MEF Standard

MEF 95

MEF Policy Driven Orchestration (PDO)

July 2021

Disclaimer

The information in this publication is freely available for reproduction and use by any recipient and is believed to be accurate as of its publication date. Such information is subject to change without notice and MEF Forum (MEF) is not responsible for any errors. MEF does not assume responsibility to update or correct any information in this publication. No representation or warranty, expressed or implied, is made by MEF concerning the completeness, accuracy, or applicability of any information contained herein and no liability of any kind shall be assumed by MEF as a result of reliance upon such information.

The information contained herein is intended to be used without modification by the recipient or user of this document. MEF is not responsible or liable for any modifications to this document made by any other party.

The receipt or any use of this document or its contents does not in any way create, by implication or otherwise:

a) any express or implied license or right to or under any patent, copyright, trademark or trade secret rights held or claimed by any MEF member which are or may be associated with the ideas, techniques, concepts or expressions contained herein; nor

b) any warranty or representation that any MEF members will announce any product(s) and/or service(s) related thereto, or if such announcements are made, that such announced product(s) and/or service(s) embody any or all of the ideas, technologies, or concepts contained herein; nor

c) any form of relationship between any MEF member and the recipient or user of this document.

Implementation or use of specific MEF standards, specifications, or recommendations will be voluntary, and no Member shall be obliged to implement them by virtue of participation in MEF Forum. MEF is a non-profit international organization to enable the development and worldwide adoption of agile, assured and orchestrated network services. MEF does not, expressly or otherwise, endorse or promote any specific products or services.

Table of Contents

List of Figures

List of Tables

1 List of Contributing Members

The following members of the MEF participated in the development of this document and have requested to be included in this list.

CMC
Futurewei
PCCW Global
Verizon

Table 1. Contributing Member Companies

2 Abstract

This document specifies how Policy-based management and modeling can be used to realize and augment the orchestration functionality defined in the MEF Lifecycle Service Orchestration (LSO) Reference Architecture (RA), MEF 55.1 [1]. It may also be used to define other types of services, such as choreography and collaboration.

This document defines what Policy is, how policy-based management is used in the LSO RA, some exemplary use cases, and architectural extensions to the LSO RA. This will enable policies to be exchanged across APIs as defined in MEF 55.1, including (but not limited to) CANTATA, ALLEGRO, and SONATA. CANTATA and ALLEGRO enable policy-based interaction between the Customer and the Provider, while SONATA is critical for policy-based interaction in a carrier supply chain.

The policy model defined in this document is developed from MEF 78.1 [2] (i.e., the MEF Core Model, or MCM). This means that the top of the policy model (i.e., MCMPolicyObject) is a subclass of the MCM.

This document uses modeling best practices (e.g., [3][4][5]), and a number of software patterns (e.g., [6][7][8]) to provide an extensible framework that can support model-driven engineering [9] as well as the needs of DevOps-inspired automation. It defines concepts and functions that can be represented to define policies, as well as associated data, exchanged at all seven of the Interface Reference Points currently defined in [1].

The policy model defined in this document is an object-oriented information model, and hence, is independent of any specific architectural paradigm (e.g., resource- or service-oriented architectures), protocol, or platform.

The policy model defined in this document can be used to represent a number of different types of policies, including (but not limited to) imperative, declarative, intent, and utility function policies. However, the focus of this document is on imperative and intent policies.

This document normatively includes the content of the following Papyrus UML files as if they were contained within this document from the following MEF GitHub Repositories:

https://github.com/MEF-GIT/MEF-General-Common/

MCM: (MCM.di, MCM.notation, and MCM.uml)

MEF_Types: (MEF_Types.di, MEF_Types.notation, and MEF_Types.uml)

3 Terminology and Abbreviations

This section defines the terms used in this document. In many cases, the normative definitions to terms are found in other documents. In these cases, the third column is used to provide the reference that is controlling, in other MEF or external documents.

Term	Definition	Reference
Abstract Class	An abstract class is a class that cannot be directly instantiated. It can have abstract or concrete subclasses.	MCM ([2])
Abstraction	Abstraction is the process of focusing on the important characteristics and behavior of a concept, and ignoring less important characteristics and behavior.	MCM ([2])
Action	An Action defines a set of operations that may be performed on a set of managed entities. 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.	THIS DOCUMENT
Capability	A set of features that are available from a Component. These features may, but do not have to, be used. Capabilities should be announced through a dedicated Interface.	THIS DOCUMENT
Class	A class is a template for defining a specific type of object that exhibits a common set of characteristics and behavior.	MCM ([2])
Concrete Class	A concrete class is a class that can be directly instantiated. Once a class has been defined as concrete in the hierarchy, all of its subclasses are required to be concrete.	MCM ([2])
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 what decision to make.	THIS DOCUMENT
Controlled Language	A language that restricts the grammar and vocabulary used.	THIS DOCUMENT
Customer	A Customer is the organization purchasing, managing, and/or using Connectivity Services from a Service Provider. This may be an end-user business organization, mobile operator, or a partner network operator.	MCM ([2])

Term	Definition	Reference
Customer (Role)	MCMCustomer is a concrete class, and specializes MCMPartyRole. It represents a particular type of MCMPartyRole that defines a set of people and/or organizations that buy, manage, or use MCMProducts from an MCMServiceProvider. The MCMCustomer is financially responsible for purchasing an MCMProduct. The MCMCustomer is the MCMPartyRole that is purchasing, managing, and/or using Services from an MCMServiceProvider.	MCM ([2])
Data Model	A data model is a representation of concepts of interest to an environment in a form that is dependent on data repository, data definition language, query language, implementation language, and/or protocol (typically, but not necessarily, all five).	MCM ([2])
Declarative Policy	A type of policy that uses statements to express the goals of the policy, but not how to accomplish those goals. Hence, state is not explicitly manipulated, and the order of statements that make up the policy is irrelevant. *In this document, Declarative Policy will refer to policies that execute as theories of a formal logic* (see below).	THIS DOCUMENT
Domain	This is an abstract class, and specializes Entity. A Domain is a collection of Entities that share a common purpose. In addition, each constituent Entity in a Domain is both uniquely addressable and uniquely identifiable within that Domain.	MCM ([2])
Event	An Event is defined as anything of importance to the management system (e.g., a change in the system being managed and/or its environment) occurring on a time-axis.	THIS DOCUMENT
Federation	A federation is the result of a process that enables a group of different systems, which may have different internal structures, to agree upon mechanisms that provide interoperability.	THIS DOCUMENT
Formal Logic	The use of inference applied to the form, or content, of a set of statements. 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.	THIS DOCUMENT

Term	Definition	Reference
Formal Methods	Formal Methods describe a set of mathematical theories, such as logic, automata, graph or set theory, and provide associated notations for describing and analyzing systems.	THIS DOCUMENT
Imperative Policy	A type of policy that uses statements to explicitly change the state of a set of targeted objects. Hence, the order of statements that make up the policy is explicitly defined. *In this document, Imperative Policy will refer to policies that are made up of Events, Conditions, and Actions.*	THIS DOCUMENT
Information Model	An information model is a representation of concepts of interest to an environment in a form that is independent of data repository, data definition language, query language, implementation language, and protocol.	MCM ([2])
Intent Policy	A type of 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 this 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 DOCUMENT
LSO (Lifecycle Service Orchestration)	Open and interoperable automation of management operations over the entire lifecycle of Layer 2 and Layer 3 Connectivity Services. This includes fulfillment, 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 service.	MEF 55.1 ([1])
LSO RA (LSO Reference Architecture)	A layered abstraction architecture that characterizes the management and control domains and entities, and the interfaces among them, to enable cooperative orchestration of Connectivity Services. Note that in this document, cooperative orchestration is NOT limited to only Connectivity Services, and may include other services as well.	MEF 55.1 ([1])

Term	Definition	Reference
Metadata	Metadata is a class that contains prescriptive and/or descriptive information about the object(s) to which it is attached. While metadata can be attached to any information model element, this document only considers metadata object instances attached to class instances and relationships.	MCM ([2])
Model Element	An element of a model. For the purposes of this document, this refers to a set of classes, attributes, operations, constraints, and/or relationships.	MCM ([2])
Object	An instance of a (concrete) class.	MCM ([2])
Partner	An organization providing Products and Services to the Service Provider (Buyer) in order to allow the Service Provider to instantiate and manage Service Components external to the Service Provider domain.	MCM ([2])
Partner (Role)	MCMPartner is a concrete class, and specializes MCMPartyRole. It represents a particular type of MCMPartyRole that provides MCMProducts and MCMServices to the MCMServiceProvider in order to instantiate and manage MCMService elements, such as MCMServiceComponents, external to the Service Provider's Domain.	MCM ([2])
Pattern	A pattern describes a named, generic, reusable solution to a problem that applies to a particular context. A pattern is not a finished design, but rather, is a reusable template that defines a set of objects, and their interactions, that can be adapted to meet the context-specific needs required to solve a problem.	[4][6][7][8]
Policy	Policy is a set of rules that are used to manage and control the changing and/or maintaining of the state of one or more managed objects.	THIS DOCUMENT
Relationship	For the purposes of this document, a relationship can be any type of association, aggregation, or composition.	MCM ([2])

Term	Definition	Reference
Role	The Role-Object pattern enables an object to adapt to the needs of different applications and contexts by transparently attaching and/or removing Role Objects. Each Role Object defines a set of responsibilities that the object has to play in that client's context. Each context may be its own application, which therefore gets decoupled from other applications. The Role-Object pattern is implemented in the MCM by aggregating Role objects, which are defined as a type of Metadata, to other objects (to enforce the separation of defining an object vs. defining responsibilities that the object has to play).	MCM ([2])
Role (MCM class)	This is an abstract class, and specializes MCMMetadata. It represents a set of characteristics and behaviors (also referred to as responsibilities) that an object takes on in a particular context. This enables an object to adapt to the needs of different clients through transparently attached role objects (as opposed to having to alter the inherent nature of the object itself). The Role Object pattern models context-specific views of an object as separate role objects that are dynamically attached to and removed from the core object to which the MCMRole objects are attached.	MCM ([2])
Service Provider	The organization providing Ethernet Service(s). Note that in this document, as well as in [1], the (Service Provider) organization is NOT limited to providing only Ethernet Services.	MCM ([2])
Service Provider (Role)	MCMServiceProvider is a concrete class, and specializes MCMPartyRole. It represents a particular type of MCMPartyRole that provides MCMProducts. This specifically includes MCMServices.	MCM ([2])
Unified Modeling Language (UML)	The objective of UML is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes.	OMG UML 2.5.1 [12]

Term	Definition	Reference
Whole-Part Relationship	A whole-part relationship is one in which one set of entities aggregates another set of entities. In such a relationship, three objects are created (the entity doing the aggregation, the set aggregated entities, and the combination of the aggregating entity and its aggregated entities). More formally, a whole-part relationship is a *partial ordering* that is reflexive, transitive, and anti-symmetric (i.e., everything is a part of itself, any part of any part of an entity is itself a part of that entity, and two distinct entities cannot be part of each other).	Various; see for example Stanford Encyclopedia of Philosophy

Table 2. Terminology and Abbreviations

4 Compliance Levels

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 (RFC 2119 [10], RFC 8174 [11]) when, and only when, they appear in all capitals, as shown here. All key words must be in bold text.

Items that are **REQUIRED** (contain the words **MUST** or **MUST NOT**) are labeled as **[Rx]** for required. Items that are **RECOMMENDED** (contain the words **SHOULD** or **SHOULD NOT**) are labeled as **[Dx]** for desirable. Items that are **OPTIONAL** (contain the words **MAY** or **OPTIONAL**) are labeled as **[Ox]** for optional**.**

5 Numerical Prefix Conventions

This document uses the prefix notation to indicate multiplier values as shown in Table 3.

Decimal		**Binary**	
Symbol	**Value**	**Symbol**	**Value**
k	10^3	Ki	2^{10}
M	10^6	Mi	2^{20}
G	10^9	Gi	2^{30}
T	10^{12}	Ti	2^{40}
P	10^{15}	Pi	2^{50}
E	10^{18}	Ei	2^{60}
Z	10^{21}	Zi	2^{70}
Y	10^{24}	Yi	2^{80}

Table 3. Numerical Prefix Conventions

6 Introduction

This specification uses UML (Unified Modeling Language) [12] to describe the salient characteristics and behavior of imperative and intent policies, and how each can affect the behavior of entities that are important to the managed environment. In particular, it explains what a MEF policy is, how behavior of a managed entity is controlled using policies and defines an information model for describing imperative and intent policies used in the MEF Lifecycle Service Orchestration Reference Architecture.

This document is intended for developers and users that need the formalism that an information model provides. An information model represents concepts, along with their relationships and semantics, to help specify an extensible and structured, shareable, information repository.

7 Introduction to Policy Management and Orchestration

This chapter defines what a Policy is, why it is important to Orchestration, and how it can be used in the MEF LSO RA.

7.1 Controlling Behavior Using Policies

Management involves monitoring the activity of a system, making decisions about how the system is acting, and performing control actions to modify the behavior of the system. The purpose of policy is to ensure that consistent decisions are made governing the behavior of a system.

Organizations are policy-driven entities. Policy is a natural way to express rules and restrictions on behavior, 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.

This project will use 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.* [13]

Policy is a mechanism for *controlling the behavior* of an Entity. Two important types of 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 must or must not do and hence, guide the decision-making process. These two types of Policies are based on *deontic logic* [14]. Their difference is shown in Figure 1.

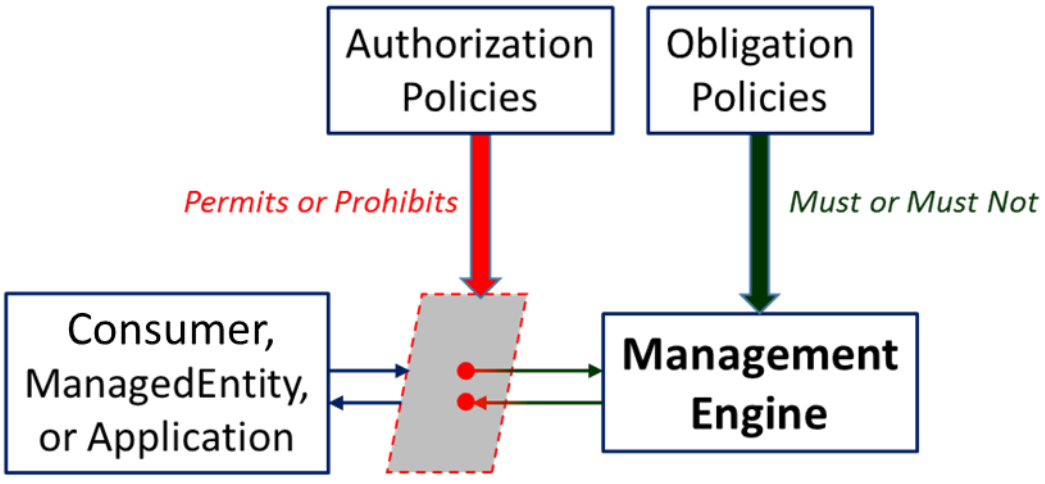


Figure 1. Deontic Policy Rules

7.1.1 Groups of Policies and the Use of Roles

In an object-oriented approach, the external behavior of an object defines how it interacts with other objects in its environment. However, in any real-world system, the number of objects is so large that it is impractical to specify policies for individual objects. Instead, mechanisms should be used to select groups of objects that have similar behaviors and/or responsibilities. One such mechanism is called *roles* [4][7]. This pattern models different context-specific views of an object as *separate Role objects*, which can be dynamically attached to and detached from the object that has those roles. Policies can be associated with sets of Roles to make management easier and to provide consistency in enforcement. Hierarchies of policies can be formed using the Composite Pattern [8], the Decorator Pattern [6], and other more complex approaches. The use of policy hierarchies enables a set of policies that apply to a parent Domain to propagate to each sub-domain that is contained in the parent Domain.

7.1.2 Can Multiple Policies Apply to a Single Object?

A Policy defines how a set of objects interact. Hence, if an object interacts with multiple objects, it can have Policies for each interaction. Note that this includes defining multiple Policies between the same two objects. For example, a particular entity (e.g., user or application) could have different access control permissions on different objects in the same server, or access to a particular object may depend on context (e.g., time of day, type of connection, and/or what role the user is logged in as).

7.1.3 Policy Subjects and Targets

Policy literature talks about policy *subjects* and policy *targets*. The subject of a policy is the entity that is executing the policy, and the target of a policy is the set of entities that are affected by the policy. This is important for the underlying logic used by imperative policies, as it defines the scope of how the policy is executed. This is examined more in the following section.

7.1.4 Authorization vs. Obligation Policies

An Authorization Policy defines the set of operations that a subject is permitted to do in terms of the operations it is authorized to perform on a target object. Permission Policies are Authorization Policies that are positive in nature (i.e., the subject is permitted to do an operation). In contrast, Prohibition Policies Authorization Policies that are negative in nature. Authorization Policies are considered *target-based*, since the operation(s) being executed by the subject are directed to a set of target entities. They affect the state of the target entities.

An Obligation Policy defines what the set of operations that a subject must (or must not) do. An important underlying assumption is typically made for Obligation Policies: all subjects act in a predictable and consistent manner, and *always* attempt to carry out Obligation Policies with *no freedom of choice*. Obligation policies are *subject-based*, since the subject is responsible for interpreting the policy and performing the set of activities specified.

Both Authorization and Obligation Policies may have constraints associated with them. Constraints are typically expressed as a predicate based on the context that the object is operating in, and/or the state of the object.

Examples include:

- John is permitted to read file F1 (positive authorization)
- John is prohibited to read, write, or execute file F3 (negative authorization)
- John is permitted to access the Code Server only if he is on the Company Intranet (positive authorization with constraint)
- A user's logon is suspended if the user fails authentication three times (positive, event triggered constraint)
- Systems whose software is less than 2.3 must not perform dynamic inventory operations (negative obligation applying to subjects that have a particular state)

Hence, the key difference between Authorization and Obligation Policies is that the former is target-based, whereas the latter is subject-based.

7.2 The Policy Continuum

The Policy Continuum [17] 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 do not work in terms of low-level constructs, such as CLI or formal logic. Similarly, actors that use low-level constructs, such as CLI, will likely not want to use policies defined in terms of high-level abstractions. This is especially true of policies written in natural language, since natural language can be very ambiguous. 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. This is illustrated in Figure 3. In this figure, two different actors are working on a common concept, called a Service Level Agreement (SLA). The business user on the left 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 (and what to do when the SLA is violated). This user (e.g., a network admin) deals in terms of low-level functions of the device. 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****?*

This is illustrated conceptually in Figure 2.

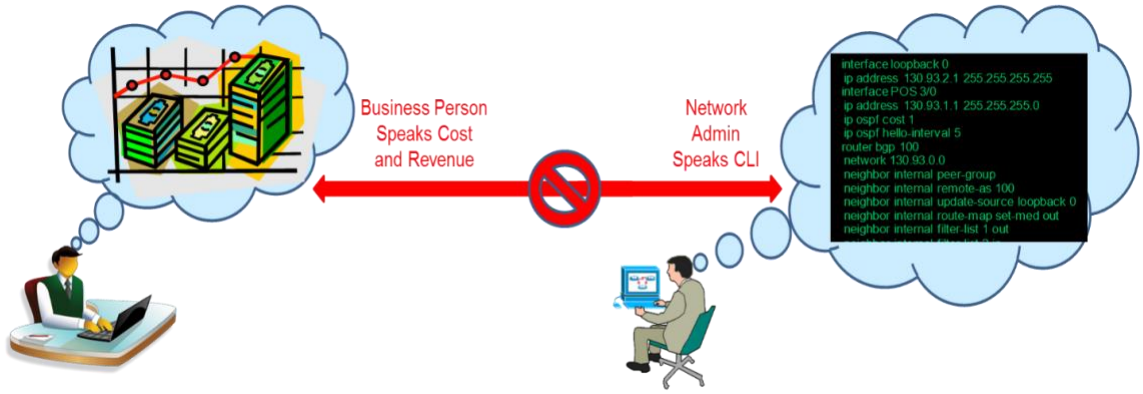


Figure 2. The Same Concept Having Different Meanings for Different Users

Note that Figure 2 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 3.

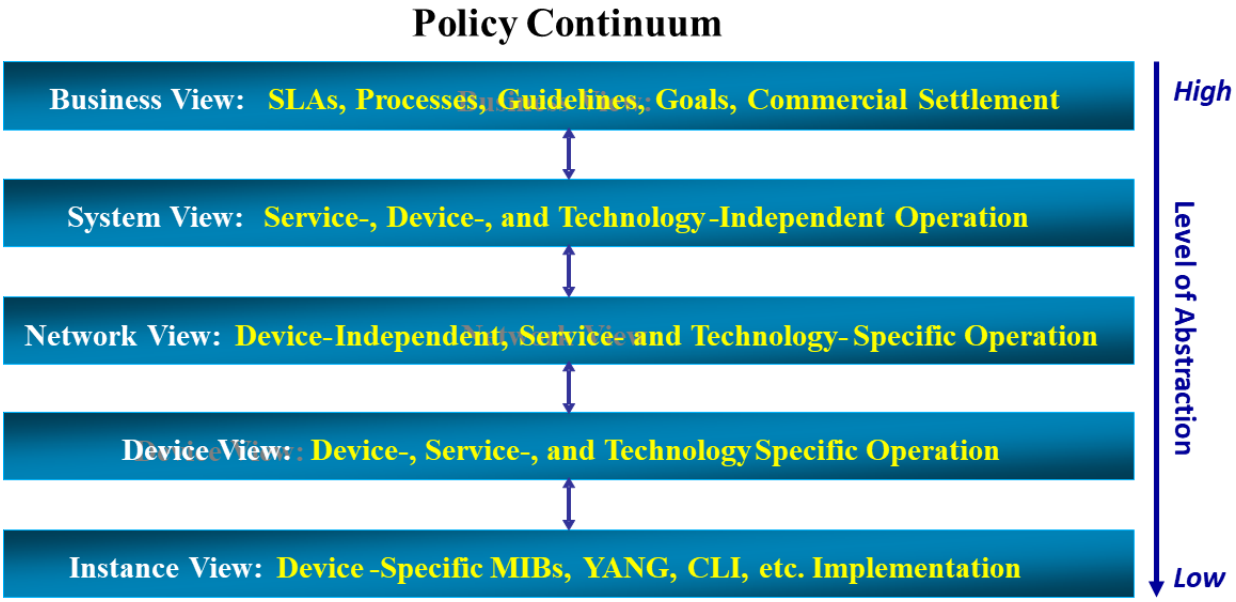


Figure 3. An Example of the Policy Continuum

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).

7.3 Proving the Correctness of a Policy

It is difficult to show that a system is operating correctly. Note that this is NOT the same as showing that it can meet its requirements through testing. Dijkstra wrote: “Program testing can be used to show the presence of bugs, but never to show their absence”.

Furthermore, other types of proofs are hard. Once again, Dijkstra wrote: “One can never guarantee that a proof is correct, the best one can say is: ‘I have not discovered any mistakes’”.

Note that automatic proof is not possible for generic programs, due to the undecidability of the halting problem (Turing). However, there is a branch of science called Formal Methods [15], which describe a set of mathematical theories, such as logic, automata, graph or set theory, (i.e., “formal”) notations for describing and analyzing systems. The advantage of such methods is that they can unambiguously describe the system and/or its properties. Formal analysis and verification techniques serve to verify that a system satisfies its specification (or help in finding out why it does not do so).

Thus, formal methods can be used to describe the system; this produces a formal specification. Then, properties about the specification can be proven; this is called formal verification. A program can then be derived from the (formal) specification; this is called formal synthesis. This can be used to increase the confidence in the reliability of the system. However, this is difficult, since most specifications are declarative and do not use precise, unambiguous language.

Modal Logics [16] are a family of formal logics that can reason about possibility and necessity. There are different modalities that can be used; the most common are temporal, deontic (“it ought to be the case that”), epistemic (“I know that”), and doxastic (“I believe that”). These logics are important, because classical logic is *static*, and for the LSO RA, truth may vary over time. Note that each of these can be complex. For example, temporal logic depends on how time is modeled (For example: Is time linear? Can time branch? Is it discrete or continuous? What about instances vs. intervals?).

Deontic logic is useful to reason about what the system does when constraints that it defines are violated. It is widely used in many applications, such as law and security systems.

7.4 Policy Usage in the MEF LSO RA

Traditionally, policy is thought of as a set of rules. Each rule expresses a set of conditions to be monitored and, if those conditions are met, one or more actions will be executed. This is one form of an *imperative* policy. However, this definition fails to take into account the different users that want to use policy, as well as the different forms that policy can take.

Different types of people use policy. Business people don’t want to express their policies in networking terminology, because typically, because typically business people understand high level technological concepts but may not be familiar with the fine details of network terminology. Similarly, networking people don’t want policies written using business concepts for the exact same reasons. However, both business and network personnel must work together to ensure that network services are managed according to the business goals of the organization. This document defines an information model that enables different types of policies to express the needs of each

constituency using the terminology and concepts of that constituency; the resulting model objects for each constituency are then used as a consensual *lingua franca.* For example, the model itself may be used to build a grammar, or set of grammars, that maps between different concepts and terminologies. This is described in [17] as the "Policy Continuum", and is shown in Figure 3.

The idea behind the Policy Continuum is that there are a number of different users of Policy (while Figure 3 has identified five, there may be more or less for specific implementations). Each level addresses a different type of user that has a different understanding of the shared entities operating at that particular level of abstraction. For example, the business view provides a description of business entities, such as customer, service and SLA, in business terms. The system view uses these same entities but adds detail that is not appropriate at the business level. In addition, the system level may introduce additional entities to support one or more business processes defined at the business level.

> *The information model serves as a common language that enables concepts in each level of the Policy Continuum to be mapped to equivalent concepts in other levels.*

This is an important point. As a simple example, consider the business definition of a VPN. Businesses view the VPN as a service to be provided to specific customers and are not necessarily concerned about the lower-level details of *how* that VPN service is managed and supported. At the system view, however, these things become important. Implementation questions, such as what type of VPN, will be asked to add detail that is necessary to be able to build the VPN. This will lead to more detailed views that focus on the definition of specific entities. This brings to the forefront *why* we have insisted on these different levels of abstraction. A Service Provider will happily sell a VPN service. Such a VPN service does *not* require the customer to be aware of lower-level technical details, such as which interior gateway protocol (IGP) the Service Provider is using. Therefore, there is no requirement to even mention the type of IGP that is being used at the business level. However, there almost certainly is a need to define the type of IGP (and other more advanced details) at lower levels of the Policy Continuum, because this affects how the Service is implemented.

This leads to another important idea, called "policy coherency". Since different people have different ideas of what a policy is and what it is telling them, we need a means to be able to translate between different levels of abstraction. In effect, we need a set of model mappings that tie the different abstractions together. Referring to the above VPN example, we need to be able to tie the high-level specification of the VPN to an approach (i.e., which type of VPN are we going to use) that has a particular implementation (e.g., the particular CLI commands necessary for a particular router to support this type of VPN). This means that the shared data must be of a form that facilitates syntactic adaptation and/or semantic mediation between the different levels. Put another way, the formalism that is the Policy Continuum enables policies at one level of abstraction to be transformed to policies at another level of abstraction.

Policies are therefore *not* restricted to just being used at a single Reference Point associated with a single Service Provider. In fact, policies can provide consistent and repeatable behavior across multiple related Reference Points (e.g., the Sonata and Interlude Reference Points) of multiple operators in a supply chain. For example:

- Customer: Always connect to an Access Point that has the least cost

- Business application: Maximize revenue
- SOF: Coordinate provisioning of VPN in specified Partner Domains
- ICM: Instantiate Gold Service Function Chaining for Gold Users
- ECM: Increase drop probability by 10% if traffic exceeds limit

In addition, the same user may need to use different types of policies. For example, consider the transformation of the high-level policy "John gets Gold Service". Assume that this policy takes context into account. This means that:

- On some days, all applications for John get Gold Service
- On some days, multimedia and voice applications for John get Gold Service, but policies are used to determine which lower class of service specific applications get if there is a lack of resources (assuming that Gold Service is not the highest policy level)
- On some days, the system correlates current environmental conditions to past recorded conditions and takes the action decided from the past (meaning that some applications get Gold Service, some get Silver Service, and some may even get Bronze Service)
- The execution strategy for the decomposition of this policy is (likely) first imperative, which is followed by a set of declarative and/or imperative policies (e.g., in the first example, imperative policies can easily be used to set each application John is using to Gold Service; however, in the second example, a declarative optimization function may be used to map the applications that John wants to run onto the system's available resources).

8 MEF Policy Model (MPM)

The MPM is a UML object-oriented information model, derived from the MCM, that contains key model elements for representing policies of different types. This document focuses specifically on representing imperative and declarative policies.

8.1 The Purpose of a Policy Model

One of the current problems with network management is that it is not linked to the business processes that run the network. For example, only authorized users should be allowed to change the configuration of a network device. Otherwise, this violates fundamental business processes, and makes it very difficult for the overall state of the network to be tracked and updated. For every configuration, no matter how large or how small, there are defined processes that govern how a configuration file is built, who must approve it, when it can be scheduled for installation, and what to do if something goes wrong. ***Ultimately, the business and operational policies that govern the construction and deployment of configuration changes are more important than the configuration changes themselves!***

Process is everything. The network is not a "fat dumb pipe" that is made up of individual interfaces; businesses don't operate or sell interfaces! Businesses operate and manage services according to the priority and contractual obligations that the business enters into. This mandates intelligent processes that can manage the rich functionality of a network, and ensure that changes to network devices follow approved processes.

8.2 How Policy is Modeled

Policy-Based Management (PBM) is defined as the usage of policy rules to manage the configuration and behavior of one or more entities. More formally, PBM is a methodology that describes one or more applications that manage one or more systems according to a set of rules. These rules take the form of policies that are applied to components of the system in order to better and more efficiently manage those components. One mechanism to do this is to use finite state machines; in this approach, policy rules are used to control the transition from the current state to a new state. Note that in this way, we can achieve true end-to-end control, as opposed to having "just" device- or element-level control without PBM, since the behavior of each component is captured by the states defined in the finite state machine.

What makes PBM different from other approaches is its use of policies to control the behavior of managed entities. As stated previously, implicit in the definition of a PBM system is the use of a management methodology – in the above example, a finite state machine (though clearly other methods are also possible) – to manage the life-cycle aspects of entities.

Why does PBM use policies? The reason is to be able to control the behavior of a managed system in a predictable and consistent fashion. In order to do this, the characteristics of the system that is being managed must be able to be represented in as much detail as required. Then, policies can be defined that govern each state of the managed object – from creation to deployment to destruction. Without policies, there is no way to coordinate the behavior (e.g., the state and state transitions) of the objects being managed. Furthermore, there is no way to guarantee consistent behavior and reaction to events.

How PBM uses policies is critical to the implementation of a PBM system. Many current PBM systems are focused on a particular component in a system, or a set of features, that must be controlled. For example, many Quality of Service (QoS) PBM systems are designed to control a small subset of the features of a device, such as a router. The worry, of course, is the interaction between the QoS features and other features of the router – what if the QoS PBM system makes an adjustment that adversely affects the delivery of some other service or feature that the router is supporting? The answer, of course, is for a PBM system to holistically manage the different components in a system, and the different services that each device supports.

8.3 Naming Rules

The MPM uses the same naming rules as those used in the MCM. The MCM uses the following rules to define the names of its model elements:

- MCM Naming rules are as follows:

  [R1] Class names **MUST** be in UpperCamelCase (i.e., the first letter is capitalized). Class names **MUST NOT** begin with any non-alphabetic character, and no spaces are allowed.

  [R2] Attribute names **MUST** be in lowerCamelCase (i.e., the first letter is lower case); attribute names **MUST NOT** begin with any non-alphabetic character except for the underscore, and no spaces are allowed. Note that attribute names that begin with an underscore are private attributes that reference an end of an association.

  [R3] Relationship names **MUST** be in UpperCamelCase (i.e., the first letter is capitalized). Relationship names **MUST NOT** begin with any non-alphabetic character, and no spaces are allowed.

- MPM naming rules are as follows:

  [R4] Each MPM class **MUST** be prefixed with “MPM”. The only exception is MCMPolicyObject, which is the top of the MPM model and is a part of MCM). This serves two purposes. First, it helps provide context to textual descriptions of these model elements. Second, it enables MPM model elements, patterns, and approaches to be compared to those of other SDOs and consortia unambiguously.

  [R5] Each attribute **MUST** be prefixed with “mpm”. For example, the attribute “continuumLevel” is named “mpmContinuumLevel”. If an attribute starts with an underscore, then “mpm” immediately follows the underscore (e.g., mpmARef).

  [R6] Each relationship **MUST** be prefixed with “MPM”. For example, the aggregation “HasPolicyTarget” is named “MPMHasPolicyTarget”.

[R7] All association classes **MUST** be suffixed with the word "Detail". For example, the association class for the above example is named "MPMHasPolicyTargetDetail". This makes it obvious that a class is an association class.

- Regarding interoperability with concepts from other SDOs:

[R8] All MCM classes that model a concept from another SDO and *change* the model of that SDO (e.g., to be able to be used in the MCM) **MUST** be prefixed with "MCMMEF". For example, the concept of a Descriptor from ETSI NFV is named "MCMMEFDescriptor".

[R9] All MPM classes that model a concept from another SDO and *change* the model of that SDO (e.g., to be able to be used in the MCM) **MUST** be prefixed with "MPMMEF".

[R10] All MCM classes that model a concept from another SDO *exactly as it is defined* in that SDO **MUST** be prefixed with "MCM", followed by the name of the SDO, followed by the class name. For example, if an SDO named Foo defined a class named Bar, and MCM imported this concept with no changes, it would be named MCMFooBar.

[R11] All MPM classes that model a concept from another SDO and *change* the model of that SDO (e.g., to be able to be used in the MCM) **MUST** be prefixed with "MPM", followed by the name of the SDO, followed by the class name. For example, if an SDO named Foo defined a class named Bar, and MPM imported this concept with no changes, it would be named MPMFooBar.

A note about associations, aggregations, compositions, and their multiplicity. The UML guidelines do not specify in detail what valid multiplicities are. In the MCM, multiplicities are important, in order to provide a robust foundation for code generation, as well as to accommodate the future incorporation of ontologies Therefore:

[O1] Association relationships **MAY** have a 0..* - 0..* multiplicity. This is because they represent a generic dependency, and one end of the association may not be instantiated yet.

[D1] Aggregation and composition relationships **SHOULD NOT** have a 0..* - 0…* multiplicity. This is because both aggregations and compositions are a type of whole-part relationship. Ontologically, it is impossible to talk about a "whole" when no "parts" exist (or vice-versa).

[D2] If there is the possibility of not instantiating an aggregation or a composition, then the cardinality of the aggregate (or composite) part **SHOULD** be 0..1, where the 0 signifies that the relationship has not yet been instantiated.

[D3] Relationships whose owner (i.e., the source of the relationship) is a value greater than 0 (e.g., 1 or 1..* or 3..7) **SHOULD** have a part multiplicity of at least 1. This is because one side of the relationship must exist, and it makes no sense to have one side of a relationship exist while the other side doesn't.

MPM defines a number of enumerations and datatypes for flexibility and extensibility. These are located in the MPM project of the MEF_Types project.

[R12] All enumerations and datatypes of MPM **MUST** be located in the following MEF GIT Repository:
https://github.com/MEF-GIT/MEF-General-Common/MPM

8.4 Overview of the MCM

This section briefly describes the top-level hierarchies of the MCM. The interested reader is referred to [2] for more details.

8.4.1 The Top Portion of the MCM

Figure 4 shows the top of the MCM class hierarchy (MCMRootEntity), the first level of inheritance (consisting of three subclasses), and relationships with their association classes.

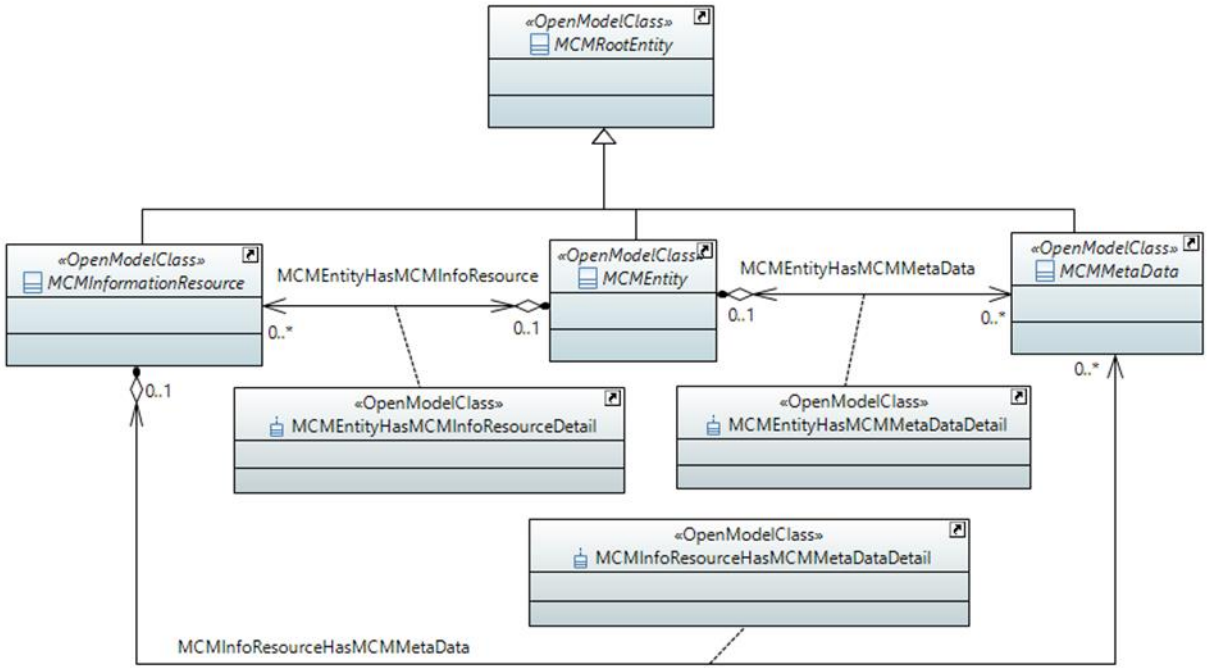


Figure 4. The Top Portion of the MCM Class Hierarchy

MCMRootEntity defines the top of the MCM class hierarchy. Its characteristics and behavior are thus inherited by all MCM classes. MCMRootEntity defines a set of attributes that enable all objects to be unambiguously named, described, and identified in a managed environment. Note that multiple inheritance is *disallowed* in MEF models.

Figure 4 shows the three subclasses of MCMRootEntity: MCMEntity, MCMInformationResource, and MCMMetaData. 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).

The three subclasses create three parallel class hierarchies that can interact with each other using the three aggregations shown in Figure 4. For example, object instances from the MCMMetaData class hierarchy are designed to be attached to object instances from the other two class hierarchies. In addition, classes from the MCMInfoResource class hierarchy are inherently related to classes from the MCMEntity class hierarchy.

The three class hierarchies are described as follows:

1) **MCMEntity**, which is the superclass for objects of interest that are important to the managed environment, and which have a separate and distinct existence. These objects can play one or more business functions, and can be managed or unmanaged (using digital mechanisms). Examples include Location (unmanaged) and Product, Service, and Resource (all three are managed).

2) **MCMInformationResource**, 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. For example, an IPAddress is an important piece of data, but it does not control its own lifecycle; rather, its lifecycle is controlled by another Resource (e.g., a DHCPServer). The use of MCMInformationResource enables the IPAddress (in this example) to be represented and associated with the correct Resource responsible for its lifecycle.

3) **MCMMetaData**, which is an object that defines descriptive and/or prescriptive information about the MCMEntity or MCMInformationResource objects that it is attached to. Examples include versioning information of an object, as well as best common practice information and context-specific usage guidelines.

Figure 4 also shows three aggregations, called MCMEntityHasMCMInfoResource, MCMEntityHasMCMMetaData, and MCMInfoResourceHasMCMMetaData.

The first aggregation defines the set of MCMInformationResource objects that are associated with a given set of MCMEntities. The second and third aggregations define the set of MCMMetaData objects that can be attached to a particular MCMEntity and a given MCMInformationResource, respectively. All three of these aggregations are implemented as association classes; this enables the Policy Pattern (see Figure 5) to be used to define policy rules that constrain which part objects (i.e., MCMInformationResource for the first aggregation, and MCMMetaData for the second and third) are attached to which MCMEntity (first or second aggregation) or MCMInformationResource (third aggregation). Note that MPMPolicyStructure is an abstract class that is the superclass of imperative, declarative, and intent policy rules.

All MCM association classes are rooted from a single superclass, called MCMRelationshipParent (which in turn is subclasses from MCMEntity); this simplifies both the design of the association classes and their implementation. The MCMPolicyStructure, which is a subclass of MCMPolicyObject, is the superclass of all policies defined in the MEF Policy Driven Orchestration project (i.e., imperative, declarative, and intent policies). The diagram below shows that an object instance of the appropriate concrete subclass of MCMPolicyStructure is related to class-level attributes and operations of an object instance of the MCMEntityHasMCMMetaDataDetail association class.

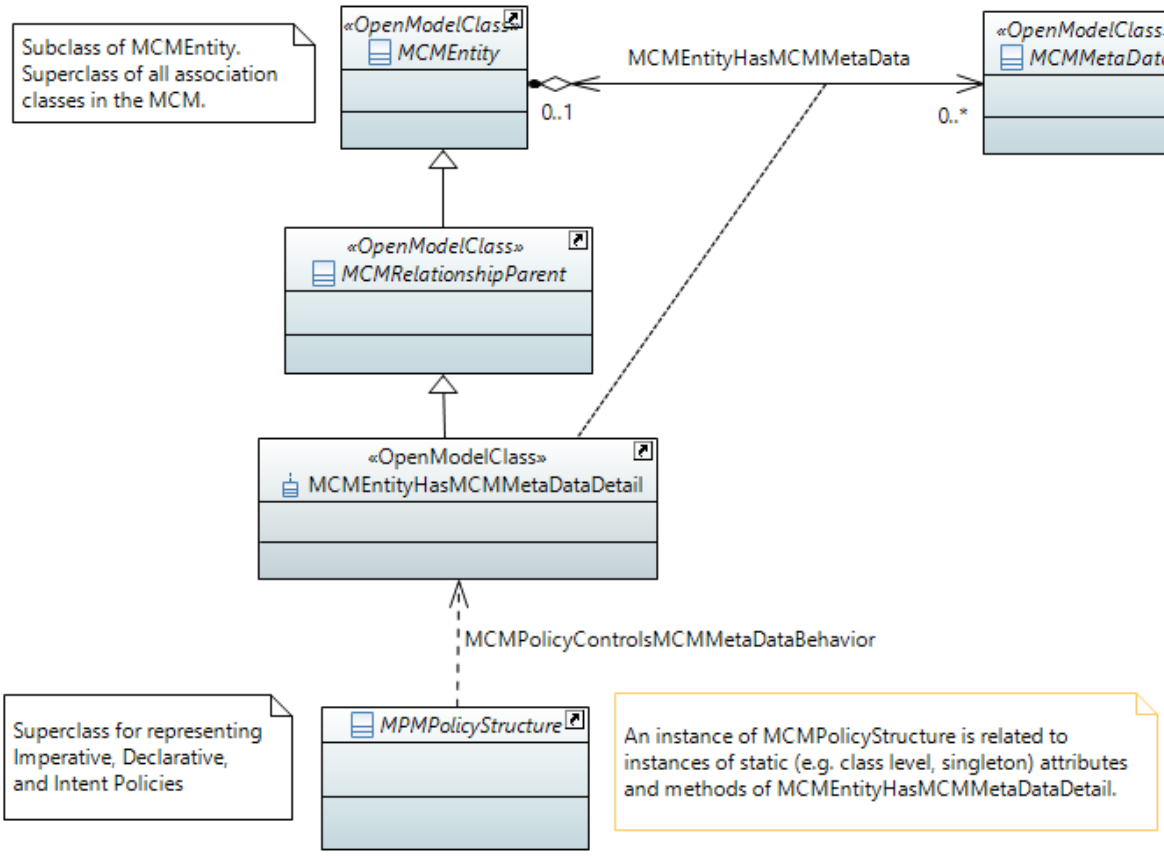


Figure 5. The Policy Pattern Applied to MCMEntityHasMCMMetaDataDetail

8.4.2 The Use of Metadata

The purpose of MCMMetaData is to describe and/or prescribe information about MCMEntity and MCMInformationResource objects. Examples include describing best current practices of using an object, instructing which version(s) of an object to use for a given situation, and to define how to manage the behavior of the system and its constituent components. This makes MCMMetaData objects different than both MCMEntities (whose purpose is to describe the constituent components of a managed system) as well as MCMInformationResource (whose purpose is to describe information that is not an inherent part of a managed entity, but which nevertheless is important information for the system being managed and is governed by an MCMEntity).

More formally, in the MCM, metadata may describe and/or prescribe information about the object(s) to which it is attached. This is done by "attaching" the metadata object to another object using a relationship, which is typically an aggregation (i.e., a type of "whole-part" relationship). This can be thought of as augmenting the description of that object, and/or attaching management and control information, to that object. Multiple metadata objects may be attached to any single object.

There is often debate as to whether something is metadata or not. In the MCM, a very simple rule is used to make this decision:

[D4] *Metadata **SHOULD** be used to describe a concept that is not part of the inherent characteristics or behavior of an object.*

For example, suppose we were designing a class to represent a Person. An attribute called birthdate would be reasonable, since it is a characteristic of all People. In contrast, an attribute called hairColor is not, since a Person may not have any hair; this could instead be conveyed using metadata. Finally, an attribute called socialSecurityNumber is a poor design for a number of reasons, including (1) social security numbers are typically used only in the US, and (2) there are a number of complex geo-political reasons involving whether a person living in the US even has a social security number.

A much better design is to realize that a social security number is one way to identify a person in a given context. Hence, a more scalable approach would be to define an association between Person and another class, called (for example) PersonalIdentifier. Note that this enables different types of identifiers (e.g., driverLicense, nameAndPassword, biometricData) to be defined a subclasses of PersonalIdentifier. Since each of these have different metadata (e.g., when they should be used), metadata could be attached to each type of identifier.

Metadata is crucial to designing and implementing model-driven software. Most information models either do not specify a metadata hierarchy, or define metadata as embedded within a class. The MCM has chosen to define a separate metadata hierarchy, because:

1) Metadata that is defined within a class makes that metadata available *only* to that class; hence, if the same concept (e.g., versioning, or periods of time within which something is applicable) pertains to other classes, the metadata *is captured as duplicate model elements* (e.g., classes, attributes, operations, constraints, and/or relationships). This creates maintenance issues, as each metadata model object needs to be separately managed.

2) Creating a metadata hierarchy enables a family of objects to be reused to represent common information and behavior that apply to other objects. For example, if the concept of a software version is needed, then defining version as metadata enables any object in the entire model to use a consistent definition of software version.

[D5] Metadata **SHOULD** be optional, since it is used to describe or prescribe the behavior and semantics of another object.

In the MCM, a separate class hierarchy supports attaching a set of metadata objects that can be optionally attached to other objects as needed (e.g., depending on context).

8.4.3 MCM Compliance

The MCM defines all common concepts that other models can use.

[D6] In principle, users of a model **SHOULD** be able to find the basic definitions of all concepts that their project needs defined in the MCM.

[D7] If a required concept is not defined in the MCM, then that concept **SHOULD** be added to either the MCM (if it is generally applicable to other models), or to a model derived from the MCM; this enables the MCM, and its derived models, to continually grow and serve the common needs of the MEF modeling community.

[D8] New concepts that are added to the MCM **SHOULD** be in the form of a small number of key model elements. Entire models **SHOULD NOT** be imported into the MCM, as they will likely not be generally applicable to other projects.

For example, if Policy was *not* defined in the MCM, and a project needed to use Policy, then that project should request that Policy be added to the MCM. This does *not* mean that the entire Policy model is added to the MCM; rather, a small set of model elements are added to the MCM hierarchy so that a common Policy model can be built. This is how Policy is currently defined in the MCM.

Note that most projects will need to reference multiple model elements. For example, the Sonata Ordering project will need to use classes, attributes, and relationships from at least the MCMUnManagedEntity hierarchy (e.g., locations and physical entities), MCMManagedEntity hierarchy (e.g., Product, and possible Service, as well as their associated Definitions), MCMParty hierarchy (e.g., people and organizations), MCMBusinessObject hierarchy (e.g., orders and order items), and MCMMetaData hierarchy.

[D9] If a project needs to add model elements (e.g., classes, attributes, relationships, operations, constraints) to the MCM, it **SHOULD** conform to the principles in this section.

The following sections define MCM model elements. Classes are not individually designated as mandatory or optional, because the set of classes that are implemented depends on the application being realized.

[R13] If a class is implemented, then any mandatory model elements defined by that class **MUST** also be implemented.

[R14] Requirement [R13] means that any inherited model elements defined by a class **MUST** also be implemented. In particular, overriding attributes or operations **MUST NOT** be done.

Care should be taken in defining relationships. Relationships are inherited by the classes participating in a relationship.

[D10] Subclasses that inherit relationships from their parent classes **SHOULD NOT** define a relationship that has the same behavior as inherited relationships. While this also applies to attributes and operations, it is much more common in practice to see this requirement not followed.

8.5 Design Approach of the MPM

The MPM contains model elements that treat a policy as an intelligent container that aggregates one or more components. This is shown in Figure 6 below.

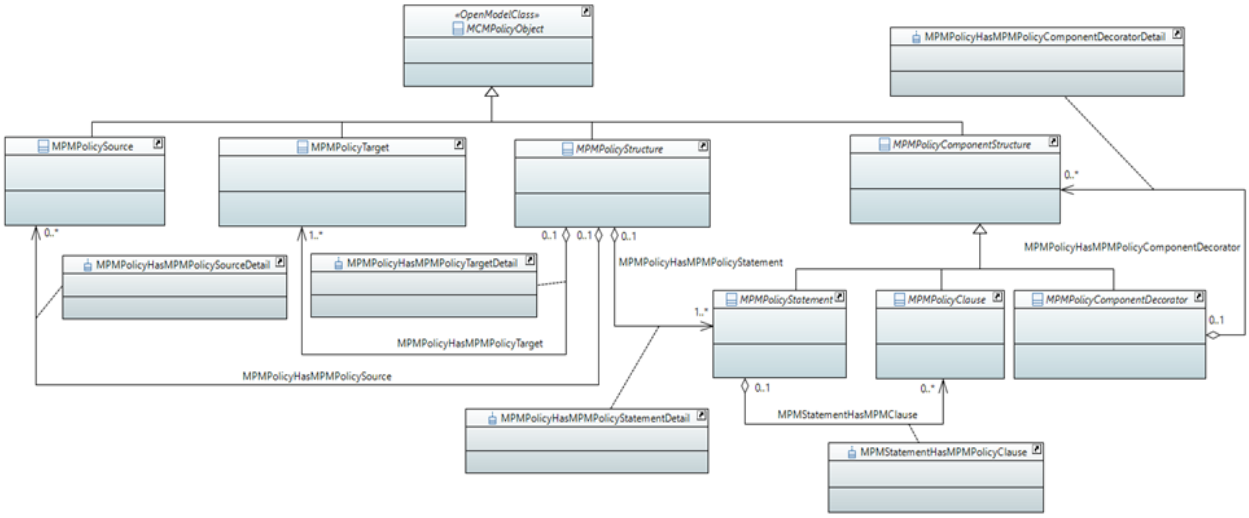


Figure 6. MPM Abstractions

MCMPolicyObject, defined in the MCM, is the root of the MPM information model. Since MCMPolicyObject is a type of MCMManagedEntity, this means that:

- All MPM classes are also managed entities
- All MPM classes can potentially be related to metadata and information resources

The MPM is made up of four types of objects. Two of them, MPMPolicyStructure and MPMPolicyComponentStructure, define hierarchies for representing policies and components of a policy, respectively. MPMPolicySource represents a set of objects that authored the policy, and MPMPolicyTarget represents a set of objects that may be affected by a policy.

8.5.1 PolicyContainer

A PolicyContainer is a collection of statements, policy components, and metadata that define the overall *structure* of the policy. A PolicyContainer defines whether the Policy is an Imperative, Declarative, or Intent Policy. That in turn determines what types of PolicyComponents the PolicyContainer is made up of.

The PolicyContainer defines the type of policy rule; the contents of the policy rule are defined by one or more MPMPolicyStatements. This is reflected in the multiplicity of the MPMPolicyHasMPMPolicyStatement aggregation.

[R15] All MPMPolicy objects **MUST** contain at least one MPMPolicyStatement.

8.5.2 Types of Policies

There are three main types of policy paradigms that are used in the PDO: imperative, declarative, and intent policies. While other types of policies are certainly possible (e.g., utility functions), the use of these three policies paradigms provides sufficient flexibility to address the currently identified needs of the PDO project.

8.5.2.1 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 7 shows the behavior of an imperative policy.

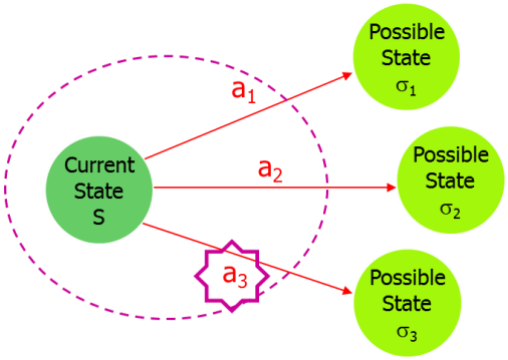


Figure 7. The Imperative Policy Paradigm

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 behavior 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).

8.5.2.2 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 *not defined*. In so doing, side effects are reduced.

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 Section 8.5.2.3), which are written in a (controlled) natural language and then *translated* to a different form. Figure 8 shows the behavior of a declarative policy.

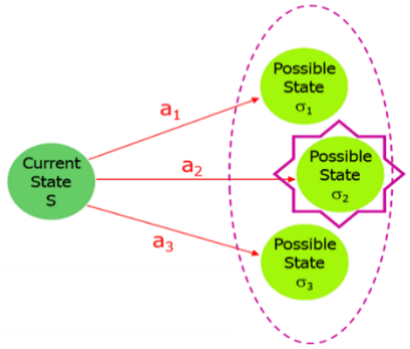


Figure 8. The Declarative Policy Paradigm

In declarative policies, there isn't really the notion of an *action;* Figure 8 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 following is an example of a declarative policy from OpenStack Congress. Note that declarative policies are expressed as *logical predicates*.

Define the following policy:

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

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 users 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 is used to manage group-membership.

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

8.5.2.3 Intent Policies

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 this 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 9.

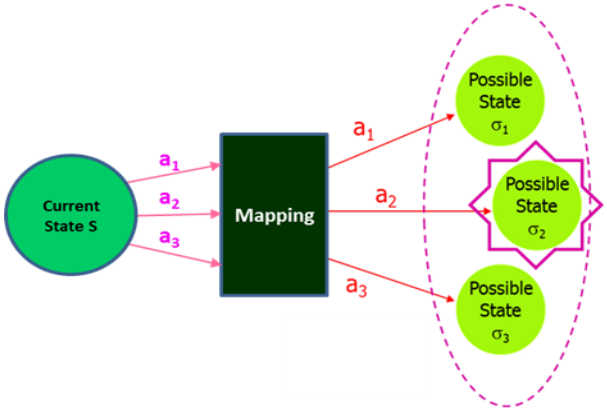


Figure 9. The Intent Policy Paradigm

The advantage of Intent is its ability to express policies using concepts and terminology that are familiar to the Consumer (e.g., Buyer or requestor of a service). This is, of course, its disadvantage, since natural languages are typically ambiguous. In the PDO project, we use the information model as a data dictionary (e.g., a central source of truth), so that the model can be used to help satisfy the needs of the translation in a common way.

An example of an intent policy that uses mapping in several different ways is the following:

Provide John Gold Service

The mapping component needs to translate the above statement to a form that other components can understand. In this example:

- Provide is mapped to an assignment
- John is recognized as a Customer
- GoldService is recognized as a type of SLA

Hence, a mapping of the above intent policy may look like:

John is assigned GoldService

where:

- The Customer (John) is an instance of the Person class (a subclass of Party) and is then assigned the Customer PartyRole (since a Party aggregates 0..n PartyRoles)
  - John is mapped to a concrete representation (e.g., an IP address range)
- The GoldService SLA is used to identify the types of applications that John may use
  - Each application runs under GoldService
  - The combination of the SLA and application can be used to determine the requirements of each application on the infrastructure
  - The SLA may itself be mapped to related concepts, such as incentives and violations

The above mappings utilize concepts in the information model, along with information from other management entities (e.g., an LDAP directory that provides an authorized Customer list).

Note that in intent policies, technical terms are avoided. The above example only uses concepts that are visible to the business user, such as the name of a Customer and the desired class of service (as advertised by the Service Provider).

In particular, the example of the declarative OpenStack Congress policy does *not* fit our definition of intent (even though that policy is, in fact, declarative). This is because it uses a number of technical terms, such as "network" and "VM". To be an intent policy, it would have to be reworded, such as:

Only allow public communication using either the Internet or a known Service Provider

The mapping logic would then translate "public communication" to a public network, and "known Service Provider" to a private network known to the owner of this VM.

Note that the OpenStack Congress could be a translation from an intent policy to a policy at a lower level of abstraction.

8.6 MCMPolicyObject

This is an abstract class, and specializes MCMManagedEntity. It is the root of the MEF Policy Model (MPM). In other words, all other classes of the MPM are subclasses of this class. This simplifies code generation and reusability. It also enables different types of MCMMetadata objects to be attached to any appropriate subclass of MCMPolicyObject.

No attributes or relationships are currently defined for this class.

8.7 The MPMPolicyStructure Hierarchy

The structure of this class hierarchy is shown in Figure 10. This class hierarchy is defined to facilitate adding new types of policies later.

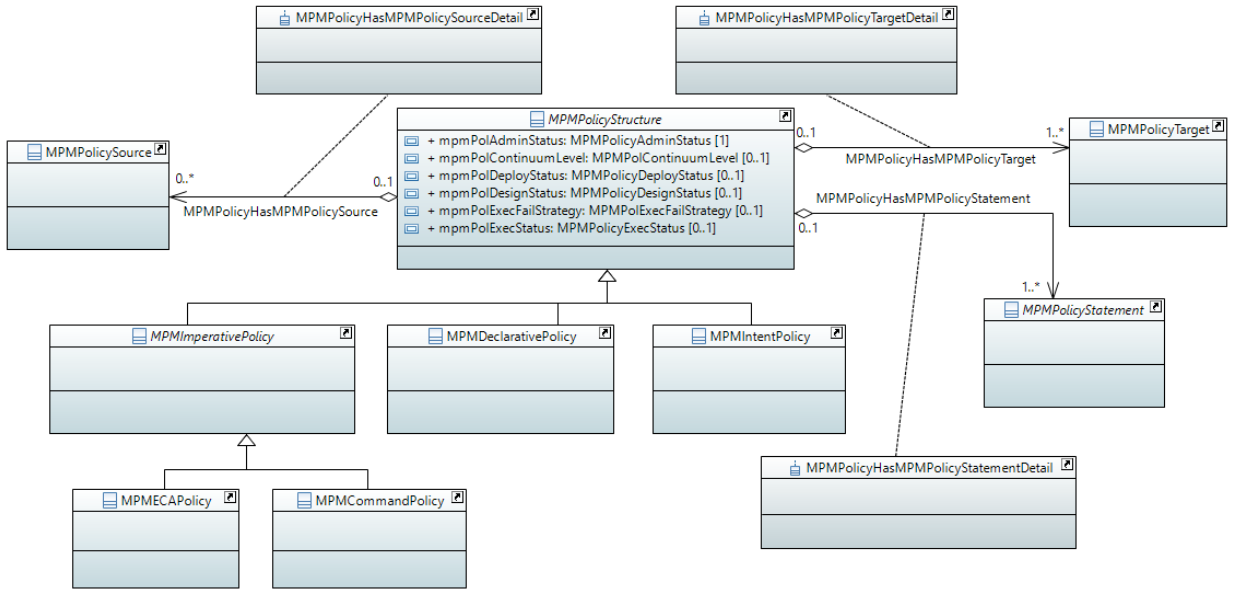


Figure 10. The MPMPolicyStructure Class Hiearchy

An MPMPolicy may take the form of an individual policy or a set of compound (i.e. embedded) policies. However, some types of MPMPolicies only make sense as individual policies. This requirement is supported by applying the composite pattern to subclasses of the MPMPolicyStructure class that can support compound policies. For example, imperative policies can support embedded policies (e.g., as nested if-then statements), while declarative and intent policies cannot.

8.7.1 MPMPolicyStructure Class Definition

This is a mandatory abstract class. It defines the structure of an MPMPolicy. In this model, the type of Policy (e.g., imperative, declarative, intent) is represented by a subclass of the MPMPolicyStructure class, which is a type of PolicyContainer. The type of PolicyContainer then defines the set of MPMPolicyComponentStructure objects that it may contain.

This release will define imperative, declarative, and intent policies.

Table 4 defines the attributes of the MPMPolicyStructure class.

Attribute Name	Description
mpmPolAdminStatus : MPMPolicyAdminStatus[1..1]	This is a mandatory enumerated non-negative integer attribute that defines the current administrative status of this particular MPMPolicy object. The allowable values of this enumeration are defined by the MPMPolicyAdminStatus enumeration.
mpmPolContinuumLevel: MPMPolContinuumLevel[0..1]	This is an optional enumerated non-negative integer attribute. It defines the level of abstraction, as represented by the Policy Continuum Level, of this particular MPMPolicy. The allowable values of this enumeration are defined by the MPMPolContinuumLevel enumeration.
mpmPolDeployStatus : MPMPolicy-DeployStatus[0..1]	This is an optional enumerated, non-negative integer attribute. It is used to indicate whether this MPMPolicy can or cannot be deployed by the policy management system. This attribute enables the policy manager to know which MPMPolicies are currently deployed, and may be useful for the policy execution system for planning the staging of MPMPolicies. The allowable values of this enumeration are defined by the MPMPolicyDeployStatus enumeration.
mpmPolDesignStatus : MPMPolicy-DesignStatus[0..1]	This is an optional enumerated, non-negative integer whose value defines the current design status of this MPMPolicy object. The allowed set of values are defined in the MPMPolicyDesignStatus enumeration.
mpmPolExecFailStrategy: MPM-PolExecFailStra-tegy[0..1]	This is an optional enumerated, non-negative integer attribute. It is used to define what actions, if any, should be taken by this particular MPMPolicy if it fails to execute correctly. Note that some systems may not be able to support all options specified in this enumeration. For example, if rollback is NOT supported by the system, then options 2 and 3 may be skipped, and options 4 and 5 be used in their place. The allowable values of this enumeration are defined by the MPMPolExecFailStrategy enumeration.
mpmPolExecStatus: MPMPolicyExecStatus[0..1]	This is an optional enumerated non-negative enumerated integer whose value defines the current execution status of

	this MPMPolicy object. The allowed set of values are defined in the MPMPolicyExecStatus enumeration.

Table 4. Attributes of the MPMPolicyStructure Class

Figure 11 shows the operations for this class, and Table 5 defines the operations for this class.

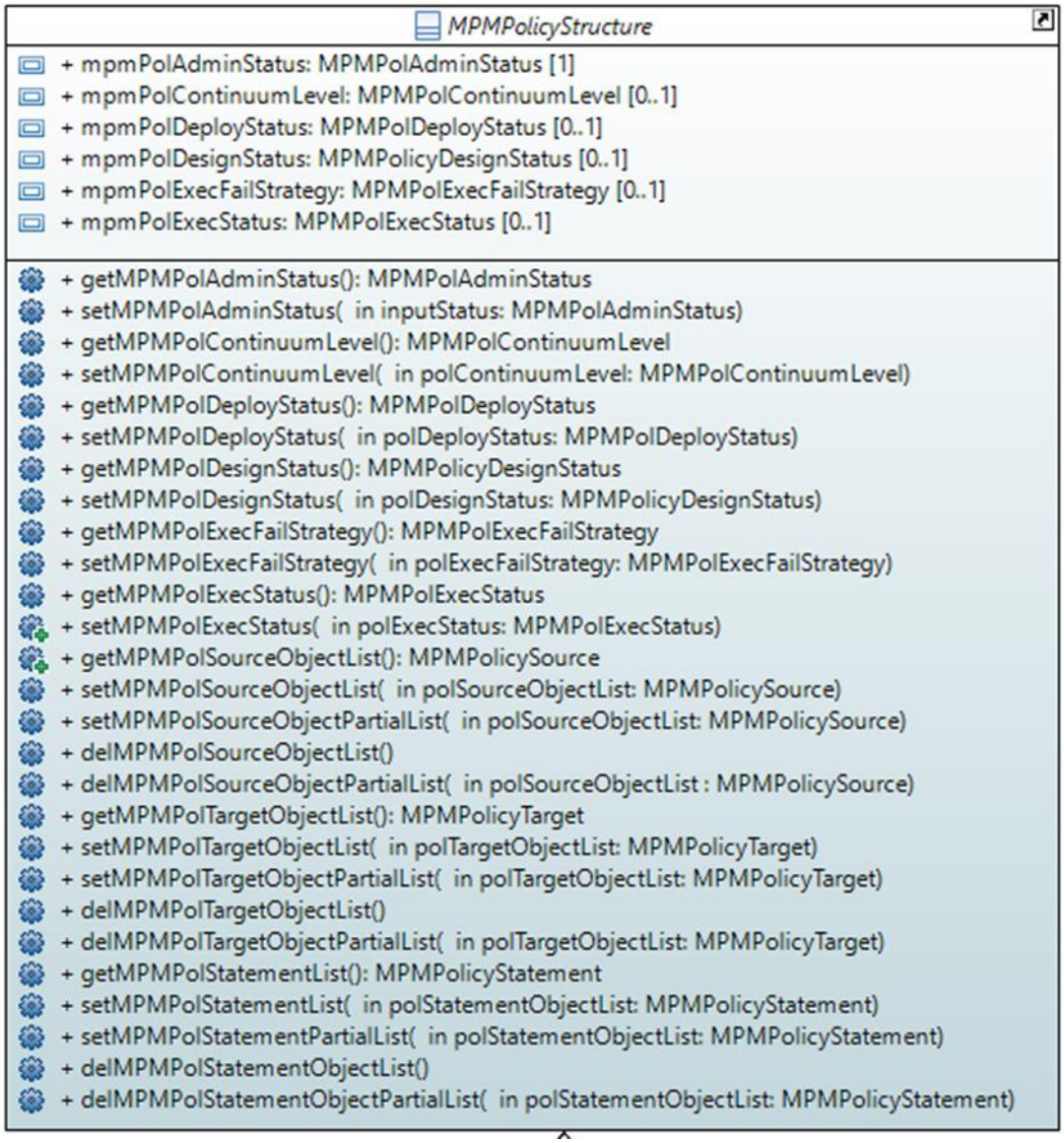


Figure 11. Operations of the MPMPolicyStructure Class

Operation Name	Description
getMPMPolAdminStatus() : MPMPolicyAdminStatus[1..1]	This operation returns the current administrative status of this particular MPMPolicy object, which is defined by the MPMPolicyAdminStatus enumeration. This operation takes no input parameters. **[R16]** If the mpmPolAdminStatus attribute does not have a value, then this operation **MUST** return an error.
setMPMPolAdminStatus(in inputStatus : MPMPolicyAdminStatus[1..1])	This operation sets the value of the current administrative status of this particular MPMPolicy object. This operation takes a single input parameter, called inputStatus, which defines the new value for the mpmPolAdminStatus attribute. The allowable values of this input parameter are defined by the MPMPolicyAdminStatus enumeration.
getMPMPolContinuumLevel() : MPMPolContinuumLevel[1..1]	This operation returns the level of abstraction, as represented by the Policy Continuum Level, of this particular MPMPolicy. The return value of this operation is defined by the MPMPolContinuumLevel enumeration. This operation takes no input parameters. **[D11]** If the mpmPolContinuumLevel attribute does not have a value, then this operation **SHOULD** return a NULL string.
setMPMPolContinuumLevel(in polContinuumLevel : MPMPolContinuumLevel[1..1])	This operation sets the level of abstraction, as represented by the Policy Continuum Level, of this particular MPMPolicy. This operation takes a single input parameter, called polContinuumLevel, which defines the new value for the mpmPolContinuumLevel attribute. The allowable values of this input parameter are defined by the MPMPolContinuumLevel enumeration.
getMPMPolDeployStatus() : MPMPolicyDeployStatus[1..1]	This operation returns the current deployment status of this particular MPMPolicy, which is defined by the MPMPolicyDeployStatus enumeration. This operation takes no input parameters. **[D12]** If the mpmPolDeployStatus attribute does not have a value, then this

	operation **SHOULD** return a NULL string.
setMPMPolDeployStatus(in polDeployStatus : MPMPolicyDeployStatus[1..1])	This operation sets the current deployment status of this particular MPMPolicy. This operation takes a single input parameter, called polDeployStatus, which defines the new value for mpmPolDeployStatus attribute. The allowable values of this input parameter are defined by the MPMPolicyDeployStatus enumeration.
getMPMPolDesignStatus() : MPMPolicyDesignStatus[1..1]	This operation returns the current design status of this particular MPMPolicy object, which is defined by the MPMPolicyDesignStatus enumeration. This operation takes no input parameters. **[R17]** If the mpmPolDesignStatus attribute does not have a value, then this operation **MUST** return an error.
setMPMPolDesignStatus(in polDesignStatus : MPMPolicyDesignStatus[1..1])	This operation sets the value of the current design status of this particular MPMPolicy object. This operation takes a single input parameter, called polDesignStatus, which defines the new value for the mpmPolDesignStatus attribute. The allowable values of this input parameter are defined by the MPMPolicyDesignStatus enumeration.
getMPMPolExecFailStrategy() : MPMPolExecFailStrategy[1..1]	This operation returns the current strategy for dealing with execution failures. This defines what actions, if any, should be taken by this particular MPMPolicy if it fails to execute correctly. The return value of this operation is defined by the MPMPolExecFailStrategy enumeration. This operation takes no input parameters. **[D13]** If the mpmPolExecFailStrategy attribute does not have a value, then this operation **SHOULD** return a NULL string.
setMPMPolExecFailStrategy(in polExecFailStrategy : MPMPolExecFailStrategy[1..1])	This operation sets the current strategy for dealing with execution failures. This defines what actions, if any, should be taken by this particular MPMPolicy if it fails to execute correctly. The allowable values of this enumeration are defined by the MPMPolExecFailStrategy enumeration.

getMPMPolExecStatus() : MPMPolicyExecStatus[1..1]	This operation returns the current execution status of this MPMPolicy object. The return value of this operation is defined by the MPMPolicyExecStatus enumeration. This operation takes no input parameters. **[D14]** If the mpmPolExecStatus attribute does not have a value, then this operation **SHOULD** return a NULL string.
setMPMPolExecStatus(in polExecStatus : MPMPolicyExecStatus[1..1])	This operation sets the current execution status of this MPMPolicy object. The allowed set of values are defined in the MPMPolicyExecStatus enumeration.
getMPMPolSourceObjectList() : MPMPolicySource[1..*]	This operation retrieves the set of MPMPolicySource objects that are contained in this particular MPMPolicyStructure object. This is obtained by following the MPMPolicyHasMPMPolicySource aggregation. Each instance of this aggregation defines an MPMPolicySource object, which is then added to the return value of this operation. The return value of this operation is an array of one or more MPMPolicySource objects. This operation takes no input parameters. **[D15]** If this MPMPolicyStructure object does not instantiate this aggregation, then this operation **SHOULD** return a NULL MPMPolicySource object.
setMPMPolSourceObjectList(in polSourceObjectList : MPMPolicySource[1..*])	This operation defines a new set of MPMPolicySource objects that will be contained in this particular MPMPolicyStructure object. This operation takes a single input parameter, called polSourceObjectList, which defines a set of one or more MPMPolicySource objects. If this MPMPolicyStructure object already has a set of one or more MPMPolicySource objects that it contains, then those MPMPolicySource objects will be deleted by first, deleting the accompanying association class, and second, deleting the corresponding association. Then, a new association (an instance of MPMPolicyHaMPMPolicySource) is created for each MPMPolicySource object in the polSourceObjectList parameter.

	[D16] Every association created **SHOULD** have a new association class created to realize the semantics of that association.
setMPMPolSourceObjectPartialList (in polSourceObjectList : MPMPolicySource[1..*])	This operation defines a new set of MPMPolicySource objects that will be contained in this particular MPMPolicyStructure object. This operation takes a single input parameter, called polSourceObjectList, which defines a set of one or more MPMPolicySource objects. If this MPMPolicyStructure object already has a set of one or more MPMPolicySource objects that it contains, then those MPMPolicySource objects are ignored. Then, a new association (an instance of MPMPolicyHaMPMPolicySource) is created for each MPMPolicySource object in the polSourceObjectList. **[D17]** Every association created **SHOULD** have a new association class created to realize the semantics of that association. **[R18]** Any association between this MPMPolicyStructure object and other MPMPolicySource objects that is not specified in the polSourceObjectList **MUST NOT** be affected.
delMPMPolSourceObjectList()	This operation removes all instances of the MPMPolicyHasMPMPolicySource aggregation, and its association classes, that enables this particular MPMPolicyStructure object to contain any MPMPolicySource objects. This operation does NOT affect either the MPMPolicySource object or the MPMPolicyStructure object; it just deletes the association between this MPMPolicyStructure object and this MPMPolicySource object. This operation has no input parameters.

delMPMPolSourceObjectPartialList(in polSourceObjectList : MPMPolicySource[1..*])	This operation removes the association, and its association class, for each MPMPolicySource object in the polSourceObjectList that is contained by this particular MPMPolicyStructure object. This operation takes a single input parameter, called polSourceObjectList, that defines the set of MPMPolicySource objects that will be unlinked from this particular MPMPolicyStructure object. This operation does NOT affect either the MPMPolicyStructure object or the MPMPolicySource object; it just deletes the association between this MPMPolicyStructure object and this MPMPolicySource object. **[R19]** Any association between this MPMPolicyStructure object and other MPMPolicySource objects that is not specified in the polSourceObjectList **MUST NOT** be affected.
getMPMPolTargetObjectList() : MPMPolicyTarget[1..*]	This operation retrieves the set of MPMPolicyTarget objects that are contained in this particular MPMPolicyStructure object. This is obtained by following the MPMPolicyHasMPMPolicyTarget aggregation. Each instance of this aggregation defines an MPMPolicyTarget object, which is then added to the return value of this operation. The return value of this operation is an array of one or more MPMPolicyTarget objects. This operation takes no input parameters. **[D18]** If this MPMPolicyStructure object does not instantiate this aggregation, then this operation **SHOULD** return a NULL MPMPolicyTarget object.
setMPMPolTargetObjectList(in polTargetObjectList : MPMPolicyTarget[1..*])	This operation defines a new set of MPMPolicyTarget objects that will be contained by this particular MPMPolicyStructure object. This operation takes a single input parameter, called polTargetObjectList, which defines a set of one or more MPMPolicyTarget objects. If this MPMPolicyStructure object already has a set of one or more MPMPolicyTarget objects that

	it refers to, then those MPMPolicyTarget objects will be deleted by first, deleting the accompanying association class, and second, deleting the corresponding association. Then, a new association (an instance of MPMPolicyHasMPMPolicyTarget) is created for each MPMPolicyTarget object in the polTargetObjectList parameter. **[D19]** Every association created **SHOULD** have a new association class created to realize the semantics of that association.
setMPMPolTargetObjectPartialList (in polTargetObjectList : MPMPolicyTarget[1..*])	This operation defines a new set of MPMPolicyTarget objects that will be contained by this particular MPMPolicyStructure object. This operation takes a single input parameter, called polTargetObjectList, which defines a set of one or more MPMPolicyTarget objects. If this MPMPolicy-Structure object already has a set of one or more MPMPolicyTarget objects that it contains, then those MPMPolicyTarget objects are ignored. Then, a new association (an instance of MPMPolicyHaMPMPolicyTarget) is created for each MPMPolicyTarget object in the polTargetObjectList. **[D20]** Every association created **SHOULD** have a new association class created to realize the semantics of that association. **[R20]** Any association between this MPMPolicyStructure object and other MPMPolicyTarget objects that is not specified in the polTargetObjectList **MUST NOT** be affected.
delMPMPolTargetObjectList()	This operation removes all instances of the MPMPolicyHasMPMPolicyTarget aggregation, and its association classes, that enables this particular MPMPolicyStructure object to refer to any MPMPolicyTarget objects. This operation does NOT affect either the MPMPolicyTarget object or the MPMPolicyStructure object; it just deletes the association between this MPMPolicyStructure

	object and this MPMPolicyTarget object. This operation has no input parameters.
delMPMPolTargetObjectPartialList(in polTargetObjectList : MPMPolicyTarget[1..*])	This operation removes the association, and its association class, for each MPMPolicyTarget object in the polSourceObjectList that is associated with this particular MPMPolicyStructure object. This operation takes a single input parameter, called polTargetObjectList, that defines the set of MPMPolicyTarget objects that will be unlinked from this particular MPMPolicyStructure object. This operation does NOT affect either the MPMPolicyStructure object or the MPMPolicyTarget object; it just deletes the association between this MPMPolicyStructure object and this MPMPolicyTarget object. **[R21]** Any association between this MPMPolicyStructure object and other MPMPolicyTarget objects that is not specified in the polTargetObjectList **MUST NOT** be affected.
getMPMPolStatementList() : MPMPolicyStatement[1..*]	This operation retrieves the set of MPMPolicyStatement objects that are contained in this particular MPMPolicyStructure object. This is obtained by following the MPMPolicyHasMPMPolicyStatement aggregation. Each instance of this aggregation defines an MPMPolicyStatement object, which is then added to the return value of this operation. The return value of this operation is an array of one or more MPMPolicyStatement objects. This operation takes no input parameters. **[D21]** If this MPMPolicyStructure object does not instantiate this aggregation, then this operation **SHOULD** return a NULL MPMPolicyStatement object.
setMPMPolStatementList (in polStatementObjectList : MPMPolicyStatement[1..*])	This operation defines a new set of MPMPolicyStatement objects that are contained by this particular MPMPolicyStructure object. This operation takes a single input parameter, called polStatementObjectList, which defines a set of

	one or more MPMPolicyStatement objects. If this MPMPolicyStructure object already has a set of one or more MPMPolicyStatement objects that it refers to, then those MPMPolicyStatement objects will be deleted by first, deleting the accompanying association class, and second, deleting the corresponding association. Then, a new association (an instance of MPMPolicyHaMPMPolicyStatement) is created for each MPMPolicyStatement object in the polStatementObjectList parameter. **[D22]** Every association created **SHOULD** have a new association class created to realize the semantics of that association.
setMPMPolStatementPartialList(in polStatementObjectList: MPMPolicyStatement[1..*])	This operation defines a new set of MPMPolicyStatement objects that refer to this particular MPMPolicyStructure object. This operation takes a single input parameter, called polStatementObjectList, which defines a set of one or more MPMPolicyStatement objects. If this MPMPolicyStructure object already has a set of one or more MPMPolicyStatement objects that it refers to, then those MPMPolicyStatement objects are ignored. Then, a new association (an instance of MPMPolicyHaMPMPolicyStatement) is created for each MPMPolicyStatement object in the polStatementObjectList. **[D23]** Every association created **SHOULD** have a new association class created to realize the semantics of that association.
delMPMPolStatementObjectList()	This operation removes all instances of the MPMPolicyHaMPMPolicyStatement aggregation, and its association classes, that enables this particular MPMPolicyStructure object to refer to any MPMPolicyStatement objects. This operation does NOT affect either the MPMPolicyStatement object or the MPMPolicyStructure object; it just deletes the association between this MPMPolicyStructure object and this MPMPolicyStatement object. This operation has no input parameters.

delMPMPolStatementObjectPartialList(in polStatementObjectList: MPMPolicyStatement[1..*])	This operation removes the association, and its association class, for each MPMPolicyStatement object in the polStatementObjectList that is associated with this particular MPMPolicyStructure object. This operation takes a single input parameter, called polStatementObjectList, that defines the set of MPMPolicyStatement objects that will be unlinked from this particular MPMPolicyStructure object. This operation does NOT affect either the MPMPolicyStructure object or the MPMPolicyStatement object; it just deletes the association between this MPMPolicyStructure object and this MPMPolicyStatement object. **[R22]** Any association between this MPMPolicyStructure object and other MPMPolicyStatement objects that is not specified in the polStatementObjectList **MUST NOT** be affected.

Table 5. Operations of the MCMPolicyStructure Class

8.7.2 MPMPolicyStructure Relationships

The MPMPolicyStructure class defines three aggregation relationships, as shown in Figure 10.

8.7.2.1 The MPMPolicyHasMPMPolicySource Aggregation

The MPMPolicyHasMPMPolicySource aggregation is an optional aggregation, and defines the set of MPMPolicySource objects that are attached to this particular MPMPolicyStructure object. The semantics of this aggregation are defined by the MPMHasPolicySourceDetail association class.

MPMPolicySource objects are used for authorization policies, as well as to enforce deontic and alethic logic.

The multiplicity of this aggregation is 0..1 - 0..n. This means that it is an optional aggregation (i.e., the “0” part of the 0..1 cardinality). If this aggregation is instantiated (i.e., the “1” part of the 0..1 cardinality), then zero or more MCMPolicySource objects can wrap this particular MCMPolicyStructure object. The 0..* cardinality enables an MCMPolicyStructure object to be defined without having to define an associated MCMPolicySource object for it to aggregate. The semantics of this aggregation are defined by the MPMHasPolicySourceDetail association class. This enables the management system to control which set of concrete subclasses of MCMPolicyStructure can aggregate which types of MPMPolicySource objects.

[O2] An MPMPolicyStructure object, or any of its subclasses, **MAY** aggregate zero or more MPMPolicySource objects.

The MPMHasPolicySourceDetail is a concrete association class, and defines the semantics of the MPMHasPolicySource aggregation. The attributes and relationships of this class can be used to define which MPMPolicySource objects can be attached to which particular set of MPMPolicyStructure objects. These semantics can be further enhanced by using the Policy Pattern to define policy rules that constrain which part objects (i.e., MPMPolicySource) are attached to which object. Note that MCMPolicyStructure is an abstract class that is the superclass of imperative, declarative, and intent policy rules.

8.7.2.2 The MPMPolicyHasMPMPolicyTarget Aggregation

The MPMPolicyHasMPMPolicyTarget aggregation is a mandatory aggregation, and defines the set of MPMPolicyTarget objects that are attached to this particular MPMPolicyStructure object. The semantics of this aggregation are defined by the MPMHasPolicyTargetDetail association class.

MPMPolicyTarget objects are MCMManagedEntity objects whose state and/or behavior will be affected by the execution of a set of MPMPolicy objects.

The multiplicity of this aggregation is 0..1 - 1..n. This means that this aggregation is optional (i.e., the "0" part of the 0..1 cardinality). If this aggregation is instantiated (i.e., the "1" part of the 0..1 cardinality), then one or more MPMPolicyTarget objects can be aggregated by this particular MPMPolicyStructure object. Note that the cardinality on the part side (MPMPolicyTarget) is 1..*; this cardinality was chosen to make explicit that any MPMPolicy object must contain at least one MPMPolicyTarget object to be considered a valid policy rule. Otherwise, there are no objects to apply the MPMPolicy to.

[R23] An MPMPolicyStructure object, or any of its subclasses, **MUST** aggregate one or more MPMPolicyTarget objects.

The MPMHasPolicyTargetDetail object is a concrete association class, and defines the semantics of the MPMHasPolicyTarget aggregation. The attributes and relationships of this class can be used to define which MPMPolicyTarget objects can be attached to which particular set of MPMPolicyStructure objects. These semantics can be further enhanced by using the Policy Pattern to define policy rules that constrain which part objects (i.e., MPMPolicyTarget) are attached to which object. Note that MCMPolicyStructure is an abstract class that is the superclass of imperative, declarative, and intent policy rules.

8.7.2.3 The MPMPolicyHasMPMPolicyStatement Aggregation

This is an mandatory aggregation, and defines the set of MPMPolicyStatement objects that are attached to this particular MPMPolicyStructure object. The attachment of different MPMPolicyStatement objects changes the content, and hence the behavior, of a given MPMPolicyStructure object. The semantics of this aggregation are defined by the MPMHasPolicyStatementDetail association class.

MPMPolicyStatement objects define the content of a given MPMPolicyStructure object. Every MPMPolicyStructure object consists of one or more MPMPolicyStatement objects. An MPMPolicyStatement object may be decorated by zero or more MPMPolicyComponentDecorator objects.

The multiplicity of this aggregation is 0..1 - 1..n. This means that this aggregation is optional (i.e., the "0" part of the 0..1 cardinality). If this aggregation is instantiated (i.e., the "1" part of the 0..1 cardinality), then one or more MPMPolicyStatement objects can be aggregated by this particular MPMPolicyStructure object. Note that the cardinality on the part side (MPMPolicyStatement) is 1..*; this cardinality was chosen to make explicit that any MPMPolicy object must contain at least one MPMPolicyStatement object to be considered a valid policy rule. Otherwise, the MPMPolicy is malformed, and does not contain any content statements.

[R24] An MPMPolicyStructure object, or any of its subclasses, **MUST** aggregate one or more MPMPolicyStatement objects.

The MPMHasPolicyStatementDetail object is a concrete association class, and defines the semantics of the MPMHasPolicyStatement aggregation. The attributes and relationships of this class can be used to define which MPMPolicyStatement objects can be attached to which particular set of MPMPolicyStructure objects. These semantics can be further enhanced by using the Policy Pattern to define policy rules that constrain which part objects (i.e., MPMPolicyStatement) are attached to which object. Note that MCMPolicyStructure is an abstract class that is the superclass of imperative, declarative, and intent policy rules.

8.7.3 MPMPolicyStructure Subclasses

The MPMPolicyStructure class currently defines three subclasses, which are described in the following subsections. Figure 12 will be used to describe each of these three subclasses.

8.7.3.1 MPMImperativePolicy Class Definition

This is a mandatory abstract class, which is a type of PolicyContainer that is used to represent imperative policy rules. An imperative policy explicitly defines how the state of the target MCMManagedEntity objects will be affected. This version of this specification supports two types of imperative policy rules: (1) ECA policy rules and (2) commands.

Figure 12 shows the attributes of this class, and Table 6 defines the attributes for this class.

Figure 12. MPMPolicyStructure Subclasses

Attribute Name	Description
mpmImpPolPriority : Integer[0..1]	This is an optional non-negative integer attribute that defines the priority of this particular MPMImperativePolicy object. A larger value indicates a higher priority. Priority can be used to resolve conflicts among policy actions. For example, given a set of conflicting policy rules, it can be used to define which policy rule will will execute. It can also be used to define the execution order of a set of policy rules. **[O3]** A default value of 0 **MAY** be assigned.
mpmImpPolExecStrategy : MPMImpPolExecStrategy[1]	This is a mandatory non-negative integer attribute that defines the execution strategy of this particular MPMImperativePolicy object. The execution strategy consists of the order that actions will execute, and whether encountering an error terminates the process of executing actions or not. **[R25]** If no actions are contained in this MPMImperativePolicy class, then an error **MUST** be returned.

Table 6. Attributes of the MPMImperativePolicy Class

Table 7 defines the operations for this class.

Operation Name	Description
getMPMImpPolPriority() : Integer[1..1]	This operation returns the current value of the mpmImpPolRulePriority attribute. This operation takes no input parameters. **[R26]** If the mpmImpPolRulePriority attribute does not have a value, then this operation **MUST** return an error.
setMPMImpPolPriority (in polRulePriority : Integer[1..1])	This operation sets the value of the mpmImpPolRulePriority attribute. This operation takes a single input parameter, called polRulePriority, which defines the new value for the mpmImpPolRulePriority attribute. **[R27]** The value of the mpmImpPolRulePriority attribute **MUST** be a non-negative integer.
getMPMImpPolExecStrategy() : MPMImpPolExecStrategy [1..1]	This operation returns the current value of the mpmImpPolExecStrategy attribute. This operation takes no input parameters. **[R28]** If the mpmImpPolExecStrategy attribute does not have a value, then this operation **MUST** return an error.
setMPMImpPolExecStrategy(in newStrategy : MPMImpPolExecStrategy[1..1])	This operation sets the value of the mpmImpPolExecStrategy attribute. This operation takes a single input parameter, called newStrategy, which defines the new value for the mpmImpPolExecStrategy attribute. Valid values are defined by the MPMImpPolExecStrategy enumeration.

Table 7. Operations of the MPMImperativePolicy Class

8.7.3.1.1 MPMECAPolicy Class Definition

This is a mandatory concrete class, whose superclass is MPMImperativePolicy. An MPMECA-Policy is a PolicyContainer that aggregates a set of events, conditions, and actions into an imperative policy rule known as an Event-Condition-Action (ECA) policy rule. This has the following semantics:

IF the event portion of the policy rule evaluates to TRUE
  IF the condition portion of the policy rule evaluates to TRUE
    THEN actions in the action portion of the policy rule may be executed
  ENDIF
ENDIF

In the above definition:

- An **event** is a Boolean statement that represents something that happens or is happening that triggers a decision-making process to start
- A **condition** is a Boolean statement that is an evaluation in a decision-making process
- An **action** is a Boolean statement that defines an atomic computation that is executed as a result of a decision-making process

The event, condition, and action portions of an MPMECAPolicy will be referred to as Event, Condition, and Action Statements (to differentiate them from Event, Condition, and Action objects). The Event, Condition, and Action Statements are all Boolean statements (i.e., a statement that produces a value of either true or false). An MPMECAPolicy refines the notion of an MPMImperativePolicy by mandating that at least one Event or Condition Statement is present, and at least one Action Statement is present.

[R29] An MPMECAPolicy **MUST** contain at least one Event Statement or at least one Condition Statement.

[R30] An MPMECAPolicy **MUST** contain at least one Action Statements.

Any Boolean statement can be combined with another Boolean statement to form compound Boolean statements using any of the logical connectives (i.e., AND, OR, and NOT). This realizes the concept of a portion of an MPMECAPolicy evaluating to true. For example, if an event Boolean clause is true, that satisfies the first IF statement in the above pseudocode. As another example, the event portion of an MPMECAPolicy may consist of two or more Boolean statements; this enables the evaluation of the event portion to be determined by the Boolean value of each statement according to the logical connectives that are present in the event portion. Boolean statements are realized by the MPMBooleanStatement class (see section 8.8.3.2). Note that other types of MPMPolicyStatements may be combined with one or more MPMBooleanStatements for any of the Event, Condition, and Action Statements. In addition, any Event, Condition, or Action Statement can be decorated with a concrete subclass of the MPMPolicyComponentDecorator class (see section 8.8.5).

An MPMECAPolicy must have an action portion that is made up of one or more MPMPolicyStatements. An MPMECAPolicy can have a null event or condition, but not both.

The following requirements summarize the structural semantics of an MPMECAPolicy.

[O4] An MPMECAPolicy **MAY** contain one or more Event Statements.

[R31] If an MPMECAPolicy does not contain an Event Statement, the Condition Statement **MUST** both trigger the start of the decision-making process and evaluate the decision.

[O5] An MPMECAPolicy **MAY** contain one or more Condition Statements.

[R32] If an MPMECAPolicy does not contain a Condition Statement, the Event Statement **MUST** both trigger the start of the decision-making process and evaluate the decision.

[O6] Either the Event Statement or the Condition Statement, but not both, **MAY** be NULL in an MPMECAPolicy.

The following requirements summarize the behavioral semantics of an MPMECAPolicy.

[R33] An MPMECAPolicy **MUST** contain one or more MPMBooleanStatements.

[O7] An MPMECAPolicy **MAY** contain other types of MPMPolicyStatements.

[R34] Each of the Event, Condition, and Action Statements **MUST** contain one or more MPMBooleanStatements.

[O8] Any MPMBooleanStatement **MAY** contain other types of MPMPolicyStatements, as long as their addition does not prevent the MPMBooleanStatement from evaluating to either true or false.

[O9] Any MPMBooleanStatement **MAY** be decorated by one or more concrete subclasses of the MPMPolicyComponentDecorator class.

There are currently no attributes or methods defined for this class. Its purpose is to provide a concrete realization of a particular type of MPMImperativePolicy with the above semantics.

8.7.3.1.2 MPMCommandPolicyRule Class Definition

This is a mandatory concrete class, whose superclass is MPMImperativePolicy. An MPMCommandPolicy is a PolicyContainer that contains one or more Action Statements. Stylistically, it corresponds to the imperative mood in English.

[R35] An MPMCommandPolicy **MUST** contain one or more Action Statements.

[R36] An instance of this class **MUST NOT** contain Event or Condition Statements.

[R37] Each Action Statement **MUST** contain one or more MPMBooleanStatements.

[O10] Any MPMBooleanStatement **MAY** contain other types of MPMPolicyStatements, as long as their addition does not prevent the MPMBooleanStatement from evaluating to either true or false.

[O11] Any MPMBooleanStatement **MAY** be decorated by one or more concrete subclasses of the MPMPolicyComponentDecorator class.

The difference between an MPMCommandPolicy and an MPMECAPolicy is that the former only has a set of Action Statements, whereas the latter has either an Event and/or a Condition Statement in addition to an Action Statement.

There are currently no attributes or methods defined for this class. Its purpose is to provide a concrete realization of a particular type of MPMImperativePolicy that does not need Event and Condition Statements.

8.7.3.2 MPMDeclarativePolicy Class Definition

This is a mandatory concrete class, which is a type of PolicyContainer that is used to represent declarative policy rules. Figure 12 shows the attributes and operations of this class.

A declarative policy uses statements to express the goals of the policy, but not how to accomplish those goals.

In this document, Declarative Policy will refer to policies that execute as theories of a formal logic.

[R38] A Declarative Policy **MUST** be written using propositional, predicate, or a higher form of a formal logic.

Table 8 defines the attributes for this class.

Attribute Name	Description
mpmDecPolLogicType : MPMFormalLogicType[1..1]	This is a mandatory non-negative enumerated integer. It defines the type of formal logic used by this MPMDeclarativePolicy object. Allowed values of this enumeration are defined by the MPMFormalLogicType enumeration.

Table 8. Attributes of the MPMDeclativePolicy Class

Table 9 defines the operations for this class.

Operation Name	Description
getMPMDeclLogicType() : MPMFormalLogicType[1..1]	This operation returns the current value of the mpmDecPolLogicType attribute. This operation takes no input parameters. **[R39]** If the mpmDecPolLogicType attribute does not have a value, then this operation **MUST** return an error.

setMPMDeclLogicType(in polLogicType : MPMFormalLogicType[1..1])	This operation sets the value of the mpmDecPolLogicType attribute. This operation takes a single input parameter, called polLogicType, which defines the new value for the mpmDecPolLogicType attribute. The allowed values for the mpmDecPolLogicType attribute are defined by the MPMFormalLogicType attribute.

Table 9. Operations of the MPMDeclativePolicy Class

8.7.3.3 MPMIntentPolicy Class Definition

This is a mandatory concrete class, which is a type of PolicyContainer that is used to represent intent policy rules. Figure 12 shows the attributes and operations of this class.

An intent policy is a type of policy that uses statements from a restricted natural language to express the goals of the policy, but not how to accomplish those goals. An Intent Policy is written in a Controlled Language (i.e., a language that restricts the grammar and vocabulary used). In particular, formal logic syntax is not used. This version of this specification will use a restricted version of English. Controlled languages simplify machine translation of the source content, and enable the source content to be translated to other types of languages. An example of a Controlled Language is Attempto Controlled English; most Domain Specific Languages (DSLs) are also Controlled Languages.

[R40] An Intent Policy **MUST** be written in a Controlled Language.

[O12] An Intent Policy **MAY** be written in a DSL.

An example of a DSL for use by MPMIntentPolicy objects is provided in Appendix A. In general, each statement in an Intent Policy may require the translation of one or more of its terms to a form that another MCMManagedEntity can understand.

In general, a newly written intent is likely to not be directly executable. This is because of ambiguities in using a Controlled Language, as well as the use of more abstract comments. For example, a Customer might be referred to by name; this would need to be translated to a form that is machine processable (e.g., an IP address).

Table 10 defines the attributes for this class.

Attribute Name	Description
mpmIntentTranslationStatus : MPMIntentTranslationStatus[1..1]	This is a mandatory non-negative enumerated integer, and defines the status of the translation of the content of this MPMIntentPolicy. Allowed values of this enumeration are defined in the MPMIntentTranslationStatus enumeration. **[R41]** If the value of the mpmIntentTranslationStatus attribute is not 2 (i.e., SUCCESS), then this MPMIntentPolicy **MUST NOT** be executed.

Table 10. Attributes of the MPMIntentPolicy Class

Table 11 defines the operations for this class.

Operation Name	Description
getMPMIntentTranslationStatus() : MPMIntentTranslationStatus[1..1]	This operation returns the current value of the mpmIntentTranslationStatus attribute. This operation takes no input parameters. **[R42]** If the mpmIntentTranslationStatus attribute does not have a value, then this operation **MUST** return an error.
setMPMIntentTranslationStatus(in intentTranslationStatus : MPMIntentTranslationStatus[1..1])	This operation sets the value of the mpmIntentTranslationStatus attribute. This operation takes a single input parameter, called intentTranslationStatus, which defines the new value for the mpmIntentTranslationStatus attribute. Valid values are defined by the MPMIntentTranslationStatus enumeration.

Table 11. Operations of the MPMIntentPolicy Class

8.8 MPMPolicyComponentStructure Class Hierarchy

The structure of the top portion of this class hierarchy is shown in Figure 13. This class hierarchy is defined to facilitate adding new types of policy components later. The main "worker" class is MPMPolicyStatement; concrete subclasses of this class are aggregated by all types of policy rules (i.e., concrete subclasses of MPMPolicyStructure). An MPMPolicyStatement object may optionally be made up of MPMPolicyClause objects. MPMPolicyComponentDecorator is used to define optional objects (or parts of an object) to decorate, or wrap, concrete subclasses of MPMPolicyStatement and/or MPMPolicyClause objects.

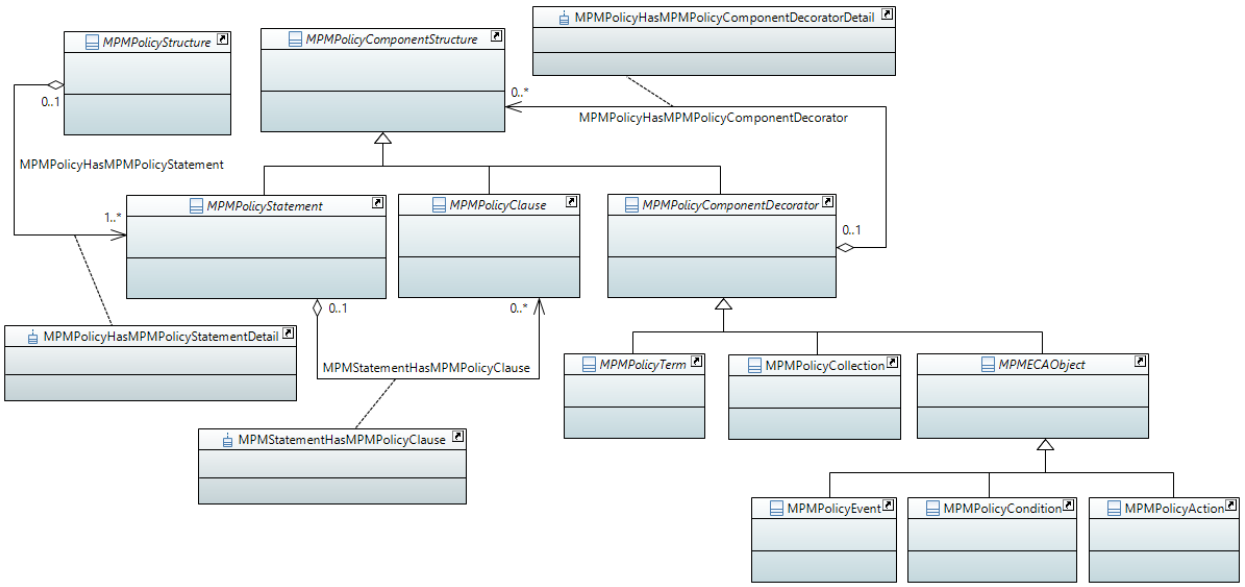


Figure 13. The Top Portion of the MPMPolicyComponentStructure Hierarchy

8.8.1 MPMPolicyComponentStructure Class Definition

This is a mandatory abstract class. It is the superclass for all types of components that may be contained in a particular type of an MPMPolicy. In this model, the type of Policy (e.g., imperative, declarative, intent) is a type of PolicyContainer. The type of PolicyContainer defines the type of MPMPolicyStructureComponent objects that it can contain.

This release will define imperative, declarative, and intent policies. However, the structure of this hierarchy is defined to facilitate adding new types of policies later.

This version of this specification does not define any attributes for this class. Its main purpose is from an ontological perspective, as it is used as the superclass for all types of components that can be contained by all types of policies that are defined by the MPM.

8.8.2 MPMPolicyComponentStructure Relationships

The MPMPolicyComponentStructure class is involved in one aggregation, which is called the MPMPolicyHasMPMPolicyComponentDecorator aggregation. This is an optional aggregation,

and defines the set of MPMPolicyComponentDecorator objects that wrap, or decorate, this particular MPMPolicyComponentStructure object. An MPMPolicyComponentStructure object may be decorated by zero or more MPMPolicyComponentDecorator objects. The semantics of this aggregation are defined by the MPMPolicyHasMPMPolicyComponentDecoratorDetail association class.

The attachment of different MPMPolicyComponentDecorator objects changes the syntax, semantics, and behavior of a given MPMPolicyComponentStructure object.

The multiplicity of this aggregation is 0..1 - 0..n. This means that this aggregation is optional (i.e., the "0" part of the 0..1 cardinality). If this aggregation is instantiated (i.e., the "1" part of the 0..1 cardinality), then zero or more MPMPolicyComponentDecorator objects can decorate this particular MPMPolicyComponentStructure object. The 0..* cardinality enables an MPMPolicyComponentStructure object to be defined without having to define an associated MPMPolicyComponentDecorator object for it to decorate.

The MPMPolicyHasMPMPolicyComponentDecoratorDetail object is a concrete association class, and defines the semantics of the MPMPolicyHasMPMPolicyComponentDecorator aggregation. The attributes and relationships of this class can be used to define which MPMPolicyComponentDecorator objects can decorate this particular set of MPMPolicyComponentStructure objects. These semantics can be further enhanced by using the Policy Pattern to define policy rules that constrain which part objects (i.e., MPMPolicyComponentDecorator) are attached to which object. Note that MCMPolicyStructure is an abstract class that is the superclass of imperative, declarative, and intent policy rules.

8.8.3 MPMPolicyComponentStructure Subclasses: MPMPolicyStatements

This section describes the main subclasses of the MPMPolicyComponentStructure hierarchy that define MPMPolicyStatement classes. This class and its concrete subclasses, frequently use the MPMPolicyClause class (this is described in section 8.8.4) and subclasses of MPMPolicyComponentDecorator (this is described in section 8.8.5).

8.8.3.1 MPMPolicyStatement Class Definition

This is a mandatory abstract class. It separates the representation of an MPMPolicy from its implementation.

An MPMPolicy, regardless of its structure and semantics, can be abstracted into a set of statements, which are instances of this class. Each statement can optionally be abstracted into a set of clauses, which are instances of MPMPolicyClause (see section 8.8.4). Each clause is made up of a set of policy elements. Thus, the type of MPMPolicyStructure determines the type of statements that it can contain; this in turn determines the types of clauses and policy elements that are allowed by this type of statement.

There are two ways to enforce the semantics of restricting the type of MPMPolicyStatements that can be contained in a particular type of MPMPolicyStructure:

- Use the MPMPolicyHasMPMPolicyStatementDetail association class
- Define OCL

The first method avoids the use of OCL, but is harder to implement. It uses the model elements of the MPMPolicyHasMPMPolicyStatementDetail association class to define explicit semantics to restrict the type of MPMPolicyStatement, and their decorations, that can be contained by this particular type of MPMPolicyStructure. The second is easier, since OCL is a formal language that enables these semantics to be easily defined. However, some implementations do not support OCL, so the particular choice of which method to use is left to the implementer.

Figure 14 shows the attributes, operations, and relationships of the MPMPolicyStatement class.

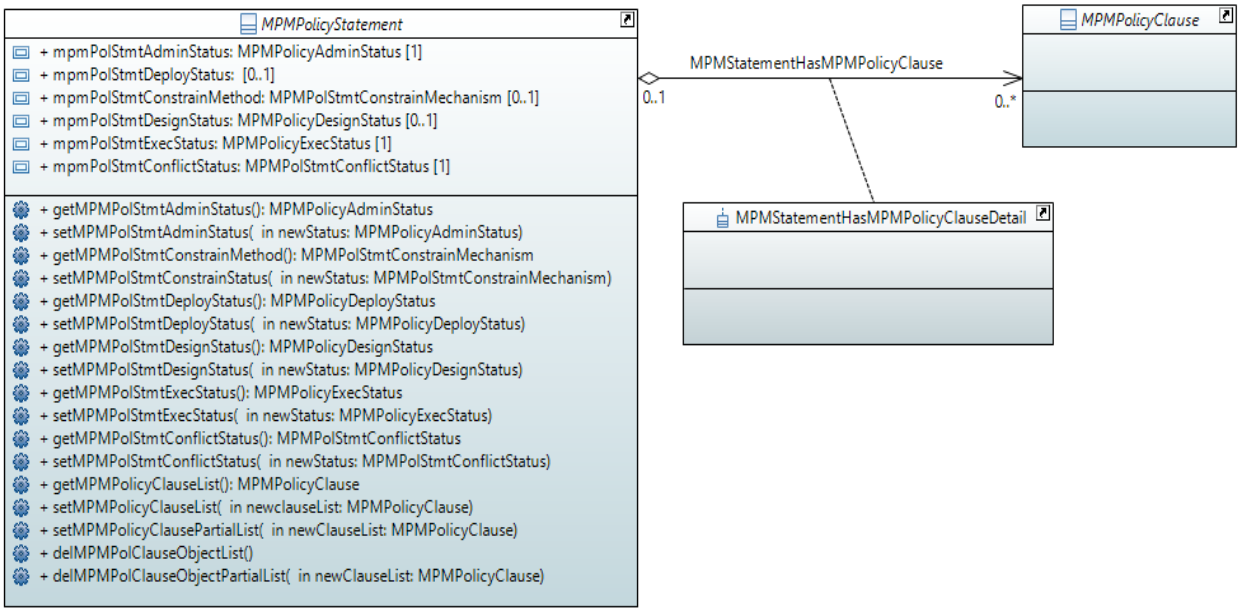


Figure 14. The MPMPolicyStatement Class

Table 12 defines the attributes for this class.

Attribute Name	Description
mpmPolStmtAdminStatus : MPMPolicyAdminStatus[1..1]	This is a mandatory enumerated non-negative integer attribute that defines the current administrative status of this particular MPMPolicy-Statement object. The allowable values of this enumeration are defined by the MPMPolicyAdminStatus enumeration.
mpmPolStmtConstrainMethod : MPMPolMethodConstrainMechanism[0..1]	This is a non-negative enumerated integer, and defines the mechanism used to constrain which concrete subclasses of MPMPolicyStatement can be used with this particular concrete subclass of MPMPolicyStructure. Allowed values are defined in the MPMPolStmtConstrainMechanism enumeration.
mpmPolStmtDeployStatus : MPMPolStatementDeployStatus[0..1]	This is an optional enumerated, non-negative integer attribute. It is used to indicate whether this MPMPolicyStatement can or cannot be deployed by the policy management system. This attribute enables the policy manager to know which MPMPolicies are currently deployed, and may be useful for the policy execution system for planning the staging of MPMPolicies. The allowable values of this enumeration are defined by the MPMPolicyDeployStatus enumeration.
mpmPolStmt-DesignStatus : MPMPolicy-DesignStatus[0..1]	This is an optional enumerated, non-negative integer whose value defines the current design status of this MPMPolicyStatement object. The allowed set of values are defined in the MPMPolicyDesignStatus enumeration.
mpmPolStmtExecStatus : MPMPolicyExecStatus[1..1]	This is a mandatory enumerated non-negative enumerated integer whose value defines the current execution status of this MPMPolicyStatement object. The allowed set of values are defined in the MPMPolicyExecStatus enumeration.
mpmPolStmtConflictStatus : MPMPolStmtConflictStatus[1..1]	This is an optional enumerated, non-negative integer whose value defines whether this particular MPMPolicyStatement has, or ever had, a conflict with another MPMPolicyStatement. The allowed set of

	values are defined in the MPMPolStmtConflictStatus enumeration. **[R43]** If the value of this attribute is not "RESOLVED" or "NONE", then this MPMPolicyStatement object **MUST NOT** be used.

Table 12. Attributes of the MPMPolicyStatement Class

Table 13 defines the operations for this class.

Operation Name	Description
getMPMPolStmtAdminStatus() : MPMPolicyAdminStatus[1..1]	This operation returns the current value of the mpmPolStmtAdminStatus attribute. This operation takes no input parameters. **[R44]** If the mpmPolStmtAdminStatus attribute does not have a value, then this operation **MUST** return an error.
setMPMPolStmtAdminStatus(in newStatus : MPMPolicyAdminStatus[1..1])	This operation sets the value of the mpmPolStmtAdminStatus attribute. This operation takes a single input parameter, called newStatus, which defines the new value for the mpmPolStmtAdminStatus attribute. Valid values are defined by the MPMPolicyAdminStatus enumeration.
getMPMPolStmtConstrainMethod() : MPMPolStmtConstrainMechanism[1..1]	This operation returns the current value of the mpmPolConstrainMethod attribute. This operation takes no input parameters. **[R45]** If the mpmPolConstrainMethod attribute does not have a value, then this operation **MUST** return an error.
setMPMPolStmtConstrainMethod (in newStatus : MPMPolStmtConstrainMechanism[1..1])	This operation sets the value of the mpmPolConstrainMethod attribute. This operation takes a single input parameter, called newStatus, which defines the new value for the mpmPolConstrainMethod attribute. Valid values are defined by the MPMPolStmtConstrainMechanism enumeration.
getMPMPolStmtDeployStatus() : MPMPolicyDeployStatus[1..1]	This operation returns the current value of the mpmPolStmtDeployStatus attribute. This operation takes no input parameters. **[R46]** If the mpmPolStmtDeployStatus attribute does not have a value, then this operation **MUST** return an error.
setMPMPolStmtDeployStatus(in newStatus : MPMPolicyDeployStatus[1..1])	This operation sets the value of the mpmPolStmt-DeployStatus attribute. This operation takes a single input parameter, called newStatus, which defines the new value for the mpmPolStmtDeployStatus attribute. Valid values are defined by the MPMPolicyDeployStatus enumeration.

getMPMPolStmtDesignStatus() : MPMPolicyDesignStatus[1..1]	This operation returns the current value of the mpmPolStmtDesignStatus attribute. This operation takes no input parameters. **[R47]** If the mpmPolStmtDesignStatus attribute does not have a value, then this operation **MUST** return an error.
setMPMPolStmtDesignStatus(in newStatus : MPMPolicyDesignStatus[1..1])	This operation sets the value of the mpmPolStmt-DesignStatus attribute. This operation takes a single input parameter, called newStatus, which defines the new value for the mpmPolStmtDesignStatus attribute. Valid values are defined by the MPMPolicyDesignStatus enumeration.
getMPMPolStmtExecStatus() : MPMPolicyExecStatus[1..1]	This operation returns the current value of the mpmPolStmtExecStatus attribute. This operation takes no input parameters. **[R48]** If the mpmPolStmtExecStatus attribute does not have a value, then this operation **MUST** return an error.
setMPMPolStmtExecStatus(in newStatus : MPMPolicyExecStatus[1..1])	This operation sets the value of the mpmPolStmtExecStatus attribute. This operation takes a single input parameter, called newStatus, which defines the new value for the mpmPolStmtExecStatus attribute. Valid values are defined by the MPMPolicyExecStatus enumeration.
getMPMPolStmtConflictStatus() : MPMPolStmtConflictStatus[1..1])	This operation returns the current value of the mpmPolStmtConflictStatus attribute. This operation takes no input parameters. **[R49]** If the mpmPolStmtConflictStatus attribute does not have a value, then this operation **MUST** return an error.
setMPMPolStmtConflictStatus(in newStatus : MPMPolStmtConflictStatus[1..1])	This operation sets the value of the mpmPolStmtConflictStatus attribute. This operation takes a single input parameter, called newStatus, which defines the new value for the mpmPolStmtConflictStatus attribute. Valid values are defined by the MPMPolStmtConflictStatus enumeration.
getMPMPolicyClauseList() : MPMPolicyClause[1..*]	This operation retrieves the set of MPMPolicyClause objects that are contained in this particular MPMPolicyStatement object.

	This is obtained by following the MPMStatementHasMPMPolicyClause aggregation. Each instance of this aggregation defines an MPMPolicyClause object, which is then added to the return value of this operation. The return value of this operation is an array of one or more MPMPolicyClause objects. This operation takes no input parameters. **[D24]** If this MPMPolicyStatement object does not instantiate this aggregation, then this operation **SHOULD** return a NULL MPMPolicyClause object.
setMPMPolicyClauseList(in newClauseList : MPMPolicyClause[1..*])	This operation defines a new set of MPMPolicyClause objects that will be contained in this particular MPMPolicyStatement object. This operation takes a single input parameter, called newClauseList, which defines a set of one or more MPMPolicyClause objects. If this MPMPolicyStatement object already has a set of one or more MPMPolicyClause objects that it contains, then those MPMPolicyClause objects will be deleted by first, deleting the accompanying association class, and second, deleting the corresponding association. Then, a new association (an instance of MPMStatementHasMPMPolicyClause) is created for each MPMPolicyClause object in the newClauseList parameter. **[D25]** Every association created **SHOULD** have a new association class created to realize the semantics of that association.

setMPMPolicyClausePartialList(in newClauseList : MPMPolicyClause[1..*])	This operation defines a new set of MPMPolicyClause objects that will be contained in this particular MPMPolicyStatement object. This operation takes a single input parameter, called newClauseList, which defines a set of one or more MPMPolicyClause objects. If this MPMPolicyStatement object already has a set of one or more MPMPolicyClause objects that it contains, then those MPMPolicyClause objects are ignored. Then, a new association (an instance of MPMStatementHasMPMPolicyClause) is created for each MPMPolicyClause object in the newClauseList. **[D26]** Every association created **SHOULD** have a new association class created to realize the semantics of that association. **[R50]** Any association between this MPMPolicyStatement object and other MPMPolicyClause objects that is not specified in the newClauseList **MUST NOT** be affected.
delMPMPolClauseObjectList()	This operation removes all instances of the MPMStatementHasMPMPolicyClause aggregation, and its association classes, that enables this particular MPMPolicyStatement object to contain any MPMPolicyClause objects. This operation does NOT affect either the MPMPolicyClause object or the MPMPolicyStatement object; it just deletes the association between this MPMPolicyStatement object and this MPMPolicyClause object. This operation has no input parameters.

delMPMPolClauseObjectPartialList(in newClauseList: MPMPolicyClause[1..*])	This operation removes the association, and its association class, for each MPMPolicyClause object in the newClauseList that is contained by this particular MPMPolicyStatement object. This operation takes a single input parameter, called newClauseList, that defines the set of MPMPolicyClause objects that will be unlinked from this particular MPMPolicyStatement object. This operation does NOT affect either the MPMPolicyStatement object or the MPMPolicyClause object; it just deletes the association between this MPMPolicyStatement object and this MPMPolicyClause object. **[R51]** Any association between this MPMPolicyStatement object and other MPMPolicySource objects that is not specified in the newClauseList **MUST NOT** be affected.

Table 13. Operations of the MPMPolicyStatement Class

The MPMPolicyStatement class participates in two aggregations.

8.8.3.1.1 The MPMPolicyHasMPMPolicyStatement Aggregation

This is an aggregation that was defined in section 8.7.2.3. This defines the set of MPMPolicyStatement objects that form the content of a given MPMPolicyStructure.

8.8.3.1.2 The MPMStatementHasMPMPolicyClause Aggregation

This is an aggregation that defines the set of MPMPolicyClause objects that make up this particular MPMPolicyStatement.

This aggregation enables the content of an MPMPolicyStatement to be changed without affecting the rest of the MPMPolicy.

The multiplicity of this aggregation is 0..1 - 0..n. This means that it is an optional aggregation (i.e., the “0” part of the 0..1 cardinality). If this aggregation is instantiated (i.e., the “1” part of the 0..1 cardinality), then zero or more MPMPolicyClause objects define the content of this particular MPMPolicyStatement object. The 0..* cardinality enables an MPMPolicyStatement object to be defined without having to define an associated MPMPolicyClause object for it to aggregate. The semantics of this aggregation are defined by the MPMStatementHasMPMPolicyClauseDetail association class. This enables the management system to control which set of concrete subclasses of MCMPolicyStatement can aggregate which types of MPMPolicyClause objects.

The MPMStatementHasMPMPolicyClauseDetail is a concrete association class, and defines the semantics of the MPMStatementHasMPMPolicyClause aggregation. The attributes and

relationships of this class can be used to define which MPMPolicyClause objects can be aggregated by which particular set of MCMPolicyStatement objects. These semantics can be further enhanced by using the Policy Pattern to define policy rules that constrain which part objects (i.e., MPMPolicyClause) are attached to which MCMPolicyStatement object. Note that MCMPolicyStructure is an abstract class that is the superclass of imperative, declarative, and intent policy rules.

8.8.3.2 MPMBooleanStatement Class Definition

An MPMBooleanStatement specializes an MPMPolicyStatement, and defines a statement that evaluates to either true or false.

An MPMBooleanStatement may be made up of one or more Boolean clauses, which is a subclass of the MPMPolicyClause class (see section 8.8.4). This is modeled using the MPMStatementHasMPMPolicyClause aggregation.

Boolean expressions correspond to propositional formulas in logic. Hence, an MPMBooleanStatement may be used by imperative, declarative, and intent policies.

Figure 15 shows the MPMBooleanStatement class, along with its sibling classes.

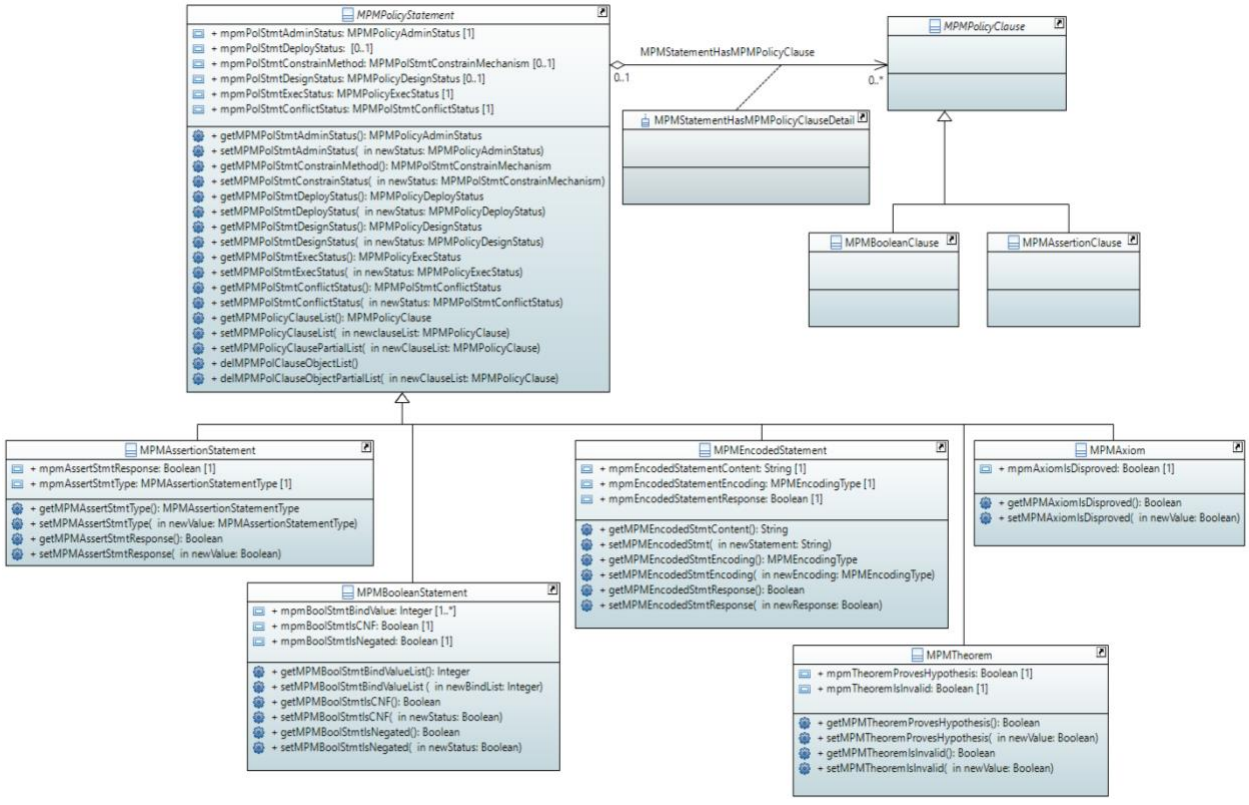


Figure 15. Subclasses of the MPMPolicyStatement Class

Table 14 defines the attributes for the MPMBooleanStatement class.

Attribute Name	Description
mpmBoolStmtBindValue : Integer[1..*]	This is a mandatory array of positive integers that defines the order in which constituent terms bind to this MPMBooleanStatement. For example, the Boolean expression "((A AND B) OR (C AND NOT (D OR E)))" has the following binding order: terms A and B have a bind value of 1; term C has a binding value of 2, and terms D and E have a binding value of 3. **[R52]** All values in this attribute **MUST** be greater than 0.
mpmBoolStmtIsCNF : Boolean[1..1]	This is a mandatory Boolean attribute. If the value of this attribute is TRUE, then this MPMBooleanStatement is in Conjunctive Normal Form. Otherwise, it is in Disjunctive Normal Form.
mpmBoolStmtIsNegated: Boolean[1..1]	This is a mandatory Boolean attribute. If the value of this attribute is TRUE, then this (entire) MPMBooleanStatement is negated.

Table 14. Attributes of the MPMBooleanStatement Class

Table 15 defines the operations for the MPMBooleanStatement class.

Operation Name	Description
getMPMBoolStmt-BindValueList() : Integer[1..*]	This operation returns the current value of the mpmBoolStmtBindValue attribute, which is an array of positive integers. This operation takes no input parameters. **[R53]** If the mpmBoolStmtBindValue attribute does not have a value, then this operation **MUST** return an error.
setMPMBoolStmt-BindValueList (in newBindList : Integer[1..1])	This operation sets the value of the mpmBoolStmtBindValue attribute. This operation takes a single input parameter, called newBindList, which defines the new value(s) for the mpmBoolStmtBindValue attribute. The newBindList is an array of non-zero positive integers. **[R54]** All values in this attribute **MUST** be greater than 0.
getMPMBoolStmtIsCNF() : Boolean[1..1]	This operation returns the current value of the mpmBoolStmtIsCNF attribute. This operation takes no input parameters. **[R55]** If the mpmBoolStmtIsCNF attribute does not have a value, then this operation **MUST** return an error.

setMPMBoolStmtIsCNF (in newStatus : Boolean[1..1])	This operation sets the value of the mpmBoolStmtIsCNF attribute. This operation takes a single input parameter, called newStatus, which defines the new value for the mpmBoolStmtIsCNF attribute.
getMPMBoolStmtIsNegated() : Boolean[1..1]	This operation returns the current value of the mpmBoolStmtIsNegated attribute. This operation takes no input parameters. **[R56]** If the mpmBoolStmtIsNegated attribute does not have a value, then this operation **MUST** return an error.
setMPMBoolStmtIsNegated (in newStatus : Boolean[1..1])	This operation sets the value of the mpmBoolStmtIsNegated attribute. This operation takes a single input parameter, called newStatus, which defines the new value for the mpmBoolStmtIsNegated attribute.

Table 15. Operations of the MPMBooleanStatement Class

Note that there are no operations that retrieve the number of MPMBooleanClause objects from an MPMBooleanStatement. This is because of two reasons. First, the MPMBooleanStatement object inherits the MPMStatementHasMPMPolicyClause aggregation from its superclass. Second, the MPMBooleanStatement can aggregate more than one type of MPMPolicyClause object.

8.8.3.3 MPMAssertionStatement Class Definition

An MPMAssertionStatement is a collection of 2 or more MPMAssertionClauses (see section 8.8.4.1.). The canonical form of an MPMAssertionStatement is a 3-tuple, containing three MPMAssertionClauses:

<pre-condition, post-condition, invariant>

In this definition

- pre-conditions are predicates that must be true in order for a method or function to execute
- post-conditions are predicates that must be true after a method or function has executed
- attributes are predicates that must be true during the life of method or function execution

This 3-tuple is especially useful when reasoning about whether a computer program is correct. An enumeration (MPMAssertionStatementType) is defined that specifies what types of MPMAssertionClauses are used by this particular MPMAssertionStatement.

Figure 15 shows the MPMAssertionStatement class.

Table 16 defines the attributes for this class.

Attribute Name	Description
mpmAssertStmtResponse : Boolean[1..1]	This is a mandatory Boolean attribute that provides a Boolean response for this MPMAssertionStatement. This enables this MPMAssertionStatement to be combined with other subclasses of an MPMPolicyStatement that provide a Boolean value that defines the status as to their correctness and/or evaluation state. This enables this object to be used to construct more complex MPMPolicyStatements.
mpmAssertStmtType : MPMAssertionStatementType[1..1]	This is a mandatory enumerated non-negative integer attribute that defines the composition of this particular MPMAssertionStatement object. The allowable values of this enumeration are defined by the MPMAssertionStatementType enumeration.

Table 16. Attributes of the MPMAssertionStatement Class

Table 17 defines the operations for this class.

Operation Name	Description
getMPMAssertStmtResponse() : Boolean[1..1]	This operation returns the current value of the mpmAssertStmtResponse attribute. This operation takes no input parameters. **[R57]** If the mpmAssertStmtResponse attribute does not have a value, then this operation **MUST** return an error.
setMPMAssertStmtResponse (in newValue : Boolean[1..1])	This operation sets the value of the mpmAssertStmtResponse attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmAssertStmtResponse attribute.
getMPMAssertStmtType() : MPMAssertionStatementType[1..1]	This operation returns the current value of the mpmAssertStmtType attribute. This operation takes no input parameters. **[R58]** If the mpmAssertStmtType attribute does not have a value, then this operation **MUST** return an error.
setMPMAssertStmtType(in newValue : MPMAssertionStatementType[1..1])	This operation sets the value of the mpmAssertStmtType attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmAssertStmtType attribute.

Table 17. Operations of the MPMAssertionStatement Class

Note that there are no operations that retrieve the number of MPMAssertionClause objects from an MPMAssertionStatement. This is because of two reasons. First, the MPMAssertionStatement object inherits the MPMStatementHasMPMPolicyClause aggregation from its superclass. Second, the MPMAssertionStatement can aggregate more than one type of MPMPolicyClause object.

8.8.3.4 MPMEncodedStatement Class Definition

An MPMEncodedStatement represents a policy statement as an encoded object. This class defines a generalized extension mechanism for representing MPMPolicyStatements that have not been modeled with other MPMPolicyComponentStructure objects.

This class encodes the contents of the policy clause directly into the attributes of the MPMEncodedStatement. Hence, MPMEncodedStatement objects are reusable at the object level, whereas other types of MPMPolicyStatement objects are reusable at the individual policy expression level.

The benefit of an MPMEncodedStatement is that it enables direct encoding of the text of the MPMPolicyStatement, without having the "overhead" of using other objects. However, note that while this method is efficient, it does not reuse other MPMPolicyComponentStructure objects. Furthermore, its potential for reuse is reduced, as only MPMPolicies that can use the exact encoding of this clause can reuse this object.

Figure 15 shows the MPMEncodedStatement class.

Table 18 defines the attributes for this class.

Attribute Name	Description
mpmEncodedStatementContent : String[1..1]	This is a mandatory string attribute that defines the content of this particular MPMEncodedStatement object. It works with another class attribute, called mpmEncodedStatementEncoding, which defines how to interpret the value of this attribute (e.g., as a string or reference). These two attributes form a tuple, and together enable a machine to understand the syntax and value of this object instance.
mpmEncodedStatementEncoding : MPMEncodingType[1..1]	This is a mandatory enumerated non-negative integer attribute, and defines how to interpret the value of the mpmEncodedStatementContent class attribute. These two attributes form a tuple, and together enable a machine to understand the syntax and value of the encoded clause for the object instance of this class.

mpmEncodedStatementResponse : Boolean[1..1]	This is a mandatory Boolean attribute that emulates a Boolean response of this statement, so that it may be combined with other subclasses of the MPMPolicyStatement that provide a Boolean value that defines their correctness and/or evaluation state. This enables this object to be used to construct more complex Boolean clauses.

Table 18. Attributes of the MPMEncodedStatement Class

Table 19 defines the operations for this class.

Operation Name	Description
getMPMEncodedStmtContent() : String[1..1]	This operation returns the current value of the mpmEncodedStatementContent attribute. This operation takes no input parameters. **[R59]** If this attribute does not have a value, then this operation **MUST** return an error.
setMPMEncodedStmtContent (in newStatement : String[1..1])	This operation sets the value of the mpmEncodedStatementContent attribute. This operation takes a single input parameter, called newStatement, which defines the new value for the mpmEncodedStatementContent attribute. **[R60]** The value of the mpmEncodedStatementContent attribute **MUST NOT** be empty or NULL.
getMPMEncodedStmtEncoding() : MPMEncodingType[1..1]	This operation returns the current value of the mpmEncodedStatementEncoding attribute. This operation takes no input parameters. Valid values are defined in the MPMEncodingType enumeration. **[R61]** If this attribute does not have a value, then this operation **MUST** return an error.
setMPMEncodedStmtEncoding(in newEncoding : MPMEncodingType[1..1])	This operation sets the value of the mpmEncodedStatementContent attribute. This operation takes a single input parameter, called newEncoding, which defines the new value for the mpmEncodedStatementEncoding attribute.

getMPMEncodedStmtResponse() : Boolean[1..1]	This operation returns the current value of the mpmEncodedStatementResponse attribute. This operation takes no input parameters. **[R62]** If this attribute does not have a value, then this operation **MUST** return an error.
setMPMEncodedStmtResponse(in : newResponse : Boolean[1..1])	This operation sets the value of the mpmEncodedStatementResponse attribute. This operation takes a single input parameter, called newResponse, which defines the new value for the mpmEncodedStatementResponse attribute.

Table 19. Operations of the MPMEncodedStatement Class

Note that there are no operations that retrieve the number of MPMPolicyClause objects from an MPMEncodedStatement. This is because the MPMEncodedStatement object inherits the MPMStatementHasMPMPolicyClause aggregation from its superclass.

8.8.3.5 MPMTheorem Class Definition

An MPMTheorem is a type of MPMPolicyStatement that has the following characteristics:

1) it is non-self-evident
2) it can be proven to be true

The proof of a theorem is defined by the set of MPMPolicyClauses that it is associated with. Specifically, two or more MPMPremiseClause (see section 8.8.4.3.1) objects must have all been proven to be true, which makes the associated MPMConclusionClause (see section 8.8.4.3.2) true. This is found by following the MPMStatementHasMPMPolicyClause aggregation (see section 8.8.3.1.2).

[R63] An MPMTheorem object **MUST** have previously been proven to be true in order for it to be used.

Figure 15 shows the MPMTheorem class.

Table 20 defines the attributes for this class.

Attribute Name	Description
mpmTheorem-ProvesHypthesis : Boolean[1..1]	This is a mandatory Boolean attribute. If the value of this attribute is TRUE, then this MPMTheorem proves a previously unknown hypothesis. Otherwise, it is the result of previously known axioms and/or other theorems.

mpmTheoremIsInvalid : Boolean[1..1]	This is a mandatory Boolean attribute. If the value of this attribute is TRUE, then this MPMTheorem was rendered incorrect due to one of its dependent axioms or theorems, that was previously true, being proved false. This requires revisiting all MPMPolicyStatements that depended on it. **[R64]** If the value of this attribute is FALSE, then the system **MUST** set the mpmPolStmtExecStatus to ERROR for this MPMTheorem.

Table 20. Attributes of the MPMTheorem Class

Table 21 defines the operations for this class.

Operation Name	Description
getMPMTheoremProves-Hypothesis() : Boolean[1..1]	This operation returns the current value of the mpmTheoremProvesHypothesis attribute. This operation takes no input parameters. **[R65]** If this attribute does not have a value, then this operation **MUST** return an error.
setMPMTheoremProves-Hypothesis(in newValue : Boolean[1..1])	This operation sets the value of the mpmTheoremProvesHypothesis attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmTheoremProvesHypothesis attribute.
getMPMTheoremIsInvalid() : Boolean[1..1]	This operation returns the current value of the mpmTheoremIsInvalid attribute. This operation takes no input parameters. **[R66]** If this attribute does not have a value, then this operation **MUST** return an error.
setMPMTheoremProves-Hypothesis(newValue : Boolean[1..1])	This operation sets the value of the mpmTheoremIsInvalid attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmTheoremIsInvalid attribute. **[R67]** The value of the mpmTheoremIsInvalid attribute **MUST** be either true or false. **[R68]** If the value of this attribute is FALSE, then the system **MUST** set the mpmPolStmtExecStatus to ERROR for this MPMTheorem.

Table 21. Operations of the MPMTheorem Class

Note that there are no operations that retrieve the number of MPMPolicyClause objects from an MPMEncodedStatement. This is because the MPMEncodedStatement object inherits the MPMStatementHasMPMPolicyClause aggregation from its superclass.

8.8.3.6 MPMAxiom Class Definition

An MPMAxiom is a type of MPMStatement that is taken to always be TRUE. Hence, it serves as a premise for other types of reasoning. Axioms are linked to MPMPolicyStatements using the MPMStatementHasMPMPolicyClause aggregation (see section 8.8.3.1.2).

[R69] An MPMAxiom object **MUST** be defined as true in order for it to be used.

Figure 15 shows the MPMAxiom class.

Table 22 defines the attributes for this class.

Attribute Name	Description
mpmAxiomIsDisproved : Boolean[1..1]	This is a mandatory Boolean attribute. If the value of this attribute is TRUE, then this MPMAxiom has been proven FALSE. This requires revisiting all MPMPolicyStatements that depended on it. **[R70]** If the value of this attribute is FALSE, then the system **MUST** set the mpmPolStmtExecStatus to ERROR for this MPMAxiom.

Table 22. Attributes of the MPMAxiom Class

Table 23 defines the operations for this class.

Operation Name	Description
getMPMAxiomIsDisproved() : Boolean[1..1]	This operation returns the current value of the mpmAxiom-IsDisproved attribute. This operation takes no input parameters. **[R71]** If this attribute does not have a value, then this operation **MUST** return an error.

setMPMAxiomIsDisproved (newValue : Boolean[1..1])	This operation sets the value of the mpmAxiomIsDisproved attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmAxiomIsDisproved attribute. **[R72]** The value of the mpmAxiomIsDisproved attribute **MUST** be either true or false. **[R73]** If the value of this attribute is FALSE, then the system **MUST** set the mpmPolStmtExecStatus to ERROR for this MPMAxiom.

Table 23. Operations of the MPMAxiom Class

Note that there are no operations that retrieve the number of MPMPolicyClause objects from an MPMEncodedStatement. This is because the MPMEncodedStatement object inherits the MPMStatementHasMPMPolicyClause aggregation from its superclass.

8.8.4 MPMPolicyComponentStructure Subclasses: MPMPolicyClause

An MPMPolicyClause is a mandatory abstract class whose subclasses define different types of clauses that are used to create the content for different types of MPMPolicies. An MPMPolicyClause serves as a convenient aggregation point for assembling other objects that make up an MPMPolicyStatement. An MPMPolicyClause, along with its subclasses, is shown in Figure 16.

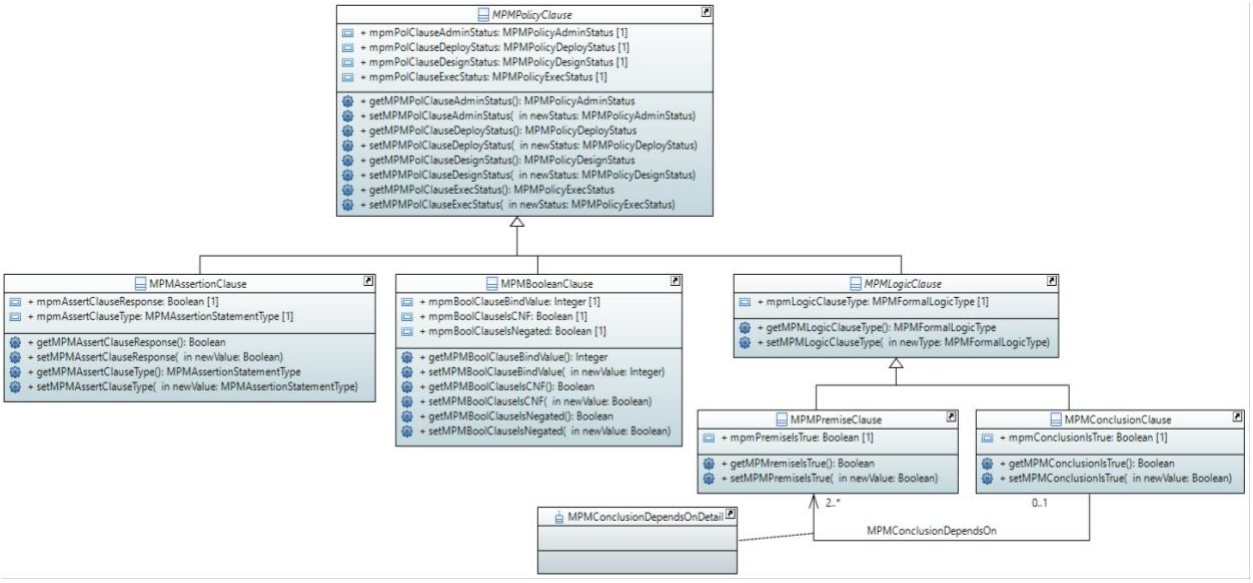


Figure 16. MPMPolicyClause and its Subclasses

An MPMPolicyClause is, as its name implies, a clause (i.e., a part of a statement), and defines all or part of the content of an MPMPolicyStatement. The decorator pattern is used to enable an extensible set of objects to "wrap" the MPMPolicyClause; this enables the contents of a MPMPolicyClause to be adjusted dynamically at runtime without affecting other objects.

MPMPolicyClauses are objects in their own right, which facilitates their reuse. MPMPolicyClauses can aggregate a set of any of the subclasses of MPMPolicyComponentDecorator.

Figure 16 shows the MPMPolicyClause class and its subclasses.

Table 24 defines the attributes for this class.

Attribute Name	Description
mpmPolClause-AdminStatus : MPMPolicy-AdminStatus[1..1]	This is a mandatory enumerated non-negative integer attribute that defines the current administrative status of this particular MPMPolicyClause object. The allowable values of this enumeration are defined by the MPMPolicyAdminStatus enumeration.
mpmPolClause-DeployStatus : MPMPolicy-DeployStatus[1..1]	This is an optional enumerated, non-negative integer attribute. It is used to indicate whether this MPMPolicyClause can or cannot be deployed by the policy management system. This attribute enables the policy manager to know which MPMClauses are currently deployed for a given MPMPolicyStatement, and may be useful for the policy execution system for planning the staging of MPMPolicies. The allowable values of this enumeration are defined by the MPMPolicyDeployStatus enumeration.
mpmPolClause-DesignStatus : MPMPolicy-DesignStatus[1..1]	This is an optional enumerated, non-negative integer whose value defines the current design status of this MPMPolicyClause object. The allowed set of values are defined in the MPMPolicyDesignStatus enumeration.
mpmPolClause-ExecStatus : MPMPolicy-ExecStatus[1..1]	This is a mandatory enumerated non-negative enumerated integer whose value defines the current execution status of this MPMPolicyClause object. The allowed set of values are defined in the MPMPolicyExecStatus enumeration.

Table 24. Attributes of the MPMPolicyClause Class

Table 25 defines the operations for this class.

Operation Name	Description
getMPMPolClauseAdmin-Status() : MPMPolicy-AdminStatus [1..1]	This operation returns the current value of the mpmPolClauseAdminStatus attribute. This operation takes no input parameters. **[R74]** If the mpmPolClauseAdminStatus attribute does not have a value, then this operation **MUST** return an error.
setMPMPolClauseAdmin-Status(in newStatus : MPMPolicyAdminStatus[1..1])	This operation sets the value of the mpmPolClause-AdminStatus attribute. This operation takes a single input parameter, called newStatus, which defines the new value for the mpmPolClauseAdminStatus attribute. Valid values are defined by the MPMPolicyAdminStatus enumeration.

getMPMPolClauseDeploy-Status() : MPMPolicy-DeployStatus [1..1]	This operation returns the current value of the mpmPolClauseDeployStatus attribute. This operation takes no input parameters. **[R75]** If the mpmPolClauseDeployStatus attribute does not have a value, then this operation **MUST** return an error.
setMPMPolClauseDeploy-Status(in newStatus : MPMPolicyDeployStatus[1..1])	This operation sets the value of the mpmPolClause-DeployStatus attribute. This operation takes a single input parameter, called newStatus, which defines the new value for the mpmPolStmtDeployStatus attribute. Valid values are defined by the MPMPolicyDeployStatus enumeration.
getMPMPolClauseDesign-Status() : MPMPolicy-DesignStatus [1..1]	This operation returns the current value of the mpmPolClauseDesignStatus attribute. This operation takes no input parameters. **[R76]** If the mpmPolClauseDesignStatus attribute does not have a value, then this operation **MUST** return an error.
setMPMPolClauseDesign-Status(in newStatus : MPMPolicyDesignStatus[1..1])	This operation sets the value of the mpmPolClause-DesignStatus attribute. This operation takes a single input parameter, called newStatus, which defines the new value for the mpmPolClauseDesignStatus attribute. Valid values are defined by the MPMPolicyDesignStatus enumeration.
getMPMPolClauseExec-Status() : MPMPolicy-ExecStatus [1..1]	This operation returns the current value of the mpmPolClauseExecStatus attribute. This operation takes no input parameters. **[R77]** If the mpmPolClauseExecStatus attribute does not have a value, then this operation **MUST** return an error.
setMPMPolClauseExec-Status(in newStatus : MPMPolicyExecStatus[1..1])	This operation sets the value of the mpmPolClause-ExecStatus attribute. This operation takes a single input parameter, called newStatus, which defines the new value for the mpmPolClauseExecStatus attribute. Valid values are defined by the MPMPolicyExecStatus enumeration.

Table 25. Operations of the MPMPolicyClause Class

8.8.4.1 MPMAssertionClause Class Definition

An assertion is a predicate (i.e., a Boolean-valued function), connected to a point in a program, that should evaluate to true at that point in the program's execution. An MPMAssertionClause may be used by different MPMPolicyStatements.

[O13] An MPMPolicyStatement **MAY** contain zero or more MPMAssertionClauses.

[O14] An MPMAssertionClause **MAY** be used with zero or more other MPMPolicyClauses.

Figure 16 shows the MPMAssertionClause class.

Table 26 defines the attributes for this class.

Attribute Name	Description
mpmAssertClause-Response : Boolean[1..1]	This is a mandatory Boolean attribute that provides a Boolean response for this clause. This enables this MPMAssertionClause to be combined with other subclasses of an MPMPolicyClause and/or an MPMPolicyStatement that provide a Boolean value that defines the status as to their correctness and/or evaluation state. This enables this object to be used to construct more complex MPMPolicyClauses and MPMPolicyStatements.
mpmAssertClauseType : MPMAssertionStatementType[1..1]	This is a mandatory enumerated non-negative integer attribute that defines the composition of this particular MPMAssertionClause object. The allowable values of this enumeration are defined by the MPMAssertionStatementType enumeration.

Table 26. Attributes of the MPMAssertionClause Class

Table 27 defines the operations for this class.

Operation Name	Description
getMPMAssertClause-Response() : Boolean[1..1]	This operation returns the current value of the mpmAssertClauseResponse attribute. This operation takes no input parameters. **[R78]** If the mpmAssertClauseResponse attribute does not have a value, then this operation **MUST** return an error.

setMPMAssertClause-Response(in newValue : Boolean[1..1])	This operation sets the value of the mpmAssertClause-Response attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmAssertClauseResponse attribute.
getMPMAssertClauseType() : MPMAssertionStatementType[1..1]	This operation returns the current value of the mpmAssertClauseType attribute. This operation takes no input parameters. **[R79]** If the mpmAssertClauseType attribute does not have a value, then this operation **MUST** return an error.
setMPMAssertClauseType(in newValue : MPMAssertionStatementType[1..1])	This operation sets the value of the mpmAssertClauseType attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmAssertClauseType attribute.

Table 27. Operations of the MPMAssertionClause Class

8.8.4.2 MPMBooleanClause Class Definition

A Boolean clause has the canonical form of a {variable, operator, value} 3-tuple, which evaluates to either true or false. Boolean clauses may be joined together using logical connectives (e.g., AND and OR). A Boolean clause may also be negated. A Boolean clause may be made up of a combination of the Boolean constants true or false, along with Boolean-typed variables, Boolean-valued operators, and Boolean-valued functions.

[O15] An MPMPolicyStatement **MAY** contain zero or more MPMBooleanClauses.

Figure 16 shows the MPMBooleanClause class.

Table 28 defines the attributes for this class.

Attribute Name	Description
mpmBoolClauseBindValue : Integer[1..1]	This is a mandatory array of positive integers that defines the order in which constituent terms bind to this MPMBooleanClause. For example, the Boolean expression "((A AND B) OR (C AND NOT (D OR E)))" has the following binding order: terms A and B have a bind value of 1; term C has a binding value of 2, and terms D and E have a binding value of 3. **[R80]** All values in this attribute **MUST** be greater than 0.

mpmBoolClauseIsCNF : Boolean[1..1]	This is a mandatory Boolean attribute. If the value of this attribute is TRUE, then this MPMBooleanClause is in Conjunctive Normal Form. Otherwise, it is in Disjunctive Normal Form.
mpmBoolClauseIsNegated : Boolean[1..1]	This is a mandatory Boolean attribute. If the value of this attribute is TRUE, then this (entire) MPMBooleanClause is negated.

Table 28. Attributes of the MPMBooleanClause Class

Table 29 defines the operations for this class.

Operation Name	Description
getMPMBoolClauseBind-Value() : Integer[1..1]	This operation returns the current value of the mpmBoolStmtBindValue attribute, which is an array of positive integers. This operation takes no input parameters. **[R81]** If the mpmBoolStmtBindValue attribute does not have a value, then this operation **MUST** return an error.
setMPMBoolClauseBind-Value(in newValue : Integer[1..1])	This operation sets the value of the mpmBoolClauseBindValue attribute. This operation takes a single input parameter, called newValue, which defines the new value(s) for the mpmBoolClauseBindValue attribute. The newValue is an array of non-zero positive integers. **[R82]** The value of the mpmBoolClauseBindValue attribute **MUST** be a positive (non-zero) integer.
getMPMBoolClauseIsCNF() : Boolean[1..1]	This operation returns the current value of the mpmBoolClauseIsCNF attribute. This operation takes no input parameters. **[R83]** If the mpmBoolClauseIsCNF attribute does not have a value, then this operation **MUST** return an error.
setMPMBoolClauseBind-Value(in newValue : Boolean[1..1])	This operation sets the value of the mpmBoolClauseIsCNF attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmBoolClauseIsCNF attribute. **[R84]** The value of the mpmBoolClauseIsCNF attribute **MUST** be a Boolean value.

getMPMBoolClauseIsNegated() : Boolean[1..1]	This operation returns the current value of the mpmBoolClauseIsNegated attribute. This operation takes no input parameters. **[R85]** If the mpmBoolClauseIsNegated attribute does not have a value, then this operation **MUST** return an error.
setMPMBoolClauseIsNegated (in newValue : Boolean[1..1])	This operation sets the value of the mpmBoolClauseIsNegated attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmBoolClauseIsNegated attribute. **[R86]** The value of the mpmBoolClauseIsNegated attribute **MUST** be a Boolean value.

Table 29. Operations of the MPMBooleanClause Class

8.8.4.3 MPMLogicClause Class Definition

An MPMLogicClause is an abstract class that is the superclass for different types of clauses that are used in declarative policies. This type of clause is limited to MPMAssertionStatements, MPMTheorems, and MPMAxioms.

[O16] An MPMAssertionStatement **MAY** contain zero or more MPMLogicClauses.

[O17] An MPMTheorem **MAY** contain zero or more MPMLogicClauses.

[O18] An MPMAxiom **MAY** contain zero or more MPMLogicClauses.

Figure 16 shows the MPMLogicClause class and its subclasses.

Table 30 defines the attributes for this class.

Attribute Name	Description
mpmLogicClauseType : MPMFormalLogicType[1..1]	This is a mandatory enumerated non-zero integer attribute that defines the formal logic system that this particular MPMLogicClause uses. Allowed values are defined by the MPMFormalLogicType enumeration.

Table 30. Attributes of the MPMLogicClause Class

Table 31 defines the operations for this class.

Operation Name	Description
getMPMogicClauseType() : MPMFormalLogicType[1..1]	This operation returns the current value of the mpmLogicClauseType attribute. This operation takes a single input parameter, called newType, which defines the new value for the mpmLogicClauseType attribute. **[R87]** If this attribute does not have a value, then this operation **MUST** return an error.
setMPMogicClauseType(in newType : MPMFormalLogicType[1..1]	This operation sets the value of the mpmLogicClauseType attribute. This operation takes no input parameters.

Table 31. Operations of the MPMLogicClause Class

8.8.4.3.1 MPMPremiseClause Class Definition

An MPMPremiseClause is a declarative clause that is intended to justify a conclusion (represented by an MPMConclusionClause; see section 8.8.4.3.2).

Figure 16 shows the MPMPremiseClause class and its subclasses.

Table 32 defines the attributes for this class.

Attribute Name	Description
mpmPremiseIsTrue : Boolean[1..1]	This is a mandatory Boolean attribute. If the value of this attribute is TRUE, then this MPMPremiseClause has been proven TRUE.

Table 32. Attributes of the MPMPremiseClause Class

Table 33 defines the operations for this class.

Operation Name	Description
getMPMPremiseIsTrue() : Boolean[1..1]	This operation returns the current value of the mpmPremiseIsTrue attribute. This operation takes no input parameters. **[R88]** If this attribute does not have a value, then this operation **MUST** return an error.
setMPMPremiseIsTrue (in newValue : Boolean[1..1])	This operation sets the value of the mpmPremiseIsTrue attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmPremiseIsTrue attribute.

Table 33. Operations of the MPMPremiseClause Class

This class participates in a single association, called MPMConclusionDependsOn. This is described in section 8.8.4.3.2.

8.8.4.3.2 MPMConclusionClause Class Definition

An MPMConclusionClause is a declarative clause that is entailed (i.e., logically proves to be true) from its set of associated MPMPremiseClauses.

Figure 16 shows the MPMConclusionClause class and its subclasses.

Table 34 defines the attributes for this class.

Attribute Name	Description
mpmConclusionIs-True : Boolean[1..1]	This is a mandatory Boolean attribute. If the value of this attribute is TRUE, then this MPMConclusionClause has been proven TRUE.

Table 34. Attributes of the MPMConclusionClause Class

Table 35 defines the operations for this class.

Operation Name	Description
getMPMConclusionIsTrue() : Boolean[1..1]	This operation returns the current value of the mpmConclusionIsTrue attribute. This operation takes no input parameters. **[R89]** If this attribute does not have a value, then this operation **MUST** return an error.
setMPMConclusionIsTrue(in newValue : Boolean[1..1])	This operation sets the value of the mpmConclusionIsTrue attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmConclusionIsTrue attribute.

Table 35. Operations of the MPMConclusionClause Class

This class defines a single relationship, called MPMConclusionDepends on, as shown in Figure 16.

The MPMConclusionDependsOn association is an optional association, and defines the set of MPMPremiseClause objects that are attached to this particular MPMConclusionClause object. The

semantics of this association are defined by the MPMConclusionDependsOnDetail association class.

An MPMConclusionClause requires at least two MPMPremiseClause objects. In addition, all associated MPMPremiseClause objects must evaluate to TRUE in order for this particular MPMConclusionClause to be TRUE.

[R90] An MPMConclusionClause object **MUST** be associated with two or more MPMPremiseClause objects.

[R91] An MPMConclusionClause object **MUST NOT** be evaluated as TRUE unless all of its associated MPMPremiseClause objects are also TRUE.

The multiplicity of this association is 0..1 - 2..n. This means that it is an optional association (i.e., the "0" part of the 0..1 cardinality). If this association is instantiated (i.e., the "1" part of the 0..1 cardinality), then two or more MPMPremiseClause objects are associated with this particular MPMConclusionClause object. Specifically, this means that the MPMConclusionClause *depends* on the two or more MPMPremiseClause objects. The 2..* cardinality prescribes a minimum number (2) of MPMPremiseClause objects to be associated with this particular MPMConclusionClause object. The semantics of this association are defined by the MPMConclusionDependsOnDetail association class. This enables the management system to control which set of concrete subclasses of MPMPremiseClause objects can be associated with which types of MPMConclusionClause objects.

The MPMConclusionDependsOnDetail is a concrete association class, and defines the semantics of the MPMConclusionDependsOn association. The attributes and relationships of this class can be used to define which MPMPremiseClause objects can be associated with which particular set of MPMConclusionClause objects. These semantics can be further enhanced by using the Policy Pattern to define policy rules that constrain which part objects (i.e., MPMPremiseClause) are attached to which MPMConclusionClause object. Note that MCMPolicyStructure is an abstract class that is the superclass of imperative, declarative, and intent policy rules.

8.8.5 MPMPolicyComponentStructure Subclasses: MPMPolicyComponentDecorators

The Decorator Pattern [6] is a design pattern that allows behavior to dynamically be added to an object, *without affecting the behavior of other objects from the same class*. More specifically, this pattern enables *all or part of one object* to *wrap* another object. In effect, this means that the decorated object may intercept a call to the object it is wrapping, and insert attributes or execute methods before and/or after the wrapped object executes.

Hence, the decorator pattern provides a flexible alternative to subclassing for extending functionality where different behaviors are required (e.g., dependent on context). In addition, subclassing *statically* defines the characteristics and behavior of an object at compile time, whereas the decorator pattern can change the characteristics and behavior of an object at run time.

Figure 17 shows the MPMPolicyComponentDecorator class and its subclasses and relationships.

8.8.5.1 MPMPolicyComponentDecorator Class Definition

This is a mandatory class, and is used to implement the decorator pattern. This means that any concrete subclass of MPMPolicyComponentDecorator can wrap any concrete subclass of MPMPolicyStatement and/or MPMPolicyClause.

Figure 17 shows the MPMPolicyComponentDecorator class and its subclasses.

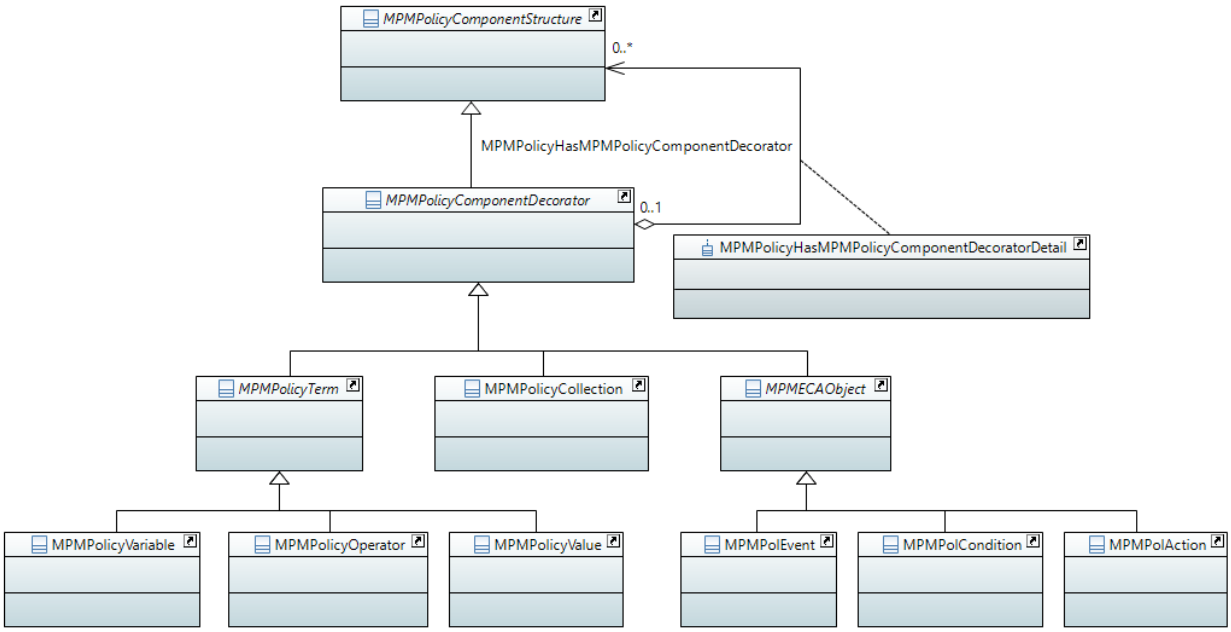


Figure 17. MPMPolicyComponentDecorator Subclasses

Figure 18 shows the attributes and operations for the MPMPolicyComponentDecorator class.

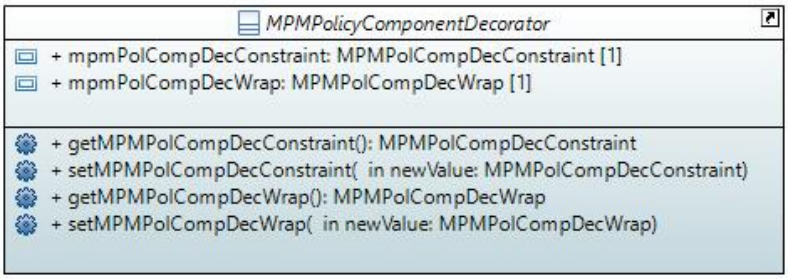


Figure 18. MPMPolicyComponentDecorator Attributes and Operations

Table 36 defines the attributes for this class.

Attribute Name	Description
mpmPolCompDecConstraint : MPMPolCompDecConstraint[1..1]	This is a mandatory non-negative enumerated integer attribute that defines the language used, if any, that this MPMPolicyComponentDecorator subclass uses to constrain object that it is wrapping. Valid values are defined by the MPMPolCompDecConstraint enumeration. **[O19]** A default value of 2 (NONE) **MAY** be defined.
mpmPolCompDecWrap : MPMPolCompDec-Wrap[1..1]	This is an optional attribute that defines if this decorated object should be wrapped before and/or after the wrapped object is executed. Valid values are defined by the MPMPolCompDecWrap enumeration

Table 36. Attributes of the MPMPolicyComponentDecorator Class

Table 37 defines the operations for this class.

Operation Name	Description
getMPMPolCompDecConstraint() : MPMPolCompDecConstraint[1..1]	This operation returns the current value of the mpmPolCompDecConstraint attribute. This operation takes no input parameters. **[R92]** If this attribute does not have a value, then this operation **MUST** return an error.

setMPMPolCompDecConstraint(in newValue : MPMPolCompDecConstraint[1..1])	This operation sets the value of the mpmPolCompDecConstraint attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmPolCompDecConstraint attribute.
getMPMPolCompDecWrap() : MPMPolCompDecWrap[1..1]	This operation returns the current value of the mpmPolCompDecValue attribute. This operation takes no input parameters. **[R93]** If this attribute does not have a value, then this operation **MUST** return an error.
setMPMPolCompDecWrap(in newValue : MPMPolCompDecWrap[1..1])	This operation sets the value of the mpmPolCompDecValue attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmPolCompDecValue attribute.

Table 37. Operations of the MPMPolicyComponentDecorator Class

8.8.5.2 MPMPolicyTerm Hierarchy

This is a mandatory abstract class that is the parent of MPMPolicy objects that can be used to define a standard way to test or set the value of a variable. It does this by defining a 3-tuple, in the form {variable, operator, value}, where each element of the 3-tuple is defined by a concrete subclass of the appropriate type (i.e., MPMPolicyVariable, MPMPolicyOperator, and MPMPolicyValue classes, respectively).

For event and condition clauses and statements, this is typically written as:
<variable> <operator> <value>.

For action clauses and statements, this is typically written as:
<operator> <variable> <value>.

Note that generic test and set expressions do not have to only use objects that are subclasses of MPMPolicyTerm. The utility of the subclasses of MPMPolicyTerm is in the ability of its subclasses to define a generic framework for implementing get and set expressions *directly from the model*. This enables a dynamic programming environment to construct get and set expressions at runtime.

Figure 19 shows the MPMPolicyTerm class and its subclasses.

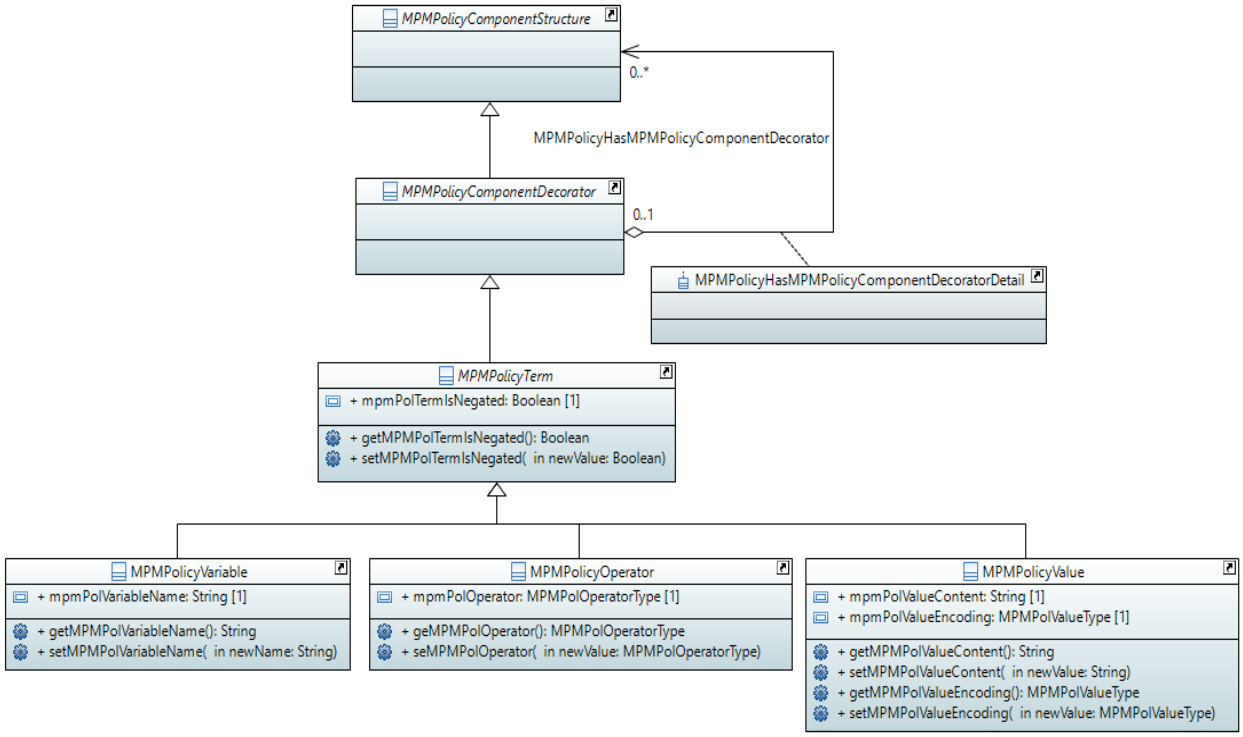


Figure 19. MPMPolicyTerm Hierarchy

Table 38 defines the attributes for this class.

Attribute Name	Description
mpmPolTermIsNegated : Boolean[1..1]	This is a mandatory Boolean attribute. If the value of this attribute is TRUE, then this (entire) MPMTerm is negated.

Table 38. Attributes of the MPMPolicyTerm Class

Table 39 defines the operations for this class.

Operation Name	Description
getMPMPolTermIsNegated() : Boolean[1..1]	This operation returns the current value of the mpmPolTermIsNegated attribute. This operation takes no input parameters. **[R94]** If the mpmPolTermIsNegated attribute does not have a value, then this operation **MUST** return an error.

setMPMPolTermIsNegated(in newValue : Boolean[1..1])	This operation sets the value of the mpmPolTermIsNegated attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmPolTermIsNegated attribute.

Table 39. Operations of the MPMPolicyTerm Class

8.8.5.2.1 MPMPolicyVariable Class Definition

This is a mandatory concrete class that defines information that forms a part of an MPMPolicyClause or MPMPolicyStatement. It specifies a concept or attribute that represents a variable, which should be compared to a value, using a particular type of operator. Since this is a subclass of the MPMPolicyComponentDecorator class, its value may be able to be changed dynamically at runtime using the decorator pattern.

[O20] The value of an MPMPolicyVariable class **MAY** be able to be changed dynamically at runtime using the decorator pattern.

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

MPMPolicyVariables are used to abstract the representation of an MPMPolicyClause (or MPMPolicyStatement) from its implementation. Some MPMPolicyVariable objects must therefore be restricted in the values and/or the data type that they may be assigned. For example, port numbers cannot be negative, and they cannot be floating-point numbers. These and other constraints may be defined in two different ways:

1. use MPMPolicyComponentDecorator attributes to constrain just that individual object
2. use the MPMPolicyClauseHasDecoratorDetail association class attributes to constrain the relationship between the concrete subclass of the MPMPolicyClause (or MPMPolicyStatement) and the concrete subclass of the MPMPolicyVariable class

Figure 19 shows the MPMPolicyVariable class and its subclasses.

Table 40 defines the attributes for this class.

Attribute Name	Description
mpmPolVariableName : String[1..1]	This is a mandatory string attribute that contains the name of this MPMPolicyVariable.

Table 40. Attributes of the MPMPolicyVariable Class

Table 41 defines the operations for this class.

Operation Name	Description
getMPMPolVariableName() : String[1..1]	This operation returns the current value of the mpmPolVariableName attribute. This operation takes no input parameters. **[R95]** If the mpmPolVariableName attribute does not have a value, then this operation **MUST** return an error.
setMPMPolVariableName(in newName : String[1..1])	This operation sets the value of the mpmPolVariableName attribute. This operation takes a single input parameter, called newName, which defines the new value for the mpmPolVariableName attribute. **[R96]** The value of the mpmPolVariableName attribute **MUST NOT** be empty or NULL.

Table 41. Operations of the MPMPolicyVariable Class

8.8.5.2.2 MPMPolicyOperator Class Definition

This is a mandatory concrete class for modeling different types of operators that are used in an MPMPolicyClause or MPMPolicyStatement.

The restriction of the type of operator used in an MPMPolicyClause or MPMPolicyStatement constrains the semantics that can be expressed in that MPMPolicyClause or MPMPolicyStatement. It is typically, but does not have to be, used with MPMPolicyVariable and MPMPolicyValue objects to form an MPMPolicyClause or MPMPolicyStatement.

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

Since this is a subclass of the MPMPolicyComponentDecorator class, its value may be able to be changed dynamically at runtime using the decorator pattern.

[O21] The value of an MPMPolicyOperator class **MAY** be able to be changed dynamically at runtime using the decorator pattern.

Figure 19 shows the MPMPolicyOperator class and its subclasses.

Table 42 defines the attributes for this class.

Attribute Name	Description
mpmPolOperator : MPMPolOperatorType[1..1]	This is a mandatory enumerated non-negative integer attribute. The allowable values of this enumeration are defined by the MPMPolOperatorType enumeration.

Table 42. Attributes of the MPMPolicyOperator Class

Table 43 defines the operations for this class.

Operation Name	Description
getMPMPolOperator() : MPMPolOperatorType[1..1]	This operation returns the current value of the mpmPolOperator attribute. This operation takes no input parameters. **[R97]** If the mpmPolOperator attribute does not have a value, then this operation **MUST** return an error.
setMPMPolOperator(in newValue : MPMPolOperatorType[1..1])	This operation sets the value of the mpmPolOperator attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmPolOperator attribute.

Table 43. Operations of the MPMPolicyOperator Class

8.8.5.2.3 MPMPolicyValue Class Definition

The MPMPolicyValue class is a mandatory concrete class for modeling different types of values and constants that occur in an MPMPolicyClause or an MPMPolicyStatement.

MPMPolicyValues objects are used to abstract the representation of an MPMPolicyClause or an MPMPolicyStatement from its implementation. Therefore, the design of the MPMPolicyValue object depends on two important factors. First, just as with MPMPolicyVariable objects, some types of MPMPolicyValue objects are restricted in the values and/or the data type that they may be assigned. Second, there is a high likelihood that specific applications will need to use their own variables that have specific meaning to a particular application.

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

1. use MPMPolicyClauseComponentDecorator attributes to constrain just that individual object

2. use the MPMPolicyClauseHasDecoratorDetail association class attributes to constrain the relationship between the concrete subclass of the MPMPolicyClause (or MPMPolicyStatement) and the concrete subclass of the MPMPolicyVariable class

The value of an MPPolicyValue object is typically compared to the value of an MPMPolicyVariable object using the type of operator defined in an MPMPolicyOperator object. However, other objects may be used instead of an MPMPolicyVariable object, and other operators may be defined in addition to those defined in the MPMPolicyOperator class.

Since this is a subclass of the MPMPolicyComponentDecorator class, its value may be able to be changed dynamically at runtime using the decorator pattern.

[O22] The value of an MPMPolicyValue class **MAY** be able to be changed dynamically at runtime using the decorator pattern.

Figure 19 shows the MPMPolicyValue class and its subclasses.

Table 44 defines the attributes for this class.

Attribute Name	Description
mpmPolValueContent: String[1..1]	This is a mandatory string attribute that defines the value of this MPMPolicyValue object. Its datatype is defined by the mpmPolValueEncoding class attribute
mpmPolValueEncoding: MPMPolValueType[1..1]	This is a mandatory enumerated non-negative integer attribute that defines the datatype of the mpmPolValueContent class attribute. The allowable values of this enumeration are defined by the MPMPolValueType enumeration.

Table 44. Attributes of the MPMPolicyValue Class

Table 45 defines the operations for this class.

Operation Name	Description
getMPMPolValueContent() : String[1..1]	This operation returns the current value of the mpmPolValueContent attribute. This operation takes no input parameters. **[R98]** If the mpmPolValueContent attribute does not have a value, then this operation **MUST** return an error.
setMPMPolValueContent(in newValue : String[1..1])	This operation sets the value of the mpmPolValueContent attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmPolValueContent attribute. **[R99]** The value of the mpmPolValueContent attribute **MUST NOT** be empty.

getMPMPolValueEncoding() : MPMPol-ValueType [1..1]	This operation returns the current value of the mpmPolValueContent attribute. This operation takes no input parameters. **[R100]** If the mpmPolValueContent attribute does not have a value, then this operation **MUST** return an error.
setMPMPolValueContent(in newValue : MPMPolValue-Type [1..1])	This operation sets the value of the mpmPolValueContent attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmPolValueContent attribute.

Table 45. Operations of the MPMPolicyValue Class

8.8.5.3 MPMECAObject Hierarchy

The MPMECAObject abstract class is used to define three concrete subclasses, one each to represent the concepts of reusable events, conditions, and actions. They are called MPMPolicyEvent, MPMPolicyCondition, and MPMPolicyAction, respectively.

Figure 20 shows the MPMECAObject class and its subclasses.

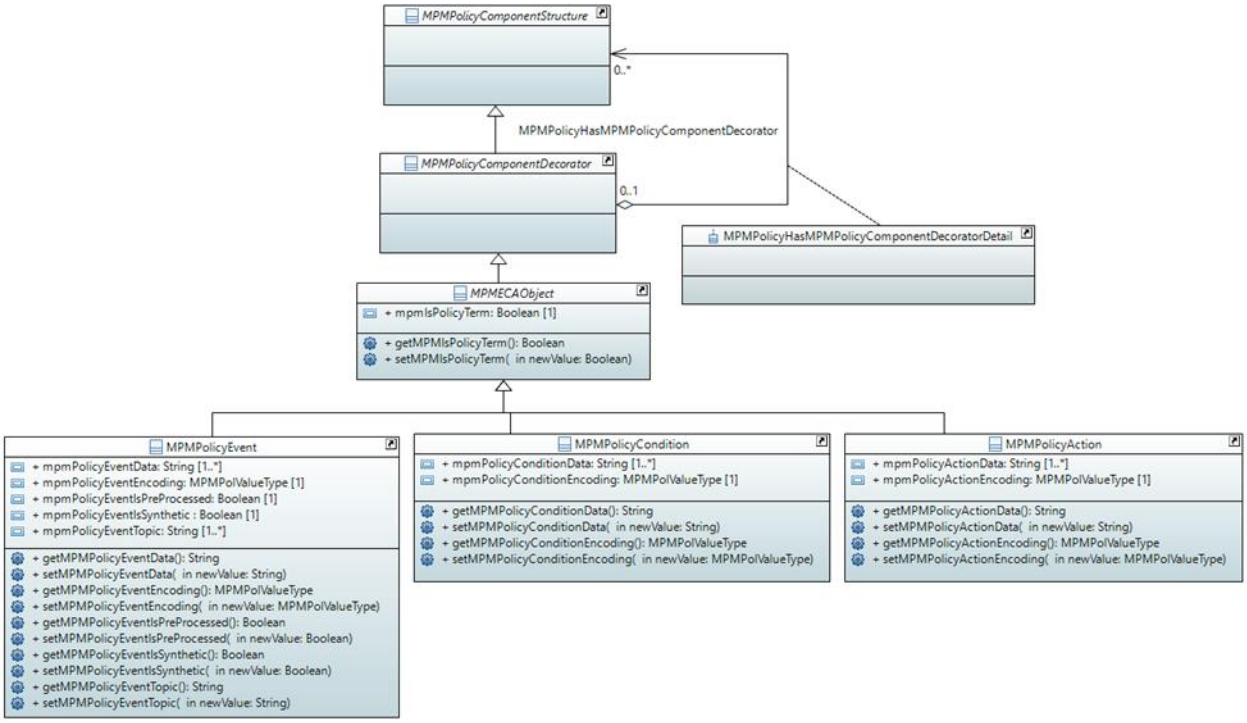


Figure 20. MPMECAObject Class and its Subclasses

MPMECAObjects provide two different ways to construct MPMPolicyClauses. The first is for the MPMECAObject to be used as either an MPMPolicyVariable or an MPM PolicyValue, and the second is for the MPMECAObject to contain the entire clause text for an MPMPolicyVariable or an MPMPolicyValue. For example, suppose it is desired to define a policy condition clause with the text “queueDepth > 10”. Two approaches could satisfy this as follows:

Approach #1 (canonical form):

MPMPolicyCondition.mpmPolicyConditionData contains the text 'queueDepth'
MPMPolicyOperator.mpmPolOpType is set to '1' (greater than)
MPMPolicyValue.mpmPolValContent is set to '10'

Approach #2 (MPMECAComponent represents the entire clause):

MPMPolicyCondition.mpmPolicyConditionData contains the text 'queueDepth > 10'

In both of the above approaches, MPMPolicyCondition.mpmPolicyConditionEncoding is set to ‘1’ (string).

The main advantage of MPMECAObjects is that they provide a machine-processable mechanism for defining MPMPolilcyClauses at runtime.

8.8.5.3.1 MPMECAObject

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

Figure 20 shows the MPMECAObject class and its subclasses.

Table 46 defines the attributes for this class.

Attribute Name	Description
mpmIsPolicyTerm : Boolean[1..1]	This is a mandatory Boolean attribute. If the value of this attribute is TRUE, then this MPMECAObject is used as the value of an MPMPolicyTerm to construct an MPMPolicyClause (this is approach #1 in section 8.8.5.3 above). If the value of this attribute is FALSE, then this MPMECAObject contains the text of the entire corresponding MPMPolicyClause (this is approach #2 in section 8.8.5.3 above).

Table 46. Attributes of the MPMECAObject Class

Table 47 defines the operations for this class.

Operation Name	Description
getMPMIsPolicyTerm() : Boolean[1..1]	This operation returns the current value of the mpmIsPolicyTerm attribute. This operation takes no input parameters. **[R101]** If the mpmIsPolicyTerm attribute does not have a value, then this operation **MUST** return an error.
getMPMIsPolicyTerm (in newValue : Boolean[1..1])	This operation sets the value of the mpmIsPolicyTerm attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmIsPolicyTerm attribute.

Table 47. Operations of the MPMECAObject Class

8.8.5.3.2 MPMPolicyEvent Class Definition

This is a mandatory concrete class that represents the concept of an Event that is applicable to a policy management system. Such an Event is defined as anything of importance to the management system (e.g., a change in the system being managed and/or its environment) occuring at a specific point in time.

It should be noted that instances of this class are not themselves events. Rather, instances of this class appear in MPMPolicyClause objects to describe what types of events the MPMPolicy is triggered by and/or uses.

MPMPolicyEvent objects can be used as part of an MPMPolicyStatement or an MPMPolicyClause object, since they are subclasses of the MPMPolicyComponentDecorator class; this means that they can wrap any concrete subclass of MPMPolicyComponentStructure, such as the concrete subclasses of MPMPolicyStatement and MPMPolicyClause.

Information from events that trigger MPMPolicies need to be made available for use in condition and action clauses, as well as in appropriate decorator objects. Application-specific subclasses (such as one for using YANG notifications as policy events) need to define how the information from the environment or event is used to trigger the evaluation of the MPMPolicyCondition subclass.

[D27] If the MPMPolicyEvent class is extended by subclassing, then that subclass **SHOULD** define how the set of events represented by the MPMPolicyEvent subclass is triggered.

Figure 20 shows the MPMPolicyEvent class and its subclasses.

Table 48 defines the attributes for this class.

Attribute Name	Description
mpmPolicyEventData : String[1..*]	This is a mandatory attribute that defines an array of strings. Each string in the array represents an attribute name and value of an Event object. The format of each string is defined as a {name:value} tuple. The 'name' part is the name of the MPMPolicyEvent attribute, and the 'value' part is the value of that attribute. For example, if the value of this attribute is: {('startTime':'08:00'), ('endTime':'17:00'), ('date':'2016-05-11'), ('timeZone':'-08:00')} then this attribute contains four properties, called startTime, endTime, date, and timeZone, whose values are 0800, 1700, May 11 2016, and Pacific Standard Time, respectively. This attribute works with another class attribute, called mpmPolicyEventEncoding, which defines how to interpret this attribute. These two attributes form a tuple, and together enable a machine to understand the syntax and value of the data carried by the object instance of this class.
mpmPolicyEvent-Encoding : PolValueType[1..1]	This is a mandatory non-zero enumerated integer attribute, and defines how to interpret the mpmPolicyEventData class attribute. These two attributes form a tuple, and together enable a machine to understand the syntax and value of the data carried by the object instance of this class. Allowed values are defined in the MPMPolValueType enumeration.
mpmPolicyEventIsPreProcessed : Boolean[1..1]	This is an optional Boolean attribute. If the value of this attribute is TRUE, then this MPMPolicyEvent has been pre-processed by an external entity, such as an Event Service Bus, before it was received by the Policy Management System.
mpmPolicyEventIsSynthetic : Boolean[1..1]	This is an optional Boolean attribute. If the value of this attribute is TRUE, then this MPMPolicyEvent has been produced by the Policy Management System. If the value of this attribute is FALSE, then this MPMPolicyEvent has been produced by an entity in the system being managed.
mpmPolicyEvent-Topic : String[1..*]	This is a mandatory array of string attributes, and contains the subject(s) that describe the nature of this PolicyEvent.

Table 48. Attributes of the MPMPolicyEvent Class

Table 49 defines the operations for this class.

Operation Name	Description
getMPMPolicyEventData() : String[1..*]	This operation returns the current value of the mpmPolicyEventData attribute, which is an array of one or more strings. This operation takes no input parameters. **[D28]** If the mpmPolicyEventData attribute does not have a value, then this operation **SHOULD** return a NULL string.
setMPMPolicyEventData(in newValue : String[1..*])	This operation sets the value of the mpmPolicyEventData attribute. This operation takes a single input parameter, called newValue, which defines an array of one or more strings for the mpmPolicyEventData attribute. **[R102]** The value of the mpmPolicyEventData attribute **MUST NOT** be an empty string.
getMPMPolicyEvent-Encoding() : MPMPolValueType[1..1]	This operation returns the current value of the mpmPolicyEvent-Encoding attribute. This operation takes no input parameters. **[R103]** If this attribute does not have a value, then this operation **MUST** return an error.
setMPMPolicyEvent-Encoding(in newValue : MPMPolValue-Type[1..1])	This operation sets the value of the mpmPolicyEventEncoding attribute. This operation takes a single input parameter, called newValue, which defines the new value for this attribute.
getMPMPolicyEventIsPreProcessed() : Boolean[1..1]	This operation returns the current value of the mpmPolicyEventIsPreProcessed attribute, which is a Boolean attribute. This operation takes no input parameters. **[R104]** If the mpmPolicyEventIsPreProcessed attribute does not have a value, then this operation **MUST** return an error.
setMPMPolicyEventIsPreProcessed(in newValue : Boolean[1..1])	This operation sets the value of the mpmPolicyEventIsPreProcessed attribute. This operation takes a single input parameter, called newValue, which defines the new value of this attribute.
getMPMPolicyEventIsSynthetic() : Boolean[1..1]	This operation returns the current value of the mpmPolicyEventIsSynthetic attribute, which is a Boolean attribute. This operation takes no input parameters.

	[R105] If the mpmPolicyEventIsSynthetic attribute does not have a value, then this operation **MUST** return an error.
setMPMPolicyEventIsSynthetic(in newValue : Boolean[1..1])	This operation sets the value of the mpmPolicyEventIsSynthetic attribute. This operation takes a single input parameter, called newValue, which defines the new value of this attribute.
getMPMPolicyEvent-Topic() : String[1..*]	This operation returns the current value of the mpmPolicyEvent-Topic attribute, which is an array of one or more strings. This operation takes no input parameters. **[R106]** If the mpmPolicyEventData attribute does not have a value, then this operation **MUST** return an error.
setMPMPolicyEvent-Topic(in newValue : String[1..*])	This operation sets the value of the mpmPolicyEventTopic attribute. This operation takes a single input parameter, called newValue, which defines an array of one or more strings for the mpmPolicyEventTopic attribute. **[R107]** The value of the mpmPolicyEventTopic attribute **MUST NOT** be an empty string.

Table 49. Operations of the MPMPolicyEvent Class

8.8.5.3.3 MPMPolicyCondition Class Definition

This is a mandatory concrete class that represents the concept of a Condition that will determine whether or not the set of Actions in this MPMPolicy should be executed or not.

MPMPolicyCondition objects can be used as part of an MPMPolicyStatement or an MPMPolicyClause object, since they are subclasses of the MPMPolicyComponentDecorator class; this means that they can wrap any concrete subclass of MPMPolicyComponentStructure, such as the concrete subclasses of MPMPolicyStatement and MPMPolicyClause.

Application-specific subclasses of this class (such as one for processing YANG) need to define how the information from the environment is used by this subclass.

[D29] If the MPMPolicyCondition class is extended by subclassing, then it **SHOULD** define how it uses information from the managed environment.

Figure 20 shows the MPMPolicyCondition class and its subclasses.

Table 50 defines the attributes for this class.

Attribute Name	Description
mpmPolicyConditionData : String[1..*]	This is a mandatory attribute that defines an array of strings. Each string in the array represents an attribute name and value of an MPMPolicyCondition object. The format of each string is defined as a {name:value} tuple. The 'name' part is the name of the MPMPolicyCondition attribute, and the 'value' part is the value of that attribute. For example, if the value of this attribute is: {(‘sourcePort’:‘8080’), (‘destPort’:‘8080’)} then this attribute contains two properties, called sourcePort and destPort, whose values are both “8080”. This attribute works with another class attribute, called mpmPolicyConditionEncoding, which defines how to interpret this attribute. These two attributes form a tuple, and together enable a machine to understand the syntax and value of the data carried by the object instance of this class.
mpmPolicyConditionEncoding : PolValueType[1..1]	This is a mandatory non-zero enumerated integer attribute, and defines how to interpret the mpmPolicyConditionData class attribute. These two attributes form a tuple, and together enable a machine to understand the syntax and value of the data carried by the object instance of this class. Allowed values are defined in the MPMPolValueType enumeration.

Table 50. Attributes of the MPMPolicyCondition Class

Table 51 defines the operations for this class.

Operation Name	Description
getMPMPolicyConditionData() : String[1..*]	This operation returns the current value of the mpmPolicyConditionData attribute. This operation takes no input parameters. **[R108]** If the mpmPolicyConditionData attribute does not have a value, then this operation **MUST** return an error.

setMPMPolicyConditionData(in newValue : String[1..*])	This operation sets the value of the mpmPolicyConditionData attribute. This operation takes a single input parameter, called newValue, which defines the new value for the mpmPolicyConditionData attribute. **[R109]** The value of the mpmPolicyConditionData attribute **MUST NOT** be an empty string.
getMPMPolicyConditionEncoding() : MPMPolValueType[1..1]	This operation returns the current value of the mpmPolicyConditionEncoding attribute. This operation takes no input parameters. **[R110]** If this attribute does not have a value, then this operation **MUST** return an error.
setMPMPolicyConditionEncoding(in newValue : MPMPolValueType[1..1])	This operation sets the value of the mpmPolicyConditionEncoding attribute. This operation takes a single input parameter, called newValue, which defines the new value for this attribute.

Table 51. Operations of the MPMPolicyCondition Class

8.8.5.3.4 MPMPolicyAction Class Definition

This is a mandatory concrete class that represents the concept of an Action, which is a part of an MPMECAPolicy. The Action may be executed when both the event and the condition clauses of its owning MPMECAPolicy evaluate to true.

MPMPolicyAction objects can be used as part of an MPMPolicyStatement or an MPMPolicyClause object, since they are subclasses of the MPMPolicyComponentDecorator class; this means that they can wrap any concrete subclass of MPMPolicyComponentStructure, such as the concrete subclasses of MPMPolicyStatement and MPMPolicyClause.

Application-specific subclasses of this class (such as one for processing YANG) need to define how the information from the environment is used by this subclass.

[D30] If the MPMPolicyAction class is extended by subclassing, then it **SHOULD** define how it uses information from the managed environment.

The execution of this action is determined by its MPMECAPolicy container, and any applicable MPMPolicyMetadata objects that are attached to that MPMECAPolicy container.

MPMPolicyAction objects can be used in three different ways:

- as part of an MPMPolicyClause (e.g., var = MPMPolicyAction.mpmPolicyActionData)
- as a standalone MPMPolicyClause (e.g., the mpmPolicyActionData attribute contains text that defines the entire action clause, and the mpmPolicyActionEncoding attribute defines the datatype of the mpmPolicyActionData attribute)

- to invoke one or more MPMPolicyActions in a different MPMECAPolicy

In the third case, note that this is NOT invoking a different MPMECAPolicy, but rather, invoking an MPMPolicyAction that is contained in a different MPMECAPolicy.

The problem with an MPMECAPolicy calling MPMECAPolicy is best illustrated with the following example:

- MPMECAPolicy A is currently executing
- MPMPolicyAction A1 executes successfully
- MPMPolicyAction A2 calls MPMECAPolicy B
- MPMPolicyAction A3 is either waiting to execute, or is executing

When MPMECAPolicy B is called, it presumably should execute under the scope of control of MPMECAPolicy A (since Policy A has not finished executing). However, calling another MPMECAPolicy means that now, the event clause of Policy B should be activated. It is very difficult to ensure that the next thing the Policy Engine does is determine if the event clause of B is satisfied or not.

Furthermore, what happens to MPMPolicyAction A3? Is MPMECAPolicy B supposed to finish execution before MPMPolicyAction A3? This requires additional logic (priorities do not work here!), which requires communication between the policy engine and both MPMECAPolicy A and MPMECAPolicy B.

Even if these problems are solved, what happens if MPMPolicyAction A3 fails, and the mpmPolExecFailStrategy has a value of 2 (i.e., if an action fails, then a rollback must be performed)? Does MPMECAPolicy B also get rolled back?

Therefore, for this version of MPM, an MPMPolicyAction can only call another MPMPolicyAction.

[R111] An MPMPolicyAction **MUST NOT** call another MPMPolicy.

[O23] An MPMPolicyAction **MAY** invoke one or more MPMPolicyActions in a different MPMECAPolicy

Figure 20 shows the MPMPolicyAction class and its subclasses.

Table 52 defines the attributes for this class.

Attribute Name	Description
mpmPolicyAction-Data : String[1..*]	This is a mandatory attribute that defines an array of strings. Each string in the array is a 2-tuple, consisting of a single character defining how this attribute is used, and the value of an attribute name of an MPMPolicyAction object. Since this attribute could represent a term in an MPMPolicyClause (e.g., var = MPMPolicyAction.mpmPolicyActionData), a complete MPMPolicyClause (e.g., the mpmPolicyActionData attribute contains text that defines the entire action clause), or the name of a MPMPolicyAction to invoke, each element in the string array is prepended with one of the following strings: o 'v:' (or 'variable:'), to denote a term in an MPMPolicyClause o 'c:' (or 'clause:'), to denote an entire MPMPolicyClause o 'a:' (or 'action:'), to invoke a MPMPolicyAction in a different MPMPolicy For example, if the value of this attribute is: {('t':'set destPort to 80'), ('a':'call PortHandlingAction')} then this attribute contains two actions. The first is the action portion of an MPMPolicyClause, and sets the variable destPort to a value of 80. The second calls the MPMPolicyAction named 'PortHandling-Action'. This attribute works with another class attribute, called mpmPolicyActionEncoding, which defines how to interpret this attribute. These two attributes form a tuple, and together enable a machine to understand the syntax and value of the data carried by the object instance of this class.
mpmPolicyAction-Encoding : MPMPolValueType[1..1]	This is a mandatory non-zero enumerated integer attribute, and defines how to interpret the mpmPolicyActionData class attribute. These two attributes form a tuple, and together enable a machine to understand the syntax and value of the data carried by the object instance of this class. Allowed values are defined in the MPMPolValueType enumeration.

Table 52. Attributes of the MPMPolicyAction Class

Table 53 defines the operations for this class.

Operation Name	Description
getMPMPolicyAction-Data() : String[1..*]	This operation returns the current value of the mpmPolicyAction-Data attribute. This operation takes no input parameters.

	[R112] If the mpmPolicyActionData attribute does not have a value, then this operation **MUST** return an error.
setMPMPolicyActionData(in newValue : String[1..*])	This operation sets the value of the mpmPolicyActionData attribute. This operation takes a single input parameter, called newValue, which defines the new value for this attribute. **[R113]** The value of the mpmPolicyActionData attribute **MUST NOT** be an empty string.
getMPMPolicyActionEncoding() : MPMPolValueType[1..1]	This operation returns the current value of the mpmPolicyActionEncoding attribute. This operation takes no input parameters. **[R114]** If this attribute does not have a value, then this operation **MUST** return an error.
setMPMPolicyActionEncoding(in newValue : MPMPolValueType[1..1])	This operation sets the value of the mpmPolicyActionEncoding attribute. This operation takes a single input parameter, called newValue, which defines the new value for this attribute.

Table 53. Operations of the MPMPolicyAction Class

8.8.5.4 MPMPolicyCollection

An MPMPolicyCollection is an optional concrete class that enables a collection (e.g., set, bag, or other, more complex, collections of elements) of arbitrary objects to be defined and used as part of an MPMPolicyClause.

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

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

Therefore, this standard defines the concept of a collection of entities, called an MPMPolicyCollection. Conceptually, the items to be collected (e.g., events or conditions) are aggregated in one or more MPMPolicyCollection objects of the appropriate type.

Another example is for an MPMPolicyCollection object to aggregate logic blocks (including MPMDeclarativePolicies) to execute.

The computation(s) represented by an MPMPolicyCollection may be part of a larger MPMPolicyClause, since MPMPolicyCollection is a subclass of MPMPolicyComponentDecorator, and can be used to decorate an MPMPolicyClause.

Figure 21 shows the attributes and operations of the MPMPolicyCollection class.

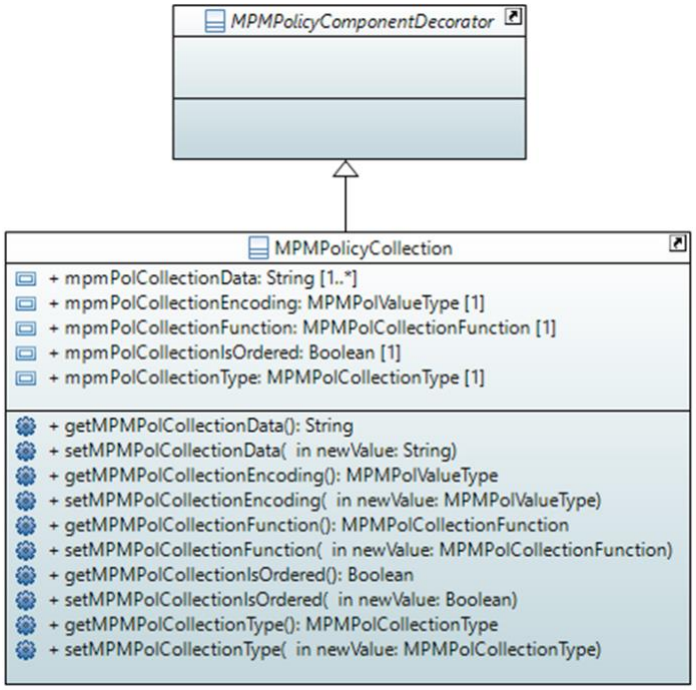


Figure 21. MPMPolicyCollection

Table 54 defines the attributes for this class.

Attribute Name	Description
mpmPolCollection-Data : String[1..*]	This is a mandatory attribute that defines an array of strings. Each string in the array defines a domain-specific identifier of an object that is collected by this object instance. This attribute works with another class attribute, called mpmPolicyCollectionEncoding, which defines how to interpret this attribute. These two attributes form a tuple, and together enable a machine to understand the syntax and value of the data carried by the object instance of this class.
mpmPolCollection-Encoding : MPMPolValueType[1..1]	This is a mandatory non-zero enumerated integer attribute, and defines how to interpret the mpmPolCollectionData class attribute. These two attributes form a tuple, and together enable a machine to understand the syntax and value of the data carried by the object instance of this class. Allowed values are defined in the MPMPolValueType enumeration.
mpmPolCollection-Function : MPMPolCollectionFunction[1..1]	This is a mandatory non-zero enumerated integer attribute, and defines how this collection is used (e.g., is it a collection of objects for an event, or for logic processing, or other functions). Allowed values are defined in the MPMPolCollectionFunction enumeration.
mpmPolCollectionIsOrdered : Boolean[1..1]	This is a mandatory Boolean attribute. If the value of this attribute is TRUE, then all elements in this instance of this MPMPolicyCollection object are ordered.
mpmPolCollectionType : MPMPolCollectionType[1..1]	This is a mandatory non-zero enumerated integer attribute, and defines the type of collection that this object instance is. Allowed values are defined in the MPMPolCollectionType enumeration.

Table 54. Attributes of the MPMPolicyCollection Class

Table 55 defines the operations for this class.

Operation Name	Description
getMPMPolCollectionData() : String[1..*]	This operation returns the current value of the mpmPolCollectionData attribute. This operation takes no input parameters. **[R115]** If the mpmPolCollectionData attribute does not have a value, then this operation **MUST** return an error.

setMPMPolCollectionData(in newValue : String[1..*])	This operation sets the value of the mpmPolCollectionData attribute. This operation takes a single input parameter, called newValue, which is an array of strings that defines the new value for the mpmPolCollectionData attribute. **[R116]** The value of the mpmPolCollectionData attribute **MUST NOT** be an empty string.
getMPMPolicyCollectionEncoding() : MPMPolValueType[1..1]	This operation returns the current value of the mpmPolicyCollectionEncoding attribute. This operation takes no input parameters. **[R117]** If this attribute does not have a value, then this operation **MUST** return an error.
setMPMPolicyCollectionEncoding(in newValue : MPMPol-ValueType[1..1])	This operation sets the value of the mpmPolicyCollectionEncoding attribute. This operation takes a single input parameter, called newValue, which defines the new value for this attribute.
getMPMPolCollection-Function() : MPMPol-CollectionFunction[1..1]	This operation returns the current value of the mpmPolicyCollectionFunction attribute. This operation takes no input parameters. **[R118]** If this attribute does not have a value, then this operation **MUST** return an error.
setMPMPolCollection-Function(in newValue : MPMPolCollection-Function [1..1])	This operation sets the value of the mpmPolicyCollectionFunction attribute. This operation takes a single input parameter, called newValue, which defines the new value for this attribute.
getMPMPolCollectionIsOrdered() : Boolean[1..1]	This operation returns the current value of the mpmPolicyCollectionIsOrdered attribute. This operation takes no input parameters. **[R119]** If this attribute does not have a value, then this operation **MUST** return an error.
setMPMPolCollectionIsOrdered (in newValue : Boolean[1..1])	This operation sets the value of the mpmPolicyCollectionIsOrdered attribute. This operation takes a single input parameter, called newValue, which defines the new value for this attribute.
getMPMPolCollectionType() : MPMPolCollectionType [1..1]	This operation returns the current value of the mpmPolicyCollectionType attribute. This operation takes no input parameters.

	[R120] If this attribute does not have a value, then this operation **MUST** return an error.
setMPMPolCollectionType(in newValue : MPMPolCollectionType[1..1])	This operation sets the value of the mpmPolicyCollectionType attribute. This operation takes a single input parameter, called newValue, which defines the new value for this attribute.

Table 55. Operations of the MPMPolicyCollection Class

8.9 MPMPolicySource

This is an optional class that defines a set of MCMManagedEntity objects that authored, or are otherwise responsible for, this MPMPolicy. Note that an MPMPolicySource does NOT evaluate or execute MPMPolicies. Its primary use is for auditability and the implementation of deontic and/or alethic logic.

It is recommended that an MPMPolicySource object is mapped to a role or set of roles (e.g., using the role-object pattern). This enables role-based or policy-based access control to be used to restrict which MCMManagedEntity objects can author a given policy.

[D31] An MPMPolicySource object **SHOULD** be mapped to a subclass of MCMPolicyRole.

Figure 22 shows the MPMPolicySource and MPMPolicyTarget classes.

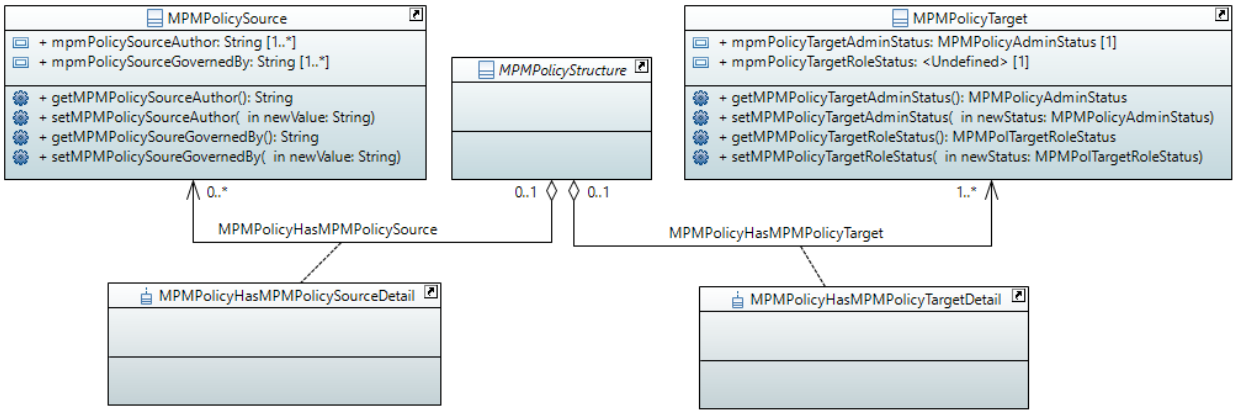


Figure 22. PolicySource and PolicyTarget

The purpose of the MPMPolicySource object is to provide a mechanism for business logic to be inserted into the model to manipulate the objects that serve as the authors or are responsible for this MPMPolicy.

Table 56 defines the attributes for this class.

Attribute Name	Description
mpmPolicy-SourceAuthor : String[1..*]	This is an optional attribute that defines an array of strings. Each string in the array is a 2-tuple, consisting of a single character defining the type of object that contains the Author, and the name of an MCMParty or MCMPartyRole object class. The mapping of the first character is defined as follow: ‘o’: MCMOrganization ‘p’: MCMPerson ‘r’: MCMPartyRole For example, if this attribute contains the following values: {(‘o’:‘CustomerSupport’), (‘r’:‘CSRole’)} then the first 2-tuple identifies an MCMOrganization named ‘CustomerSupport’, and the second identifies an MCMPartyRole named ‘CSRole’. **[R121]** This attribute **MUST NOT** contain a prefix (i.e., a character before the quote) of more than 1 character. **[R122]** This attribute **MUST NOT** contain a NULL or empty string.
mpmPolicySourceGovernedBy : String[1..*]	This is an optional attribute that defines an array of strings. Each string in the array is a 2-tuple, consisting of a single character defining the type of object that governs this MPMPolicy, and the name of an MCMParty or MCMPartyRole object class. The mapping of the first character is defined as follow: ‘o’: MCMOrganization ‘p’: MCMPerson ‘r’: MCMPartyRole For example, if this attribute contains the following values: {(‘o’:‘CustomerSupport’), (‘r’:‘CSRole’)} then the first 2-tuple identifies an MCMOrganization named ‘CustomerSupport’, and the second identifies an MCMPartyRole named ‘CSRole’. **[R123]** This attribute **MUST NOT** contain a prefix (i.e., a character before the quote) of more than 1 character. **[R124]** This attribute **MUST NOT** contain a NULL or empty string.

Table 56. Attributes of the MPMPolicySource Class

Table 57 defines the operations for this class.

Operation Name	Description
getMPMPolicy-SourceAuthor() : String[1..*]	This operation returns the current value of the mpmPolicySource-Author attribute. This operation takes no input parameters. **[R125]** If the mpmPolicySourceAuthor attribute does not have a value, then this operation **MUST** return an error. **[R126]** This attribute **MUST** be a two-tuple.
setMPMPolicy-SourceAuthor(in newValue : String[1..*])	This operation sets the value of the mpmPolicySourceAuthor attribute. This operation takes a single input parameter, called newValue, which defines the new value for this attribute. The newValue attribute is an array of 2-tuples. **[R127]** The value of the mpmPolicySourceAuthor attribute **MUST NOT** be a NULL or empty string.
getMPMPolicySourceGovernedBy() : String[1..*]	This operation returns the current value of the mpmPolicySourceGovernedBy attribute. This operation takes no input parameters. **[R128]** If this attribute does not have a value, then this operation **MUST** return an error. **[R129]** This attribute **MUST** be a two-tuple.
setMPMPolicySourceGovernedBy(in newValue : String[1..*])	This operation sets the value of the mpmPolicySourceGovernedBy attribute. This operation takes a single input parameter, called newValue, which defines the new value for this attribute. The newValue attribute is an array of 2-tuples. **[R130]** The value of the mpmPolicySourceGovernedBy attribute **MUST NOT** be an empty string.

Table 57. Operations of the MPMPolicySource Class

8.10 MPMPolicyTarget

This is a mandatory class that defines a set of MCMManagedEntity objects that an MPMPolicy is applied to.

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

[R131] If a proposed MPMPolicyTarget object is in a state that enables changes to be made to it, and if it can process those changes, it **MUST** have its mpmPolicyTargetEnabled Boolean attribute set to a value of TRUE.

It is recommended that an MPMPolicyTarget object is mapped to a role or set of roles (e.g., using the role-object pattern). This enables role-based or policy-based access control to be used to restrict which MCMManagedEntity objects can be affected by a given policy.

[D32] An MPMPolicyTarget object **SHOULD** be mapped to a subclass of MCMPolicyRole.

Figure 22 shows the MPMPolicySource and MPMPolicyTarget classes.

Table 58 defines the attributes for this class.

Attribute Name	Description
mpmPolicyTargetAdminStatus : MPMPolicyAdminStatus[1..1]	This is a mandatory enumerated non-negative integer attribute that defines the current administrative status of this particular MPMPolicyTarget object. The allowable values of this enumeration are defined by the MPMPolicyAdminStatus enumeration.
mpmPolicyTarget-RoleStatus : MPMPolTargetRoleStatus[1..1]	This is a mandatory enumerated non-negative integer attribute that defines the current readiness of this particular MPMPolicyTarget object to take on the PolicyTargetRole. The allowable values of this enumeration are defined by the MPMPolicyTargetRoleStatus enumeration.

Table 58. Attributes of the MPMPolicyTarget Class

Table 59 defines the operations for this class.

Operation Name	Description
getMPMPolicyTargetAdminStatus() : MPMPolicyAdminStatus[1..1]	This operation returns the current administrative status of this particular MPMPolicyTarget object, which is defined by the MPMPolicyAdminStatus enumeration. This operation takes no input parameters. **[R132]** If the mpmPolicyTargetAdminStatus attribute does not have a value, then this operation **MUST** return an error.
setMPMPolicyTargetAdminStatus(in newStatus : MPMPolicyAdminStatus[1..1])	This operation sets the value of the current administrative status of this particular MPMPolicyTarget object. This operation takes a single input parameter, called newStatus, which defines the new value for the mpmPolAdminStatus attribute. The allowable values of this input parameter are defined by the MPMPolicyAdminStatus enumeration.
getMPMPolicyTargetRole-Status() : MPMPolicy-RoleStatus[1..1]	This operation returns the current readiness of this particular MPMPolicyTarget object to play the role of a PolicyTarget. This operation takes no input parameters. **[R133]** If the mpmPolicyTargetRoleStatus attribute does not have a value, then this operation **MUST** return an error.
setMPMPolicyTargetRole-Status(in newStatus : MPMPolicyRoleStatus[1..1])	This operation sets the value of the mpmPolicyTarget-RoleStatus attribute of this particular MPMPolicyTarget object. This operation takes a single input parameter, called newStatus, which defines the new value for the mpmPolicyTargetRoleStatus attribute. The allowable values of this input parameter are defined by the MPMPolicyRoleStatus enumeration.

Table 59. Operations of the MPMPolicyTarget Class

9 MPM Datatypes and Enumerations

This section will list the common data types and enumerations defined for the MPM model only.

9.1 Introduction

The MEF Types additions for the MPM project are defined in a folder labelled "MPM" in the MEF_Types GitHub project. There are currently eight enumerations and one datatype defined.

9.2 MPM Enumerations

The following Enumerations are defined in the current MPM Model.

9.2.1 MPMPolicyAdminStatus

This is an Enumeration that defines the current administrative status of this MPMPolicy object. It is used by multiple MPMPolicyObjects to define a consistent set of definitions for administrative status for each MPMPolicy object that uses it. It is defined in Table 60.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R134]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R135]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	ENABLED	This literal means that this MPMPolicy object is ENABLED and can be used operationally.
3	DISABLED	This literal means that this MPMPolicy object is DISABLED and MUST NOT be used operationally. **[R136]** This means that this MPMPolicy object **MUST NOT** be used operationally.
4	IN_TEST	This literal indicates that MPMPolicy object is in a special test mode. **[D33]** This means that this MPMPolicy object **SHOULD NOT** be used operationally.

Table 60. MPMPolicyAdminStatus Enumeration Definition

9.2.2 MPMPolContinuumLevel

This is an Enumeration that defines the current Policy Continuum Level (i.e., level of abstraction) of this particular MPMPolicy. It is defined in Table 61.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R137]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R138]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	BUSINESS	This literal indicates that the Policy Continuum Level for this MPMPolicy is the BUSINESS level. This means that this MPMPolicy is targeted at end-users, Customers, and business people (e.g., Product Managers or Business Analysts). This group of users typically does NOT have familiarity with application development or networking details.
3	APP_DEVELOPER	This literal indicates that the Policy Continuum Level for this MPMPolicy is the APPLICATION DEVELOPER level. This means that this MPMPolicy is targeted at Application Developers. This group of users typically does NOT have familiarity with business or networking details.
4	NETWORK_ ADMIN	This literal indicates that the Policy Continuum Level for this MPMPolicy is the NETWORK ADMINISTRATOR level. This means that this MPMPolicy is for network administrators (e.g., people technically proficient at configuring and managing networks). This group of users typically does NOT have familiarity with business or application development details.
5	CUSTOM	This literal indicates that the Policy Continuum Level for this MPMPolicy is a CUSTOM level. This means that this MPMPolicy is NOT targeted at business, application development, or network administrator constituencies as defined in the above literals.

Table 61. MPMPolContinuumLevel Enumeration Definition

9.2.3 MPMPolicyDeployStatus

This is an Enumeration that defines whether this MPMPolicy can currently be deployed or not by the Policy Management System. This enables the policy manager to know which MPMPolicies are currently deployed or not, and may be useful for the policy execution system for planning the staging of MPMPolicies. It is defined in Table 62.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R139]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R140]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	DEPLOYED	This literal means that this MPMPolicy object has been deployed. The mpmPolAdminStatus class attribute defines whether this MPMPolicy object is enabled, in test, or disabled, and in conjunction with this attribute, determines whether this MPMPolicy can be operationally used or not. **[R141]** An MPMPolicy **MUST NOT** be used operationally unless the value of this enumeration is 2 and the value of the MPMAdminStatus enumeration is also 2.
3	READY_TO_BE_DEPLOYED	This literal indicates that this MPMPolicy object is ready to be deployed. **[R142]** This means that this MPMPolicy object **MUST NOT** be used operationally.
4	NOT_DEPLOYED	This literal indicates that this MPMPolicy object has NOT been deployed. **[R143]** This means that this MPMPolicy object **MUST NOT** be used operationally.

Table 62. MPMPolicyDeployStatus Enumeration Definition

9.2.4 MPMPolicyDesignStatus

This is an enumeration that defines the design status of this MPMPolicyStructure. It is defined in Table 63.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R144]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R145]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	DESIGN_FINISHED	This literal means that the design of this MPMPolicyRule has been completed.
3	DESIGN_BEING_MODIFIED	This literal means that the design of this MPMPolicyRule was complete, but is currently being modified.
4	DESIGN_ IN_PROCESS	This literal means that the design of this MPMPolicyRule is in process and has NOT been completed.
5	DESIGN_NOT_ STARTED	This literal means that the design of this MPMPolicyRule has NOT started.

Table 63. MPMPolicyDesignStatus Enumeration Definition

9.2.5 MPMPolicyExecStatus

This is an Enumeration that defines the current execution status of this MPMPolicyStructure. It is defined in Table 64.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R146]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R147]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	EXECUTED_ WITHOUT_ ERRORS	This literal indicates that this MPMPolicy object has completed execution without any errors.

3	EXECUTING_IN_PROCESS	This literal indicates that this MPMPolicy object is currently executing, but has not yet finished.
4	EXECUTION_ABORTED	This literal indicates that this MPMPolicy object did not complete execution; one or more errors occured, and execution was aborted.
5	EXECUTION_FAILED	This literal indicates that this MPMPolicy object did not complete execution due to a failure that stopped execution.
6	EXECUTION_CONFLICT_UNRESOLVED	This literal indicates that this MPMPolicy object encountered a conflict during runtime execution, but no corrective action has been taken to remedy this conflict.
7	EXECUTION_CONFLICT_ROLLBACK	This literal indicates that this MPMPolicy object encountered a conflict during runtime execution, and that a rollback of its actions was successfully completed.
8	EXECUTION_CONFLICT_ERROR	This literal indicates that this MPMPolicy object encountered a conflict during runtime execution, and that execution had to be aborted.
9	EXECUTION_TIMEOUT	This literal indicates that this MPMPolicy object has exceeded the allowed time to execute, and was aborted.
10	NOT_EXECUTED	This literal indicates that this MPMPolicy object has not yet ben executed.

Table 64. MPMPolicyExecStatus Enumeration Definition

9.2.6 MPMPolExecFailStrategy

This is an Enumeration that defines what actions, if any, should be taken by this MPMPolicy if it fails to execute correctly. It is defined in Table 65.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R148]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R149]** This means that this MPMPolicy object **MUST NOT** be used operationally.

2	ROLLBACK_ ALL	This literal indicates that rollback of all actions taken should be attempted. After that, execution MUST be attempted to be stopped. **[R150]** Execution **MUST** be attempted to be stopped after the rollback has finished.
3	ROLLBACK_ SINGLE	This literal indicates that rollback of only the action that failed should be attempted. After that, execution MUST be attempted to be stopped. **[R151]** Execution **MUST** be attempted to be stopped after the rollback has finished.
4	STOP	This literal indicates that execution MUST be attempted to be stopped WITHOUT trying to rollback any actions that failed. **[R152]** Execution **MUST** be attempted to be stopped as soon as an error has been detected.
5	IGNORE	This literal indicates that the failure of any action SHOULD be ignored WITHOUT trying to rollback any actions that failed. **[D34]** Execution **SHOULD** continue even though one or more errors occurred.

Table 65. MPMPolExecFailStrategy Enumeration Definition

9.2.7 MPMImpPolExecStrategy

This is an Enumeration that defines the current execution strategy of this MPMPolicyStructure. The execution strategy consists of the order that actions will execute, and whether encountering an error terminates the process of executing actions or not. It is defined in Table 66.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R153]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R154]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	EXECUTE_ FIRST_AND_STOP	This literal indicates that only the first Action should be executed. The mpmPolExecStatus attribute is then populated with the result.

3	EXECUTE_ LAST_AND_STOP	This literal indicates that only the last Action should be executed. The mpmPolExecStatus attribute is then populated with the result.
4	EXECUTE_HIGH_ PRIORITY_AND_ STOP	This literal indicates that only the highest priority Action should be executed. The mpmPolExecStatus attribute is then populated with the result. **[D35]** If there is no action with a priority greater than 0, then ALL Actions **SHOULD** be executed. **[R155]** If multiple Actions all have the same highest priority, then those Actions **MUST** all be executed.
5	EXECUTE_ALL_ IN_PRIORITY_ ORDER	This literal indicates that all Actions will be executed in priority order. The mpmPolExecStatus attribute is then populated with the result. **[D36]** If an error occurs, execution **SHOULD** continue. **[R156]** If multiple Actions all have the same priority, then those Actions **MUST** all be executed in priority order.
6	EXECUTE_ALL_ IN_ORDER_ UNTIL_ERROR	This literal indicates that all Actions will be executed in priority order. The mpmPolExecStatus attribute is then populated with the result. **[R157]** If an error occurs, execution **MUST** terminate. **[R158]** If multiple Actions all have the same priority, then those Actions **MUST** all be executed in priority order.
7	EXECUTE_AS_IS	This literal indicates that Action will be executed in the order that they are contained, without regard to priority. The mpmPolExecStatus attribute is then populated with the result. **[R159]** If an error occurs, execution **MUST** terminate.

Table 66. MPMImpPolExecStrategy Enumeration Definition

9.2.8 MPMPolCollectionType

This is an Enumeration that defines the type of collection that this MPMPolicyCollection uses.It is defined in Table 67.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R160]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R161]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	SET	This literal defines the datatype of this MPMPolicyCollection object to be a set. **[R162]** A set is an unordered collection of elements that **MUST NOT** have duplicates.
3	BAG	This literal defines the datatype of this MPMPolicyCollection object to be a bag. **[O24]** A bag is an unordered collection of elements that **MAY** have duplicates.
4	DICTIONARY	This literal defines the datatype of this MPMPolicyCollection object to be a dictionary. A dictionary is a table that associates a key with a value. **[R163]** A dictionary **MUST NOT** have duplicate, null, or empty keys.

Table 67. MPMPolCollectionType Enumeration Definition

9.2.9 MPMPolCollectionFunction

This is an Enumeration that defines how the objects in this MPMPolicyCollectionFunction are used in a particular MPMPolicyStatement. It is defined in Table 68.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R164]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R165]** This means that this MPMPolicy object **MUST NOT** be used operationally.

2	EVENT_ COLLECTION	This literal defines a collection of objects that are to be used to populate an MPMPolicyStatement that is used in the event clause for an MPMImperativePolicy.
3	CONDITION_ COLLECTION	This literal defines a collection of objects that are to be used to populate an MPMPolicyStatement that is used in the condition clause for an MPMImperativePolicy.
4	ACTION_ COLLECTION	This literal defines a collection of objects that are to be used to populate an MPMPolicyStatement that is used in the action clause for an MPMImperativePolicy.
5	LOGIC_ COLLECTION	This literal defines a collection of objects that are to be used to populate an MPMPolicyStatement that is used in an MPMDeclarativePolicy.
6	INTENT_ COLLECTION	This literal defines a collection of objects that are to be used to populate an MPMPolicyStatement that is used in an MPMIntentPolicy.

Table 68. MPMPolCollectionFunction Enumeration Definition

9.2.10 MPMPolStmtConstrainMechanism

This is an Enumeration that defines the type of constraint mechanism used between a concrete subclass of MPMPolicyStructure and a concrete subclass of MPMPolicyStatement. It is defined in Table 69.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R166]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R167]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	NONE	This literal indicates that NO mechanism is used to constrain which subclasses of MPMPolicyStatement can be used with this particular subclass of MPMPolicyStructure.
3	ASSOCIATION_ CLASS	This literal indicates that the MPMPolicyHasMPMPolicy-StatementDetail association class is used to constrain which subclasses of MPMPolicyStatement can be used with this particular subclass of MPMPolicyStructure.

4	OCL	This literal indicates that OCL is used to constrain which subclasses of MPMPolicyStatement can be used with this particular subclass of MPMPolicyStructure. The current version of OCL is 2.4.
5	ALLOY	This literal indicates that the Alloy language is used to constrain which subclasses of MPMPolicyStatement can be used with this particular subclass of MPMPolicyStructure. Alloy is a declarative language that can be used to specify the structure of a system textually.
6	FOML	This literal indicates that the FOML language is used to constrain which subclasses of MPMPolicyStatement can be used with this particular subclass of MPMPolicyStructure. FOML is a logic rule language that supports object modeling, analysis, and inference.

Table 69. MPMPolStmtConstrainMechanism Enumeration Definition

9.2.11 MPMAssertionStatementType

This is an Enumeration that defines the set of MPMAssertionClauses that make up this particular MPMAssertionStatement. It is defined in Table 70.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R168]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R169]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	CONTRACT	This literal indicates that this MPMAssertionStatement is made up of a standard set of MPMAssertionClauses (i.e., pre-condition, post-condition, and invariant). This forms a software contract.
3	PRE_AND_POST	This literal indicates that this MPMAssertionStatement contains only pre- and post-condition MPMAssertionClauses.
4	PRE_AND_ INVAR	This literal indicates that this MPMAssertionStatement contains only pre-condition and Invariant MPMAssertionClauses.

5	POST_AND_ INVAR	This literal indicates that this MPMAssertionStatement contains only post-condition and Invariant MPMAssertionClauses.
6	PRE_ONLY	This literal indicates that this MPMAssertionStatement contains only pre-condition MPMAssertionClauses.
7	POST_ONLY	This literal indicates that this MPMAssertionStatement contains only post-condition MPMAssertionClauses.
8	INVAR_ONLY	This literal indicates that this MPMAssertionStatement contains only Invariant MPMAssertionClauses.
9	OTHER	This literal indicates that this MPMAssertionStatement is made up of a non-standard set of MPMAssertionClauses.

Table 70. MPMAssertionStatementType Enumeration Definition

9.2.12 MPMPolStmtConflictStatus

This is an Enumeration that defines the types of conflicts that this particular MPMPolicyStatement has or has had. It is defined in Table 71.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R170]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R171]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	NONE	This literal indicates that this MPMPolicyStatement has never had a conflict.
3	RESOLVED	This literal indicates that this MPMPolicyStatement has had one or more conflicts in the past, but all have been resolved.
4	CONFLICT	This literal indicates that this MPMPolicyStatement currently has a conflict that has not been resolved.
5	UNABLE_TO_ RESOLVE	This literal indicates that this MPMPolicyStatement has a conflict that was unable to be resolved.

Table 71. MPMPolStmtConflictStatus Enumeration Definition

9.2.13 MPMFormalLogicType

This is an Enumeration that defines the type of logic theory used. It is defined in Table 72.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R172]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R173]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	PROPOSITIONAL_LOGIC	This literal defines the type of logic used as propositional logic. This defines the manipulation of a set of propositions, possibly with logical connectives, to prove or disprove a conclusion. Propositional logic does not deal with logical relationships and properties that involve the parts of a statement smaller than the statement itself.
3	MODAL_LOGIC	This literal defines the type of logic used as modal logic. This defines logic theories that deal with necessity and possibility.
4	DEONTIC_ LOGIC	This literal defines the type of logic used as deontic logic. This defines logic theories that deal with obligation, permission, and related concepts.
5	ALETHIC_ LOGIC	This literal defines the type of logic used as alethic logic. This defines logic theories that deal with logical necessity, possibility, or impossibility.
6	EPISTEMIC_ LOGIC	This literal defines the type of logic used as epistemic logic. This defines logic theories that deal with reasoning about knowledge.
7	DOXASTIC_ LOGIC	This literal defines the type of logic used as doxastic logic. This defines logic theories that deal with reasoning about beliefs.
8	TEMPORAL_ LOGIC	This literal defines the type of logic used as temporal logic.

		This defines logic theories that deal with propositions qualified in terms of time.
9	DESCRIPTION_ LOGIC	This literal defines the type of logic used as description logic. This defines families of logic theories that are subsets of first-order logic. They are typically decidable.
10	FIRST_ORDER_ LOGIC	This literal defines the type of logic used as first order logic. This defines logic theories that are an extension of propositional logic to include predicates and quantification.
11	HIGHER_ ORDER_LOGIC	This literal defines the type of logic used as second and higher order logic. This defines an extension of first order logic to include quantification over relations.
12	FUZZY_LOGIC	This literal defines the type of logic used as fuzzy logic. This defines a family of many-values logic theories in which the truth values may be any real number between 0 and 1 inclusive. This models partial truths, or confidence in something being true.

Table 72. MPMFormalLogicType Enumeration Definition

9.2.14 MPMIntentTranslationStatus

This is an Enumeration that defines the current status of translating an MPMIntentPolicy. It is defined in Table 73.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R174]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R175]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	SUCCESS	This literal indicates that the translation of the MPMIntentPolicy content was successful. **[O25]** This means that this MPMPolicy object **MAY** be used operationally.

3	IN_PROGRESS	This literal indicates that the translation of the MPMIntentPolicy content is currently being done. **[R176]** This means that this MPMPolicy object **MUST NOT** be used operationally.
4	IN_TEST	This literal indicates that the translation of the MPMIntentPolicy content is in a special test mode, and is currently being tested. **[D37]** This means that this MPMPolicy object **SHOULD NOT** be used operationally.
5	FAILED	This literal indicates that the translation of the MPMIntentPolicy content has failed. **[R177]** This means that this MPMPolicy object **MUST NOT** be used operationally.

Table 73. MPMIntentTranslationStatus Enumeration Definition

9.2.15 MPMPolCompDecConstraint

This is an Enumeration that defines the language used, if any, that this MPMPolicyComponentDecorator subclass uses to constrain objects that it is wrapping. It is defined in Table 74.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R178]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R179]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	NONE	This literal means that NO CONSTRAINTS are applied by this MPMPolicyComponentDecorator subclass.
3	OCL2.4+	This literal indicates that the decorated object is constrained using OCL version 2.4 (the current version as of this standard) and higher.
4	OCL2.3-	This literal indicates that the decorated object is constrained using OCL version 2.0 - 2.3.
5	OCL1.x	This literal indicates that the decorated object is constrained using OCL version 1.x.

6	QVT_REL	This literal indicates that the decorated object is constrained using the QVT Relations Language.
7	QVT_OP	This literal indicates that the decorated object is constrained using the QVT Operational Language.
8	ALLOY	This literal indicates that the decorated object is constrained using the Alloy Language. Alloy is a language for describing constraints, and uses a SAT solver to guarantee correctness.
9	ASCII	This literal indicates that the decorated object is constrained using ASCII text (as instructons). This enumeration is NOT recommended (since it is informal, and hence, not verifiable), but is included for completeness. **[D38]** This enumeration **SHOULD NOT** be used.

Table 74. MPMPolCompDecConstraint Enumeration Definition

9.2.16 MPMPolCompDecWrap

This is an Enumeration that defines if this decorated object should be wrapped before and/or after the wrapped object is executed. It is defined in Table 75.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R180]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R181]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	BEFORE	This literal indicates that this decorated object should be applied BEFORE the wrapped object is executed.
3	AFTER	This literal indicates that this decorated object should be applied AFTER the wrapped object is executed.
4	BOTH	This literal indicates that this decorated object should be applied BEFORE AND AFTER the wrapped object is executed.

Table 75. MPMPolCompDecWrap Enumeration Definition

9.2.17 MPMPolTargetRoleStatus

This is an Enumeration that defines the current readiness of this particular MPMPolicyTarget object to take on the PolicyTargetRole. It is defined in Table 76.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R182]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R183]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	READY	This literal indicates that this MPMPolicyTarget is able to take on the role of being a PolicyTarget.
3	NOT_READY	This literal indicates that this MPMPolicyTarget is NOT able to take on the role of being a PolicyTarget.
4	IN_PROGRESS	This literal indicates that this MPMPolicyTarget is currently preparing itself to be able to take on the role of being a PolicyTarget.
5	IN_TEST	This literal indicates that this MPMPolicyTarget is currently in a special test mode. **[D39]** This MPMPolicyTarget **SHOULD NOT** be used operationally.

Table 76. MPMPolTargetRoleStatus Enumeration Definition

9.2.18 MPMPolOperatorType

This is an Enumeration that defines the type of MPMPolicyOperator that this current object instance is defined as. It is defined in Table 77.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R184]** This means that this MPMPolicy object **MUST NOT** be used operationally.

1	INIT	This literal indicates that this object is ready to be initialized. **[R185]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	GREATER_THAN	This literal defines this MPMPolicyOperator as the 'greater than' operator.
3	GREATER_THAN _OR_EQUAL_TO	This literal defines this MPMPolicyOperator as the 'greater than or equal to' operator.
4	LESS_THAN	This literal defines this MPMPolicyOperator as the 'less than' operator.
5	LESS_THAN_OR EQUAL_TO	This literal defines this MPMPolicyOperator as the 'less than or equal to' operator.
6	EQUAL_TO	This literal defines this MPMPolicyOperator as the 'equal to' operator.
7	IN	This literal defines this MPMPolicyOperator as the 'IN' operator.
8	SET	This literal defines this MPMPolicyOperator as the 'SET' operator.
9	CLEAR	This literal defines this MPMPolicyOperator as the 'CLEAR' operator.
10	BETWEEN	This literal defines this MPMPolicyOperator as the 'BETWEEN than' operator.
11	REGEX_PERL	This literal defines this MPMPolicyOperator as a regular expression operator that is compatible with PERL.
12	REGEX_ POSIX_BRE	This literal defines this MPMPolicyOperator as a regular expression operator that is compatible with POSIX Basic Regular Expressions.
13	REGEX_ POSIX_ERE	This literal defines this MPMPolicyOperator as a regular expression operator that is compatible with POSIX Extended Regular Expressions.

Table 77. MPMPolOperatorType Enumeration Definition

9.2.19 MPMPolValueType

This is an Enumeration that defines the datatype of certain class attributes (e.g., mpmPolValueContent, mpmPolicyEventEncoding, mpmPolicyConditionEncoding, mpmPolicyActionEncoding, and mpmPolicyCollectionEncoding). It is defined in Table 78.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R186]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R187]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	STRING	This literal indicates that the datatype of this attribute is a STRING.
3	INTEGER	This literal indicates that the datatype of this attribute is an INTEGER.
4	BOOLEAN	This literal indicates that the datatype of this attribute is a Boolean.
5	FLOAT	This literal indicates that the datatype of this attribute is a FLOATING-POINT number.
6	DATE_TIME	This literal indicates that the datatype of this attribute is a DateTime.
7	GUID	This literal indicates that the datatype of this attribute is a Globally Unique ID (GUID).
8	UUID	This literal indicates that the datatype of this attribute is a Universally Unique ID (UUID).
9	URI	This literal indicates that the datatype of this attribute is a Universal Resource Identifier (URI).
10	DN	This literal indicates that the datatype of this attribute is a Distinguished Name.
11	FQDN	This literal indicates that the datatype of this attribute is a Fully Qualified Domain Name (FQDN).
12	FQPN	This literal indicates that the datatype of this attribute is a Fully Qualified Path Name (FQPN).
13	NULL	This literal indicates that the datatype of this attribute is a NULL. This is used with the NULL object pattern.

Table 78. MPMPolValueType Enumeration Definition

9.3 MPM Datatypes

The following Datatypes are defined in the current MPM Model.

9.3.1 MPMEncodingType

This is a custom datatype that defines the type of encoding used as part of the MPMPolicyObjectID. It is defined in Table 79.

Enum Value	Literal Name	Description
0	ERROR	This literal indicates that an error has occured. **[R188]** This means that this MPMPolicy object **MUST NOT** be used operationally.
1	INIT	This literal indicates that this object is ready to be initialized. **[R189]** This means that this MPMPolicy object **MUST NOT** be used operationally.
2	TEXT	This literal defines the encoding to be ASCII TEXT.
3	XML	This literal defines the encoding to be XML.
4	YANG	This literal defines the encoding to be YANG.
5	primaryKey	This literal defines the encoding to be a primary key.
6	foreignKey	This literal defines the encoding to be a foreign key.
7	GUID	This literal defines the encoding to be a Globally Unique Identifier.
8	UUID	This literal defines the encoding to be a Universally Unique ID.
9	URI	This literal defines the encoding to be a Uniform Resource Identifier.
10	FQDN	This literal defines the encoding to be a Fully Qualified Domain Name.
11	FQPN	This literal This literal defines the encoding to be a Fully Qualified Path Name.
12	stringInstanceID	This literal defines the encoding to be the canonical representation, in ASCII, of an instance ID of this object.

Table 79. AdminState Enumeration Definition

10 References

[1] MEF, “Lifecycle Service Orchestration: Reference Architecture and Framework”, MEF 55.1, January 2021

[2] MEF, “MEF Core Model (MCM)”, MEF 78.1, July 2020

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

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

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

[6] Gamma, E., Helm, R. Johnson, R., Vlissides, J., “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, Nov, 1994. ISBN 978-0201633610

[7] Bäumer, D. Riehle, W. Siberski, M. Wulf, “The Role Object Pattern”, Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97), ACM Press, 1997, Page 218-228

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

[9] Schmidt, D.C., “Model-Driven Engineering”, IEEE Computer, 2006

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

[11] Internet Engineering Task Force RFC 8174, “Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words”, May 2017

[12] Object Management Group, OMG Unified Modeling Language TM (OMG UML), Version 2.5.1, December 2017.

[13] J. Strassner, “Policy-Based Network Management”, Morgan Kaufman, Sep 2003. ISBN 978-1558608597

[14] https://plato.stanford.edu/entries/logic-deontic/

[15] https://users.ece.cmu.edu/~koopman/des_s99/formal_methods/

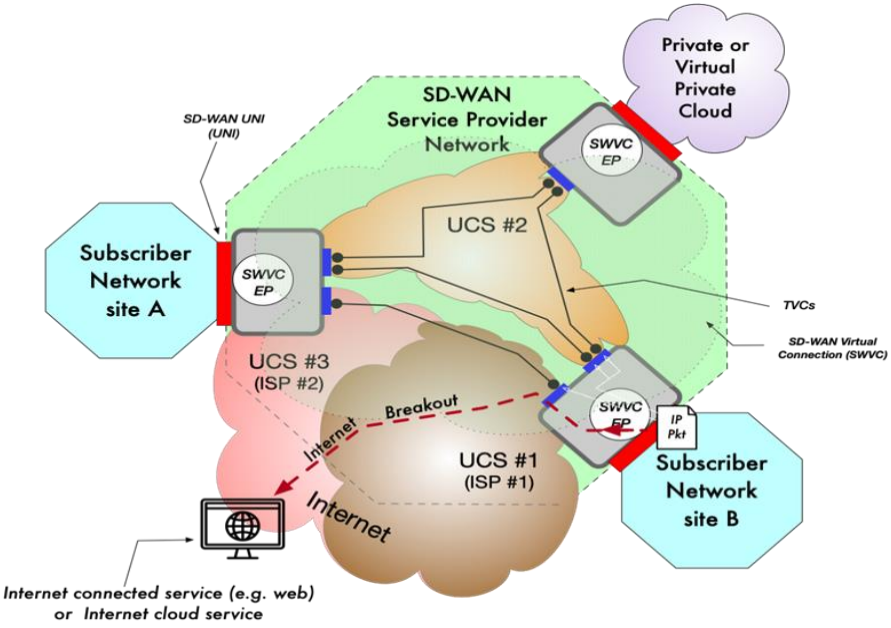
[16] https://plato.stanford.edu/entries/logic-modal/

[17] Strassner, J., Agoulmine, N., Lehtihet, E.: "FOCALE - A Novel Autonomic Networking Architecture", ITSSA Journal 3(1), 64-79, 2007.

Appendix A Exemplary MPMIntentPolicy Language Description

This is an example DSL (Domain Specific Language) that provides an intent program to connect multiple SD-WAN users between multiple sites.

// Create an SD-WAN between multiple Subscriber Sites – BUSINESS USER VERSION
// The SD-WAN Edge is situated between the SD-WAN UNI, on its Subscriber side, and the
// UCS UNIs of one or more UCSs on its network side. The SD-WAN Edge receives ingress
// IP Packets over the SD-WAN UNI; determines how they should be handled according to
// routing information, applicable policies, other service attributes, and information about the
// UCSs; and if appropriate, forwards them over one of the available UCS UNIs. Similarly, it
// receives packets over the UCS UNIs and determines how to handle them, including
// forwarding them on over the SD-WAN UNI to the Subscriber Network, if appropriate. The
// SD-WAN Edge thus implements all of the data plane functionality of the SD-WAN service
// that is not provided by a UCS.



// RED:BEGIN, END for business; app dev DSL will use this for control structures as well
// BLUE:KEYWORDS
// GREEN:INPUT FROM USER
// BLACK:comments

```
BEGIN                                   // could be a space and then DEFINE on same line
    DEFINE SD-WAN EDGES "A", "B", "VPC"  // this creates the associated sites and UNIs
END
// Define the SWVC Connectivity
BEGIN
    DEFINE CONNECTIVITY                 // connects multiple SD-WAN Edges
    USING SERVICE PROVIDER              // under control of the Service Provider
    BEGIN DEFINE UNDERLAY "2"           // defines UCS2
        "A" CONNECTS TO "VPC"           // TVC connects Edges A and VPC
        "VPC" CONNECTS TO "B"           // TVC connects Edges VPC and B
        "B" CONNECTS TO "A"             // TVC connects Edges B and A
    END
END
BEGIN DEFINE CONNECTIVITY
    USING INTERNET                      // Each TVC appears twice since this is a
                                        // single-ended service
    BEGIN DEFINE UNDERLAY "1"           // Internet used as an underlay by ISP1
        "B" CONNECTS TO INTERNET BREAKOUT  // Internet Breakout (single-ended
                                           // service, only appears once)
        "B" CONNECTS TO "A"             // TVC between B and A
    END
    BEGIN DEFINE UNDERLAY "3"           // Internet used as an underlay by ISP2
        "A"  CONNECTS TO "B"            // TVC between A and B, 2nd occurrence
                                        // due to Internet
    END
END
```