ETSI GR ENI 003 V1.1.1 (2018-05)

GROUP REPORT

Experiential Networked Intelligence (ENI); Context-Aware Policy Management Gap Analysis

Reference

DGR/ENI-003

Keywords

management, network, policy management

Contents

Foreword

This Group Report (GR) has been produced by ETSI Industry Specification Group (ISG) Experiential Networked Intelligence (ENI).

Modal verbs terminology

In the present document "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

Introduction

A critical foundation of experiential networked intelligence is context- and situation-awareness. The present document will analyse work done in various SDOs on policy management in general, and context-aware policy management specifically, to determine what work can be reused from external SDOs, and what work needs to be developed in ENI. When gaps are found on existing interfaces that have been developed by other SDOs and ISG ENI needs to reuse, then the recommendation on how these gaps should be filled will be discussed in co-operation with the SDO that defined these interfaces within Phase 1 and beyond. The requirements documented in the present document will be considered during the architecture design work.

1 Scope

The present document analyses the work done in various SDOs and open source consortia on policy-based modelling. This information will be used to develop a specification for a context-aware, policy-based management model and architecture for enhancing the operator experience through the use of network intelligence.

2 References

2.1 Normative references

Normative references are not applicable in the present document.

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] IETF draft-ietf-supa-generic-policy-info-model-03: "Generic Policy Information Model for Simplified Use of Policy Abstractions (SUPA)", May 30, 2017.

[i.2] Strassner, J.: "Policy-Based Network Management", Morgan Kaufman, ISBN 978-1558608597, September 2003.

[i.3] Liskov, B.H.: Wing, J.M., "A Behavioral Notion of subtyping", ACM Transactions on Programming languages and Systems 16 (6): 1811 - 1841, 1994.

[i.4] Martin, R.C.: "Agile Software Development, Principles, Patterns, and Practices", Prentice-Hall, 2002, ISBN: 0-13-597444-5.

[i.5] Strassner, J.: editor: "MEF Technical Specification: Policy-Driven Orchestration", Call for Comments v0.7, August 2017.

[i.6] Riehle, D.: "Composite Design Patterns", Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97), ACM Press, 1997, pages 218-228.

[i.7] 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.

[i.8] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Design Patterns - Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994, ISBN 0-201-63361-2.

[i.9] 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.

[i.10] MEF: "Lifecycle Service Orchestration Architecture: Reference Architecture and Framework", MEF 55, March, 2016.

[i.11] MEF: "Service Orchestration Functionality", Call for Comments v0.7, August 2017.

[i.12] MEF: "Policy Driven Orchestration Kickoff v3".

NOTE: Available at https://wiki.mef.net/display/MTA/PDO+Contributions.

[i.13] Dey, A.: "Providing architectural support for building context-aware applications". Ph.D. Thesis (2000).

[i.14] MEF: "MEF Core Model", Call for Comments v0.95, January 2018.

[i.15] ETSI ISG ENI(17)0002-014r3: "Improved Operator Experience through Experiential Networked Intelligence (ENI)", first edition, May 2017.

[i.16] IETF draft-ietf-supa-generic-policy-data-model-04: "Generic Policy Data Model for Simplified Use of Policy Abstractions (SUPA)", June 18, 2017.

[i.17] Standford Encyclopedia of Philosophy: "Deontic Logic".

NOTE: Available at https://plato.stanford.edu/entries/logic-deontic/.

[i.18] Standford Encyclopedia of Philosophy: "Modal Logic".

NOTE: Available at https://plato.stanford.edu/entries/logic-modal/.

[i.19] IEFT: "Simplified Use of Policy Abstractions (supa)".

NOTE: Available at https://datatracker.ietf.org/wg/supa/about/.

[i.20] IETF draft-ietf-supa-policy-based-management-framework-03: "SUPA Policy-based Management Framework", July 2017.

[i.21] IETF draft-cheng-supa-applicability-01: "Applicability of SUPA", March 2017.

[i.22] TM Forum: "Information Framework (SID); Common Business Entities - Policy", GB922 Policy, Release 14.5.1, March 2015 (part of Release 17.0).

[i.23] TM Forum: "Information Framework (SID); Common Business Entities - Root Business Entities", GB922 Root, Release 17.0.0, June 2017 (part of Release 17.0).

[i.24] Strassner, J.: "Using the MEF Core Model in ONAP", December 2017.

[i.25] MEF PDO CfC: "Policy-Driven Orchestration", v0.8, February 2018.

[i.26] MEF: "Lifecycle Service Orchestration (LSO): Reference Architecture and Framework", MEF 55, March 2016.

[i.27] IETF RFC 3060: "Policy Core Information Model -- Version 1 Specification".

3 Definitions and abbreviations

3.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

abstraction: process of focusing on the important characteristics and behaviour of a concept, and realizing this as a set of one or more elements in an information or data model

NOTE: When applied to modelling, it defines a generic set of characteristics and behaviours for a class that all of its subclasses inherit.

action: set of operations that may be performed on a set of managed entities. It represents a transformation or processing in the system being modelled

NOTE: An Action either maintains the state, or transitions to a new state, of the targeted managed entities. The execution of an Action may be influenced by applicable attributes and metadata [i.25].

capability: set of features that are available from a component

NOTE: These features may, but do not have to, be used. All Capabilities should be announced through a dedicated Interface.

condition: set of attributes, features, and/or values that are to be compared with a set of known attributes, features, and/or values in order to determine what decision to make

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

declarative policy: type of policy that uses statements to express the goals of the policy, but not how to accomplish those goals

NOTE 1: State is not explicitly manipulated, and the order of statements that make up the policy is irrelevant.

NOTE 2: In the present document, Declarative Policy will refer to policies that execute as theories of a formal logic [i.25].

event: anything of importance to the management system (e.g. a change in the system being managed and/or its environment) occurring on a time-axis

formal logic: use of inference applied to the form, or content, of a set of statements

NOTE: The logic system is defined by a grammar that can represent the content of its sentences, so that mathematical rules may be applied to prove whether the set of statements is true or false [i.25].

formal methods: set of mathematical theories, such as logic, automata, graph or set theory, and provide associated notations for describing and analysing systems

functional block: modular unit that defines the properties, behaviour, and relationships of a part of a system. Some functional blocks may also define relationships to other functional blocks outside of its enclosing system

imperative policy: type of policy that uses statements to explicitly change the state of a set of targeted objects

NOTE 1: The order of statements that make up the policy is explicitly defined.

NOTE 2: In the present document, Imperative Policy will refer to policies that are made up of Events, Conditions, and Actions [i.25].

information model: 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

intent policy: type of policy that uses statements in a natural language to express the goals of the policy, but not how to accomplish those goals

NOTE 1: Each statement in an Intent Policy may require the translation of one or more of its terms to a form that another managed functional entity can understand [i.25].

NOTE 2: In the present document, Intent Policy will refer to policies that do not execute as theories of a formal logic. They typically are expressed in a restricted natural language and require a mapping to a form understandable by other managed functional entities.

Lifecycle Service Orchestration (LSO): open and interoperable automation of management operations over the entire lifecycle of Layer 2 and Layer 3 Connectivity Services

NOTE: This includes fulfilment, control, performance, assurance, usage, security, analytics and policy capabilities, over all the network domains that require coordinated management and control, in order to deliver the offered Service [i.26].

LSO reference architecture: high-level functional architecture that characterizes the management and control domains and entities that make up a system, and the interfaces among them, to enable cooperative orchestration of offered Services

managedEntity: manageable object that may be related to a Product, Service, and/or Resource

NOTE: A ManagedEntity has the following common semantics:

1) each has the potential to be managed;

2) each can be associated with at least one ManagementDomain;

3) each is related to Products, Resources, and/or Services of the system being managed [i.14].

management: set of procedures that are responsible for describing, organizing, controlling access to, and managing the lifecycle needs of information and entities of an organization

management abstraction: abstraction used for management purposes

managementDomain: domain whose contents are governed using a common set of management mechanisms

NOTE: A ManagementDomain is a type of ManagedEntity that has 3 key characteristics:

1) it has a set of administrators defined to perform management operations on the ManagedEntities that it contains;

2) it defines a set of applications that are responsible for different governance operations, such as monitoring, configuration, and so forth;

3) it defines a common set of management mechanisms, such as policy rules, that are used to govern the behaviour of ManagedEntities contained in the ManagementDomain [i.14].

metadata: set of objects that contains prescriptive and/or descriptive information about the object(s) to which it is attached [i.14]

NOTE: While metadata can be attached to any information model element, the present document only considers metadata attached to classes and relationships.

meta-policy: policy that governs that operation, administration, and/or management of another set of policies

orchestration: set of processes that collectively automate the management and control of digital information systems

NOTE: Orchestration processes coordinate the actions of disparate systems and functions, ensuring that they all act towards a common set of goals.

party: abstract class that represents either an individual person or a group of people

NOTE: A Party may take on zero or more PartyRoles. A group of people can also be structured as an organization made up of organizational units [i.14].

PartyRole: abstract class, and specializes Role. It represents a set of unique behaviours played by a Party in a given context [i.14]

pattern: reusable, object-oriented framework to a commonly occurring problem

policy: set of rules that are used to manage and control the changing and/or maintaining of the state of one or more managed objects

Reference Point (RP): logical point of interaction between specific ManagedEntities

resource: Resource provides capabilities that may be consumed by different internal and external users [i.14].

NOTE: In addition, a Resource may consume other Resources. A Resource has a distinct state. Resources are typically limited in quantity and/or availability. Resources may be logical or virtual in nature [i.14].

service: service represents functionality that can be used by different internal and external users (e.g. a management system and a Customer, respectively) for different purposes [i.14]

NOTE: Services may be consumed by other Services, but not by Resources. A Service has a distinct state.

Service Orchestration Functionality (SOF): set of functional blocks and associated processes to translate requests from business and customer applications to a form that the infrastructure of the Service Provider can understand, and similarly, to translate responses from the infrastructure of the Service Provider to business and customer applications

NOTE: It also manages and controls the functional components that make up the infrastructure of the Service Provider.

3.2 Abbreviations

For the purposes of the present document, the following abbreviations apply:

AI	Artificial Intelligence
ANIMA	Autonomic Networking Integrated Model and Approach
API	Application Programming Interface
CLI	Command Line Interface
DSL	Domain-Specific Language
ECA	Event-Condition-Action
ENI	Experiential Networked Intelligence
EPRIM	Eca Policy Rule Information Model
FOL	First Order Logic
FTP	File Transfer Protocol
GBP	Group Based Policy
GPIM	Generic Policy Information Model
GS	Group Specification
ICM	Infrastructure Control and Management
IP	Internet Protocol
LDAP	Lightweight Directory Access Protocol
LF	Linux Foundation®
LSO	Lifecycle Service Orchestration
LSO RA	LSO Reference Architecture
MANO	MANagement and Orchestration
MCM	MEF Core Model
MEC	Multi-access Edge Computing
MEF	MEF Forum
NFVRG	Network Functions Virtualisation Research Group
ODL	OpenDayLight project
ONAP	Open Network Automation Platform
ONF	Open Networking Foundation
ONOS	Open Network Operating System
PBM	Policy-Based Management
PDO	Policy-Driven Orchestration
RA	Reference Architecture
RDBMS	Relational DataBase Management System
RP	interface Reference Point
SAP	SoftwAre Product for enterprise resource planning
SDN	Software Defined Networks
SID	Shared Information and Data model
SLS	Service Level Specification
SNMP	Simple Network Management Protocol
SOF	Service Orchestration Functionality
SUPA	Simplified Use of Policy Abstractions
SVGA	Super Video Graphics Array
TM	TeleManagement
TMF	TeleManagement Forum
UML	Unified Modeling Language
WG	Working Group

4 Introduction and Approach

4.1 Base Assumptions

ENI has recommended the use of *context-aware* policy management in its white paper. This is because one of the most important goals of ENI is to respond to dynamic changes. This is best handled by defining the concept of *context,* and then modelling context in the system, so that the Policy Management system can detect and respond to changes in context. In this respect, the goal of ENI is to detect changes in context, and as a result, to change the working set of policies being used. This causes the behaviour of the system being managed to be adjusted to follow changes in context (according to appropriate business goals and other factors, of course) in a closed loop manner.

One of the most popular definitions of context is: "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves" [i.13]. This definition has a number of shortcomings when applied to modern system and network management, as detailed in [i.9]. Therefore, the definition of context used in ENI is:

> *"The Context of an Entity is a collection of measured and inferred knowledge that describe the state and environment in which an Entity exists or has existed".*

This definition emphasizes two types of knowledge - facts (which can be measured) and *inferred* data, which results from machine learning and reasoning processes applied to past and current context. It also includes context history, so that current decisions based on context may benefit from past decisions, as well as observation of how the environment has changed [i.9].

4.2 Introduction to Policy Management

4.2.1 Policy Definition

The purpose of policies is to ensure that consistent decisions are made governing the behaviour of a system. More specifically, [i.5] and [i.12] provide the following definition of Policy:

> *"Policy is a set of rules that is used to manage and control the changing and/or maintaining of the state of one or more managed objects."*

Organizations are policy-driven entities. Policy is a natural way to express rules and restrictions on behaviour, and then automate the enforcement of those rules and restrictions. However, the number of policies can be very large (e.g. 100 000+), and the relationships between policies can be complex. In addition, policy can change *contextually.* For example, different actions can be taken based on type of connection, time of day, and network state.

4.2.2 Uses of Policy Management

There is a distinct difference between Policies that operate in a hierarchy of systems (e.g. north-south) vs. Policies that are exchanged between systems (e.g. east-west). The former is used to control behaviour, while the latter is used to represent and possibly negotiate behaviour. This is shown in Figure 4.2-1.

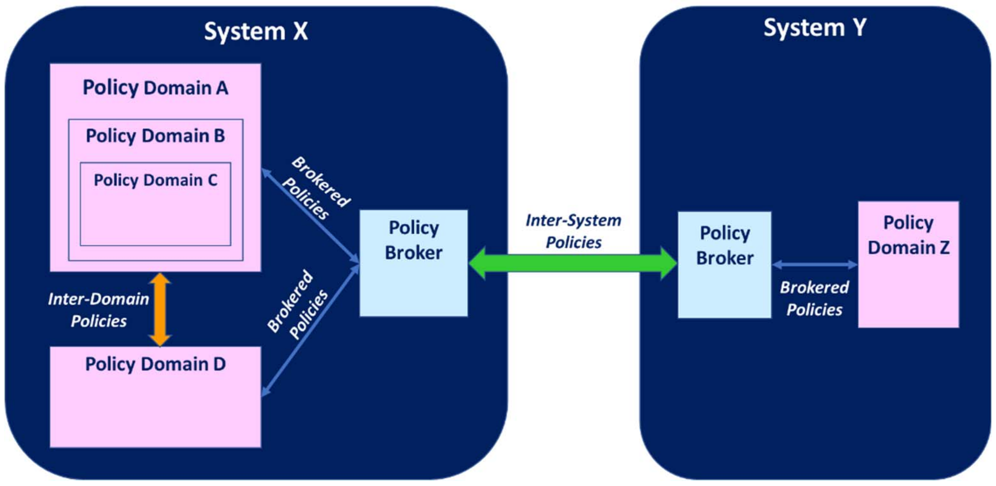


Figure 4.2-1: Policies Used in Management and Control vs. Negotiation

Systems X and Y are different systems (e.g. a Service Provider and a Content Provider) that wish to interact. Inter-System Policies can be used to negotiate behaviour, and use a functional block called a Policy Broker (see clause 4.11.4) to communicate with other Systems that use policy. For example, a Service Provider could specify that a particular service should include HD video; however, the Service Provider cannot tell the Content Provider how to build that service. Furthermore, the use of Policies enables the Content Provider to respond to the request and offer different behaviours based on the current context (e.g. upsell to 4K video, downgrade to SVGA, and so forth). This enables the behaviour of the service to be agreed upon by the Providers, and hence, provide a projected experience for the end user.

Figure 4.2-1 shows three types of Policies, defined as follows:

- Intra-Domain: a Policy exchanged between two domains that are both contained in the same higher domain (e.g. Policies exchanged between domains A and B, or domains B and C).
- Inter-Domain: a Policy exchanged between two domains that are NOT contained in the same higher domain (e.g. between domains A and D, or between B and D or C and D).
- Inter-System: a Policy exchanged between two systems.

The first case (intra-domain) defines Policies according to a strict hierarchy. Policies from the outermost domain should be obeyed by all of its contained domains. Hence, in order for such a Policy to be validated, there should not be any conflict between a newly added (or edited) Policy and all Policies that are in that domain, or in any containing domains.

The second case (inter-domain) defines Policies that are in sibling domains in the same system. The system may use one or more functional blocks to perform conflict detection and remediation.

The third case (inter-system) defines Policies that are in sibling systems. This means that each system should ensure that its policies are compatible with Policies in the systems that it is communicating with.

In general, model-driven engineering uses the abstractions of a *component,* a *module,* and a *system.* A component is the most granular level of reuse in a software-intensive system. Architecturally, it encapsulates a set of related functions, and offers services to the rest of the system via interfaces. This abstraction decouples the functionality offered by a component from its implementation.

A module is a set of related components that are assembled to form a higher-level function.

A system is a set of modules that performs a complete, working application.

Since Policies can be defined at any of these levels, and because Policy depends on context, a Policy Broker is required to communicate the semantics, constraints, and metadata associated with exchanging Policies between systems.

System X is a Carrier. System Y is a Content Provider. In each case, the brokered Policies are a proper subset of the policies from the outermost Domain (i.e. the Policies from a (hierarchical) domain represent a set of capabilities, and the brokered Policies are those policies that the containing System chooses to expose).

Within System X, four different domains are shown. Domain D is a stand-alone domain, while Domain A contains Domain B, which contains Domain C. Specifically, this means:

- Domain A can create new policies at any time.
- Domain B can create new policies as long as they do not conflict with policies of Domain A.
- All Domain B policies should not conflict with policies of Domain A.
- Domain C can create new policies as long as they do not conflict with policies of Domain A or Domain B.
- All Domain C policies should not conflict with policies of Domain A or Domain B.
- Since Domain D and Domain A are sibling domains, their policies can be independent of each other. However, care should be taken to ensure that conflicting policies of Domains D and A are not applied to the same entity at the same time.

4.2.3 Managing and Controlling Behaviour Using Policy

Two important types of behavioural policies are authorization and obligation policies. Authorization policies define what the target of a policy is permitted or not permitted to do. Obligation policies define what the management engine should or should not do, and hence, guide the decision-making process. This is based on *deontic logic* [i.17] and [i.18]. The difference between these two types of policies is shown in Figure 4.2-2.

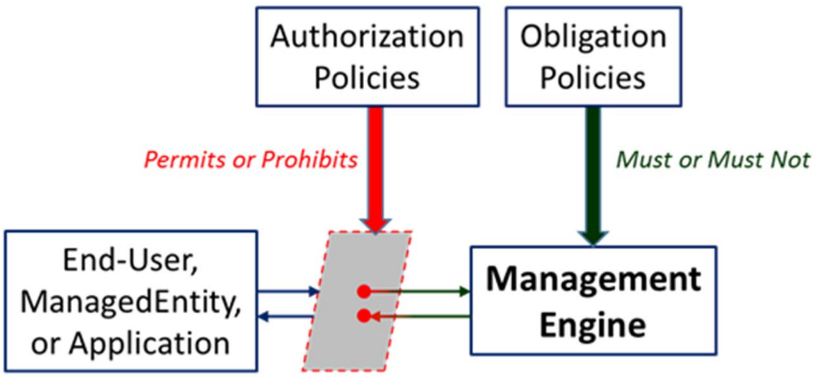
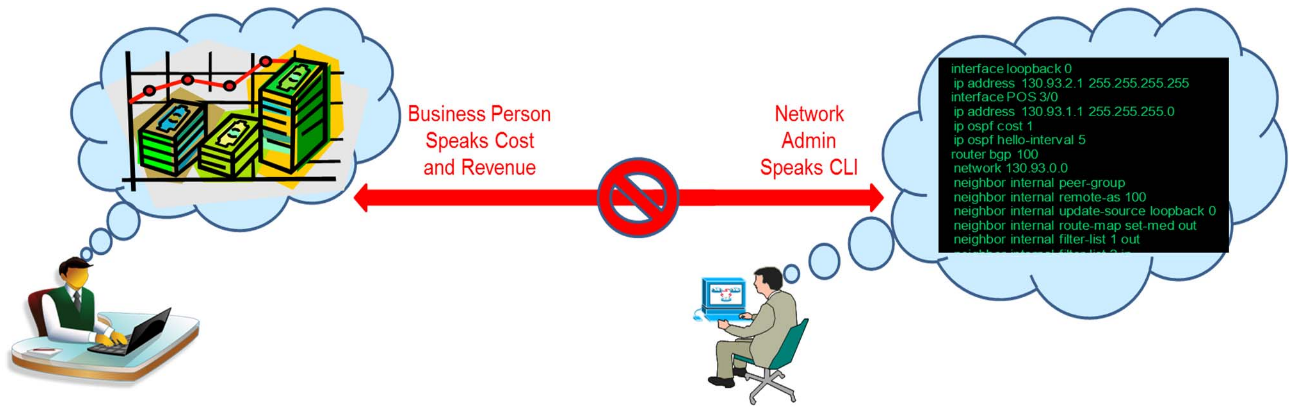


Figure 4.2-2: Policies Used in Management and Control vs. Negotiation

4.3 The Policy Continuum

The Policy Continuum [i.2] and [i.7] defines the concept of different layers of policies that are associated with different sets of actors. This concept was invented because policy is only useful to users that understand its terms and concepts. For example, business users will likely not use declarative policies, since those policies are written as a logic program (see clause 4.4.2). They also will not understand low-level formulations of policy (e.g. in CLI or YANG). In contrast, intent policies were invented to enabled restricted (i.e. controlled) languages to be used to more easily express rules in a language that is appropriate for users working at a higher level of abstraction (see clause 4.4.3). This is illustrated in Figure 4.3-1. In this figure, the business user on the left is working with a Service Level Agreement (SLA). This user thinks of an SLA in terms of cost and revenue. Cost can be further linked to remediation actions. In contrast, the user on the right thinks of how to implement the SLA. This user (e.g. a network admin) deals in terms of low-level functions of the device.



```
interface loopback 0
 ip address  130.93.2.1 255.255.255.255
interface POS 3/0
 ip address  130.93.1.1 255.255.255.0
 ip ospf cost 1
 ip ospf hello-interval 5
router bgp  100
 network 130.93.0.0
 neighbor internal peer-group
 neighbor internal remote-as 100
 neighbor internal update-source loopback 0
 neighbor internal route-map set-med out
 neighbor internal filter-list 1 out
```

Figure 4.3-1: Motivation for the Policy Continuum

The problem is:

> *Two different actors from two different constituencies will have different definitions and terminology for the same concept. This typically gives rise to two (or more) different policies to reflect these different views.* ***How can these different policies be properly associated?***

Note that Figure 4.3-1 shows the *cognitive dissonance* that arises when two different actors refer to the same term or concept (in this case, the term "SLA"), but have *different meanings* associated with that term. Both formulations are, of course, valid. The key is how to *translate* between them. This is the purpose of the Policy Continuum, which is shown in Figure 4.3-2.

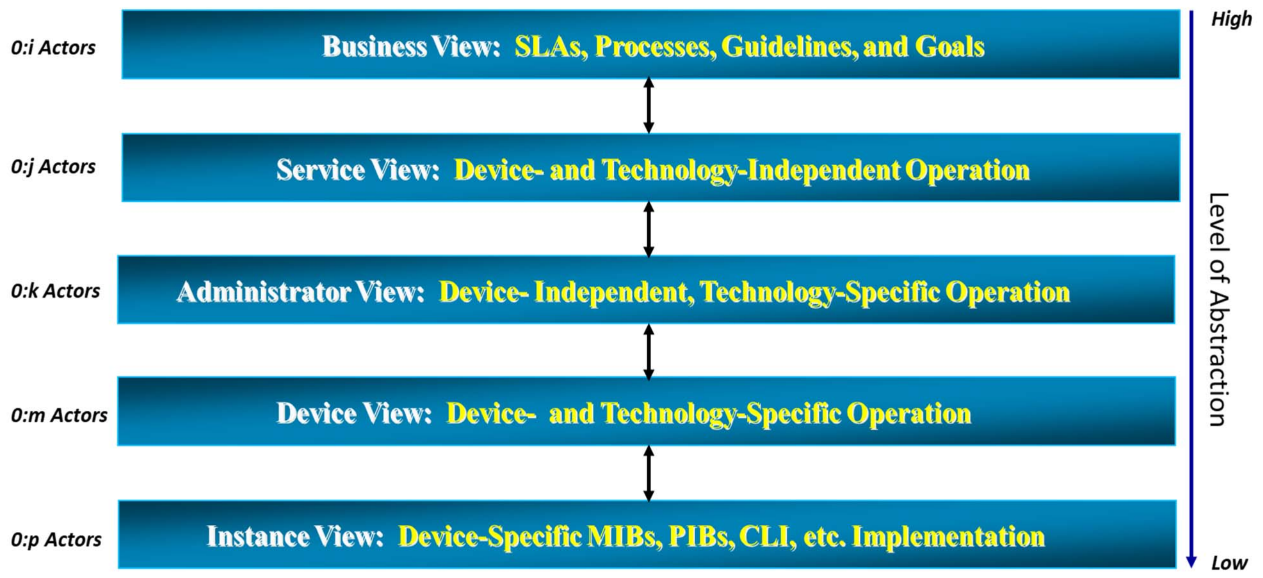

0:i Actors
Business View: SLAs, Processes, Guidelines, and Goals
0:j Actors
Service View: Device- and Technology-Independent Operation
0:k Actors
Administrator View: Device- Independent, Technology-Specific Operation
0:m Actors
Device View: Device- and Technology-Specific Operation
0:p Actors
Instance View: Device-Specific MIBs, PIBs, CLI, etc. Implementation
High
Level of Abstraction
Low

Figure 4.3-2: An Exemplar Policy Continuum

The number of continua in the Policy Continuum should be determined by the applications using it. There is no fixed number of continua. The above figure shows five, because this enables a set of much smaller translations of terms (e.g. from a representation without technology, to one with technology while being device, vendor, and technology independent, to successively lower levels that fix each of these three dimensions).

4.4 Types of Policy Paradigms

4.4.1 Considered Policy Paradigms

There are three main types of policy paradigms that are mentioned in [i.15]: imperative, declarative, and intent policies. These are defined in more detail in [i.5] and [i.12]. While other types of policies are certainly possible (e.g. functional), the use of these three policies paradigms provides sufficient flexibility to address the currently identified needs of ENI.

4.4.2 Imperative Policies

Imperative policies follow the imperative programming paradigm, which focuses on describing *how* a program operates. In this paradigm, policies are structured such that they explicitly control the transitioning of one state to another state. In this approach, only one target state is allowed to be chosen. This is done by *defining the order in which operations occur*, using programming constructs that explicitly control that order. Another important characteristic of imperative policies is that they allow side effects. Figure 4.4-1 shows the behaviour of an imperative policy.

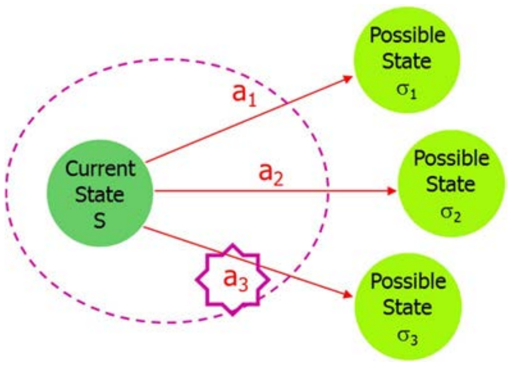

Possible State σ_1
Possible State σ_2
Possible State σ_3
Current State S
a_1
a_2
a_3

Figure 4.4-1: The Imperative Policy Paradigm

A commonly accepted and generic form of imperative policies is the ECA (Event-Condition-Action) Policy. In this paradigm:

Event:	An Event is any important occurrence in time of a change in the system being managed, and/or in the environment of the system being managed. Event include time and user actions (e.g. logon, logoff, and actions that violate an ACL).
Condition:	A condition is defined as a set of attributes, features, and/or values that are to be compared with a set of known attributes, features, and/or values in order to determine whether or not the set of Actions in that (imperative) Policy Rule can be executed or not. Examples of Conditions include matching attributes of a packet or flow, determining if sufficient resources exist for running a Service, and checking the contextual values associated with the current state with those in past states.
Action:	An action is used to control and monitor the behaviour of the system or component that a Policy Rule is applied to when the event and condition clauses are satisfied. The order of action execution, as well as how failures are treated, are determined by metadata. Examples of Actions include providing intrusion detection and/or protection, changing ACLs to grant or deny access privileges, and redirecting traffic to a backup circuit (e.g. in the case of congestion).

Each of the above three clauses are Boolean clauses. A Boolean clause is a logical statement that evaluates to either TRUE or FALSE. It may be made up of one or more terms; if more than one term, then the terms are connected using logical connectives (i.e. AND, OR, and NOT).

4.4.3 Declarative Policies

The purpose of declarative programming is to describe the set of computations that need to be done without describing how to execute those computations. In particular, the control flow of the program is *not* specified. Hence, a key characteristic of declarative programming is that the *order* of statement execution is *irrelevant*. In so doing, side effects are reduced.

In ENI, declarative programming is defined as a program that executes according to a theory defined in a formal logic. That is, declarative policies are written in a formal logic language, such as First Order Logic. This is contrasted with intent policies (see clause 4.4.4), which are written in a (controlled) natural language and then *translated* to a different form. Figure 4.4-2 shows the behaviour of a declarative policy.

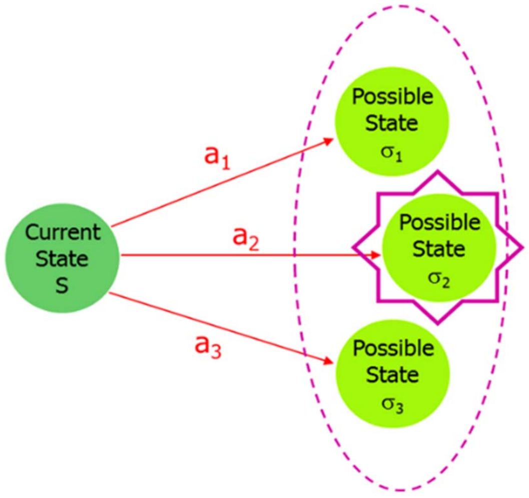

Current State S
a_1
a_2
a_3
Possible State σ_1
Possible State σ_2
Possible State σ_3

Figure 4.4-2: The Declarative Policy Paradigm

The following is an example of a declarative policy from OpenStack Congress.

"Every network attached to a VM should be a public network or a private network owned by someone in the same group as the VM owner."

EXAMPLE: This is expressed in a formal logic as follows:

```
// DEFINE PROHIBITED STATES

ERROR(VM) :-

// FIND ALL VMS IN A NETWORK

NOVA:VIRTUAL_MACHINE(VM),

NOVA:NETWORK(VM, NETWORK),

// SEE IF THIS IS A PUBLIC NETWORK

NOT NEUTRON:PUBLIC_NETWORK(NETWORK),

// IS THE OWNER OF THE NETWORK IN THE SAME GROUP AS THE OWNER OF THE VM

NEUTRON:OWNER(NETWORK, NETOWNER),

NOVA:OWNER(VM, VMOWNER),

NOT SAME_GROUP(NETOWNER, VMOWNER)

// WHICH USER ARE MEMBERS OF THE SAME GROUP

SAME_GROUP(USER1, USER2) :-

LDAP:GROUP(USER1, GROUP),

LDAP:GROUP(USER2, GROUP)
```

In the above, Nova is a Manager for VMs, Neutron is a Manager for Virtual Networks, and LDAP directory services are used to manage group-membership.

Note that declarative policies are expressed as *predicates*.

Declarative policies can be used in three different ways:

1) **Monitoring**: check if all deployed VMs obey this policy
2) **Preventative**: determine if this policy is satisfied before Nova deploys a VM
3) **Corrective**: when LDAP group membership changes, correct violations

4.4.4 Intent Policies

In ENI, an intent policy is a type of declarative policy that uses statements to express the goals of the policy, but not how to accomplish those goals. Each statement in an Intent Policy may require the translation of one or more of its terms to a form that another managed functional entity can understand.

In the present document, Intent Policy will refer to policies that do **not** execute as theories of a formal logic. They typically are expressed in a restricted natural language, and require a mapping to a form understandable by other managed functional entities. This is shown in Figure 4.4-3.

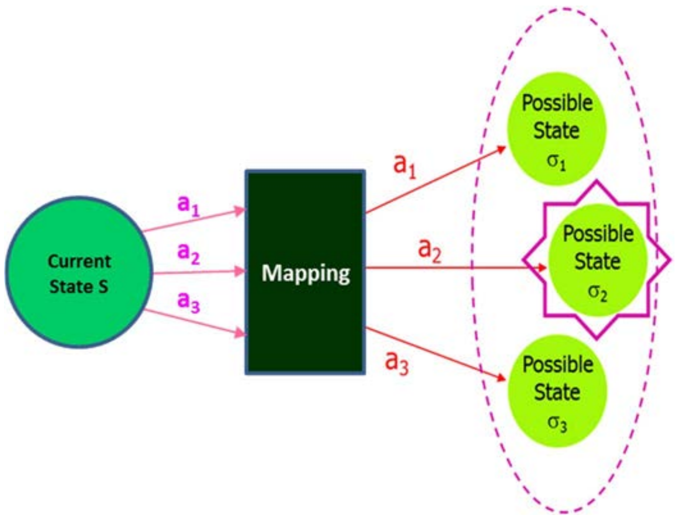

Current State S
a_1
a_2
a_3
Mapping
a_1
a_2
a_3
Possible State σ_1
Possible State σ_2
Possible State σ_3

Figure 4.4-3: The Intent Policy Paradigm

In intent policies, there is not really the notion of an *action;* Figure 4.3-5 is used to keep the symbology constant to facilitate comparison between imperative, declarative, and intent policies. Rather, in declarative and intent policies, the current state S represents the *goals* of the policy, and the possible states represent solutions that realize those goals in different ways.

The advantage of Intent is its flexibility, in that it enables its users to express policies using concepts and terminology that are familiar to the user. This is, of course, its disadvantage, since natural languages are typically ambiguous.

4.5 Policy Translation

There are several approaches to solving the translation problem. One approach is to try and standardize a set of languages, one for each level of the Policy Continuum. This will not work, since the Policy Continuum is conceptual, and the number of levels will vary based on the types of users that want to use intent, and the policies themselves depend on the types of behaviours that need to be controlled (which in turn depend on which applications and services are being used).

Therefore, a more promising approach is to use an information model as a data dictionary. This enables a set of languages to be defined, where each language uses terms from the information model to guarantee consistency and the ability to translate between languages. For example, the highest level could be limited to simple nouns and verbs (e.g. classes and relationships in a model). The next level could add linguistic processes to add synonyms to important elements in the information model. The next level could take this a step further, and map different types of phrases in a sentence to sets of model elements.

4.6 Model-Driven DSL

Imperative, declarative, and intent policies are very different in form and structure. However, as shown in Figure 4.2-2, they need to be easily related to each other. This implies a single language with a number of dialects, as opposed to multiple different languages. This resembles the relationship between a single information model and the many data models that can be derived from it. Therefore, the information model, which serves as a data dictionary, will be used to construct the underlying grammar, as well as the dialects corresponding to each level of the Policy Continuum.

The formal representation of a policy can remove ambiguities present in natural language statements. Policies that are represented using formal languages can be defined by axioms and reasoned about using automated systems. An object-oriented information model provides an extensible structure, complete with relationships, which can control the axiomatisation of the policies. This in turn simplifies the subsequent translation of policy between different continua in the Policy Continuum.

The input policy is parsed, and a parse tree is generated. Each node in the tree is tagged with a particular part-of-speech. This enables a large degree of flexibility in building and changing a grammar. For example, this front-end process can be kept the same, and used to support different grammars. This is especially attractive for realizing the Policy Continuum. If an input word is not recognized, then the system will request help from the operator.

NOTE 1: This process can be made much more robust by using a formal representation of semantics, such as ontologies. A semantic network is then formed between nodes in the model and concepts in the ontologies. This enables simple linguistics, such as synonyms, to be used. This is for further study.

NOTE 2: The actual grammar is still a work-in-progress. This work is beyond the scope of the present document, but is a candidate for a future deliverable on architectural approaches of ENI.

The model engine then examines additional model elements that are related to the identified model elements. For example, suppose that "SLA" was identified in the input policy. A model may relate the SLA class to a number of different concepts, including Role, SLS, and Application, even though they were not mentioned in the policy. This is the power of intent: it establishes a *context* that enables the system to infer additional information about the intent.

The system forms a *semantic network* (i.e. a network that represents semantic relations between its concepts). Semantic relationships include linguistic (e.g. synonymy, antonymy, meronomy), structural (e.g. do the input terms form a pattern that matches a pattern in the model), and other measures that determine how similar the input is to the model. Conceptually, this is a graph that consists of nodes (i.e. concepts and objects) and edges (i.e. semantic relations between concepts).

4.7 Policy Conflict Detection and Remediation

It is important to note that not all types of policies have policy conflicts. In particular, in the three types of policies that ENI will use, only imperative policies have this problem.

A policy conflict is defined as follows:

two actions applied to the same point that cause different, incompatible behaviours

The difference between imperative vs. declarative policies (remember that intent is a type of declarative policies) becomes immediately clear:

imperative policies are independent, siloed artefacts;
declarative policies form an integrated program

Consider the following imperative policies:

John gets GoldService
FTP gets BronzeService

Assume that both of these is analysed separately. Both pass, because they have a valid grammar. The problem arises in an environment where John has GoldService and can also use FTP services. This causes a conflict, since the first policy is assigning all services GoldService processing, while the second policy is defining FTP to receive BronzeService.

In contrast, a declarative (or intent) program would write this as part of a program. If the two policies cannot be simultaneously supported, then the program outputs a FALSE; otherwise, it evaluates to TRUE. In essence, a declarative (or intent) policy changes these from separate statements to separate facts that the system tries to prove in a single program.

4.8 Types of Logic Supported by Policies

There are a variety of formal logic styles that are supported by policy formalisms. They include deontic logic (i.e. the logic of obligation and permission) and alethic logic (i.e. the logic necessity and possibility). Deontic and alethic logic are both types of *modal* logic. A formal policy grammar can be developed to support these and other types of modal logics; however, that is beyond the scope of the present document.

4.9 The Ramifications of Changing Policies

Ideally, ENI will adapt in response to change. This implies that all or part of the current working set of policies changes accordingly. This presents to key problems:

1) Does the new set of policies conflict with the existing working set of policies?
    a) If so, which entities are affected?
    b) If so, are there policies and/or meta-policies that can be used to address the changes?
2) Does the changing of policies cause undesirable intermediate state changes?

For imperative policies, it is critical to conduct a local conflict analysis (e.g. intra- and inter-domain, as defined in clause 4.2.1) to ensure that conflicts do not arise *before* the existing policies are shut down and the new policies are installed. If these policies are going to interact with entities outside of the local domain (e.g. inter-system policies, as defined in clause 4.2.1), then the above conflicts should also be repeated between the affected domains.

For declarative policies, explicit conflicts are not allowed. Rather, two logic programs can yield different results when they are finished executing. This may be solved in a number of different ways; two exemplary ways are:

- using game-theoretic solutions to maximize expected utility, which in turn have two approaches:
    - zero-sum approach (one solution "wins" at the expense of other solutions);
    - co-operative approaches (multiple solutions partially benefit);
- using logic-based approaches, which also have two approaches:
    - reduce conflicting goals to sub-goals;
    - seek to find higher-level goals that avoid the conflict.

This work is beyond the scope of the present document, but is a candidate for a future deliverable on architectural approaches of ENI.

4.10 Other Types of Policies

There are a variety of other types of policies. These are included in this clause for completeness. For example:

Meta-policy: policy about policy. This helps coordinate the interaction between multiple policies. An example of a meta-policy is "Coordinate new peering policies with latest SLA traffic conditioning policies".

Operational policy: defines required actions without defining the goal. An example of an operational policy is "Passwords shall be changed every six months."

4.11 The MEF Policy-Driven Orchestration Project

4.11.1 Introduction to MEF PDO

A starting point for policy management and modelling in ENI is the work being done in the MEF Policy-Driven Orchestration (PDO) work. This is because this work meets the needs of the present document, and represents a good sampling of the state-of-the-art in policy management. As such, a liaison has been produced between ENI and the MEF.

As background, the MEF PDO work defines what Policy is, how Policy-based Management is used in the MEF Lifecycle Service Orchestration Reference Architecture (LSO RA), some exemplary use cases (from the point-of-view of the Service Provider and other Partners), and architectural extensions to the LSO RA.

Note that the MEF LSO RA defines a set of RPs; while most are certainly relevant to the overall architecture of ENI, it is likely that ENI will need additional RPs.

The MEF PDO project is in the process of defining a robust information model that enables imperative, declarative, intent, and other types of Policies can be modelled using the same Policy model. This model has grown out of the IETF SUPA group [i.1] and [i.16]. It is important to note that SUPA is *limited* to imperative policy modelling; however, it contains a "superstructure" (i.e. the class hierarchy is designed using a number of strict software design principles, such as the Liskov Substitution Principle [i.3] and the Single Responsibility Principle [i.4], that can be used to model all types of Policies in an extensible manner). The MEF Approach combines a top-down analysis approach using software design models with a bottom-up view to ensure that implementation in a number of different languages is straightforward.

4.11.2 The MEF PDO Information Model

The MEF PDO information model is derived from the MEF Core Model [i.14], and is shown below in Figure 4.11-1.

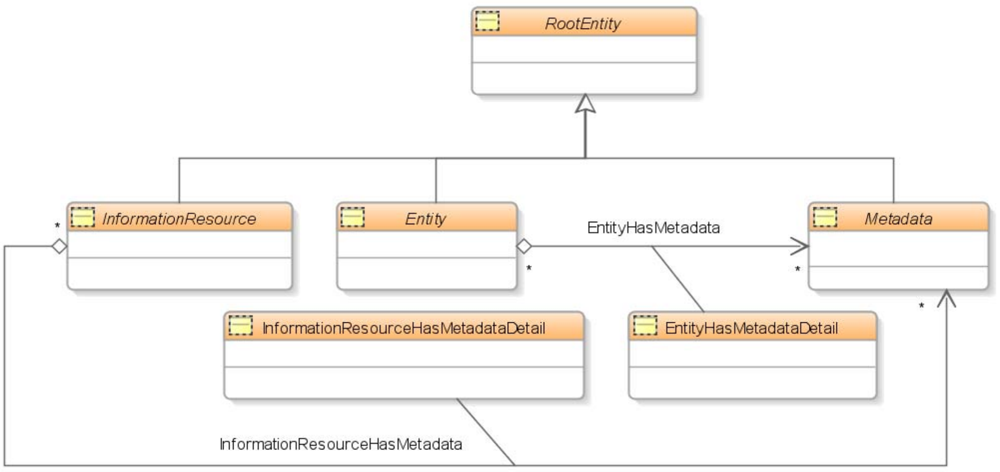


Figure 4.11-1: The MEF Core Model

RootEntity is the (single) superclass of the entire MEF model. This enables a single namespace to be defined for all MEF classes. Its characteristics and behaviour are thus inherited by all MCM classes. Note that multiple inheritance is *disallowed* in MEF models.

Figure 4.11-1 shows the three subclasses of RootEntity: Entity, InformationResource, and Metadata. The limit of three subclasses simplifies the understanding of the model, and uses classification theory to ensure that objects are organized into groups according to a set of criteria (e.g. their similarities and/or differences). This approach is called set theory when applied to mathematics.

The three subclasses create three parallel and interacting class hierarchies, and are described as follows:

- Entity, which is the superclass for objects of interest that are important to the managed environment, and which have a separate and distinct existence.
- InformationResource, which is information that is required to describe concepts owned by other Entities, but which is not an inherent part of the Entity being described.
- Metadata, which is information that describes and/or prescribes the behaviour of Entities and InformationResources.

The purpose of the Entity hierarchy is to represent the characteristics and behaviour of concepts that are important to the managed environment. Entities have a separate and distinct existence. The Entity hierarchy is the set of subclasses of the Entity class that define the externally visible characteristics and behaviour of the system being managed in more detail. The main classes in this hierarchy include ManagedEntity, UnManagedEntity, Domain, and Party.

ManagedEntity is a subclass of Entity that represents objects that have the following common semantics:

1) each has the potential to be managed;
2) each can be associated with at least one ManagementDomain;
3) each is related to other important objects of the system being managed.

Note that Figure 4.11-1 shows an aggregation, called EntityHasMetadata, that enables an Entity to optionally aggregate Metadata. This enables different Entities to reuse common descriptive and/or prescriptive information. For example, a common versioning strategy can be defined using Metadata, and then applied in a consistent and uniform fashion to all Entities that need it. Similarly, the InformationResourceHasMetadata aggregation enables InformationResources to aggregate Metadata.

The PDO information model was derived from DEN-ng [i.2], and then simplified and took the form of SUPA [i.1]. The SUPA model was used as a basis for the MEF PDO model, with DEN-ng concepts added back in.

The PDO model views all Policies, regardless of their structure or programming paradigm, as a set of statements. Each statement is made up of a set of clauses. Each statement and clause can be augmented by metadata. This produces three main sub-hierarchies:

- PolicyStructure, which is used to represent the structure of a Policy. This class, and all of its subclasses, are a *typed container* (i.e. they aggregate components of a Policy, but enforce the programming paradigm of the policy). Hence, it is impossible to put declarative or intent objects in an imperative policy (container).
- PolicyComponentStructure, which is used to represent components of a Policy. Since different types of policies require different types of structural components, and since all policies are abstracted as one or more sentences, the PolicyComponentStructure class hierarchy uses the abstraction of a clause (a portion of a statement). This, combined with the Decorator pattern [i.8], enables different types of clauses to be constructed for each Policy.
- PolicyMetadata, which is information that describes and/or prescribes the behaviour of Policies and PolicyComponents.

A Policy may be an individual Policy or a hierarchy of Policies. This is done by applying the composite pattern [i.6] and [i.8] to the PolicyStructure class, resulting in two subclasses:

1) PolicyRuleAtomic (for defining stand-alone policies); and
2) PolicyRuleComposite (for defining hierarchies of policies).

In addition, two other top-level classes are defined. These are PolicySource and PolicyTarget. A PolicySource defines a set of managed entities that authored, or are otherwise responsible for, this Policy. A PolicySource does *not* evaluate or execute SUPAPolicies. Its primary use is for auditability and the implementation of deontic and/or alethic logic. A PolicyTarget defines a set of managed entities that a Policy is applied to. A managed object should satisfy two conditions in order to be defined as a PolicyTarget. First, the set of managed entities that are to be affected by the Policy should all agree to play the role of a PolicyTarget. In general, a managed entity may or may not be in a state that enables Policies to be applied to it to change its state; hence, a negotiation process may need to occur to enable the PolicyTarget to signal when it is willing to have Policies applied to it. Second, a PolicyTarget should be able to process (directly or with the aid of a proxy) Policies.

The mapping of the top-level classes of the PDO model to the MEF Core model is:

- PolicyObject is a new subclass of Entity (a peer to ManagedEntity), and PolicyStructure, PolicyComponentStructure, PolicySource, and PolicyTarget are subclasses of PolicyObject.
- PolicyMetadata is a subclass of Metadata.

4.11.3 The MEF PDO Data Models

The original SUPA work produced a YANG data model [i.16] that has a unique "object-oriented-like" representation (especially for a data model that is hierarchical in nature). This is done by implementing three key information modeling concepts: classes, class inheritance, and associations.

Each class in the model is represented by a YANG identity and by a YANG grouping. The use of groupings enables us to define these classes abstractly. Each grouping begins with two leaves (either defined in the grouping or inherited via a uses clause), which provide common functionality. One leaf is used for the system-wide unique identifier for this instance. The second leaf is always called the entity-class. It is an identityref, which is set to the identity of the instance. The default value for this leaf is always correctly defined by the grouping.

Class inheritance (or subclassing) is done by defining an identity and a grouping for the new class. The identity is based on the parent identity, and is given a new name to represent this class. The new grouping uses the parent grouping. It refines the entity-class of the parent, replacing the default value of the entity-class with the correct value for this class.

Associations are represented by the use of instance-identifiers and association classes. Association classes are classes, using the above construction, which contain leaves representing the set of instance-identifiers for each end of the association, along with any other properties the information model assigns to the association. The two associated classes each have a leaf with an instance-identifier that points to the association class instance. Each instance-identifier leaf is defined with a "must" clause, which references the entity-class of the target of the instance-identifier, and specifies that the entity class type should be the same as, or subclassed from, a specific named class. Thus, associations can point to any instance of a selected class, or any instance of any subclass of that target.

Identities are used in this model as a means to provide simple introspection to allow an instance-identifier to be tested as to what class it represents. This allows "must" clauses to specify that the target of a particular instance-identifier leaf should be a specific class, or within a certain branch of the inheritance tree.

4.11.4 The MEF PDO Architecture

Figure 4.11-2 illustrates a high-level functional block diagram of a Policy Domain, as defined in the MEF PDO Architecture.

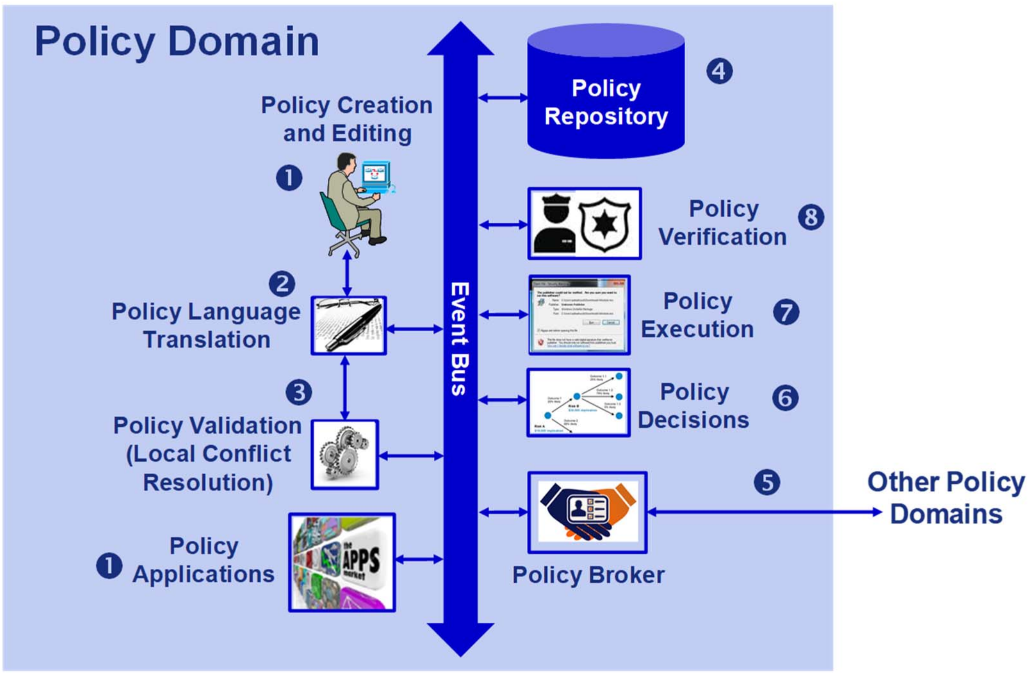


Figure 4.11-2: The MEF PDO Domain Conceptual Architecture

The Policy Creation and Editing functional block supports policy creation, editing, viewing, and deletion using graphical and command line functions.

The Policy Language Translation functional block supports translating a policy provided in one language to an equivalent set of policies in a different language.

The Policy Validation functional block supports local (i.e. intra-domain) conflict resolution. A policy conflict occurs if two or more policies cause conflicting actions to occur on the same policy target at the same time. Hence, this functional block ensures that all currently running policies do not conflict with each other.

The Policy Repository functional block supports the storage, retrieval, and querying of policies and policy components, along with common PolicyMetadata objects.

The Policy Broker functional block serves as a communication and distribution mechanism between policy domains. As such, a broker may furnish information, in the form of specs, metadata, and other entities, to a Policy Broker in other domains. The PolicyBroker is a well-known entity that all policy engines, once they register to the system, are aware of.

The Policy Decision functional block supports matching one or more policies to the needs of a managed object that requests direction in what action(s) it should take.

The Policy Execution functional block supports monitoring the operation of the policy to ensure that it executes successfully.

The Policy Verification functional block determines if an executed policy performed as expected. For example, it will note any unintended side effects, as well as determine if the set of state transitions of the policy target were exactly those that were expected.

NOTE: This work is beyond the scope of the present document, but is a candidate for a future deliverable on architectural approaches of ENI.

4.12 Approach Going Forward

The approach proposed for this Gap Analysis is:

1) Analyse the MEF PDO Policy Model, and detail its strengths and weaknesses.

2) Analyse at least the following SDOs, and compare their approach to the MEF PDO approach to see if ENI could benefit from incorporating concepts from them:
IETF SUPA, IETF L3SM, IETF L2SM, IETF NFVRG, IETF ANIMA, TMF SID,
ETSI ISG NFV MANO, ETSI ISG MEC, ONF SDN, ETSI NTECH AFI.

3) Analyse at least the following open source consortia, and compare their approach to the MEF PDO approach to see if ENI could benefit from incorporating concepts from them:
OpenStack Congress, ONF Boulder, ODL GBP, LF ONAP, ONOS.

4) Write a detailed deliverable summarizing the above points, and recommend the approach to be taken in ENI.

4.13 Key Features to be Compared in the Gap Analysis

The following is a set of key features, based on experience, which are recommended to be present in order to make maximum use of policy in network management:

NOTE: Since neither the Use Case or Requirements Group Reports are complete at this time, the feature comparison will be based on the architectural experience of the Editor.

1) A single information model is required to be used to represent the structure and semantics of Policies. Multiple information models are not to be used.

2) It is recommended that multiple data models are derived from the (single) information model.

3) The (single) information model, and each data model, are required to use a set of software design patterns to provide extensibility and consistency.

4) Formal transformations from the (single) information model are required to derive each different data model. This facilitates the development of associated software tooling.

5) It is permissible that different types of Policies are used in the same system.

6) It is recommended that different actors are represented by Roles, which are then mapped to different levels in the Policy Continuum.

7) Different actors have different concepts and terminologies for their policies. Therefore, it is recommended that the notion of a Policy Continuum, where each continuum corresponds to a given set of actors, is realized.

8) It is recommended that formal mathematical transformations between each level of the Policy Continuum and its actors are defined; this facilitates the development of APIs and DSLs.

9) Actors operate in a *context*. Context may change throughout the lifecycle of the system. Therefore, it is permissible that the context of an actor is used to map the activities of the actor to a particular level in the Policy Continuum.

10) It is required that the Policy information model is integrated into the system's information model. This facilitates relating policies to Services, Resources, Functional Blocks, and other managed entities of the system.

11) It is required that the information model is used in constructing Policy APIs and DSLs.

12) It is required that Policy, when used within a domain, act as a mechanism to manage behaviour.

13) It is required that Policy, when used between domains, act as a mechanism to negotiate behaviour.

14) It is required that Context is related to, and updated by, information from applicable functional blocks, such as analytics, so that as context changes, the management system is aware of contextual changes and can respond accordingly.

15) It is required that Context is used to select active Policies (and deselect Policies that are no longer appropriate). This enables the system to adjust its goals and behaviours accordingly to changes in context.

16) It is permissible that the system, based on contextual changes, construct new policies to address problems found and/or improve system behaviour by using operator intervention if no suitable policies currently exist.

17) A policy can be cloned and modified by the policy engine (as opposed to the policy itself being self-modified). The changes allowed by this process will be limited according to parameters defined in the architecture GS. Self-modifying policies are still in the early research stage; therefore, in this stage of ENI, it is not permitted to use self-modifying policies.

18) It is recommended that Policy is able to be augmented by metadata. Specifically, metadata is permitted to be used to describe and/or prescribe Policy functionality.

19) It is required that Policy be able to be realized using centralized and/or distributed means.

20) It is recommended that AI algorithms are used to construct policies.

5 Analysis of the MEF PDO Model

5.1 Characteristics of the MEF PDO Model

5.1.1 Comprehensive Policy Information Model

The MEF PDO Model provides a comprehensive and extensible information model for defining how different types of policies (e.g. imperative, declarative, intent, and others) can be modelled using a common framework. This information model defines a number of different management abstractions and design patterns that can be adapted for many uses. These include:

1) A management abstraction defining a policy as a container, regardless of the programming paradigm used.

2) A management abstraction defining a policy component as an object that can be aggregated by a policy.

3) A management abstraction defining a policy *clause:*

   a) A policy clause is the smallest grammatical unit that can express a logical proposition.

   b) Sentences are constructed from one or more policy clauses, which is a building block for constructing a policy *sentence.*

   c) Imperative policies are three-tuples, consisting of an event, a condition, and an action clause.

   d) Declarative policies are collections of clauses that collectively prove or disprove a proposition.

4) A management abstraction defining the concept of a decorator (e.g. all or part of an object that is attached to another object at runtime).

5) Integration with the MEF LSO RA.

NOTE: The following subjects are for further discussion:

- add discussion of how different paradigms are used - e.g. imperative for vendor-specific configuration vs. declarative for vendor-independent configuration;
- add discussion of differences in imperative model in SUPA vs. MEF;
- clarify advantages and disadvantages of APIs vs DSLs; point out that these advantages and disadvantages are independent of programming paradigm;
- relate policy continuum to info model;
- explain conflict resolution differences in imperative vs. declarative policies; intent can use negotiation to resolve conflicts;
- point that grammatical representation of policies facilitates DSLs and APIs generation, and assists in multi-policy representation;

▪ explain the "No processor should run at more than 70 % utilization" example in more detail; add additional examples.

5.1.2 YANG Data Model

The YANG data model (defined in the IETF SUPA WG) details how to construct a YANG data model directly from an information model. This approach is unique (to the best of our knowledge) in the industry, as most YANG data models are built bottom-up (e.g. without the benefit of an information model) and/or using simple transforms that do not use many important UML mechanisms. In stark contrast, the SUPA YANG model is implemented directly from the information model. In addition, it is implemented as an object model (to the extent that YANG allows that), and even defines a simple form of introspection.

5.1.3 Design Patterns Used in the MEF PDO

Another important characteristic is a set of design patterns that simplify the complexity of the MEF PDO model. These design patterns [i.8] are reusable templates that present a generic solution to a commonly occurring problem. The MEF policy model currently includes a number of design patterns, including:

- Composite pattern: a generic pattern for constructing hierarchies of objects [i.6] and [i.8].
- Role-object pattern: a generic pattern for separating roles that an object can play from the definition of the object itself [i.8].
- Decorator pattern: defines a base interface to create variations of a specific object that can be combined and nest to dynamically add and remove behaviour [i.8].
- Policy pattern: defines how to use policies to control the semantics of a relationship [i.2] and [i.5].

5.1.4 Use of the Policy Continuum

The Policy Continuum [i.7] describes how to model, using mathematical formalisms, the notion of policies that are associated to a set of constituencies. More specifically, a set of actors are associated with a particular policy, and transformations (based on the model) can then be defined to translate a policy at that level of the continuum to another form at the next level of the policy continuum (either upwards or downwards, where upwards is more abstract, and downwards is more concrete).

Figure 5.1-1 illustrates one mapping of actors to different levels of the Policy Continuum. This is an exemplary mapping, since the definition of actors and continua are specific to a particular business environment.

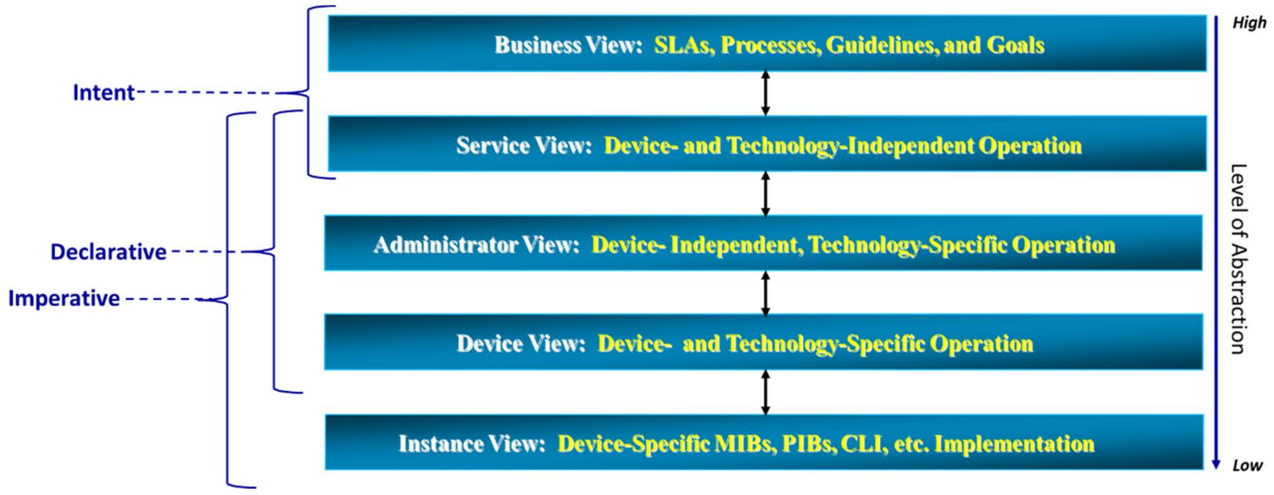

Business View: SLAs, Processes, Guidelines, and Goals
Service View: Device- and Technology-Independent Operation
Administrator View: Device- Independent, Technology-Specific Operation
Device View: Device- and Technology-Specific Operation
Instance View: Device-Specific MIBs, PIBs, CLI, etc. Implementation
Intent
Declarative
Imperative
High
Level of Abstraction
Low

Figure 5.1-1: Mapping Actors to Different Policy Paradigms

In Figure 5.1-1, each of the three types of policy are mapped to a set of actors. For each policy type, this does not mean that a particular policy type cannot be used for other actors! Rather, this figure shows where each policy type is likely to be used the most, since that particular paradigm offers the most benefit for that set of actors.

Briefly:

- Intent policies are used in the business and service views, since they enable both types of actors to specify the *goals* of the policy in *non-device-* and *technology-specific terms*:
    - Intent policies are likely not applicable to the lower three continua shown in Figure 5.1-1, since each of these continua uses technology-specific terminology, which violates the definition of an intent policy.
- Declarative policies are used in the service, administrator, and device views, since they enable logic programs to express and act on goals that are applicable to the needs of these actors:
    - Declarative policies are likely not applicable to the instance view, since that would require a device that could evaluate declarative logic.
    - Declarative policies could, of course, be used in the business view. However, declarative policies use formal logic, which is difficult for business actors to use.
- Imperative policies are used in the service, administrator, device, and instance views, since they enable actors to specify a policy using a simple syntax:
    - Imperative policies could, of course, be used in the business view. However, intent policies are judged to be easier to use.

5.1.5 Extension of the MEF LSO RA

The MEF LSO RA [i.10] defines a multi-level, multi-domain, orchestration architecture that is based on model-driven engineering principles. The MEF Service Orchestration Functionality (SOF) [i.11] and Policy-Driven Orchestration (PDO) [i.12] projects, both launched in 2017, extend the LSO RA to:

- Define the SOF as a set of nested functional blocks, where each functional block is defined by its inputs, outputs, and current state - this enables the behaviour of the functional block to be defined in a vendor-neutral, platform-neutral manner.
- Define the PDO as an extension of the MEF LSO RA, detailing how policies can be applied at each of the seven RPs of the MEF LSO RA.

NOTE: Further elaboration of this work is beyond the scope of the present document, but is a candidate for a future deliverable on architectural approaches of ENI.

5.2 Supported Policy Paradigms

5.2.1 Imperative Policy

5.2.1.1 Description

The purpose of an Imperative Policy is to achieve a goal by explicitly changing the state of a set of entities in the managed environment. An Imperative Policy explicitly defines the control flow as a set of commands to be executed.

The basic abstraction used is a set of statements that directly change computed state. An Imperative Policy is usually formulated as either a condition-action (CA) or event-condition-action (ECA) tuple, consisting of 2 or 3 Boolean clauses, respectively.

Imperative Policies is intended to be directly consumable by the device (though some devices may require translation or proxies).

5.2.1.2 Formal Model

The canonical form of an imperative Policy is a set of three clauses whose behaviour is as follows:

WHEN a set of events is received

IF the condition clause evaluates to TRUE

THEN zero or more actions in the action clause are allowed to be executed

In the above formulation, the event clause and the condition clause are required to be Boolean clauses. That is, each clause is a Boolean formula, which evaluates to TRUE if all variables in the Boolean formula can be replaced by the values TRUE or FALSE. Multiple conditions can be defined using logical connectives (i.e. AND, OR, and NOT).

This has the following logical flow:

```
IF the (Boolean) event clause evaluates to FALSE
   EXIT
ELSE IF the (Boolean) condition clause evaluates to FALSE
   EXIT
ELSE
   Examine Metadata
   Execute any applicable Actions according to Metadata settings
```

The action clause is more complex. First, it can only be evaluated if both the event clause and the condition clause evaluate to TRUE. Second, if that occurs, then metadata is used to determine if one or more actions can execute, and how the execution is performed. In the MCM [i.14], two different metadata attributes are used as follows:

- Execution strategy defines how the actions are processed. Exemplary values include "execute first action then exit", "execute all actions in priority order", and only execute actions with a priority assigned, and execute them in priority order".
- Failure strategy defines what to do if an action fails to complete successfully. Exemplary values include "ignore and continue", "rollback and exit", and "rollback and continue".

5.2.2 Declarative Policy

5.2.2.1 Description

The MEF PDO project is currently working on building a robust model of declarative policy. This work is based on work from the IETF SUPA WG [i.1] and DEN-ng [i.2].

The purpose of a Declarative Policy is to achieve a goal by describing what needs to be done without defining its implementation. Hence, control flow is not described.

The basic abstraction used is a set of statements that make up a program, where the program executes as a formal logic. Examples include SQL and OpenStack Congress.

Declarative Policies are not intended to be directly consumable by the device; they will, in general, require translation or proxies to convert the output of the policy to a form that the device can consume.

5.2.2.2 Formal Model

There are many different examples of declarative policy languages. One of the better known is OpenStack Congress, which is an extension of the Datalog programming language.

A declarative policy language consists of rules and facts. Rules state how to evaluate facts, and facts state what is known to the system. A rule consists of three parts: the head, a symbol that means "if it is known that", and a body. Thus, a rule can be defined as:

"<HEAD-part> if it is known that <BODY-part>"

The purpose of a declarative policy is to determine if a premise is TRUE, given a set of rules and facts. For example, the following two simple rules will be used to process facts:

```
ancestor(X, Y) :- parent(X, Y)
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y)
```

The rules are interpreted as:

"X is the ancestor of Y if it is known that X is the parent of Y"
"X is the ancestor of Y if it is known that X is the parent of Z and Z is the ancestor of Y"

Given the two facts:

```
parent(Conrad, Richard)
parent(Richard, John)
```

They mean that "Conrad is the parent of Richard" and "Richard is the parent of John", respectively.

If these two facts and two rules are combined into a Datalog program, then when you run the following query (which asks for the ancestors of John):

```
?-  ancestor(Conrad, X)
```

The result is Richard and John.

5.2.3 Intent Policy

5.2.3.1 Description

The purpose of Intent policy is to specify the goals of what the policy should do, but not how the policy achieves those goals. Hence, Intent should not be thought of as, for example, a single 500 line policy; rather, it should be thought of as a set of small, simple policies that work together. Hence, a single complex goal is discouraged, and should be replaced by multiple smaller, simpler goals.

As a general rule, Intent policies are expected to be on the order of 3 - 30 lines, not 500. This means that translation is significantly simplified, because many small policies are being translated, and hence, the grammar is easier to specify. Note that this also implies that simpler Intent policies may be reusable (e.g. able to be used to build up more complex intent policies).

Intent is NOT directly consumable by the device; rather, its final form leads to something that is consumable after a suitable translation process.

5.2.3.2 Formal Model

This clause includes several examples of intent policy that are being developed in the MEF PDO Project. They represent different grammars used by different actors in the Policy Continuum [i.2] and [i.7].

5.2.3.3 Example 1: Application Developer

An application developer may need to specify protocol behaviour. For example:

Prevent SNMP from entering or leaving the enterprise

This may be translated into:

Ensure that SNMP is blocked on ports at the edgeInterface of the administrative domain to prevent SNMP going out or coming in from outside the enterprise.

5.2.3.4 Example 2: Application Developer/Architect/Business Level

This example defines a common intent policy, which is to assign a certain class of service to a given set of Roles. In this case, the class of service is Gold, and the Roles are CxOs. It is assumed that there are at least four service offerings: Platinum, Gold, Silver, and Bronze, in order of decreasing capabilities and functionality.

It is assumed that Gold Service consists of a number of different applications that each require different treatment of traffic (e.g. 4K video vs. Internet browsing vs. FTP download vs. SAP). In this example, each service and the requirements of its applications are first enumerated. Then, the same is done for other services (e.g. Silver, Bronze, and best effort). A preferential allocation of Resources is then defined, where applications that belong to Gold Service have priority over applications that belong to Silver and Bronze Service. In this case, "priority" means higher bandwidth, less jitter, etc., as appropriate to the service.

Note that this brings up *starvation* issues - so maybe, the policy for how GOLD behaves is to starve lower priority services, whereas the policy for Silver and Bronze is to attempt to allocate fairly, but in a weighted fashion (so that Silver does not starve Bronze, but Silver applications do get better treatment than corresponding Bronze applications). Assuming that an implementation could further generalize into "video", then:

Gold defaults to 4K and has degradation scheme 1

Silver defaults to HD and has degradation scheme 2

Bronze defaults to SVGA and has degradation scheme 3

5.2.3.5 Business Level

Policy may invoke a number of related policies. For example, it could check to see if this Customer is a pre-existing Customer with valid credit. If not, then this customer needs to have a credit check performed before an order can be placed.

In a model-driven system, this is straightforward to do. For example, a new Customer is associated with an instance of a Customer class, which is defined in the underlying information model. Since a Customer inherits from PartyRole, which is related to Party [i.14], then:

- The Person or Company is defined as an instance of the appropriate Party class.
- The role being played is defined as an instance of the Customer class.

This makes it easy to specialize the characteristics and behaviour of individual Parties and PartyRoles.

Note that all of these actions are hidden from the business user.

5.3 MEF PDO Model Features Comparison

This clause compares the features of the current MEF PDO model to the requirements listed in clause 4.12.

Table 5.3-1: How the MEF PDO Model Meets the Requirements of clause 4.12

Requirement	Status	Comments
1a. A single information model is required to be used to represent the structure and semantics of Policies.	Fully Meets	The MEF PDO information model is designed as a module that can plug into any information model as long as it can separate structure, semantics, and metadata.
1b. It is recommended that multiple information models are not to be used.	Fully Meets	The MEF PDO model is designed as a single atomic information model.
2. It is recommended that multiple data models are derived from the (single) information model.	Fully Meets	The MEF PDO information model is independent of technology, protocol, language, and platform. In addition, the MEF PDO project will provide guidance on how to construct diverse data models in the future.
3. The (single) information model, and each data model, are required to use a set of software design patterns to provide extensibility and consistency.	Fully Meets	The MEF PDO information model uses a number of traditional software patterns; it also incorporates new software patterns that were designed specifically for policy-based management.

Requirement	Status	Comments
4. Formal transformations from the (single) information model are required to derive each different data model.	Fully Meets	The MEF PDO information model will describe in detail formal transformations for at least two different data models (currently YANG and RDBMS using SQL).
5. It is permissible that different types of Policies are used in the same system.	Fully Meets	The MEF PDO information model has been designed to support three different types of Policies: imperative, declarative, and intent. Additional types can be supported using the PDO design abstractions.
6. It is recommended that different actors are represented by Roles, which are then mapped to different levels in the Policy Continuum.	Fully Meets	The MEF PDO information model supports Roles natively.
7. It is recommended that the notion of a Policy Continuum, where each continuum corresponds to a given set of actors, is realized.	Fully Meets	Metadata can be used to augment Roles. In addition, traditional and new patterns are being developed to focus on simplifying this task.
8. It is recommended that formal mathematical transformations between each level of the Policy Continuum and its actors are defined.	Fully Meets	Metadata can be used to augment Roles to carry transformation information. In addition, traditional and new patterns are being developed to focus on simplifying this task. Finally, the work in [i.2] and [i.7] is fully supported.
9. it is permissible that the context of an actor is used to map the activities of the actor to a particular level in the Policy Continuum.	Fully Meets	The MEF PDO information model defines the notion of a context-aware policy [i.2] and [i.7].
10. It is required that the Policy information model is integrated into the system's information model.	Fully Meets	The MEF PDO information model is designed to fit into the MEF Core model, as well as any model that can separate the structure of a policy from its content and semantics.
11. It is required that the information model is used in constructing Policy APIs and DSLs.	Fully Meets	The MEF PDO information model uses a number of traditional software patterns to simplify the development of APIs and DSLs. More importantly, the MEF PDO information model uses classification theory to ensure that each class is rooted and semantically distinct; this enables model elements to be used as grammatical elements.
12. It is required that Policy, when used within a domain, act as a mechanism to manage behaviour.	Fully Meets	The MEF PDO information model defines domains, metadata, and context-aware policies to manage behaviour. This is based on work dating back to 2005.
13. It is required that Policy, when used between domains, act as a mechanism to negotiate behaviour.	Fully Meets	Same as above.
14. It is required that Context is related to, and updated by, information from applicable functional blocks…, so that as context changes, the management system is aware of such changes and can respond accordingly.	Fully Meets	The MEF PDO information model supports context-aware policies natively.
15. It is required that Context is used to select active Policies (and deselect Policies that are no longer appropriate). This enables the system to adjust its goals and behaviours accordingly to changes in context.	Fully Meets	Metadata can be used to augment context-aware policies, as well as other important elements, to meet this requirement.
16. It is permissible that the system, based on contextual changes, construct new policies to address problems found and/or improve system behaviour by using operator intervention if no suitable policies currently exist.	Fully Meets	This is clearly dependent on the nature and content of the policy being altered. Changing an IP address as a result of a planned migration is one thing - changing a traffic engineering policy is much more difficult. Predecessors of the MEF PDO information model have been used for this purpose before. Note that a policy can be cloned and modified by the policy engine (as opposed to the policy itself being self-modified). The changes allowed by this process policy can be cloned and modified by the policy engine (as opposed to the policy itself being self-modified). The changes will be limited according to parameters defined in the architecture GR.
17. It is not permitted to use self-modifying policies in this stage of ENI.	Fully Meets	See above. The MEF PDO information model does not mandate or support mechanisms to self-modify policies.

Requirement	Status	Comments
18. It is recommended that Policy is able to be augmented by metadata. Specifically, metadata is permitted to be used to describe and/or prescribe Policy functionality.	Fully Meets	The MEF PDO information model formally defines Metadata, and uses it for both description as well as prescription.
19. It is recommended that Policy is able to be realized using centralized and/or distributed means.	Fully Meets	The LSO RA is completely distributed. The MEF PDO project is also completely distributed, though it could also operate in a centralized manner.
20. It is recommended that AI algorithms are used to construct policies.	DOES NOT MEET	This is not hard to do because the MEF PDO information model uses software patterns. However, this function currently does not exist and needs to be constructed.

6 Analysis of the IETF SUPA Policy Model

6.1 Characteristics of the IETF SUPA Policy Model

6.1.1 Overview

The IETF SUPA WG (working group) was chartered in October 2015. The goals of SUPA were threefold:

1) to define a comprehensive information model for defining imperative policy rules;

2) to define YANG modules, based on the information model, to define policy rules that include commands to be sent to device-, technology-, and service-specific YANG models developed elsewhere for managing and configuring devices and services; and

3) to define a generic framework in which SUPA policies are used. Policy rules within an operator's environment can be used to express high-level, possibly network-wide policies to a network management function (within a controller, an orchestrator, or a network element).

The network management function can then control the configuration and/or monitoring of network elements and services.

Note that declarative policies, which specify the goals to be achieve but not how to achieve those goals (also called "intent-based" policies), are out of scope for the initial phase of [i.19].

The work items discussed in SUPA were:

1) SUPA Policy-based Management Framework, which explained the scope of the policy-based management framework, its elements and interfaces and how it relates to existing work of the IETF [i.20].

2) SUPA Generic Policy Information Model, a generic information model composed of policy concepts and vocabulary. It is an information model for representing policies using a common extensible framework that is independent of language, protocol, repository. It is also independent of the level of abstraction of the content and meaning of a policy [i.1].

3) SUPA Generic Policy Data Model, which consists of two YANG policy data modules. The first is a generic policy model that is meant to be extended on an application-specific basis. The second is an exemplary extension of the first generic policy model that defines rules as event-condition-action policies. Both models are independent of the level of abstraction of the content and meaning of a policy [i.16].

4) Applicability of SUPA, an applicability document providing a few examples that demonstrate how the YANG policy data models can be used to express policies that are relevant for network operators [i.21].

The information model in SUPA represents 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 different data models that use different languages, protocols, and repositories to optimize their usage. This framework is divided into three portions:

- a sub-model that represents policy rules as an intelligent container;

- a sub-model that represents different components of a policy rule (e.g. phrases and statements);
- a sub-model that defines metadata that can be attached to both policy rules and policy rule components.

The data model in SUPA includes two YANG data modules: a generic policy model that is meant to be extended on an application-specific basis, and an exemplary extension of the first (generic policy) model that defines policy rules as event-condition-action tuples. Both models are independent of the level of abstraction of the content and meaning of a policy [i.16].

6.1.2 Support for the Policy Continuum

The SUPA policy model supports the Policy Continuum as previously described in the present document (e.g. see clause 4.3). Hence, it will not be further described in this clause.

6.1.3 Policy Entities

6.1.3.1 Policy Entity Overview

Three types of entities are defined in the SUPA work. They are described in more detail in the following clauses.

6.1.3.2 Policy Rule Structure Entities

This class hierarchy is used to model different types of Policy Rules. In SUPA, a Policy Rule is an intelligent container, and aggregates Policy Rule components and metadata. Only a single type of container is defined (for imperative policy rules). However, the set of entities defined in this portion of the model enable other types of containers to be defined. It also defines PolicySource and PolicyTarget, which represent the set of entities that author this Policy and which will be affected by this policy, respectively.

This set of model elements is a subset of the functionality in the current MEF model.

6.1.3.3 Policy Rule Component Structure Entities

This class hierarchy is used to model different types of components of a Policy Rule. In SUPA, the main component of imperative policy rules is a PolicyClause. A PolicyClause is the building block for defining policy statements, and is also applicable to other types of policy rules.

The main subclasses of this include:

- types of PolicyClauses: BooleanClause and EncodedClause (this represents encoded components, such as CLI or YANG statements);
- types of events, conditions, and actions;
- types of objects that can decorate a PolicyClause (e.g. a PolicyTerm, which represents variables, operators, and values to define terms in a policy clause, as well as a Collection object).

This set of model elements is a subset of the functionality in the current MEF model.

6.1.3.4 Policy Rule Metadata Entities

This class hierarchy is used to model different types of metadata that can be used to add descriptive and/or prescriptive information to a Policy Rule or a component of a Policy Rule. There is no limit to the number of metadata objects that can be added to another object.

Two examples of such metadata are defined. Access control metadata defines different types of access control mechanisms, such as mandatory, discretionary, role-based, and attribute-based access control. Version control defines a robust mechanism for adding versioning mechanism for any SUPA object, according to the Semantic Versioning specification.

This set of model elements is a subset of the functionality in the current MEF model.

6.1.4 Design Patterns

SUPA uses two of the same design patterns (Composite [i.6], [i.8] and Decorator [i.8] that are used by the MEF PDO project. Refer to the references for a more complete description of these two patterns.

6.2 Supported Policy Paradigms

SUPA defines a single type of policy - an imperative policy model. Its structure is defined as an event-condition-action (ECA) policy rule, as described in clause 4.4.2. This type of rule is a superset of other forms, such as a condition-action policy rule.

In order to do this, SUPA first defined a framework for defining the concept of policy, called the SUPA Generic Policy Information Model (GPIM). The GPIM defines a common framework as a set of model elements (e.g. classes, attributes, and relationships) that specify a common set of policy management concepts that are independent of the type of policy (e.g. imperative, procedural, declarative, or otherwise).

Then SUPA defined a framework for defining a policy model that uses the event-condition-action (ECA) paradigm, called the SUPA ECA Policy Rule Information Model (EPRIM). 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. It also defines basic metadata classes and relationships that can be used to augment the GPIM [i.16]. The YANG data modules are derived from the GPIM and the metadata.

The SUPA GPIM is a subset of the MEF imperative policy model, defined in clause 5.2.1.

6.3 IETF SUPA Model Features Comparison

This clause compares the features of the current IETF SUPA model to the requirements listed in clause 4.12.

Table 6.3-1: How the IETF SUPA Model Meets the Requirements of clause 4.12

Requirement	Status	Comments
1a. A single information model is required to be used to represent the structure and semantics of Policies.	Fully Meets	The IETF SUPA information model is designed as a module that can plug into any information model as long as it can separate structure, semantics, and metadata.
1b. It is recommended that multiple information models are not to be used.	Fully Meets	The IETF SUPA model is designed as a single atomic information model.
2. It is recommended that multiple data models are derived from the (single) information model.	Fully Meets	The IETF SUPA information model is independent of technology, protocol, language, and platform. In addition, the IETF SUPA project has already defined a YANG data model.
3. The (single) information model, and each data model, are required to use a set of software design patterns to provide extensibility and consistency.	Fully Meets	The IETF SUPA information model uses a number of traditional software patterns.
4. Formal transformations from the (single) information model are required to derive each different data model.	Partially Meets	The IETF SUPA information model defines patterns to transform an object-oriented information model to a YANG data model. However, the transformation of other data models is not addressed.
5. It is permissible that different types of Policies are used in the same system.	Partially Meets	The IETF SUPA information model has been designed to support two different types of imperative Policies. Additional types can be supported using the IETF SUPA design abstractions.
6. It is recommended that different actors are represented by Roles, which are then mapped to different levels in the Policy Continuum.	Fully Meets	The IETF SUPA information model supports Roles natively.
7. It is recommended that the notion of a Policy Continuum, where each continuum corresponds to a given set of actors, is realized.	Fully Meets	Metadata can be used to augment Roles. In addition, traditional and new patterns are being developed to focus on simplifying this task. Finally, the IETF SUPA model defines an attribute to carry the continuum level associated with a Policy.

Requirement	Status	Comments
8. It is recommended that formal mathematical transformations between each level of the Policy Continuum and its actors are defined.	Partially Meets	The Policy Continuum is supported [i.2] and [i.7]; work to define formal mathematical transformations is currently on hold.
9. it is permissible that the context of an actor is used to map the activities of the actor to a particular level in the Policy Continuum.	Does Not Meet	Work to define formal mathematical transformations is currently on hold.
10. It is required that the Policy information model is integrated into the system's information model.	Partially Meets	The IETF SUPA information model is designed to fit into a higher-level model, as well as any model that can separate the structure of a policy from its content and semantics.
11. It is required that the information model is used in constructing Policy APIs and DSLs.	Partially Meets	The IETF SUPA information model uses a number of traditional software patterns to simplify the development of APIs and DSLs. However, this work is currently on hold.
12. It is required that Policy, when used within a domain, act as a mechanism to manage behaviour.	Partially Meets	The IETF SUPA information model was constructed to support these concepts, but their full definition is currently on hold.
13. It is required that Policy, when used between domains, act as a mechanism to negotiate behaviour.	Partially Meets	Same as above.
14. It is required that Context is related to, and updated by, information from applicable functional blocks…, so that as context changes, the management system is aware of such changes and can respond accordingly.	Partially Meets	The IETF SUPA information model was constructed to support context-aware policies (as defined in [i.2] and [i.7]); however, Context is *not* defined in the IETF SUPA model.
15. It is required that Context is used to select active Policies (and deselect Policies that are no longer appropriate). This enables the system to adjust its goals and behaviours accordingly to changes in context.	Partially Meets	Same as above.
16. It is permissible that the system, based on contextual changes, construct new policies to address problems found and/or improve system behaviour by using operator intervention if no suitable policies currently exist.	Does Not Meet	Context is not currently defined in the IETF SUPA information model.
17. It is not permitted to use self-modifying policies in this stage of ENI.	Fully Meets	There is no current mechanism defined in the IETF SUPA information model to support this feature.
18. It is recommended that Policy is able to be augmented by metadata. Specifically, metadata is permitted to be used to describe and/or prescribe Policy functionality.	Fully Meets	The IETF SUPA information model formally defines Metadata, and uses it for both description as well as prescription.
19. It is recommended that Policy is able to be realized using centralized and/or distributed means.	Fully Meets	There is nothing inherent in the design of the IETF SUPA information model to bias its use to centralized or distributed implementation.
20. It is recommended that AI algorithms are used to construct policies.	DOES NOT MEET	AI mechanisms are not currently defined in the IETF SUPA information model.

7 Analysis of the TM Forum SID Policy Model

7.1 Characteristics of the TM Forum SID Policy Model

7.1.1 Overview

The latest version of the SID Policy Model is [i.22], and is part of the 17.0 Frameworx Suite. It should be noted that this version is based on an old model of DEN-ng (3.0, released in 2003, with some important differences), though it had a minor update in 2010.

The TMF SID policy model has several important characteristics:

- It was based on supporting the Policy Continuum
- It consisted of three sets of entities:
  - Objects that model the content of a policy
  - Objects that define invariant characteristics of a policy
  - Objects that model how to build a policy-based application
- It uses three design patterns:
  - Composite pattern (for building tree-like hierarchies of objects)
  - Role-object pattern (for defining behaviour via delegation)
  - Entity-EntitySpecification (to separate invariant characteristics of an object from the individual characteristics of an object)

The following clauses elaborate on these characteristics.

7.1.2 Support for the Policy Continuum

The TMF SID policy model supported the Policy Continuum as previously described in the present document (e.g. see clause 4.3). Hence, it will not be further described in this clause.

7.1.3 Policy Entities

7.1.3.1 Policy Entity Overview

The three types of entities contained in the TMF SID policy model are described in the following clauses.

7.1.3.2 Policy Content Entities

The main types of entities of this type include PolicySet, PolicyGroup, PolicyStatement, PolicyEvent, PolicyCondition, and PolicyAction. Note that only imperative policies are defined by the TMF SID policy model.

The main difference between the PolicyEvent, PolicyCondition, and PolicyAction classes defined in the TMF SID policy model and those in the MEF PDO model are that the former use recursive aggregations, whereas the latter use the composite pattern. The ramifications of this difference are beyond the scope of the present document.

The PolicySet and PolicyGroup classes are containers that enable a group of PolicyRules as well as individual PolicyRules to be used. The motivation was to ensure backwards compatibility with IETF RFC 3060 [i.27]. Note that PolicySet has a recursive aggregation that makes its use problematic.

A PolicyStatement is similar in nature to the PolicyClause object of the MEF PDO model.

7.1.3.3 Policy Specification Entities

The main types of entities of this type include specification classes for the six types of entities defined in clause 7.1.3.2. This creates double the number of classes and relationships compared to the MEF PDO model. In addition, the specification class suffers from creating a large number of objects that do not provide the flexibility of other patterns, such as the Decorator pattern.

7.1.3.4 Policy Application Entities

The main types of entities of this type include PolicyApplication, PolicyServer, PolicyBroker, PolicyDecisionPoint, PolicyEnforcementPoint, and PolicyExecutionPoint.

The PolicyApplication class is the superclass for the other five classes. The description of each of these classes matches that of [i.2]; however, the class definitions are primitive in nature and require additional specificity to be usable.

7.1.4 Design Patterns

The TMF SID policy model uses two of the same design patterns (Composite [i.6], [i.8] and Role-object [i.8]) that are used by the MEF PDO project. Refer to the references for a more complete description of these two patterns.

The third pattern, Entity-EntitySpecification, is unique to the TMF SID. It is described in [i.23]. While it is used throughout different SID domains, recent analysis has revealed that this pattern has a number of problems, including object explosion. Therefore, it is not recommended for use.

7.2 Supported Policy Paradigms

7.2.1 Characteristics of the TM Forum SID Policy Model

The TM Forum SID model defines a single type of policy - an imperative policy model. Its structure, however, is only loosely defined. Its model is a subset of the functionality provided by the MEF imperative policy model.

7.3 TM Forum SID Model Features Comparison

This clause compares the features of the current IETF SUPA model to the requirements listed in clause 4.12.

Table 7.3-1: How the TM Forum SID Model Meets the Requirements of clause 4.12

Requirement	**Status**	**Comments**
1a. A single information model is required to be used to represent the structure and semantics of Policies.	Does Not Meet	The TM Forum SID information model is *not* designed as a module that can plug into any information model. Furthermore, the TM Forum SID model is *not* defined as a single rooted model, which makes this task very difficult.
1b. It is recommended that multiple information models are not to be used.	Does Not Meet	Same as above. Note that, by definition, a model that does not have a single root makes meeting this requirement impossible.
2. It is recommended that multiple data models are derived from the (single) information model.	Partially Meets	The TM Forum SID information model is independent of technology, protocol, language, and platform. However, the TM Forum API data model does not conform to the TM Forum SID model, and does *not* meet this requirement.
3. The (single) information model, and each data model, are required to use a set of software design patterns to provide extensibility and consistency.	Partially Meets	The TM Forum SID information model uses a limited number of traditional software patterns. However, it also uses patterns that have serious deficiencies [i.24].
4. Formal transformations from the (single) information model are required to derive each different data model.	Does Not Meet	The TM Forum SID information model does *not* define any formal transformation mechanisms.
5. It is permissible that different types of Policies are used in the same system.	Does Not Meet	The TM Forum SID information model does *not* support anything but imperative policies.
6. It is recommended that different actors are represented by Roles, which are then mapped to different levels in the Policy Continuum.	Does Not Meet	The TM Forum SID information model does *not* support Roles natively.
7. It is recommended that the notion of a Policy Continuum, where each continuum corresponds to a given set of actors, is realized.	Does Not Meet	The TM Forum SID Information Model does *not* support the notion of a Policy Continuum.
8. It is recommended that formal mathematical transformations between each level of the Policy Continuum and its actors are defined.	Does Not Meet	The TM Forum SID Information Model does *not* support the notion of a Policy Continuum.
9. it is permissible that the context of an actor is used to map the activities of the actor to a particular level in the Policy Continuum.	Does Not Meet	The TM Forum SID Information Model does *not* support the notion of context.

Requirement	Status	Comments
10. It is required that the Policy information model is integrated into the system's information model.	Does Not Meet	The TM Forum SID information model is *not* designed as a module that can plug into any information model. Furthermore, the TM Forum SID model is *not* defined as a single rooted model, which makes this task very difficult.
11. It is required that the information model is used in constructing Policy APIs and DSLs.	Does Not Meet	The TM Forum SID information model is *not* compatible with the TM Forum API data model.
12. It is required that Policy, when used within a domain, act as a mechanism to manage behaviour.	Does Not Meet	The TM Forum SID information model does *not* support the concept of a domain.
13. It is required that Policy, when used between domains, act as a mechanism to negotiate behaviour.	Does Not Meet	Same as above.
14. It is required that Context is related to, and updated by, information from applicable functional blocks…, so that as context changes, the management system is aware of such changes and can respond accordingly.	Does Not Meet	The TM Forum SID Information Model does *not* support the notion of context.
15. It is required that Context is used to select active Policies (and deselect Policies that are no longer appropriate). This enables the system to adjust its goals and behaviours accordingly to changes in context.	Does Not Meet	The TM Forum SID Information Model does *not* support the notion of context.
16. It is permissible that the system, based on contextual changes, construct new policies to address problems found and/or improve system behaviour by using operator intervention if no suitable policies currently exist.	Does Not Meet	The TM Forum SID Information Model does *not* support the notion of context.
17. It is not permitted to use self-modifying policies in this stage of ENI.	Fully Meets	There is no current mechanism defined in the TM Forum SID information model to support this feature.
18. It is recommended that Policy is able to be augmented by metadata. Specifically, metadata is permitted to be used to describe and/or prescribe Policy functionality.	Does Not Meet	The TM Forum SID Information Model does *not* support the notion of metadata.
19. It is recommended that Policy is able to be realized using centralized and/or distributed means.	Fully Meets	There is nothing inherent in the design of the TM Forum information model to bias its use to centralized or distributed implementation.
20. It is recommended that AI algorithms are used to construct policies.	DOES NOT MEET	AI mechanisms are not currently defined in the TM Forum SID information model.

Annex A:
Authors & contributors

The following people have contributed to the present document:

Rapporteur:
Dr. John Strassner, Huawei Technologies

Other contributors:
Antonio Gamelas, Portugal Telecomunications

Dr. Shucheng(Will) Liu, Huawei Technologies

Annex B: Bibliography

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

Peled, D.: "Software Reliability Methods", Springer, 2001 ISBN 0-38-795106-7.

Annex C: Change History

Date	Version	Information about changes
2017-07	001	Initial creation from ENI(17)000018, ENI(17)000033, and ENI(17)000032
2017-08	002	Implemented Change Requests: ENI(17)000035r1, ENI(17)000037, and ENI(17)000037
2017-09	003	Implemented Change Requests: ENI(17)000055r1, ENI(17)000058r2, ENI(17)000061, and ENI(17)000070r1
2017-09	004	Implemented Change Requests: ENI(17)000082 and ENI(17)000083
2018-01	005	Implemented Change Requests: ENI(18)000050 and ENI(18)000050r1
2018-02	006	Final proposed content revisions
2018-03	007	Corrected typos and spacing

History

Document history		
V1.1.1	May 2018	Publication