Network Working Group
Internet Draft
Intended status: Standard Track
Expires: July 4, 2016

J. Strassner
Huawei Technologies
J. Halpern
Ericsson
J. Coleman
Cisco Systems

January 4, 2016

Generic Policy Information Model for Simplified Use of Policy Abstractions (SUPA) draft-strassner-supa-generic-policy-info-model-03

Abstract

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.

```
**************************************************************
*                                                            *
*  Editor's note: this draft is still being changed by the   *
*  authors. However, it contains a significant rewrite and   *
*  update to reflect the new SUPA charter; hence, please     *
*  view this draft as an early pre-published version. An     *
*  updated version will follow soon.                         *
*                                                            *
**************************************************************
```

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), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.
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."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

This Internet-Draft will expire on October 26, 2015.

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 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 an interface to a network management function that takes high-level, possibly network-wide policies as input and creates element configuration snippets as output. SUPA addresses the needs of operators and application developers to represent multiple types of policy rules, which vary in the level of abstraction, to suit the needs of different actors [1], [10].

Different constituencies of users would like to use languages that use terminology and concepts that are familiar to each constituency. Rather than require multiple software systems to be used for each language, a common information model enables these different languages to be mapped to terms 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 needs different grammars using different concepts and terminologies to match their skill set. This is shown in Figure 1. 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.

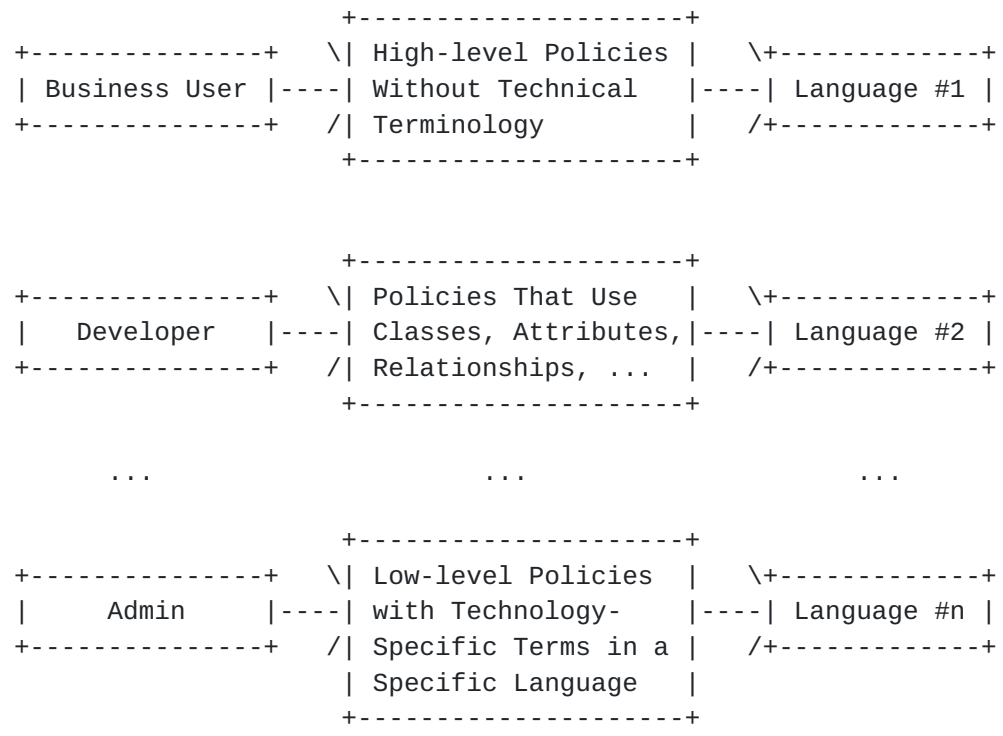


Figure 1. Different Constituencies Need Different Policies

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 any number of lower-level modules, where each lower-level module extends the concepts of the single high-level module. Conceptually, a set of model elements (e.g., classes, attributes, and relationships) are used to define the Generic Policy Information Model (GPIM); this module defines a common set of policy management concepts that are independent of the type of policy (e.g., imperative, procedural, declarative, or otherwise). Then, any number of additional modules are derived from the GPIM; each additional module MUST extend the GPIM to define a new type of policy rule by adding to the GPIM. (Note: using extensions preserves the core interoperability, as compared with modification of the base GPIM, which 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. (The Appendices describe the SUPA Logic Statement Information Model (LSIM), which shows how to extend the GPIM to represent statements that are subsets of either Propositional Logic (PL) or First-Order Logic (FOL), respectively. Both of these logics are types of declarative logic. Note that the LSIM is currently out of scope. However, it is outlined as a set of Appendices in this document to get feedback on its utility.

1.2. Changes Since Version -02

There are several main changes in this version of this document compared to the previous version (-02) of this document. They are:

1) The GPIM has been redesigned to be more compact, making it easier to construct data models
2) As part of 1), additional options for constructing data models have been added to the GPIM
3) The LSIM has been moved into an Appendix, since the latest charter makes it currently out of scope. However, it is important to ensure that the GPIM can serve as a single foundation that different types of policies can all be derived from to ensure that SUPA can interact with other SDOs, as well as for future work in the IETF.
4) Examples and figures have been added to clarify the model

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
NETCONF	Network Configuration protocol
OAM&P	Operations, Administration, Management, and Provisioning
OID	Object IDentifier
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIP	Policy Information Point
PR	Policy Repository
PXP	Policy Execution Point
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.

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 protocol (typically, but not necessarily, all three).

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.

3.2.1.3. 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 near the top-left side of the class.

3.2.1.4. 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 near the top-left side of the class.

3.2.1.5. 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.

3.2.1.6. 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 OAMP 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.

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.

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 at least create, read, query, update, and delete functions are supported.

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 SUPAPolicyBooleanClause 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 SUPAPololicyEncodedClause 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.

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. 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. 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.

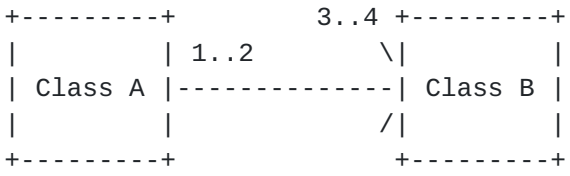
3.2.3.8. Navigability

A relationship may have a restriction on the ability of an object at one end of the relationship to access the object at the other end of the relationship. 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. In this example, the arrow would be pointing at object B.

Examples of navigability are:



This is an association. 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:

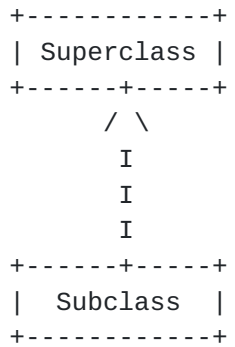
Class A depends on 3 to 4 instances of Class B, and
Class B depends on 1 to 2 instances of Class A.

3.3. Symbology

The following symbology is used in this document:

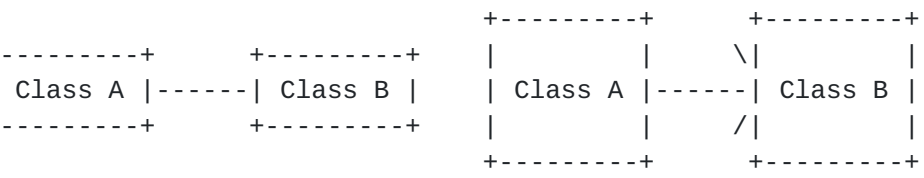
3.3.1. Inheritance

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



3.3.2. Association

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

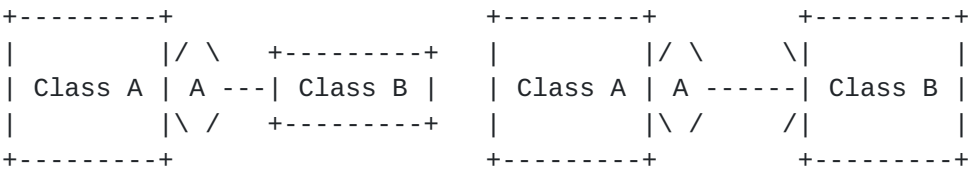


association with no navigability restrictions

association with navigability restrictions

3.3.3. Aggregation

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

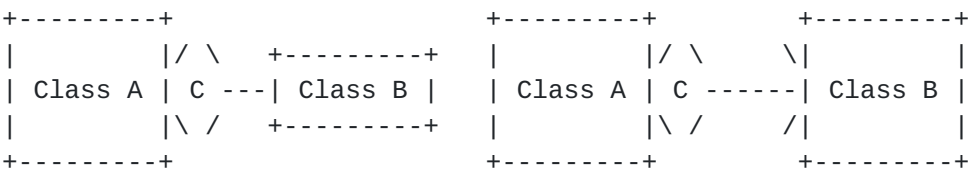


aggregation with no navigability restrictions

aggregation with navigability restrictions

3.3.4. Composition

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

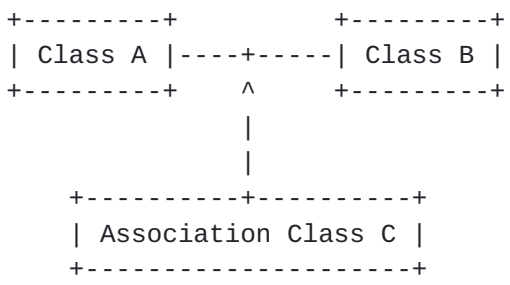


composition with no navigability restrictions

composition with navigability restrictions

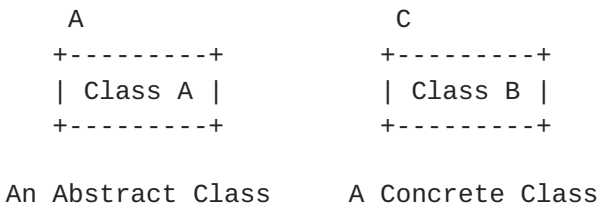
3.3.5. Association Class

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



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.



4. Policy Abstraction Architecture

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

- The GPIM defines a technology-neutral information model that can express the concept of Policy.
  - All classes, except for SUPAPolicyMetadata, inherit from SUPAPolicyObject, or one of its subclasses
  - SUPAPolicyObject and SUPAPolicyMetadata are designed to inherit from classes in another model; the GPIM does not define an "all-encompassing" model.
- This version of this document restricts the expression of Policy to a set of event-condition-action statements.
- However, the purpose of the GPIM is to enable different policies that have fundamentally different representations to share common model elements. This abstraction of the content of a Policy from its representation is supported by:
  - All policy rules (of which SUPAECAPolicyRule is the first example of a concrete class) are derived from the SUPAPolicyStructure class.
  - All objects that are components of policy rules are derived from the SUPAPolicyComponentxxx`Structure class.
  - A SUPAPolicy MUST contain at least one SUPAPolicyClause.
  - A SUPAPolicy MAY specify one or more SUPAPolicyTarget, SUPAPolicySource, and SUPAPolicyMetadata objects to augment the semantics of the SUPAPolicy
- A SUPAPolicyClause has two subclasses:
  - A SUPAPolicyBooleanClause, which is used to build SUPAECAPolicyRules from reusable objects.
  - A SUPAPolicyEncodedClause, which is used for using attributes instead of objects to construct a SUPAECAPolicyRule.
- 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.
- SUPAMetadata MAY be defined for any SUPAPolicyObject class.
- SUPAMetadata MAY be prescriptive and/or descriptive in nature.

Please see the Appendices for experimental definitions of declarative policies. Note that they also are derived from the GPIM, and extend (but do not change) the above abstractions.

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.

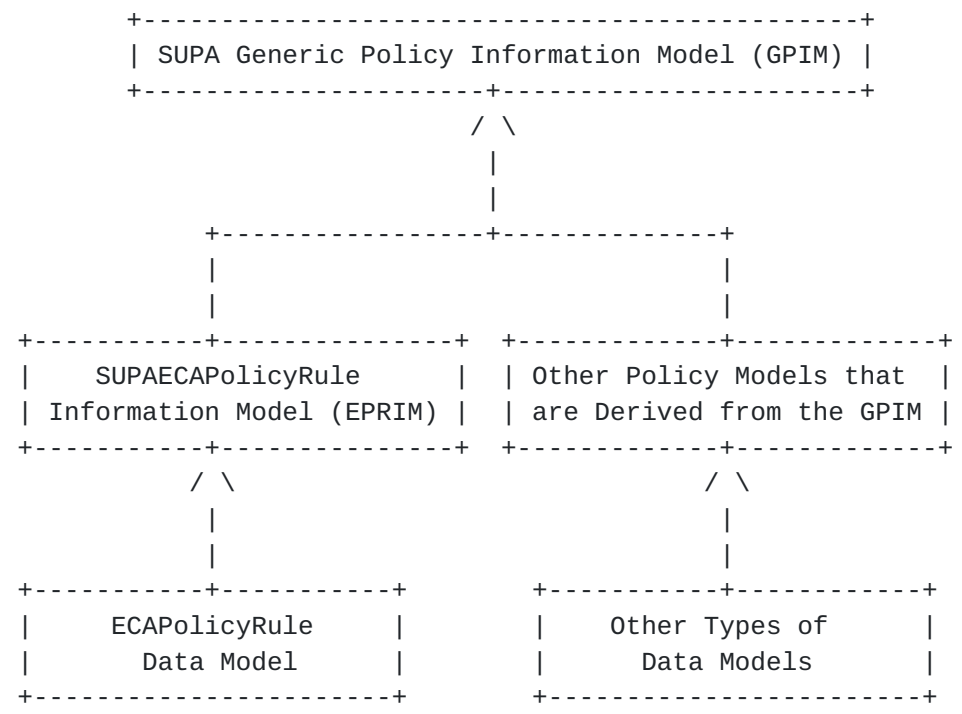


Figure 2. Overview of SUPA Policy Rule Abstractions

This draft defines the GPIM and EPRIM. Note that there is only ONE GPIM and ONE EPRIM. While both can be extended, it is important to limit the number of information models to one, in order to avoid defining conflicting concepts at this high a level of abstraction. Similarly, 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:

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

The SUPAPolicyStructure hierarchy has two main subclasses, an atomic (stand-alone) and composite (hierarchy of) PolicyRule. The SUPAPolicyComponentStructure hierarchy has two main subclasses:

- A SUPAPolicyClause (the building block of all Policies)
- A SUPAPolicyComponentDecorator, which uses the decorator pattern to define objects that make up the content of

the SUPAPolicyClause

This yields the following high-level structure:

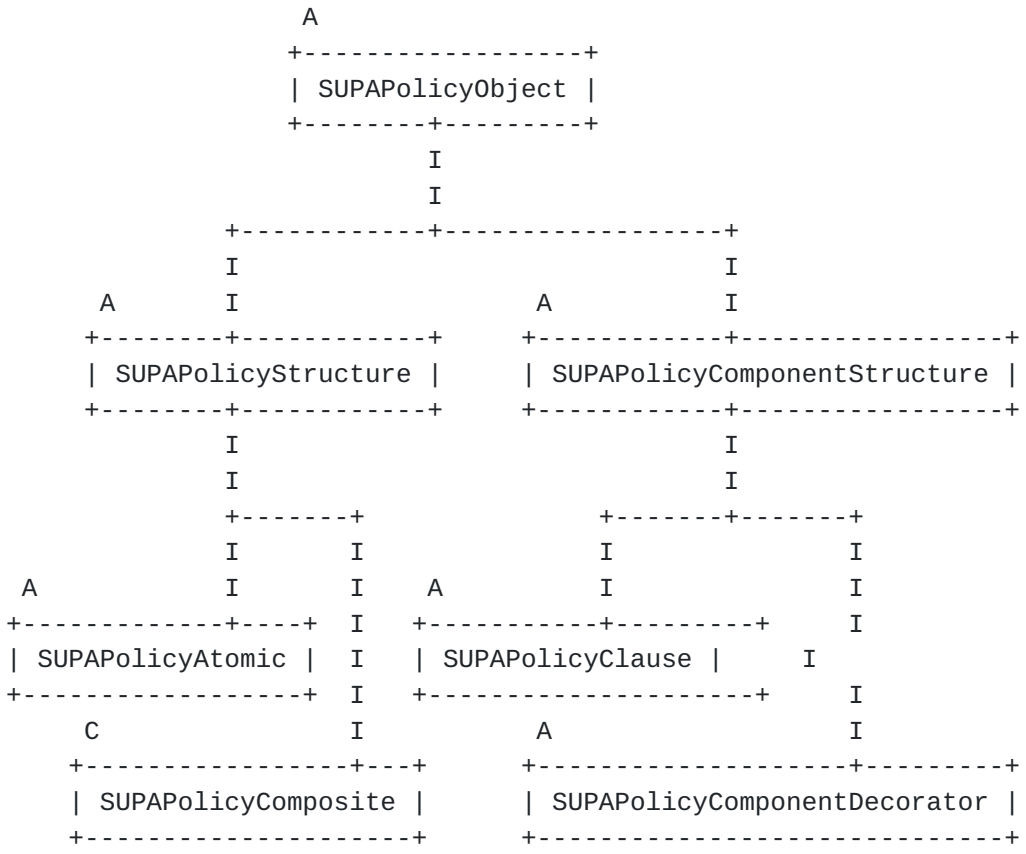


Figure 3. Functional View of the Top-Level GPIM

Note that all classes except the SUPAPolicyComposite class 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 to 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). This simplifes code generation and reusability.

4.3.2. SUPAPolicyStructure

SUPAPolicyStructure is an abstract superclass that serves as the base class for defining all types of policies (though in this version of this document, this is limited to ECA policies). It serves as a convenient aggregation point to define atomic and composite SUPAPolicies; it also enables PolicySources and/or PolicyTargets to be associated with a given set of Policies.

SUPA Policies are defined as either a stand-alone PolicyContainer (i.e., a subclass of SUPAPolicyAtomic), or a hierarchy of PolicyContainers (i.e., as an instance of or subclass of SUPAPolicyComposite). A PolicyContainer specifies the structure, content, and optionally, source, target, and metadata information for the Policy.

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. However, the ability to define different types of policy rules, let alone combine different types of policies, is NOT true of [RFC3060], [RFC3460], [4] and [6]; [RFC3060], [RFC3460], and [4] are limited to only defining condition-action rules, and [6] is limited to only defining ECA policy rules.

4.3.3. SUPAPolicyComponentStructure

SUPAPolicyComponentStructure is an abstract superclass that serves as the base class for defining the set of policy components that are used to make up a given Policy. From an information model point-of-view, this isolates the various subclasses of SUPAPolicyComponentStructure, and controls how they are used by other different elements of the SUPAPolicy hierarchy.

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. This 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 or to construct more complex clauses that form a policy statement. Note that there is no need to create the SimplePolicyCondition and ComplexPolicyCondition objects defined in [RFC3460].

Other models define additional subclasses of SUPAPolicyStatment (e.g., the EPRIM defines a SUPABooleanClause, which is specific to an ECA Policy Rule, and the LSIM (see Appendix) defines a SUPALogicClause, which is specific to declarative policies). This structure enables different types of Policies, which have different forms of content and structure, to all be represented as subclasses of SUPAPolicyClause. This enables the designer to use multiple types of Policies.

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, as if the case in [RFC3460], [4], and [6].

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 [RFC3060], [RFC3460], [4] and [6], since they lack both the abstraction of a common SUPAPolicyClause and do not use the decorator (or similar) design pattern.

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:

- SUPAPolicyTerm, which enables a clause to be defined in a canonical {variable, operator, value} form
- SUPAVendorDecoratedComponent, which enabled a custom object to be defined and then used in a SUPAPolicyClause
- SUPAPolicyCollection, which enables a collection of objects to be gathered together and associated with all or a portion of a SUPAPolicyClause
- SUPAECAComponet, 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.

4.4. The Design of the GPIM

This section describes the overall design of the GPIM.

4.4.1. Structure of Policies

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.

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 interpreted
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 PolicyMetadata 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.

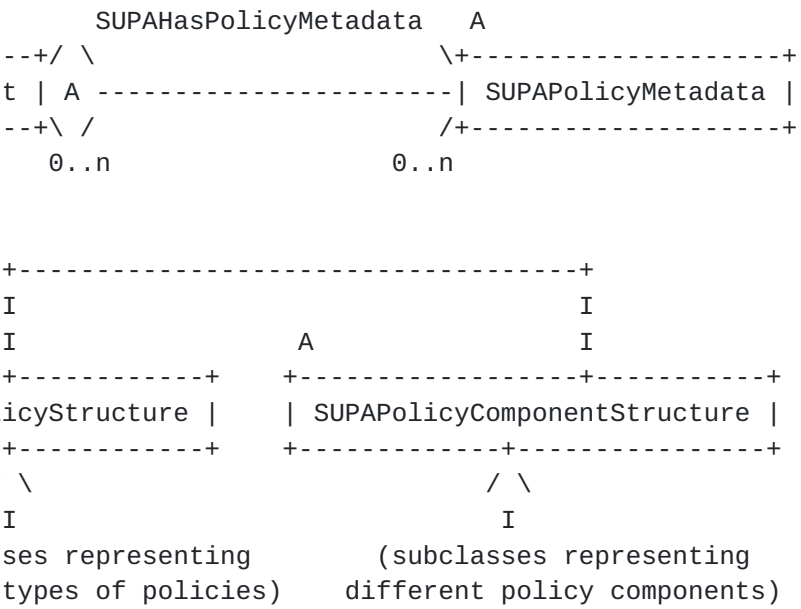


Figure 4. Structure of a Policy

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 two subclasses: SUPAPolicyStructureComposite and SUPAPolicyStructureAtomic. The former is used to represent groups of SUPAPolicyStructure objects (i.e., groups of SUPAPolicyStructureAtomic and/or SUPAPolicyStructureComposite objects), and the latter is used to

represent stand-alone Policy Rules.

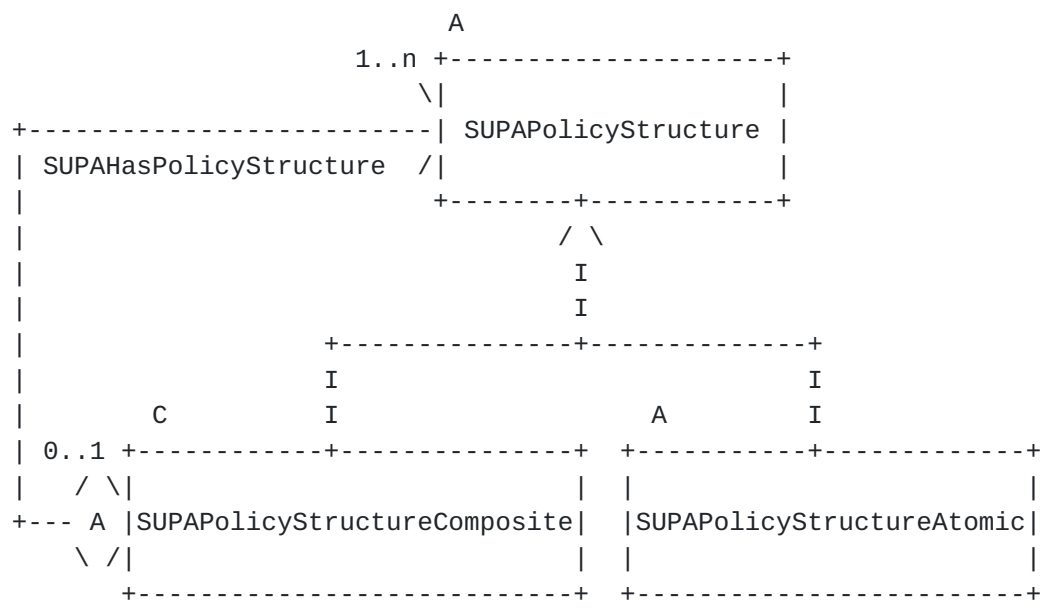


Figure 5. The Composite Pattern Applied to SUPAPolicyStructure

The SUPAHasPolicyStructure aggregation says that if it is instantiated, one or more SUPAPolicyStructure objects can be contained in a SUPAPolicyStructureComposite. This works due to inheritance. Since the SUPAPolicyStructure class is a superclass of both SUPAPolicyStructureAtomic and SUPAPolicyStructureComposite, a SUPAPolicyStructureComposite can contain either class. The SUPAHasPolicyStructure aggregation is realized as an association class, in order to manage which set of SUPAPolicyStructure objects can be aggregated by which SUPAPolicyStructureComposite object. (If a stand-alone policy rule is desired, then a concrete instance of a SUPAPolicyStructureAtomic class is created; there is no need to instantiate the SUPAHasPolicyStructure aggregation.)

SUPAPolicyStructureComposite is defined as a concrete class, so that it can be directly instantiated and used without having to subclass it. In contrast, the other five classes described in Figures 3 and 4 are all defined as abstract. This helps simplify the construction of the data model, because abstract classes cannot be instantiated (rather, they are used to define characteristics and behavior of the concepts they represent).

4.4.2. Representing an ECA Policy Rule

An ECA policy rule is a statement that consists of an event clause, a condition clause, and an action clause. Any or all of these clauses can be made into more complex Boolean statements. 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 SUPAPolicyStructureAtomic, so that the composite pattern can be applied to it.

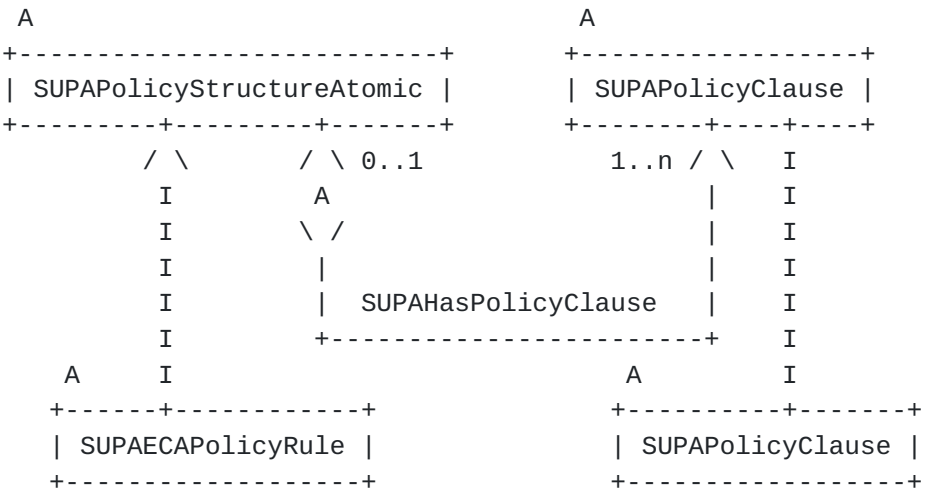


Figure 6. SUPAECAPolicyRule Aggregating SUPAPolicyClauses

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 SUPAPolicyStructureAtomic level, and not at SUPAECAPolicyRule, so that non-ECA policies can 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.

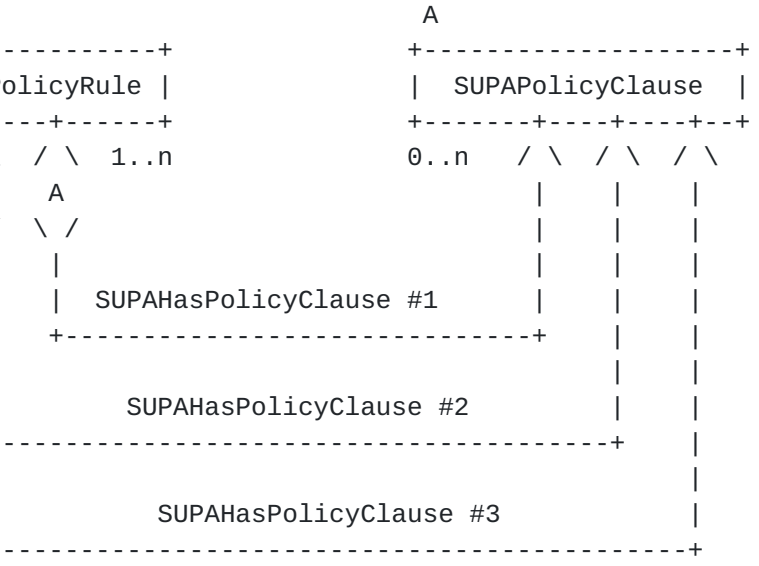


Figure 7. Instantiating a SUPAECAPolicyRule, part 1

In figure 7, SUPAECAPolicyRule is shown as "owning" these three aggregations, since it inherits them from its superclass (SUPAPolicyStructureAtomic). The three aggregations represent the event, condition, and action clauses of a policy rule. Note that each of these clauses MAY consist of one or more SUPAPolicyClauses. Similarly, each SUPAPolicyClause MAY consist of one or more clauses. In this way, simple and complex (e.g., Boolean combinations of clauses) are supported, without having to define additonal objects (as is done in [RFC3460] and [4], with the SimplePolicyCondition, CompoundPolicyCondition, SimplePolicyAction, and CompoundPolicyAction classes).

The multiplicity of the SUPAHasPolicyClause aggregation is 1..n on the aggregate side and 0..n on the part side. This means that a particular SUPAECAPolicyRule MUST have at least one SUPAPolicyClause. This cardinality is refined to 3..n for SUPAECAPolicyRules, but is defined to be 1..n 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 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) with 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. This is shown in Figure 8.

Figure 8. Instantiating a SUPAECAPolicyRule, part 2

The policy management system may use any number of different software mechanisms, such as introspection or reflection, to determine the nature of the aggregation, and select the appropriate subclass of SUPAHasPolicyClauseDetail. The three subclasses of SUPAHasPolicyClauseDetail are named SUPAHasPolicyEventDetail, SUPAHasPolicyConditionDetail, and SUPAHasPolicyActionDetail, respectively.

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. 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, and is shown in

Figure 9.

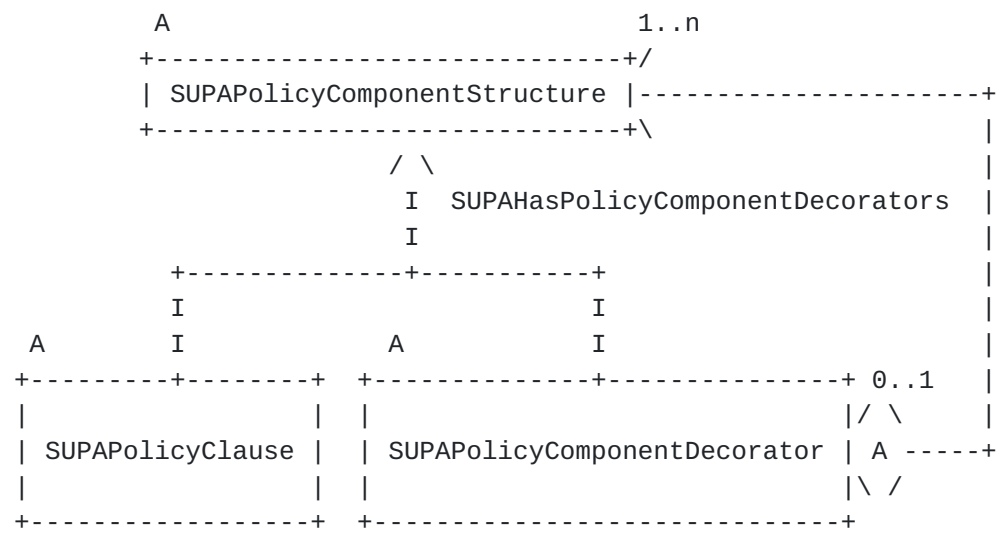


Figure 9. Subclasses of SUPAPolicyComponentStructure

While the above looks like a composite pattern, it is actually the decorator pattern [11]. As stated in 4.3, this pattern 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. Zero or more concrete subclasses of the SUPAPolicyComponentDecorator class can be used to decorate, or "wrap", any of the concrete subclasses of SUPAPolicyClause. Instead of using inheritance to statically create new classes to represent new types of object, the decorator pattern uses composition to dynamically combine smaller objects into more robust ones. 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").

More importantly, 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 statement that makes up the policy rule. Clearly, continuing to subclass 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.

This 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.

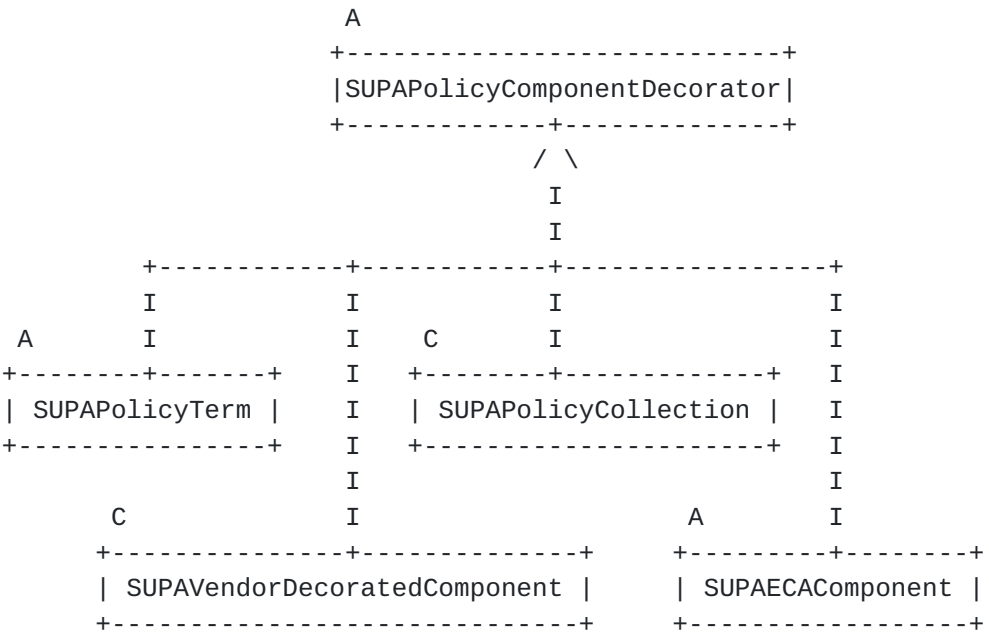


Figure 10. Subclasses of SUPAPolicyComponentDecorator

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

However, if a SUPAPolicyEncodedClause is NOT being used, then a SUPA Policy Clause 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) as a {variable, operator, value} clause
2) as an encoded object (e.g., to pass YANG or CLI code)
3) as a collection of objects that requires further processing in order to be made into a SUPAPolicyClause
4) as an Event, Condition, or Action object

The power of the decorator pattern is that these four different types of objects can be intermixed. For example, the first and last types can be combined as follows:

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

In the above rules, example (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 SUPAPolicyEncodedClause. 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 SUPAPolicyBooleanClause, 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 policy statement can be encoded using one or more SUPAPolicyEncodedClauses; this has the option of encoding the entire statement or any of its three individual clauses (event, condition, action).
2) The policy statement can be defined using one or more SUPAPolicyBooleanClauses; each of the three clauses can be defined as either a single SUPAPolicyBooleanClause, or a combination of SUPAPolicyBooleanClauses 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.

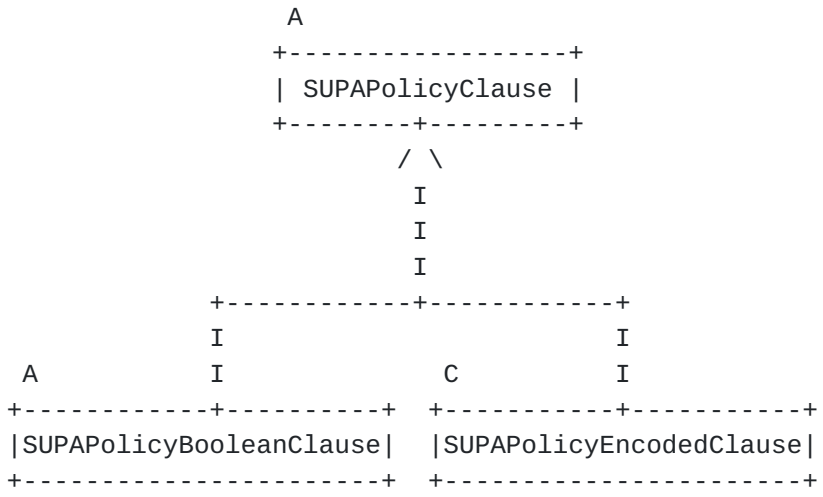


Figure 11. Subclasses of SUPAPolicyClause

SUPAPolicyBooleanClause 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.

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 two aggregations, SUPAHasPolicySource and SUPAHasPolicyTarget. Since SUPAECAPolicyRule is a subclass of SUPAPolicyStructureAtomic, which is in turn a subclass of SUPAPolicyStructure, it (and its subclasses) inherit both of these aggregations. This enables SUPAPolicySources and/or SUPAPolicyTargets to be attached to SUPAECAPolicyRules.

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.

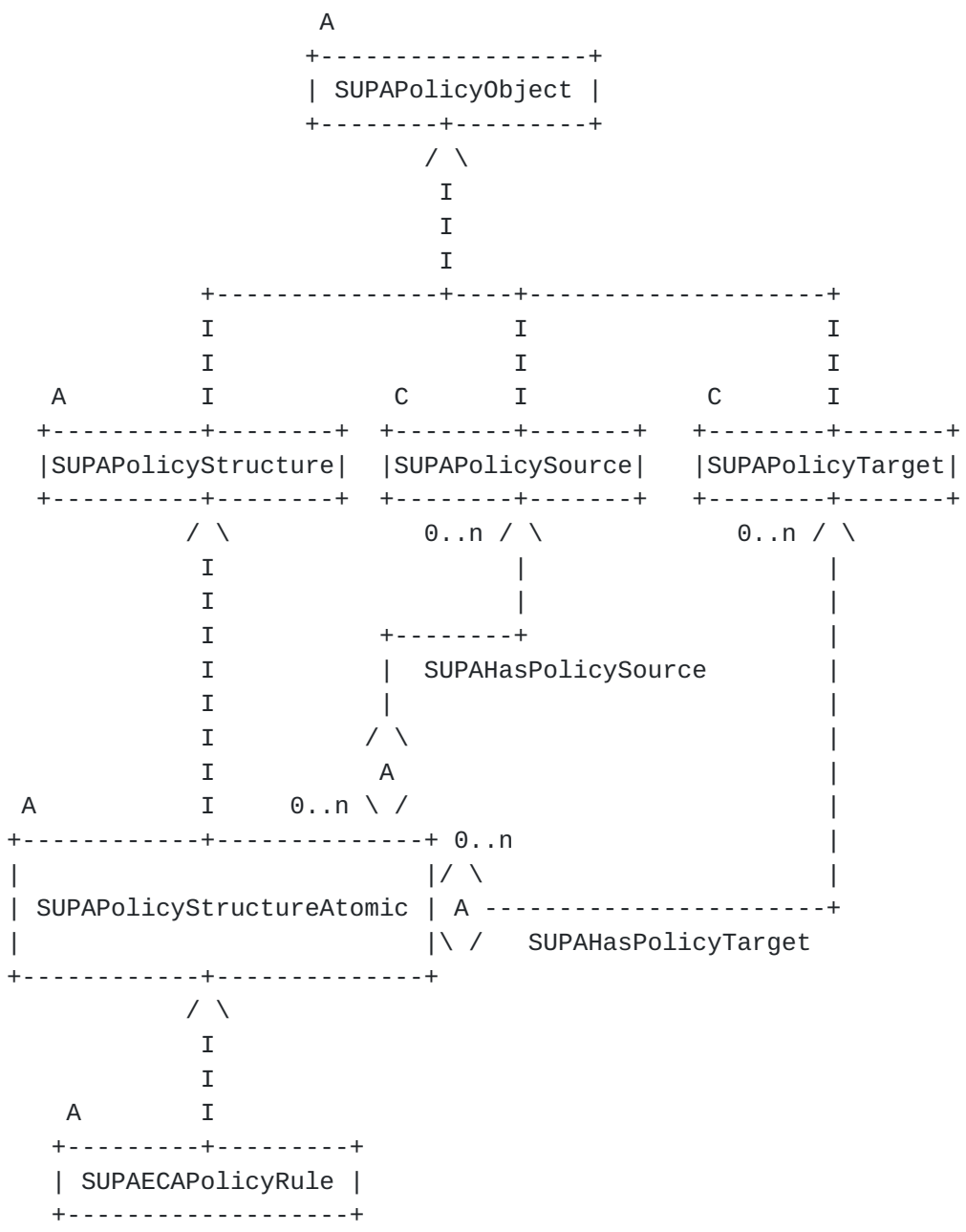


Figure 12. ECAPolicyRules, SUPAPolicySources, and PolicyTargets

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 or may not be in a state that enables SUPAPolicies to be applied to it to change its state.

2) A SUPAPolicyTarget must be able to:
   a) process (either directly or with the aid of a proxy) SUPAPolicies, and/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:

- 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)
- the Single Responsibility Principle [14]
  (every class should have responsibility over one, and only one, part of the functionality provided by the program)

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 indicative of the models 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:

- Context is derived from all applicable system inputs (e.g., OAMP data from network elements, business goals, time of day, geo-location, etc.).
- Context is then used to select a working set of Policies.
- Policies are then used to define behavior at various control points in the system.
- 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
  - ECAPolicyRule actions can be used to change the attribute values, methods, and relationships of the association class
  - This has the affect of changing how the two classes are related to each other
- 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.

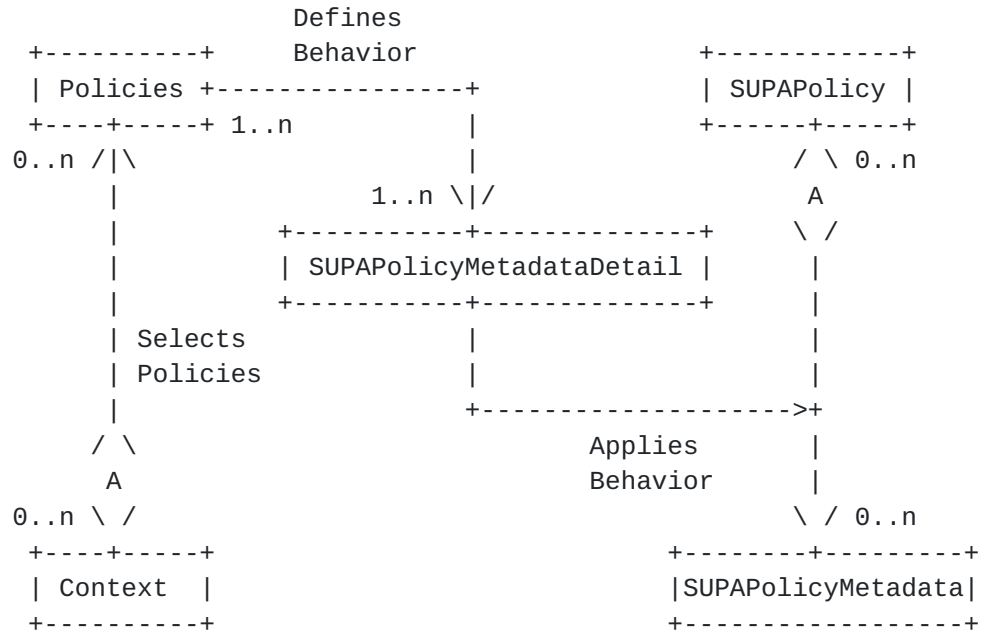


Figure 13. Context-Aware Policy Rules

Assume that the set of deployed Policies are SUPAECAPolicyRules. Then, the actions of these SUPAECAPolicyRules will, for example, change attribute values 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.

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

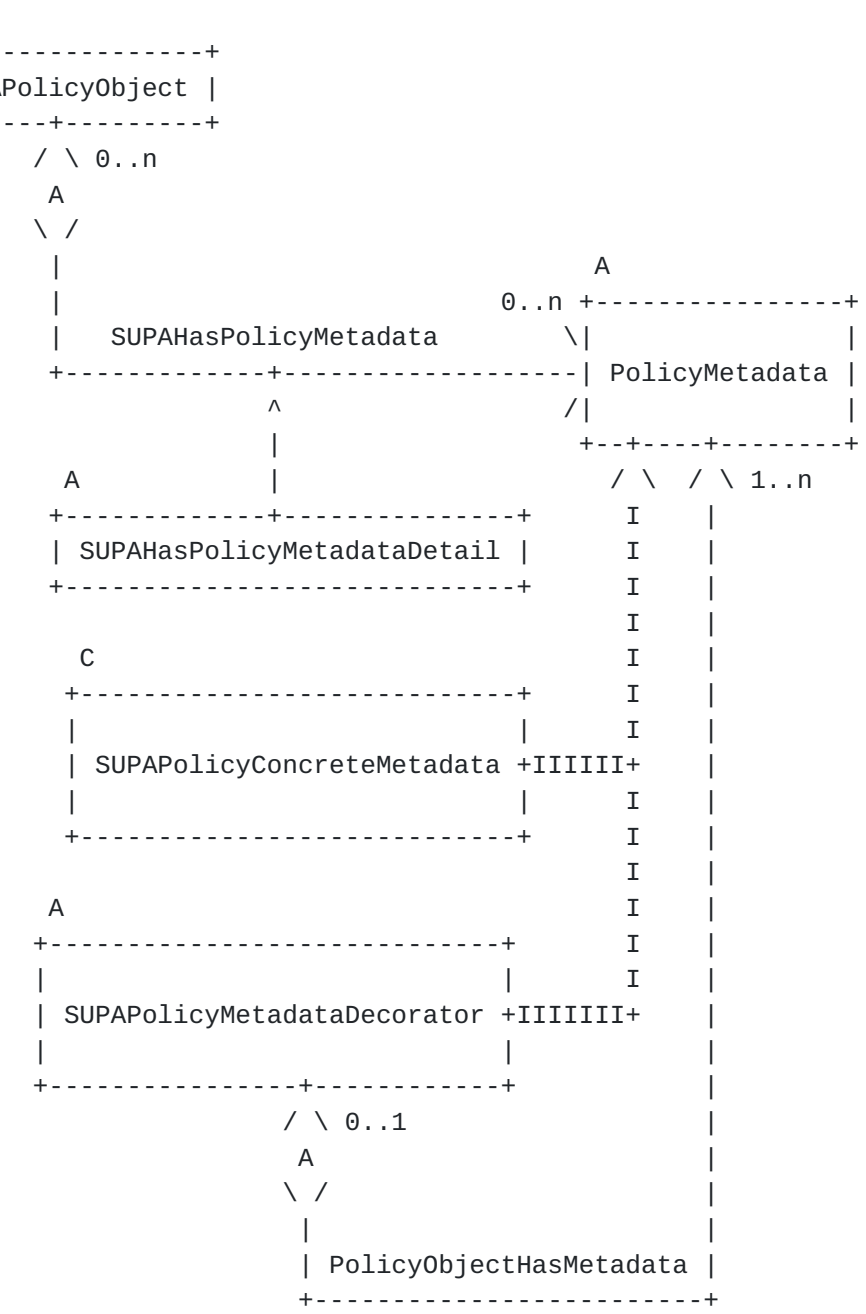


Figure 14. SUPAPolicyMetadata Subclasses and Relationships

Policy, PolicyMetadata, and PolicyMetadataDecorator are abstract; PolicyConcreteMetadata is concrete, and is the object that

instances of the PolicyMetadataDecorator subclasses are wrapped around.

4.4.7.3. Structure of SUPAPolicyMetadata

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

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 15.

Figure 15: 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. In Figure 15, indentation represents subclassing. Numbers after a class refer to the section that defines the class.

Classes, attributes, and relationships that are marked as "mandatory" MUST be part of a conformant implementation. Classes, attributes, and relationships that are marked as "optional" SHOULD 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].

5.2. The Abstract Class "SUPAPolicyObject"

This is a mandatory abstract class. Figure 16 shows the SUPAPolicyObject class, and its four subclasses.

Figure 16. 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 supaPolObjIDFormat, 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, GUIDs, UUIDs, FQDNs, URIs, and other similar mechanisms, to define the structure of an information model. Therefore, a synthetic object ID is defined using these two attributes. This can be used to facilitate mapping to different data model object schemes, such as those depending on URIs, FQDNs, UUIDs, primary key-foreign key relationships, UUIDs, and others can all be accommodated.

The two attributes work collectively, with one defining the content of the object ID and the other 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. This is based on the DEN-ng class design [2].

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 supaPolObjIDFormat, 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 "supaPolObjIDFormat"

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 supaPolObjIDFormat attribute is mapped to the following values:

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

The value 0 may be used to initialize the system, or to signal that there is a problem with thius 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 [RFC3060], as that attribute is intended to be used with just X.500 cn attributes.

5.2.2. SUPAPolicy Relationships

This section defines the relationships of the SUPAPolicy class.

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 5.18.2

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 5.18.3.

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]. For this release, the only official type of rule 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.

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 the SUPAPolicyStructure class, as shown in Figure 5. Two subclasses of SUPAPolicyStructure are defined: SUPAPolicyAtomic (for defining stand-alone policies) and SUPAPolicyComposite (for defining hierarchies of policies). Each SSUPAPolicyComposite can have zero or more instances of a concrete subclass of a SUPAPolicyAtomic class and/or a SUPAPolicyComposite class, or subclasses of either.

5.3.1. SUPAPolicyStructure Attributes

This section defines the attributes of the SUPAPolicyStructure class. 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 "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.

5.3.1.2. The Attribute "supaPolDeployStatus"

This is an optional attribute, which is an enumerated, non-negative integer. It defines the current deployment status of this SUPAPolicy. This means that both individual and groups of policies may be deployed. Both operational and test mode values are included in its definition. Values include:

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

5.3.2. SUPAPolicyStructure Relationships

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

5.3.2.2. The Association Class "SUPAHasPolicySourceDetail"

This is an optional 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.

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

5.3.2.3. The Aggregation "SUPAIsTargetOf"

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 SUPAIsTargetOfDetail 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.

5.3.2.4. The Association Class "SUPAIsTargetOfDetail"

This is an optional 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.

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

5.4. The Abstract Class "SUPAPolicyStructureAtomic"

SUPAPolicyStructureAtomic is the superclass of all of the different types of policies supported by the GPIM. For this release of this document, this is limited to ECA policy rules.

The purpose of the SUPAPolicyStructureAtomic class is to provide a control point for aggregating SUPAPolicyClauses. Since it is the superclass of each type of policy, this means that all policies will use this same, critical, abstraction.

A SUPAPolicyStructureAtomic represents a complete policy. More specifically, a SUPAPolicyStructureAtomic class represents a SUPAPolicy that can operate as a single, stand-alone, manageable object. Put another way, a SUPAPolicyStructureAtomic object can NOT be modeled as a set of hierarchical SUPAPolicy objects; if this functionality is required, then at least one SUPAPolicyStructureComposite object MUST be used.

Each SUPAPolicyStructureAtomic object (or a subclass of it) MUST have at least one SUPAPolicyClause that is used to define the content of the policy.

A SUPAPolicyStructureAtomic SHOULD have one or more instances of SUPAPolicyMetadata attached to it, so that the SUPAPolicyMetadata may provide additional descriptive and prescriptive information about the SUPAPolicyStructureAtomic object. It MAY also have one or more SUPAPolicySources and/or SUPAPolicyTargets attached to it.

SUPAPolicyStructureAtomic objects inherit the attributes defined for its parent class (SUPAPolicyStructure). For example, they can be deployed, and have an associated policy continuum level.

5.4.1. SUPAPolicyStructureAtomic Attributes

This section defines the attributes of the SUPAPolicyStructureAtomic class. This class defines the behavior of all types of atomic (i.e., stand-alone) policies, not just ECA policy rules. Therefore, 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.

5.4.1.1. The Attribute "supaPolExecStatus"

This is an optional attribute, which is an enumerated, non-negative integer. It defines the current execution status of this SUPAPolicy. Both operational and test mode values are included in its definition. Values include:

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

5.4.1.2. The Attribute "supaPolExecFailStrategy"

This is an optional non-negative, enumerated integer that defines what actions, if any, should be taken by this SUPAPolicyStructureAtomic object if it fails to execute correctly. 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 policies
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 a rollback of that SUPAPolicy (and ONLY that SUPAPolicy) is attempted. A value of 3 means that execution is stopped, but any SUPAPolicies that have been previously executed are left in their current state. A value of 4 means that any failure will be ignored, and execution continues.

5.4.1.3. The Attribute "supaPolExecFailTakeActionName"

This is an optional string attribute that identifies the name of the remediation to take if this PolicyStructure object failed to execute properly. The interpreation of this string attribute is defined by the supaPolExecFailTakeActionRes class attribute.

5.4.1.4. The Attribute "supaPolExecFailTakeActionRes"

This is an optional enumerated, non-negative integer attribute that defines how to interpet the value of the supaPolExecFailTakeActionName class attribute. Values include:

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

```
**********************************************************
*                                                        *
*  Editor's note: the above two attributes will be moved to *
*  an association class, and an association will be defined *
*  to make this more portable and powerful.              *
*                                                        *
**********************************************************
```

5.4.2. SUPAPolicyStructureAtomic Relationships

The SUPAPolicyStructureAtomic class defines a single relationship (SUPAHasPolicyClause), which is described in the following subsection.

5.4.2.1. The Aggregation "SUPAHasPolicyClause"

This is a mandatory aggregation that defines the set of SUPAPolicyClauses that are aggregated by this particular SUPAPolicyStructureAtomic instance. The semantics of this aggregation are defined by the SUPAHasPolicyClauseDetail association class.

Every SUPAPolicyStructureAtomic 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 SUPAPolicyStructureAtomic object instances.

Thus, the multiplicity of this aggregation is defined as 0..1 on the aggregate (i.e., the SUPAPolicyStructureAtomic side) and 1..n on the part (i.e., the SUPAPolicyClause side). This means that at least one SUPAPolicyClause MUST be aggregated by this SUPAPolicyStructureAtomic object. Similarly, a SUPAPolicyClause may be aggregated by this particular SUPAPolicyStructureAtomic object.

5.4.2.2. The Association Class "SUPAHasPolicyClauseDetail"

This is a mandatory 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 SUPAPolicyStructureAtomic objects.

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

5.5. The Concrete Class "SUPAPolicyStructureComposite"

This is a mandatory concrete class. This class is a type of PolicyContainer.

A SUPAPolicyStructureComposite class represents a SUPAPolicy as a hierarchy of Policy objects, where the hierarchy contains instances of SUPAPolicyStructureAtomic and/or SUPAPolicyStructureComposite objects. Each of the SUPAPolicy objects, including the outermost SUPAPolicyStructureComposite object, are separately manageable. More importantly, the SUPAPolicyStructureComposite object can aggregate any SUPAPolicyStructure subclass.

A SUPAPolicyStructureComposite SHOULD have one or more instances of SUPAPolicyMetadata attached to it, so that the SUPAPolicyMetadata may provide additional descriptive and prescriptive information about the SUPAPolicyStructureComposite object. It MAY also have one or more SUPAPolicySources and/or SUPAPolicyTargets attached to it.

5.5.1. SUPAPolicyStructureComposite Attributes

No attributes are currently defined for this class, as it functions as a pure PolicyContainer.

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

5.5.2. SUPAPolicyStructureComposite Relationships

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

5.5.2.1. The Aggregation "SUPAHasPolicy"

This is a mandatory aggregation that defines the set of SUPAPolicyStructure objects that are aggregated by this SUPAPolicyStructureComposite instance. The semantics of this aggregation are defined by the SUPAHasPolicyDetail association class.

5.5.2.2. The Association Class "SUPAHasPolicyDetail"

This is a mandatory association class, and defines the semantics of the SUPAHasPolicy aggregation. The attributes and/or relationships of this association class can be used to determine which SUPAPolicyStructure objects are aggregated by which SUPAPolicyStructureComposite objects.

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

5.6. 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.

5.6.1. SUPAPolicyComponentStructure Attributes

The SUPAPolicyComponentStructure currently defines two attributes; these are defined in the following subsections.

5.6.1.1. The Attribute "supaAllowsExternalAccess"

This is a Boolean attribute. If its value is TRUE, then external Applications can access and update the values of this SUPAPolicyComponentStructure object. This enables Applications to have controlled updating of policy components.

5.6.1.2. The Attribute "supaAllowsExternalUpdate"

This is a Boolean attribute. If its value is TRUE, then external Applications can access (but not update) the values of this SUPAPolicyComponentStructure object. This enables Applications to have controlled access to policy components.

5.6.2. SUPAPolicyComponentStructure Relationships

No relationships are currently defined for this class.

5.7. The Abstract Class "SUPAPolicyClause"

This is a mandatory abstract class that separates the representation of a SUPAPolicy from its implementation. This abstraction is missing in [RFC3060], [RFC3460], [4], and [6].

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 policies: ECA policy rules and encoded policies. Both use SUPAPolicyClauses.

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

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

5.7.1. SUPAPolicyClause Attributes

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

5.7.1.1. The Attribute "supaPolStmtAdminStatus"

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 SUPAPolicyClause into a specific administrative state, such as enabled, disabled, or in test.

Note that since a SUPAPolicy is made up of SUPAPolicyClauses, this enables all or part of a SUPAPolicy to be administratively controlled. 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 SUPAPolicyClause from being used. Values 1 and 2 mean that this SUPAPolicyClause is administratively enabled or disabled, respectively. A value of 3 means that this SUPAPolicyClause is in a special test mode and SHOULD NOT be used as part of an OAM&P policy.

5.7.1.2. The Attribute "supaPolStmtExecStatus"

This is an optional attribute, which is an enumerated non-negative integer. It defines whether this SUPAPolicyClause is currently in use and, if so, what its execution status is.

This attribute can be used to place this particular SUPAPolicyClause into a specific execution state, such as enabled, disabled, or in test. 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: In Test (i.e., cannot be used as part of an OAM&P policy)
5: Available (i.e., could be used, but currently isn't)
6: Not Available (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. 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 in a special test state. A test state signifies that it SHOULD NOT be used to evaluate OAM&P policies. Value 5 means that this SUPAPolicyClause is available, but not currently in use. A value of 6 means that it is unavailable for use.

5.7.2. SUPAPolicyClause Relationships

This class does not currently define any relationships, since the decorator pattern is used to "wrap" this object with instances of the subclasses of the SUPAPolicyComponentDecorator object.

5.8. 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 statement are directly encoded into the attributes of the SUPAEncodedClause. Note that other subclasses of SUPAPolicyClause use SUPAPolicy objects to define their content. This class provides the developer a tradeoff of efficiency vs. reusability.

This class uses two of its attributes (supaPolicyClauseContent and supaPolicyClauseFormat) for defining the content and format of a vendor-specific policy statement. This allows direct encoding of the policy statement, without having the "overhead" of using other objects. However, note that while this method is efficient, it does not reuse other SUPAPolicy objects. Rather, it can be thought of as a direct encoding of the policy statement.

5.8.1. SUPAEncodedClause Attributes

This section defines the attributes of the SUPAEncodedClause class.

5.8.1.1. The Attribute "supaClauseContent"

This is a mandatory string attribute, and defines the content of this encoded clause of this clause. It works with another attribute of the SUPAEncodedClause class, called supaClauseFormat, 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 encoded clause for the object instance of this class.

5.8.1.2. The Attribute "supaClauseFormat"

This is a mandatory string attribute, and defines the format of this encoded clause. It works with another attribute of the SUPAEncodedClause class, called supaClauseContent, which defines the content (i.e., the value) 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.

5.8.1.3. The Attribute "supaClauseResponse"

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 in more complex Boolean policy clauses.

5.8.2. SUPAEncodedClause Relationships

This class currently does not define any relationships.

5.9. The Abstract Class "SUPAPolicyComponentDecorator"

This is a mandatory aggregation, 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. In SUPA, this means that any concrete subclass of SUPAPolicyClause is wrapped by any concrete subclass of SUPAPolicyComponentDecorator, as shown in Figure 17 below.

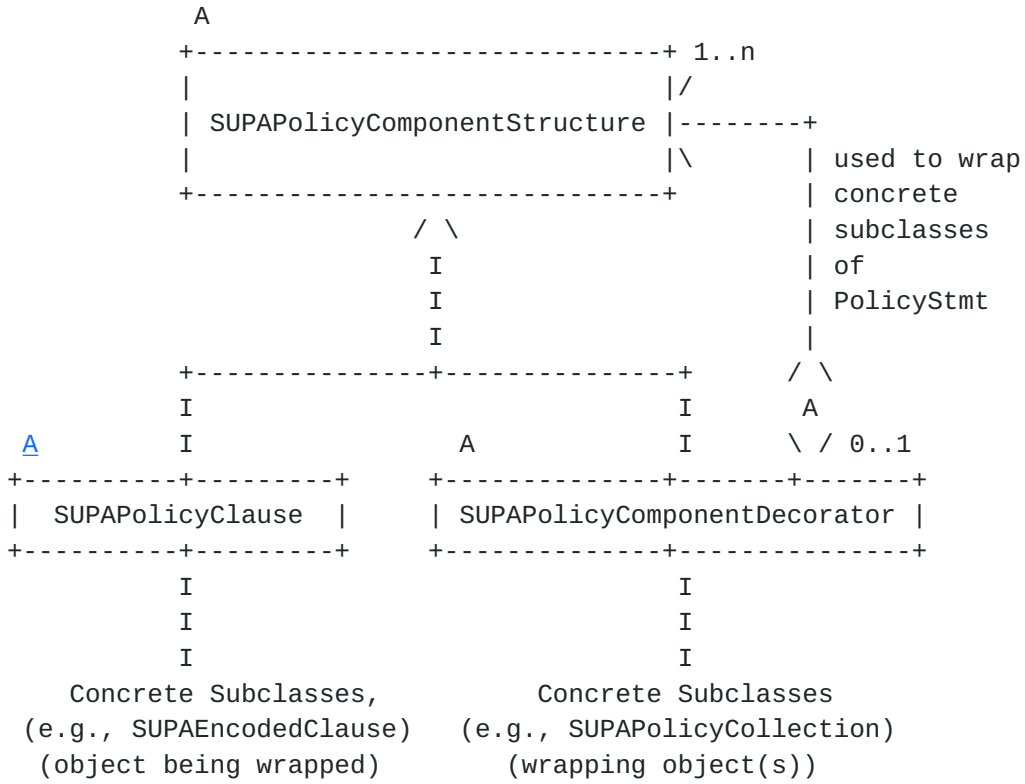


Figure 17. The PolicyComponent Decorator Pattern

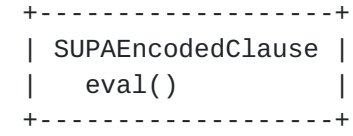
5.9.1. The Decorator Pattern

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).

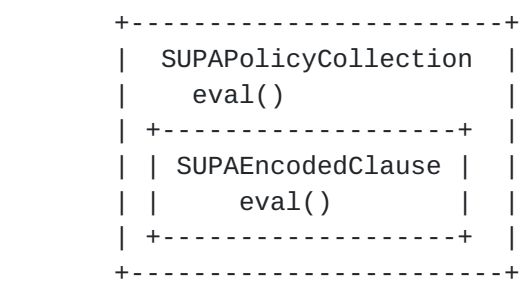
Even better, this means that SUPAPolicyComponentDecorator object instances can add attributes and/or methods to those of the concrete

instance of the chosen subclass of SUPAPolicyClause.

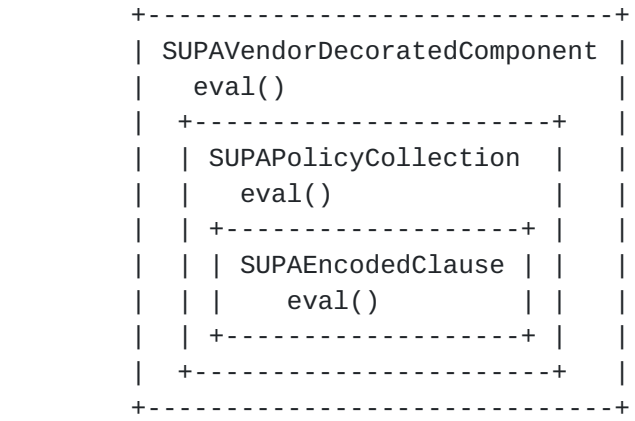
Figure 18 shows how this is done for methods. 18a shows the initial object to be wrapped; 18b shows SUPAPolicyCollection wrapping SUPAEncodedClause; 18c shows SUPAVendorDecoratedComponent wrapping SUPAPolicyCollection. When eval() is called in the outermost object (SUPAVendorDecoratedComponent), it delegates 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 SUPAVendorDecoratedComponent, which executes and returns the final result.



(a) Initial Object



(b) SUPAPolicyCollection "wraps" SUPAEncodedClause



(c) SUPAVendorDecoratedComponent "wraps" SUPAPolicyCollection

Figure 18. Conceptual Depiction of eval() Decorated Method

5.9.2. 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 SUPAHasDecoratedPolicyComponent object that is wrapping it).

5.9.2.1. The Attribute "supaPolCompConstraintEncoding"

This is an optional 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

The latest version of OCL is 2.4, but since this is considered by most the default language for specifying constraints, enumerations 1-3 are dedicated to OCL. 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.

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

This is an optional array of string attributes. Each attribute specifies a constraint to be applied using OCL 2.0. This provides a more rigorous and flexible treatment of constraints than is possible in [RFC3460]. Each string attribute is interpreted according to the value of the supaPolCompConstraintEncoding class attribute.

5.9.3. SUPAPolicyComponentDecorator Relationships

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

5.9.3.1. 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.

5.9.3.2. 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 (and its associated aggregation) to constrain the **relationship** between the concrete subclass of SUPAPolicyComponentDecorator that is wrapping the concrete subclass of SUPAPolicyClause; in contrast, class attributes of SUPAPolicyComponentDecorator (see section 5.9.2) only affect that specific subclass.

5.9.3.2.1. The Attribute "supaDecoratedConstraintsEncoding"

This is a non-negative enumerated integer that defines how to interpret each string in the supaDecoratedConstraints 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

The latest version of OCL is 2.4, but since this is considered by most the default language for specifying constraints, enumerations 1-3 are dedicated to OCL. 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.

5.9.3.2.2. The Attribute "supaDecoratedConstraints[0..n]"

This is an optional 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.

5.9.4. Illustration of Constraints in the Decorator Pattern

The following example will illustrate how the different constraints defined in sections 5.9.2 (class attribute constraints) and section 5.9.3 (relationship constraints) can be used.

Figure 19 builds a simple SUPAPolicyClause that has both types of relationships.

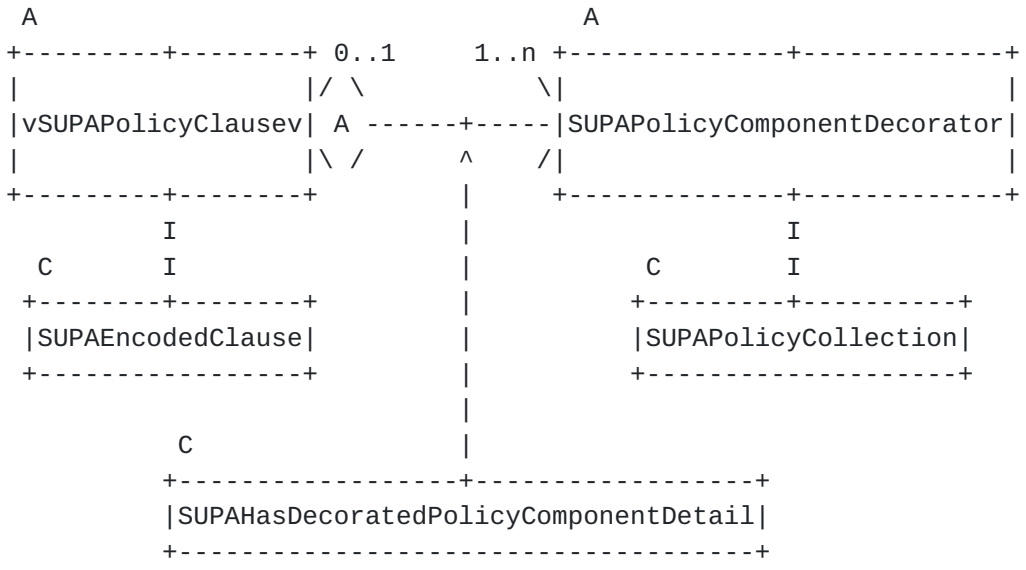


Figure 19. Constraints in the Decorator Pattern

Figure 19 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.

5.10. 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}.

A class diagram is shown in Figure 20.

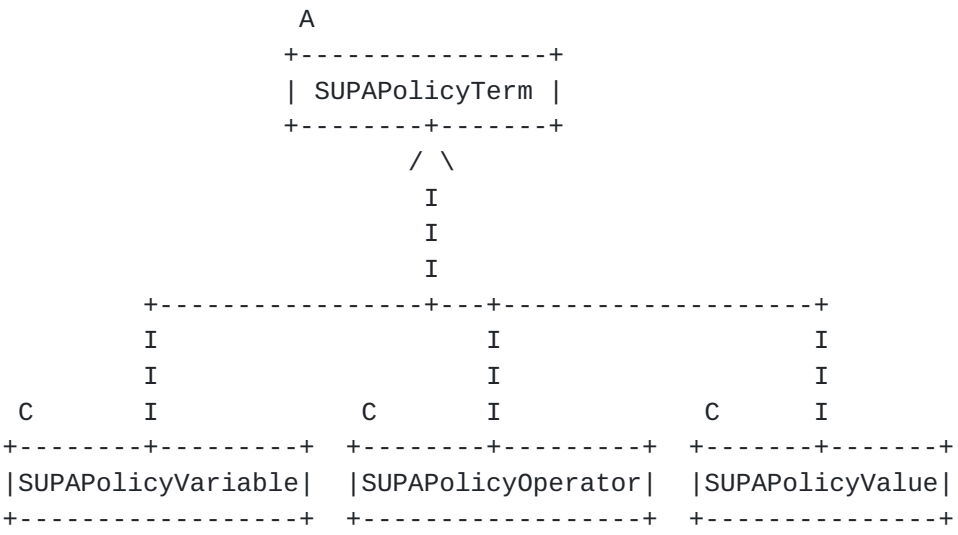


Figure 20. SUPAPolicyTerm Class Hierarchy

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 SUPAVendorDecoratedComponent 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 statements. This is in stark contrast to previous designs (e.g., [RFC3460] and [6]), which both depended on defining a broad set of subclasses of PolicyVariable and PolicyValue. (Note that [4] does not have this generic capability).

5.10.1. 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

5.10.1.1. 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.

5.10.2. SUPAPolicyTerm Relationships

Currently, no dedicated relationships are defined for the SUPAPolicyTerm class (as there is in [RFC3460] and [6]) that aggregate policy terms into any object. 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].

5.11. 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. Note that 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.

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 are 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.

5.11.1. 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).

5.11.1.1. 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:

- PolicyExplicitVariables are for CIM compatibility; note that the CIM does not contain either PolicyVariables or PolicyValues ([4])
- 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).

5.11.1.2. Object Explosion

The above three problems lead to class explosion (recall that in [RFC3060], [RFC3460], and [4], associations are implemented as classes).

In stark contrast to this approach, the approach in this document keeps the idea of the class hierarchy for backwards compatibility, but streamlines the implementation. First, the decorator pattern is an established and very used software pattern (it dates back to at least 1997). Second, the use of a single association class (i.e., SUPAHasDecoratedPolicyComponentDetail) performs many more constraints than is possible in the approaches of [RFC3460] and [4] in a much more flexible manner, due to its role as a decorator of other objects. Third, 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. Fourth, if these constraints are not honored, then there is no mechanism specified to define the statement as incorrectly formed.

5.11.1.3. Specification Ambiguities

There are a number of ambiguities in [RFC2460].

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:

- It uses keys and weak relationships, which are both concepts from relational algebra and thus, not technology-independent
- It has its own proprietary modeling language
- 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).

5.11.2. SUPAPolicyVariable Attributes

Currently, SUPAPolicyVariable defines three generic attributes, as described below.

5.11.2.1. The Attribute "supaPolVarContent"

This is a mandatory string attribute that contains the value of the SUPAPolicyVariable object instance. Its data type is defined by the supaPolVarType class attribute.

5.11.2.2. The Attribute "supaPolVarType"

This is a mandatory non-negative enumerated integer attribute that defines the data type of the supaPolVarContent attribute in this SUPAPolicyVariable object instance. Values include:

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

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).

In general, specific semantics of the above data types are NOT defined in this document, as there are differences in most when converted to a data type of a specific data model. However, constraints can be used to restrict the values that a String, Integer, Floating Point, or DateTime data type may have; this can simplify converting to a data model.

5.11.3. SUPAPolicyVariable Relationships

Currently, no relationships are defiend for the SUPAPolicyVariable class (note that the decorator pattern obviates the need for relationships such as those in [RFC3460] and [6]).

5.12. The Concrete Class "SUPAPolicyOperator"

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

A SUPAPolicyOperator is a mandatory concrete class that defines the type of operator to be applied to a SUPAPolicyClause. The restriction of the type of operator used in a SUPAPolicyClause restricts the semantics that can be expressed in that SUPAPolicyClause (e.g., a "shallow" vs. a "deep" equality comparison; the former compares just the attributes in the specified objects, while the latter compares the entire tree of objects (using the specified objects as the base of both trees).

5.12.1. Problems with the RFC3460 Version

Note that this class is NOT present in either RFC[3060] or [RFC3460]; instead, both hardwire the operator to a "MATCH" function. Quoting from [RFC3460]:

> "A simple condition models an elementary Boolean expression of the form 'variable MATCHes value". However, the formal notation of the SimplePolicyCondition, together with its associations, models only a pair, (<variable>, <value>). The 'MATCH' operator is not directly modeled -- it is implied. Furthermore, this implied 'MATCH' operator carries overloaded semantics [sic]."

In stark contrast to this, SUPA defines a SUPAPolicyOperator as a formal subclass of SUPAPolicyTerm. A single attribute, called supaPolOpType, carries the operator to be applied to the SUPAECAPolicyRule. This has the important advantage of enabling ECA policy rules of varying functionality to be created by a human or a machine. It also removes the ambiguity created by [RFC3460].

5.12.2. SUPAPolicyOperator Attributes

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

5.12.2.1. 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 (shallow)
 2:  Greater than or equal to (shallow)
 3:  Less than (shallow)
 4:  Less than or equal to (shallow)
 5:  Equal to (shallow)
 6:  Not equal to (shallow)
 7:  IN
 8:  NOT IN
 9:  SET
10:  CLEAR
11:  Greater than (deep)
12:  Greater than or equal to (deep)
13:  Less than (deep)
14:  Less than or equal to (deep)
15:  Equal to (deep)
16:  Not equal to (deep)
17:  BETWEEN
```

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.

5.12.3. SUPAPolicyOperator Relationships

Currently, no relationships are defiend for the SUPAPolicyOperator class (note that the decorator pattern obviates the need for relationships such as those in [6]).

5.13. 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

5.13.1. 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).

5.13.1.1. 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 5.11.1.1):

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).

5.13.1.2. 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).

5.13.1.3. 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.

5.13.1.4. 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:

- It uses keys and weak relationships, which are both concepts from relational algebra and thus, not technology-independent
- It has its own proprietary modeling language
- It contains a number of concepts that are not defined in UML (including overriding keys for subclasses)

5.13.1.5. 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.

5.13.1.6. 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 statement; 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.

5.13.2. SUPAPolicyValue Attributes

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

5.13.2.1. 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 supaPolValType class attribute.

5.13.2.2. The Attribute "supaPolValType"

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: 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.

5.13.3. SUPAPolicyValue Relationships

Currently, no relationships are defiend for the SUPAPolicyValue class (note that the decorator pattern obviates the need for relationships such as those in [6]).

5.14. The Concrete Class "SUPAVendorDecoratedComponent"

A SUPAVendorDecoratedComponent enables a custom, vendor-specific object to be defined and used in a SUPAPolicyClause. This should not be confused with the SUPAEncodedClause class. The SUPAVendorDecoratedComponent class represents a single, atomic, that is 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. Note that this object is not present in [RFC3060], [RFC3460], [4], [5], or [6].

5.14.1. SUPAVendorDecoratedComponent Attributes

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

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

This is a mandatory attribute that defines an array of strings. This array contains the value(s) of the SUPAVendorDecoratedComponent object instance. Its data type is defined by the supaVendorDecoratedFormat class attribute.

5.14.1.2. The Attribute "supaVendorDecoratedCompFormat"

This is a mandatory string 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:  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.

5.14.2. SUPAVendorDecoratedComponent Relationships

Currently, no relationships are defiend for the SUPAVendorDecoratedComponent class (note that the decorator pattern obviates the need for relationships such as those in [6]).

5.15. The Concrete Class "SUPAPolicyCollection"

A SUPAPolicyCollection enables a collection (e.g., set, bag, or other, more complex, collections of elements) to be defined and used as part of a SUPAPolicyClause.

5.15.1. 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 EventA followed by either one of event B or two of 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 or 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.

5.15.2. 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; this is supported, 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.

5.15.3. SUPAPolicyCollection Attributes

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

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

This is a mandatory attribute that defines an array of strings. This array defines the content of this SUPAPolicyCollection instance.

5.15.3.2. The Attribute "supaPolCollectionDataType"

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

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

5.15.3.3. The Attribute "supaPolCollectionFunction"

This is a mandatory 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

5.15.3.4. 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.

5.15.3.5. The Attribute "supaPolCollectionType"

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

Sets have a number of important functions:

- 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 witin a given dictionary.

5.15.4. SUPAPolicyCollection Relationships

Currently, no relationships are defiend for the SUPAVendorDecoratedComponent class (note that the decorator pattern obviates the need for relationships such as those in [6]).

5.16. The Concrete Class "SUPAPolicySource"

This is an optional class that defines the set of managed entities that authored, or are otherwise responsible for, this SUPAPolicyClause. 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.

5.16.1. SUPAPolicySource Attributes

Currently, no attributes are defined for the SUPAPolicySource class.

5.16.2. SUPAPolicySource Relationships

This section defines the relationships of the SUPAPolicySource class.

5.16.2.1. The Aggregation "SUPAHasPolicySource"

This is an optional association that defines the set of SUPAPolicySource objects that are associated with this particular SUPAPolicyStructure object. The multiplicity of this relationship is defined as 0..n on the aggregate (i.e., SUPAPolicyStructure) side, and 0..n on the part (i.e., SUPAPolicySource) side. This means that this relationship is optional. The semantics of this aggregation are implemented using the SUPAHasPolicySourceDetail association class.

5.16.2.2. The Association Class "SUPAHasPolicySourceDetail"

This is an optional association class that defines the semantics of the SUPAHasPolicySource aggregation. It is typically used to constrain the types of SUPAPolicyStructure objects that can aggregate a particular set of SUPAPolicySource objects.

5.16.2.2.1. 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 specific SUPAPolicyStructure object.

5.16.2.2.2. The Attribute "supaPolicySrcIsTrusted"

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

5.17. The Concrete Class "SUPAPolicyTarget"

A SUPAPolicyTarget is a set of managed entities that a SUPAPolicy is applied to. This is determined by two conditions.

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 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.

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

Figure 12 shows a class diagram of the SUPAPolicyTarget.

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.

5.17.1. SUPAPolicyTarget Attributes

Currently, no attributes are defined for the SUPAPolicyTarget class.

5.17.2. SUPAPolicyTarget Relationships

This section defines the relationships of the SUPAPolicyTarget class.

5.17.2.1. The Aggregation "SUPAHasPolicyTarget"

This is an optional aggregation that defines the set of SUPAPolicyTarget objects that can be attached to this particular SUPAPolicyStructure object. This defines the set of entities that will be operated on by this particular SUPAPolicyStructure object. The multiplicity of this relationship is defined as 0..1 on the aggregate (i.e., SUPAPolicyStructure) side, and 0..n on the part (i.e., SUPAPolicyTarget) side. The semantics of this aggregation are implemented using the SUPAIsTargetOfDetail association class.

5.17.2.2. The Association Class "SUPAHasPolicyTargetDetail"

This is an optional concrete association class that defines the semantics of the SUPAHasPolicyTarget aggregation. This enables the attributes and relationships of the SUPAHasPolicyTargetDetail association class to be used to constrain which SUPAPolicyTarget objects can be operated on by which SUPAPolicyStructure objects.

5.17.2.2.1. 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 specific SUPAPolicyStructure object.

5.17.2.2.2. 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.

5.18. 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 class defines a hierarchy of model elements that are used to define different types of metadata that can be applied to policy objects that are subclasses of the SUPAPolicyObject class. 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. This is an abstract class, and is meant to be subclassed to include more detailed metadata attributes and relationships, as appropriate to the needs of the policy management application.

5.18.1. SUPAPolicyMetadata Attributes

This section defines the attributes of the SUPAPolicyMetadata class. This class is the base class of the metadata hierarchy for policy objects.

5.18.1.1. The Attribute "supaPolMetadataDescription"

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

5.18.1.2. 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 supaPolMetadataIDFormat, 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.

5.18.1.3. The Attribute "supaPolMetadataIDFormat"

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 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 supaPolMetadataIDFormat attribute is mapped to the following values:

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

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

5.18.1.4. The Attribute "supaPolicyName"

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

5.18.2. SUPAPolicyMetadata Relationships

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

5.18.3. The Abstract Class "SUPAHasPolicyMetadataDetail"

This is a mandatory abstract 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.

5.18.3.1. 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.

5.18.3.2. 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

The latest version of OCL is 2.4, but since this is considered by most the default language for specifying constraints, enumerations 1-3 are dedicated to OCL. 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.

5.18.3.3. The Attribute "supaPolMetadataPolicyConstraints[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].

5.19. The Concrete Class "SUPAPolicyConcreteMetadata"

This class will be defined in the next release of this document.

5.20. The Abstract Class "SUPAPolicyMetadataDecorator"

This class will be defined in the next release of this document.

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 21 (only new EPRIM subclasses and their GPIM superclasses are shown).

Figure 21. The EPRIM Class Hierarchy

Specifically, the EPRIM specializes the SUPAPolicyStructureAtomic 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. Specifically, 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. In addition, 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.

Note that the EPRIM only defines new (sub)classes that are a subclass of SUPAPolicyStructure or SUPAPolicyComponentStructure. This ensures that the semantics of the GPIM are not changed while providing new functionality for ECA Policy Rules.

The overall strategy for refining the GPIM is as follows:

- SUPAECAPolicyRule is defined as a subclass of the GPIM SUPAPolicyStructureAtomic class
- A SUPAECAPolicyRule has event, condition, and action clauses
  - Conceptually, this can be viewed as three aggregations between the SUPAECAPolicyRule and each clause
  - Each aggregation uses an instance of a concrete subclass of SUPAPolicyClause; this can be a SUPABooleanClause (making it ECA-specific) or a SUPAEncodedClause (making it generic in nature)
  - Either of the above object instances may be decorated with zero or more concrete subclasses of the SUPAPolicyComponentDecorator class
- An optional set of GPIM SUPAPolicySource objects can be defined to represent the authoring of a SUPAECAPolicyRule
- An optional set of GPIM SUPAPolicyTarget objects can be defined to represent the set of managed entities that will be affected by this SUPAECAPolicyRule
- 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:

- Define three types of SUPABooleanClauses, one each for the event, condition, and action clauses that make up a SUPAECAPolicyRule
  - 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
  - Note that compound Boolean clauses may be formed using one or more SUPABooleanComposite objects with one or more SUPABooleanAtomic objects
- Define a SUPAPolicyCollectionComponent, which is used to aggregate a set of SUPAECAComponents, and complete the clause using an appropriate SUPAPolicyOperator and a corresponding SUPAPolicyValue or SUPAPolicyVariable
- 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 use it as part of a SUPAPolicyClause
- Create a new subclass of SUPAPolicyComponentDecorator that provides ECA-specific functionality, and use that to decorate a SUPAPolicyClause
- Create a new concrete subclass of subclass of SUPAECAPolicyRule 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:

- 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).
- 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).
- The action clause contains a set of actions

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.

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 construction of a SUPAECAPolicyRule is shown in Figure 22, and is explained in further detail in Section 6.4.

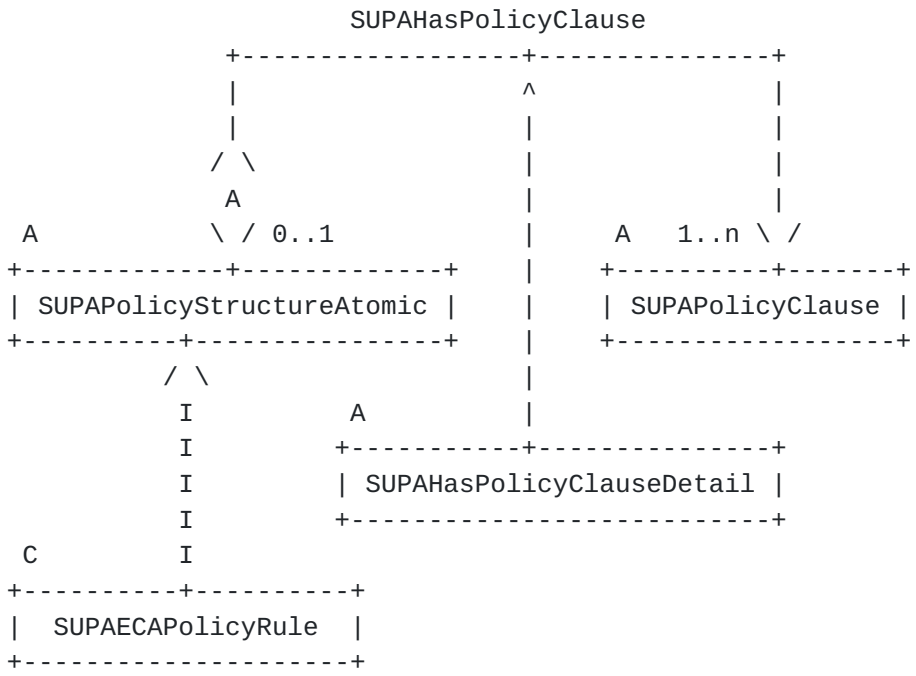


Figure 22. SUPAECAPolicyRule Clauses

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 5.9.

Each of the three types of clauses is 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, a valid event clause could be:

```
"3 A-events AND ((NOT B-event) OR 2 C-events)"
```

In either case, the output of all three clauses is either TRUE or FALSE; this facilitates combining and chaining ECAPolicyRules.

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 the SUPAECAPolicyRule MAY have PolicyMetadata 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. This is shown in Figure 23.

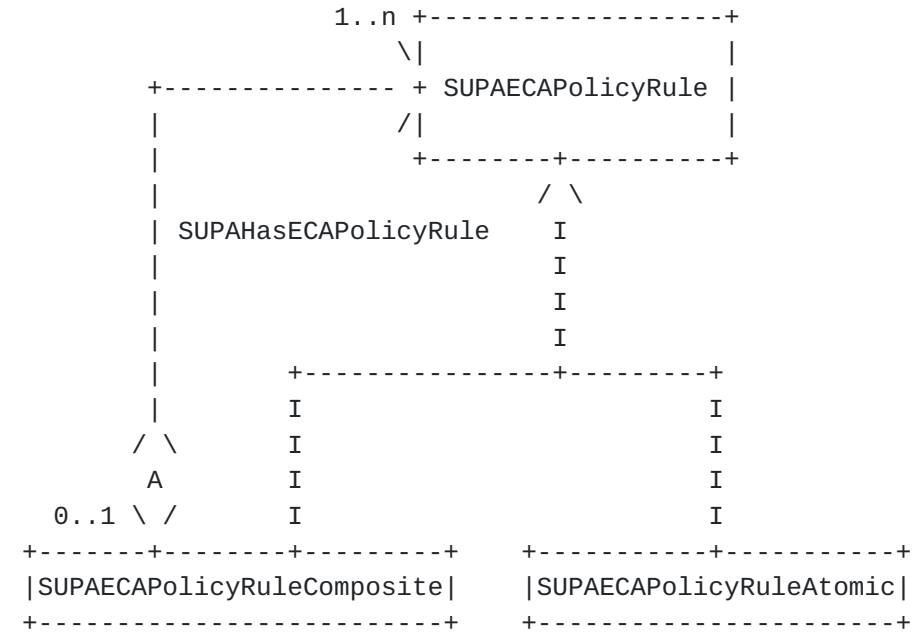


Figure 23. The Composite Pattern Applied to a SUPAECAPolicyRule

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.

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 "supaECAPolicyIsMandatory"

This is an optional Boolean attribute. If the value of this attribute is true, then this SUPAECAPolicyRule MUST be executed (i.e., its Event and Condition clauses are irrelevant, and the Action(s) specified in the Action clause MUST be executed). A default value of FALSE MAY be assigned.

6.4.1.2. The Attribute "supaECAPolicyPriority"

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.

6.4.1.3. 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.

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.

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 actions in a SUPAECAPolicyRuleComposite object are evaluated. Values include:

0: undefined
1: execute the first action and then terminate
2: execute only the highest priority action(s)
3: execute all actions regardless of their execution status
4: execute all actions until one or more actions fail

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

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

Then, if the supaECAEvalStrategy attribute value equals:

0: an error is issued
1: only Action A is executed
2: only Actions C and D are executed
3: all actions are executed, regardless of any failures
4: all actions are executed until a failure is detected, and then execution terminates

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 SUPHasECAPolicyRuleDetail association class.

6.6.3. The Association Class "SUPHasECAPolicyRuleDetail"

This is an optional association class, and defines the semantics of the SUPHasECAPolicyRule aggregation. This enables the attributes and relationships of the SUPHasECAPolicyRuleDetail 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 statement consisting of a standard structure in the form of a PolicyVariable, a PolicyOperator, and a PolicyValue. For example, this enables the following Boolean clause to be defined:

```
Foo >= Bar AND Baz
```

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

Note that in this approach, the PolicyVariable and PolicyValue terms are defined as an appropriate subclass of the SUPAPolicyComponentDecorator class; it is assumed that the PolicyOperator 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.

The addition of a negation in the above statement is provided by the supaBoolIsNegated class attribute of the SUPABooleanClause class. Individual terms of a Boolean clause can be negated by using the supaTermIsNegated Boolean attribute in the SUPAPolicyTerm class (see section 5.10).

A PolicyStatement is in Conjunctive Normal Form (CNF) if it is a conjunction (i.e., a sequence of ANDed terms), where each term is a disjunction (i.e., a sequence of ORed terms). Every statement that consists of a combination of AND, OR, and NOT operators can be written in CNF.

A PolicyStatement is in Disjunctive Normal Form (DNF) if it is a disjunction (i.e., a sequence of ORed terms), where each term is a conjunction (i.e., a sequence of ANDed terms). Every statement that consists of a combination of AND, OR, and NOT operators can be written in DNF.

The construction of more complex clauses, which consist of a set of simple clauses in conjunctive or disjunctive normal form (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 two attributes, which are defined in the following subsections.

6.7.1.1. The Attribute "supaBoolIsCNF"

This is a mandatory Boolean attribute. If the value of this attribute is TRUE, then this SUPABooleanClause is in CNF form. Otherwise, it is in DNF form.

6.7.1.2. The Attribute "supaBoolIsNegated"

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.

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. Put another way, 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.

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.

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.

6.9.1. SUPABooleanClauseComposite Attributes

A single attribute is currently defined for the SUPABooleanClauseComposite class, and is described in the

following subsection.

6.9.1.1. The Attribute "supaPolStmtBindValue"

This is an optional non-zero integer attribute, and defines the order in which terms bind to a clause. For example, the Boolean statement "((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.

6.9.2. 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 Concrete Class "SUPAHasBooleanClauseDetail"

This is a mandatory 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 at this time.

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.

6.10.1. SUPAECAComponent Attributes

No attributes are currently defined for this class.

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 any important occurrence in time of a change in the system being managed, and/or in the environment of the system being managed.

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.

6.11.1.4. The Attribute "supaPolicyEventDataType"

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

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

Enumerations 1-4 are used to provide a reference to an event object. Enumeration 5 defines the Event as a temporal value. Enumerations 6-11 are used to express the Event as a string.

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

This is a mandatory array of string attributes that contain the content of this SUPAPolicyEvent object (or set of objects).

This version of this document enables either the text describing the set of events that should be contained in the event clause of a SUPAPolicyRule or a set of event objects. The former is useful for describing common conditions, such as "if the time is before 6pm" or "if three events of type A are received and then a single event of type B or type C is received". The latter is useful for treating the event as an object, and filtering on the attributes of the event.

In the former case, the text may be entered as one or more strings. In the latter case, each string in the array is a reference to an event object.

This attribute works with another class attribute, called supaPolicyEventDataType, 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.

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.

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 "supaPolicyConditionDataType"

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

6.12.1.2. The Attribute "supaPolicyConditionData"

This is a mandatory string attribute that contains the content of this SUPAPolicyCondition object.

This attribute works with another class attribute, called supaPolicyConditionDataType, 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.

6.12.2. SUPAPolicyEvent Relationships

No relationships are currently defined for this class.

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 the SUPAECAPolicyRule container, and any applicable SUPAPolicyMetadata objects.

6.13.1. SUPAPolicyAction Attributes

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

6.13.1.1. The Attribute "supaPolicyActionDataType"

This is a mandatory non-zero enumerated integer attribute, and defines the data type of the supaPolicyActionData attribute. 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
```

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.

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

This is a mandatory string attribute that contains the content of this SUPAPolicyAction object.

This attribute works with another class attribute, called supaPolicyConditionDataType, 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.

6.13.1.3. The Attribute "supaPolicyActionResponse"

This is a mandatory non-negative enumerated integer attribute that defines the execution status of this particular SUPAPolicyAction. Values include:

0: undefined
1: executed with no errors
2: executed with at least one error
3: failed to execute

6.13.2. SUPAPolicyAction Relationships

No relationships are currently defined for this class.

7. Examples

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. Acknowledgments

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

11. References

This section defines normative and informative references for this document.

11.1. Normative References

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

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

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

[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.

11.2. 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

[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., editors, "SUPA Proposition", IETF Internet draft, draft-klyus-supa-proposition-01, July 4015

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

[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

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

Jason Coleman
Cisco Systems
124 Copper Lake Lane
Georgetown Tx 78628
Email: routerjockey@me.com

Appendix A. Mathematical Logic Terminology and Symbology

Appendix B. SUPA Logic Statement Information Model

Appendix C. Brief Analyses of Previous Policy Work