of
                subtyping", ACM Transactions on Programming languages
                and Systems, 16 (6): 1811 - 1841, 1994

   [14]   Martin, R.C., "Agile Software Development, Principles,
                 Patterns, and Practices", Prentice-Hall, 2002,
                 ISBN: 0-13-597444-5

   [15]   Halpern, J., Strassner, J., "Generic Policy Data Model
                 for Simplified Use of Policy Abstractions (SUPA)"
                 draft-ietf-supa-generic-policy-data-model-00,
                 July 13, 2016

   [16]   Wang, Y., Esposito, F., Matta, I., Day, J., "RINA: An
                 Architecture for Policy-based Dynamic Service
                 Management", Tech Report BUCS-TR-2013-014, 2013

   [17]   Meyer, B., "Object-Oriented Software Construction",
                 Prentice Hall, second edition, 1997 ISBN 0-13-629155-4

   [18]   http://semver.org/

   [19]   ISO/IEC:2004(E), "Data elements and interchange
                 formats -- Information interchange -- Representation
                 of dates and times", 2004

Authors' Addresses

   John Strassner
   Huawei Technologies
   2330 Central Expressway
   Santa Clara, CA 95138 USA
   Email: john.sc.strassner@huawei.com

   Joel Halpern
   Ericsson
   P. O. Box 6049
   Leesburg, VA 20178
   Email: joel.halpern@ericsson.com

   Sven van der Meer
   LM Ericsson Ltd.
   Ericsson Software Campus
   Garrycastle
   Athlone
   N37 PV44
   Ireland
   Email: sven.van.der.meer@ericsson.com

Appendix A.   Brief Analyses of Previous Policy Work

   This appendix describes of some of the important problems with
   previous IETF policy work., and describes the rationale for
   taking different design decisions in this document.

A.1.   PolicySetComponent vs. SUPAPolicyStructure

   The ability to define different types of policy rules is not
   present in [RFC3060] and [RFC3460], because both are based on [4],
   and this ability is not present in [4]. [RFC3060], [RFC3460], and
   [4] are all limited to CA (condition-action) policy rules. In
   addition, events are NOT defined. These limitations mean that
   RFC3060], [RFC3460], and [4] can only represent CA Policy Rules.

   In contrast, the original design goal of SUPA was to define a
   single class hierarchy that could represent different types of
   policies (e.g., imperative and declarative). Hence, it was
   decided to make SUPAPolicyStructure generic in nature, so that
   different types of policies could be defined as subclasses. This
   enables a single Policy Framework to support multiple types of
   policies.

A.2.   Flat Hierarchy vs. SUPAPolicyComponentStructure

   Figure 26 shows a portion of the class hierarchy of [RFC3460].

                           A
                      +--------+
                      | Policy |
                      +----+---+
                          / \
                           |
                           |
            +---------+------+------+-----------+----------+
            | | | | | |
       A | | A | | A |
   +-----+-----+ | +--------+--------+ | +------+-------+
   | PolicySet | | | PolicyCondition | | | PolicyAction |
   +-----+-----| | +--------+--------+ | +------+-------+
        / \ | / \ | / \
         | | | | |
        ... | ... | ...
              A | A |
         +-------+--------+   +-------+-----+
         | PolicyVariable | | PolicyValue |
         +-------+--------+   +-------+-----+
                / \   / \
                 | |
                ...   ...

            Figure 26.   Simplified Class Hierarcy of [RFC3460]

RFC3060], [RFC3460], and [4] defined PolicyConditions and
PolicyActions as subclasses of Policy (along with PolicySet,
which is the superclass of PolicyRules and PolicyGroups). This
means that there is no commonality between PolicyConditions and
PolicyActions, even though they are both PolicyRule components.
From an object-oriented point-of-view, this is incorrect, since a
PolicyRule aggregates both PolicyConditions and PolicyActions.

In addition, note that both PolicyVariables and PolicyValues are
siblings of PolicyRules, PolicyConditions, and PolicyActions. This
is incorrect for several reasons:

> o a PolicyRule cannot rectly contain PolicyVariables or
>   PolicyValues, so they shouldn't be at the same level of the
>   class hierarchy
> o both PolicyConditions and PolicyActions can contain
>   PolicyVariables and PolicyValues, which implies that both
>   PolicyVariables and PolicyValues should be lower in the
>   class hierarchy

Note that in the current version of [4], PolicyVariable and
PolicyValue are both deleted. There are other changes as well,
but they are beyond the scope of this Appendix.

The original design goal of SUPA was to define a single class
hierarchy that could represent different types of policies and
policy components. This cannot be accomplished in [RFC3460], since
there is no notion of a policy component (or alternatively,
PolicyCondition, PolicyAction, PolicyVariable, and PolicyValue are
all components at the same abstraction level, which is clearly not
correct). Hence, SUPA defined the SUPAPolicyComponentStructure
class to capture the concept of a reusable policy component.

In summary, SUPAPolicyStructure subclasses define the structure of
a policy in a common way, while SUPAPolicyComponentStructure
subclasses define the content that is contained in the structure
of a policy, also in a common way.

A.3. PolicyRules and PolicyGroups vs. SUPAPolicyRules

A PolicySetComponent is an aggregation, implemented as an
association class, that "collects instances of PolicySet
subclasses into coherent sets of Policies". This is a recursive
aggregation, with multiplicity 0..n - 0..n, on the PolicySet
class.

Since this is a recursive aggregation, it means that a PolicySet
can aggregate zero or more PolicySets. This is under-specified,
and can be interpreted in one of two ways:

   1. A PolicySet subclass can aggregate any PolicySet subclass
      (PolicyRules can aggregate PolicyRules and PolicyGroups, and
      vice-versa)
   2. PolicyRules can aggregate PolicyRules, and PolicyGroups can
      aggregate PolicyGroups, but neither class can aggregate the
      other type of class

Both interpretations are ill-suited for policy-based management.
The problem with the first is that if PolicyGroup is the mechanism
for grouping, why can a PolicyRule aggregate a PolicyGroup? This
implies that PolicyGroups are not needed. The problem with the
second is that PolicyGroups cannot aggregate PolicyRules (which
again implies that PolicyGroups are not needed).

Furthermore, there are no mechanisms defined in the [RFC3460]
model to prevent loops of PolicyRules. This is a problem, because
EVERY PolicyRule and PolicyGroup inherits this recursive
aggregation.

This is why this document uses the composite pattern. First, this
pattern clearly shows what object is aggregating what other
object (i.e., a SUPAECAPolicyRuleAtomic cannot aggregate a
SUPAECAPolicyRuleComposite). Second, it does not allow a
SUPAECAPolicyRule to be aggregated by another SUPAECAPolicyRule
(this is discussed more in the following subsection).

A.3.1. Sub-rules

Sub-rules (also called nested policy rules) enable a policy rule to
be contained within another policy rule. These have very complex
semantics, are very hard to debug, and provide limited value. They
also require a complex set of aggregations (see section A.4.).

The main reason for defining sub-rules in [RFC3460] is to enable
"complex policy rules to be constructed from multiple simpler
policy rules". However, the composite pattern does this much more
efficiently than a simple recursive aggregation, and avoids the
ambiguous semantics of a recursive aggregation. This latter point
is important, because if PolicyRule and/or PolicyGroup is
subclassed, then all subclasses still inherit this recursive
aggregation, along with its ambiguous semantics.

A.4. PolicyConditions and PolicyActions vs. SUPAECAComponent

There is no need to use the SimplePolicyCondition and
ComplexPolicyCondition objects defined in [RFC3460], since the
SUPAPolicyComponentStructure uses the decorator pattern (see
section 5.7) to provide more extensible types of conditions than is
possible with those classes. This also applies for the
SimplePolicyAction and the ComplexPolicyAction classes defined in
[RFC3460].

More importantly, this removes the need for a complex set of
aggregations (i.e., PolicyComponent, PolicySetComponent,
PolicyConditionStructure, PolicyConditionInPolicyRule,
PolicyConditionInPolicyCondition, PolicyActionStructure,
PolicyActionInPolicyRule, and PolicyActionInPolicyAction).
Instead, ANY SUPAECAComponent is defined as a decorator (i.e., a
subclass of SUPAPolicyComponentDecorator), and hence, Any
SUPAECAComponent is wrapped onto a concrete subclass of
SUPAPolicyClause using the SAME aggregation
(SUPAHasDecoratedPolicyComponent). This is a significantly simpler
design that is also more powerful.

A.5. The SUPAPolicyComponentDecorator Abstraction

One of the problems in building a policy model is the tendency to
have a multitude of classes, and hence object instances, to
represent different combinations of policy events, conditions, and
actions. This can lead to class and/or relationship explosion, as
is the case in [RFC3460], [4], and [6].

For example, [RFC3460] defines five subclasses of PolicyCondition:
PolicyTimePeriodCondition, VendorPolicyCondition,
SimplePolicyCondition, CompoundPolicyCondition, and
CompoundFilterCondition. Of these:

    o PolicyTimePeriodCondition is a data structure, not a class
    o VendorPolicyCondition represents a condition using two
      attributes that represent a multi-valued octet string
    o SimplePolicyCondition, CompoundPolicyCondition, and
      CompoundFilterCondition all have ambiguous semantics

SimplePolicyCondition represents an ordered 3-tuple, in the form
{variable, match, value}. However, the match operator is not
formally modeled. Specifically, "the 'match' relationship is to
be interpreted by analyzing the variable and value instances
associated with the simple condition". This becomes problematic
for several cases, such as shallow vs. deep object comparisons.
More importantly, this requires two separate aggregations
(PolicyVariableInSimplePolicyCondition and
PolicyValueInSimplePolicyCondition) to associate variables and
values to the SimplePolicyCondition, respectively. Since [RFC3460]
defines all relationships as classes, this means that the
expression "Foo > Bar" requires a total of FIVE objects (one each
for the variable and value, one for the SimplePolicyCondition, and
one each to associate the variable and value with the
SimplePolicyCondition).

This is exacerbated when SimplePolicyConditions are used to build
CompoundPolicyConditions. In addition to the above complexity
(which is required for each SimplePolicyCondition), a new
aggregation (PolicyConditionInPolicyCondition) is required to
aggregation PolicyConditions. Thus, the compound expression:
"((Foo > Bar) AND (Foo < Baz))" requires a total of THIRTEEN
objects (five for each of the terms being ANDed, plus one for the
CompoundPolicyCondition, and two to aggregate each term to the
CompoundPolicyCondition).

Note that in the above examples, the superclasses of each of the
relationships are omitted for clarity. In addition, [RFC3460] is
built using inheritance; this means that if a new function is
required, a new class must be built (e.g., CompoundFilterCondition
is a subclass, but all it adds is one attribute).

In constrast, the Decorator Pattern enables behavior to be
selectively added to an individual object, either statically or
dynamically, without having to build association classes. In
addition, the decorator pattern uses composition, instead of
inheritance, to avoid class explosion. This means that a new
variable, value, or even condition class can be defined at
runtime, and then all or part of that class can dynamically wrap
an existing object without need for recompilation and
redeployment.

A.6. The Abstract Class "SUPAPolicyClause"

This abstraction is missing in [RFC3060], [RFC3460], [4], and [6].
SUPAPolicyClause was abstracted from DEN-ng [2], and a version of
this class is in the process of being added to [5]. However, the
class and relationship design in [5] differs significantly from
the corresponding designs in this document.

SUPAPolicyClause further reinforces the different between a policy
rule and a component of a policy rule by abstracting the content
of a policy rule as a reusable object. This is fundamental for
enabling different types of policy rules (e.g., imperative and
declarative) to to be represented using the same constructs.

A.7. Problems with the RFC3460 Version of PolicyVariable

The following subsections define a brief, and incomplete, set of
problems with the implementation of [RFC3460] (note that [RFC3060
did not define variables, operators, and/or values).

A.7.1. Object Bloat

[RFC3460] used two different and complex mechanisms for providing
generic get and set expressions. PolicyVariables were subclassed
into two subclasses, even though they performed the same semantic
function. This causes additional problems:

> o PolicyExplicitVariables are for CIM compatibility; note that
>   the CIM does not contain either PolicyVariables or
>   PolicyValues ([4])
> o PolicyImplicitVariable subclasses do not define attributes;
>   rather, they are bound to an appropriate subclass of
>   PolicyValue using an association

Hence, defining a variable is relatively expensive in [RFC3460],
as in general, two objects and an association must be used. The
objects themselves do not define content; rather, their names are
used as a mechanism to identify an object to match. This means
that an entire object must be used (instead of, for example, an
attribute), which is wasteful. It also make it difficult to
adjust constraints at runtime, since the constraint is defined in
a class that is statically defined (and hence, requires
recompilation and possibly redeployment if it is changed).

A.7.2. Object Explosion

The above three problems lead to class explosion (recall that in
[RFC3060], [RFC3460], and [4], associations are implemented as
classes).

In contrast to this approach, the approach in this document keeps
 the idea of the class hierarchy for backwards compatibility, but
streamlines the implementation. Specifically:

1. The decorator pattern is an established and very used
   software pattern (it dates back to at least 1994 [11]).
2. The use of a single association class
   (i.e., SUPAHasDecoratedPolicyComponentDetail) can represent
   more constraints than is possible in the approaches of
   [RFC3460] and [4] in a much more flexible manner, due to its
   function as a decorator of other objects.
3. Note that there is no way to enforce the constraint matching
   in [RFC3460] and [6]; the burden is on the developer to
   check and see if the constraints specified in one class are
   honored in the other class.
4. If these constraints are not honored, there is no mechanism
   specified to define the clause as incorrectly formed.

A.7.3. Specification Ambiguities

There are a number of ambiguities in [RFC3460].

First, [RFC3460] says: "Variables are used for building individual
conditions". While this is true, variables can also be used for
building individual actions. This is reflected in the definition
for SUPAPolicyVariable.

Second, [RFC3460] says: "The variable specifies the property of a
flow or an event that should be matched when evaluating the
condition." While this is true, variables can be used to test many
other things than "just" a flow or an event. This is reflected in
the SUPAPolicyVariable definition.

Third, the [RFC3460] definition requires the use of associations
in order to properly constrain the variable (e.g., define its
data type, the range of its allowed values, etc.). This is both
costly and inefficient.

Fourth, [RFC3460] is tightly bound to the DMTF CIM schema [4].
The CIM is a data model (despite its name), because:

   o It uses keys and weak relationships, which are both concepts
     from relational algebra and thus, not technology-independent
   o It has its own proprietary modeling language
   o It contains a number of concepts that are not defined in UML
     (including overriding keys for subclasses)

Fifth, the class hierarchy has two needless classes, called
SUPAImplicitVariable and SUPAExplicitVariable. These classes do
not define any attributes or relationships, and hence, do not
add any semantics to the model.

Finally, in [RFC3460], defining constraints for a variable is
limited to associating the variable with a PolicyValue. This is
both cumbersome (because associations are costly; for example,
they equate to a join in a relational database management system),
and not scalable, because it is prone to proliferating PolicyValue
classes for every constraint (or range of constraints) that is
possible. Therefore, in SUPA, this mechanism is replaced with
using an association to an association class that defines
constraints in a much more general and powerful manner (i.e.,
the SUPAHasDecoratedPolicyComponentDetail class).

A.8. Problems with the RFC3460 Version of PolicyValue

   The following subsections define a brief, and incomplete, set of
   problems with the implementation of [RFC3460] (note that [RFC3060
   did not define variables, operators, and/or values).

A.8.1. Object Bloat

   [RFC3460] defined a set of 7 subclasses; three were specific to
   networking (i.e., IPv4 Address, IPv6 Address, MAC Address) and 4
   (PolicyStringValue, PolicyBitStringValue, PolicyIntegerValue, and
   PolicyBooleanValue) were generic in nature. However, each of these
   objects defined a single class attribute. This has the same two
   problems as with PolicyVariables (see section 5.9.1.1):

   1. Using an entire object to define a single attribute is very
      wasteful and expensive
   2. It also make it difficult to adjust constraints at runtime,
      since the constraint is defined in a class that is statically
      defined (and hence, requires recompilation and possibly
      redeployment if it is changed).

A.8.2. Object Explosion

   [RFC3460] definition requires the use of associations
   in order to properly constrain the variable (e.g., define its
   data type, the range of its allowed values, etc.). This is both
   costly and inefficient (recall that in [RFC3060], [RFC3460], and
   [4], associations are implemented as classes).

A.8.3. Lack of Constraints

   There is no generic facility for defining constraints for a
   PolicyValue. Therefore, there is no facility for being able to
   change such constraints dynamically at runtime.

A.8.4. Tightly Bound to the CIM Schema

   [RFC3460] is tightly bound to the DMTF CIM schema [4]. The CIM is
   a data model (despite its name), because:

      o It uses keys and weak relationships, which are both concepts
        from relational algebra and thus, not technology-independent
      o It has its own proprietary modeling language
      o It contains a number of concepts that are not defined in UML
        (including overriding keys for subclasses)

A.8.5. Specification Ambiguity

   [RFC3460] says: It is used for defining values and constants used
   in policy conditions". While this is true, variables can also be
   used for building individual actions. This is reflected in the
   SUPAPolicyVariable definition.

A.8.6. Lack of Symmetry

   Most good information models show symmetry between like components.
   [RFC3460] has no symmetry in how it defines variables and values.
   In contrast, this document recognizes that variables and values
   are just terms in a clause; hence, the only difference in the
   definition of the SUPAPolicyVariable and SUPAPolicyValue classes
   is that the content attribute in the former is a single string,
   whereas the content attribute in the latter is a string array.
   In particular, the semantics of both variables and values are
   defined using the decorator pattern, along with the attributes of
   the SUPAPolicyComponentDecorator and the
   SUPAHasDecoratedPolicyComponentDetail classes.

SUPA   W. Liu
Internet Draft   J. Strassner
Intended status: Informational   G. Karagiannis
Expires: January 2017   Huawei Technologies
                                                            M. Klyus
                                                          NetCracker
                                                              J. Bi
                                                Tsinghua University
                                                             C. Xie
                                                      China Telecom
                                                      July 22, 2016

                    SUPA policy-based management framework
              draft-liu-supa-policy-based-management-framework-02

Status of this Memo

Copyright Notice

Abstract

   Simplified Use of Policy Abstractions (SUPA) defines base YANG data
   models to encode policy, which will point to device-, technology-,
   and service-specific YANG models developed in other working groups.
   Policy rules within an operator's environment can be used to express
   high-level, possibly network-wide policies to a network management
   function (within a controller, an orchestrator, or a network element).
   The network management function can then control the configuration
   and/or monitoring of network elements and services. This document
   describes the SUPA basic framework, its elements and interfaces.

Table of Contents

1. Introduction

   The rapid growth in the variety and importance of traffic flowing
   over increasingly complex enterprise and service provider network
   architectures makes the task of network operations and management
   applications and deploying new services much more difficult. In
   addition, network operators want to deploy new services quickly and
   efficiently. Two possible mechanisms for dealing with this growing
   difficulty are the use of software abstractions to simplify the
   design and configuration of monitoring and control operations, and
   the use of programmatic control over the configuration and operation
   of such networks. Policy-based management can be used to combine
   these two mechanisms into an extensible framework.

Policy rules within an operator's environment can be used to express high-level, possibly network-wide policies to a network management function (within a controller, an orchestrator, or a network element). The network management function can then control the configuration and/or monitoring of network elements and services.

Simplified Use of Policy Abstractions (SUPA) will define a generic policy information model (GPIM) [SUPA-info-model] for use in network operations and management applications. The GPIM defines concepts and terminology needed by policy management indepednent of the form and content of the policy rule. The ECA Policy Rule Information Model (EPRIM) [SUPA-info-model] extends the GPIM to define how to build policy rules according to the event-condition-action paradigm.

Both the GPIM and the EPRIM are targeted at controlling the configuration and monitoring of network elements throughout the service development and deployment lifecycle. The GPIM and the EPRIM will both be translated into corresponding YANG [RFC6020][RFC6020bis] modules that define policy concepts, terminology, and rules in a generic and interoperable manner; additional YANG modules may also be defined from the GPIM and/or EPRIM to manage specific functions.

The key benefit of policy management is that it enables different network elements and services to be instructed to behave the same way, even if they are programmed differently. Management applications will benefit from using policy rules that enable scalable and consistent programmatic control over the configuration and monitoring of network elements and services.

2. Framework for Generic Policy-based Management

   This section briefly describes the design and operation of the SUPA policy-based management framework.

2.1. Overview

   Figure 1 shows a simplified functional architecture of how SUPA is used to define policies for creating network element configuration and monitoring snippets. SUPA uses the GPIM to define a consensual vocabulary that different actors can use to interact with network elements and services. The EPRIM defines a generic structure for imperative policies. The GPIM, as well as the combination of the GPIM and EPRIM, are converted to generic YANG data modules.

In one possible approach, SUPA Generic & ECA Policy YANG Data
modules together with the Resource and Service YANG data models
specified in IETF (which define the specific elements that will be
controlled by policies) are used by the Service Interface Logic.
This Service Interface Logic creates appropriate input mechanisms
for the operator to define policies (e.g., a web form or a script)
for creating and managing the network configuration. The operator
interacts with the interface, which is then translated to
configuration snippets.

Note that YANG models may not exist. In this case, the SUPA generic
policy YANG data modules serve as an extensible basis to develop new
YANG data models for the Service Interface Logic to create
appropriate input mechanisms for the operator to define policies.
This transfers the work specified by the Resource and Service YANG
data models specified in IETF into the Service Interface Logic,
which is then translated to configuration snippets.

```
                              +---------------------+
   +----------+          \|          SUPA          |
   |   IETF   |---+----+  Information Models |
   +----------+   |   /|     GPIM and EPRIM   |
                  |     +---------+-----------+
   Assignments    |               | Defines Policy Concepts
   and Manage     |              \|/
     Content      |     +---------+-----------+
                  |  \|      SUPA Generic     |
              +----+      & ECA Policy      |
                  /|     YANG Data modules  |
                  +---------+-----------+
                        *  Possible Approach
   +---------------------------*---------------------------+
   |  Management System        *                           |
   |                          \*/                          |
   |           Fills  +---------+---------+  +-------------+  |
   | +--------+ Forms \| Service Interface |/ |Resource and |/ | +----+
   | |Operator|--------+     Logic         +--|Service YANG |----|IETF|
   | +--------+ Runs  /| (locally defined  |\ | Data Models |\ | +----+
   |           scripts |forms, scripts,...)|  +-------------+  |
   |                  +---------+---------+                  |
   |                          \|/                           |
   |                   +-------+--------+                   |
   |                   | Local Devices  |                   |
   |                   | and Management |                   |
   |                   |    Systems     |                   |
   |                   +----------------+                   |
   +-------------------------------------------------------+
```

Figure 1 SUPA Framework

Figure 1 is exemplary. The Operator actor shown in Figure 1 can interact with SUPA in other ways not shown in Figure 1. In addition, other actors (e.g., an application developer) that can interact with SUPA are not shown for simplicity.

The EPRIM defines an Event-Condition-Action (ECA) policy as an example of imperative policies. An ECA policy rule is activated when its event clause is true; the condition clause is then evaluated and, if true, signals the execution of one or more actions in the action clause. Imperative policy rules require additional management functions, which are explained in section 2.2 below.

Figure 2 shows a SUPA Policy Model creating and communicating policy rules to two different Network Manager and Network Controller elements.

The Generic Policy Information Model (GPIM) was used to construct policies. The GPIM defines generic policy concepts, as well as two types of policies: ECA policy rules and declarative policy statements.

An ECA policy rule is activated when its event clause is true; the condition clause is then evaluated and, if true, signals the execution of one or more actions in the action clause. This type of policy explicitly defines the current and desired states of the system being managed.

A set of Generic Policy Data Models are then created from the GPIM. These YANG data model policies are then used to control the configuration of network elements that model the service(s) to be managed using policy.

```
                       OSS/BSS/Orchestrator
                             / \
                              C
                             \ /
 +---------------------------+--------------------------------+
 |                     SUPA Policy Model                      |
 |        +----------------------------------+               |
 |        | Generic Policy Information Model  |               |
 |        +----+-----------------------+----+               |
 |             D                        \D/                  |
 |             D          +-----------+-------------+        |
 |             D          | ECAPolicyRule Information |       |
 |             D          | Model (EPRIM)             |       |
 |             D          +-----------+-------------+        |
 |   +---------------D---------------------D---------------+ |
 |   |         \D/    SUPA Policy DM       D               | |
 |   |+---------------+----------+         D               | |
 |   || Generic Policy Data Model |        D               | |
 |   |+-------------------+-------+         D               | |
 |   |           \D/                \D/                    | |
 |   |         +--+-------------------+-------------+ |     | |
 |   |         |    ECA PolicyRule Data Model       | |     | |
 |   |         +------------------------------------+ |     | |
 |   +--------------------------+---------------------+ |   | |
 |  +---------------------------+------------------------+  |
 +---------------------------------|---------------------------+
               +-------------+--------+
               \C/                    \C/  NETCONF/RESTCONF
 +---------------+----------+  +-------+--------------------+
 |     EMS/NMS/Controller    |  |     EMS/NMS/Controller     |
 |   +-------------------+   |  |   +-------------------+   |
 |   | Network Service &  |   |  |   | Network Service &  |   |
 |   | Resource Data Models|  |  |   | Resource Data Models|  |
 |   +-------------------+   |  |   +-------------------+   |
 +---+---+---+-------------+  +-----+---+---+-------------+
    / \ / \ / \                  / \ / \ / \
     C   C   C                    C   C   C
    \ / \ / \ /                  \ / \ / \ /
    NE1 NE2 NEn                  NE1 NE2 NEn
```

                   Figure 2 SUPA Policy Model Framework

In Figure 2:
A double-headed arrow with Cs means communication;
A double-headed arrow with Ds means derived from.

The network elements used in this framework are:

SUPA Policy Model: represents one or more policy modules that
contain the following entities:

Generic Policy Information Model: a model for defining policy
rules that are independent of data repository, data definition,
query, and implementation languages, and protocol. This model is
abstract and is used for design; it MUST be turned into a data model
for implementation.

Generic Policy Data Model: a model of policy rules for that are
dependent of data repository, data definition, query, and
implementation languages, and protocol.

ECA Policy Rule Information Data Model (EPRIM): represents a policy
rule as a statement that consists of an event clause, a condition
clause, and an action clause. This type of Policy Rule explicitly
defines the current and desired states of the system being managed.
This model is abstract and is used for design; it MUST be turned
into a data model for implementation.

ECA Policy Rule Data Model: a model of policy rules derived from
EPRIM, consist of an event clause, a condition clause, and an action
clause.

EMS/NMS/Controller: represents one or more entities that are able
to control the operation and management of a network infrastructure
(e.g., a network topology that consists of Network Elements).

Network Service & Resource Data Models: models of the service as
well as physical and virtual network topology including the resource
attributes (e.g., data rate or latency of links) and operational
parameters needed to support service deployment over the network
topology.

Network Element (NE), which can interact with local or remote
EMS/NMS/Controller in order to exchange information, such as
configuration information, policy enforcement capabilities, and
network status.

   Relationship among Policy, Service and Resource models can be illustrated by
the
   figure below.

```
        +----------------+                    +----------------+
        |    Policy      |        (1)          |    Service     |
        |                |********************|                |
        |    ( SUPA )    |                    |  ( L3SM, ... )  |
        +--------------+                      +---------------+
              **                                        **
                **                                    **
                  **                                **
           (2)   **                            **    (3)
                  **                          **
                    **                      **
                      **                  **
              +-------------------+
              |     Resource       |
              |                    |
              | (Inventory, ... )  |
              +-------------------+
```

         Figure 3 Relationship among Policy, Service and Resource

   In Figure 3:
   (1) policy manages and can adjust service behavior as necessary
   (2) policy manages and can adjust resource behavior as necessary
   (3) resource hosts service; changing resources may change service
   behavior as necessary

   Policies are used to manage behavior. Policies can be applied to
   services and resources. More importantly, policies can be used to
   manage how resources are allocated and assigned to services. This
   enables a single policy to manage one or multiple services and
   resources as well as their dependencies.

2.2. Operation

   SUPA can be used to define various types of policies, including
   policies that affect services and/or the configuration of
   individual or groups of network elements. SUPA can be used by a
   centralized and/or distributed set of entities for creating,
   managing, interacting with, and retiring policy rules.

   The SUPA scope is limited to policy information and data models.
   SUPA will not define network resource data models or network
   service data models; both are out of scope. Instead, SUPA will make
   use of network resource data models defined by other WGs or SDOs.

   Declarative policies that specify the goals to achieve but not how
   to achieve those goals (also called "intent-based" policies) are out
   of scope for the initial phase of SUPA.

2.3. The GPIM and the EPRIM

   The GPIM provides a common vocabulary for representing concepts
   that are common to expressing different types of policy, but which
   are independent of language, protocol, repository, and level of
   abstraction.

   This enables different policies at different levels of abstraction
   to form a continuum, where more abstract policies can be translated
   into more concrete policies, and vice-versa. For example, the
   information model can be extended by generalizing concepts from an
   existing data model into the GPIM; the GPIM extensions can then be
   used by other data models.

   The SUPA working group develops models for expressing policy at
   different levels of abstraction. Specifically, two models are
   envisioned (both of which are contained in the Generic Policy
   Information Model block in Figure 1:

   1. a generic model (the GPIM) that defines concepts and vocabulary
      needed by policy management systems independent of the form and
      content of the policy

   2. a more specific model (the EPRIM) that refines the GPIM to
      specify policy rules in an event-condition-action form

2.4. Creation of Generic YANG Modules

   An information model is abstract. As such, it cannot be directly
   instantiated (i.e., objects cannot be created directly from it).
   Therefore, both the GPIM, as well as the combination of the GPIM
   and the EPRIM, are translated to generic YANG modules.

   SUPA will provide guidelines for translating the GPIM (or the
   combination of the GPIM and the EPRIM) into concrete YANG data
   models that define how to manage and communicate policies between
   systems. Multiple imperative policy YANG data models may be
   instantiated from the GPIM (or the combination of the GPIM and the
   EPRIM). In particular, SUPA will specify a set of YANG data models
   that will consist of a base policy model for representing policy
   management concepts independent of the type or structure of a
   policy, and as well, an extension for defining policy rules
   according to the ECA paradigm.

   The process of developing the GPIM, EPRIM and the derived/translated
   YANG data models is realized following the sequence shown below.
   After completing this process and if the implementation of the YANG

data models requires it, the GPIM and EPRIM and the
derived/translated YANG data models are updated and synchronized.

(1)=>(2)=>(3)=>(4)=>(3')=>(2')=>(1')

Where, (1)=GPIM; (2)=EPRIM; (3)=YANG data models; (4)=
Implementation; (3')= update of YANG data models; (2')=update of
EPRIM; (1') = update of GPIM

The YANG module derived from the GPIM contains concepts and
terminology for the common operation and administration of policy-
based systems, as well as an extensible structure for policy rules
of different paradigms. The YANG module derived from the EPRIM
extends the generic nature of the GPIM to represent policies using
an event-condition-action structure.

The above sequence allows for the addition of new, as well as editing
of existing model elements in the GPIM and EPRIM. In practice, the
implementation sequence may be much simpler. Specifically, it is
unlikely that the GPIM will need to be changed. In addition, changes
to the EPRIM will likely be focused on fine-tuning the behavior
offered by a specific set of model elements.

3. Security Considerations

   TBD

4. IANA Considerations

   This document has no actions for IANA.

5. Contributors

   The following people all contributed to creating this document,
   listed in alphabetical order:

        Ying Chen, China Unicom
        Luis M. Contreras, Telefonica I+D
        Dan Romascanu, Avaya
        J. Schoenwaelder, Jacobs University, Germany
        Qiong Sun, China Telecom

6. Acknowledgments

   This document has benefited from reviews, suggestions, comments and
   proposed text provided by the following members, listed in
   alphabetical order: Andy Bierman, Benoit Claise, Joel Halpern, Bert
   Wijnen, Tianran Zhou.

   Part of the initial draft of this document was picked up from
   previous documents, and this section lists the acknowledgements from
   them.

   From "SUPA Value Proposition" [Klyus2016]

   The following people all contributed to creating this document,
   listed in alphabetical order:

7. References

7.1. Normative References

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
             Requirement Levels", BCP 14, RFC 2119, March 1997.

7.2. Informative References

   [RFC3198] Westerinen, A., Schnizlein, J., Strassner, J.,
   Scherling, M., Quinn, B., Herzog, S., Huynh, A., Carlson, M., Perry,
   J., Waldbusser, S., "Terminology for Policy-Based Management", RFC
   3198, November, 2001

[RFC6020] M. Bjorklund, "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, October 2010.

[RFC6020bis] M. Bjorklund, "The YANG 1.1 Data Modeling Language", IETF Internet draft, draft-ietf-netmod-rfc6020bis-14, June 2016.

[RFC7285] R. Alimi, R. Penno, Y. Yang, S. Kiesel, S. Previdi, W. Roome, S. Shalunov, R. Woundy "Application-Layer Traffic Optimization (ALTO) Protocol", September 2014

[SUPA-info-model] J. Strassner, J. Halpern, S. van der Meer, "Generic Policy Information Model for Simplified Use of Policy Abstractions (SUPA)", IETF Internet draft, draft-ietf-supa-generic-policy-info-model-01, July 2016

[TR235] J. Strassner, ed., "ZOOM Policy Architecture and Information Model Snapshot", TR245, part of the TM Forum ZOOM project, October 26, 2014

[Karagiannis2015] G. Karagiannis, ed., "Problem Statement for Simplified Use of Policy Abstractions (SUPA)", IETF Internet draft, draft-karagiannis-supa-problem-statement-07, June 5, 2015

[Klyus2016] M. Klyus, ed., "SUPA Value Proposition", IETF Internet draft, draft-klyus-supa-value-proposition-00, Mar 21, 2016

[Zhou2015] C. Zhou, ed., "The Framework of Simplified Use of Policy Abstractions (SUPA)", draft-zhou-supa-framework-02, May 08, 2015

Authors' Addresses

   Will(Shucheng) Liu
   Huawei Technologies
   Bantian, Longgang District, Shenzhen 518129
   P.R. China
   Email: liushucheng@huawei.com

   John Strassner
   Huawei Technologies
   2330 Central Expressway
   Santa Clara, CA 95138 USA
   Email: strazpdj@gmail.com

   Georgios Karagiannis
   Huawei Technologies
   Hansaallee 205, 40549 Dusseldorf
   Germany
   Email: Georgios.Karagiannis@huawei.com

   Maxim Klyus
   NetCracker
   Kozhevnicheskaya str.,7 Bldg. #1
   Moscow, Russia
   E-mail: klyus@netcracker.com

   Jun Bi
   Tsinghua University
   Network Research Center, Tsinghua University
   Beijing 100084
   P.R. China
   Email: junbi@tsinghua.edu.cn

   Chongfeng Xie
   China Telecom Beijing Research Institute
   China Telecom Beijing Information Science&Technology Innovation Park
   Beiqijia Town Changping District Beijing 102209 China
   Email: xiechf@ctbri.com.cn