ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/4256684

Issues in Designing a Policy Language for Distributed Management of IT Infrastructures

Conference Paper · May 2007

DOI: 10.1109/INM.2007.374767 · Source: IEEE Xplore

CITATIONS
32

READS
140

4 authors:

Dakshi Agrawal
IBM
61 PUBLICATIONS **4,192** CITATIONS

SEE PROFILE

Seraphin B. Calo
IBM
153 PUBLICATIONS **2,443** CITATIONS

SEE PROFILE

Kang-Won Lee
IBM
103 PUBLICATIONS **2,251** CITATIONS

SEE PROFILE

Jorge Lobo
University Pompeu Fabra
205 PUBLICATIONS **4,940** CITATIONS

SEE PROFILE

All content following this page was uploaded by Jorge Lobo on 22 May 2014.

The user has requested enhancement of the downloaded file.

SAMSUNG 1028

Issues in Designing a Policy Language for Distributed Management of IT Infrastructures

Dakshi Agrawal, Seraphin Calo, Kang-Won Lee, Jorge Lobo
IBM T. J. Watson Research Center
{agrawal,scalo,kangwon,jlobo}@us.ibm.com

***Abstract*— The objectives of this paper are twofold. First, we introduce a novel policy language, called CIM-SPL (Simple Policy Language for CIM) that complies with the CIM (Common Information Model) Policy Model and fully incorporates CIM constructs. Currently, the CIM standards from Distributed Management Task Force (DMTF) include a Policy Model, but there is no satisfactory way to render this policy model. CIM-SPL is a language that has been defined for this purpose. Second, we address design and implementation issues for policy languages in general, and for CIM-SPL in particular. The design of CIM-SPL was inspired by our previous experiences in designing various policy languages (e.g., PDL from Bell Laboratories, and ACPL from IBM) and lessons learned from studying other well-known policy languages (e.g., Ponder from Imperial College). We will discuss our design choices, evaluating the pros and cons of various alternatives. The ideas presented in this paper are meant to shed light on our design decisions, and provide guidance for those who want to build a CIM-based policy system or some other policy system in the future.**

I. Introduction

The relative cost of IT system management is ever increasing compared to the cost of the infrastructure itself. In 2005, operations labor costs accounted for over 70% of CIO budgets, with hardware, software and services costs taken together making up less than 30%[1]. To reduce management cost, the industry has been exploring various ways to minimize the complexity and overhead of system management (e.g., Autonomic Computing Initiative from IBM and N1 Initiative from Sun Microsystems). Policy-based management has been proposed as a practical means to achieve this goal. The key idea of policy-based management is to allow system administrators to specify declarative policy rules that govern the behavior of their IT systems instead of relying on *ad hoc* mechanisms such as running customized scripts based on experience and rules of thumb. In this way, the complexity of system management can be reduced, and also, by employing various policy analysis technologies [1], one can validate the correctness and consistency of policies automatically, thus increasing the reliability and predictability of the system's behavior.

Another important trend in the management of IT systems is the use of the Common Information Model (CIM) standards from the Distributed Management Task Force (DMTF). These standards provide mechanisms for the definition and use of management information for systems, networks, applications and services. The ambitious goal of these standards is to develop: (a) a unified model to represent all IT components; (b) a means to store management information in CIM data structures; and, (c) a standard way to access and update CIM data repositories. Currently, there are several initiatives dedicated to the implementation of CIM standards (e.g. OpenPegasus) to facilitate storing and distribution of CIM information. At the same time, domain-specific CIM-related activities are underway in various key areas, including networks, storage systems, and servers.

From the above observations, it is evident that a policy-based management system based on the CIM standards could be used for system management in many application domains. The CIM standards already include a Policy Model, and it has been recognized by the DMTF Policy Workgroup that what is now needed is an effective, simple language for rendering the CIM Policy Model. Although there are a number of policy languages that have been proposed by academia [2] and industry [3], [4], [5], [6], there has been very little activity aimed at maintaining compatibility with the CIM Policy Model or incorporating CIM constructs. To our best knowledge, CQL (CIM Query Language) [6] is the only language that has been suggested for expressing policy rules that complies with the CIM data model. However, writing a policy using CQL has proved onerous (as will be discussed in subsequent sections of this paper), partly because CQL was developed as a query language rather than a policy language.

To remedy this situation, the authors have proposed a novel policy language, called CIM-SPL (Simple Policy Language for CIM) that complies with the CIM Policy Model and fully incorporates CIM constructs. CIM-SPL was presented to the DMTF Policy Workgroup where it has been approved for a preliminary release and is waiting approval from the DMTF Board for a public release [7]. The main goals of this paper are to describe CIM-SPL, and, more importantly, discuss our design choices, evaluating the pros and cons of various alternatives.

The remainder of the paper is structured as follows: Section II presents background on CIM and CIM Policy Models, and other policy languages. Section III introduces the basic syntax for writing policy rules in CIM-SPL. Section IV provides further details on the syntax and semantics of CIM-SPL by introducing policy groups. Section V provides a concrete example of a policy written in CIM-SPL to illustrate several

[1]The operational labor cost varies anywhere from 70% to 80% in the surveys and studies conducted by market research firms such as IDC and the Gartner Group.

salient features of the language. Section VI discusses design principles and constraints that guided the development of CIM-SPL, and how those design decisions affect policy-based management systems that may use CIM-SPL. Finally, Section VII concludes the paper with a summary of our results and a discussion of future research directions.

II. Background

In IT systems, policies represent externalized logic that determines the behavior of managed systems, networks, applications, and services [8]. Thus a policy language depends on two information models, one for the resources being managed, and another for the policies themselves. CIM-SPL is built on top of the DMTF standards for the Common Information Model (CIM) of managed elements, including the CIM Policy Model.

A. CIM System Model

The Common Information Model (CIM) is an object-oriented model defined by the DMTF that can be used to describe various components of IT systems, including software and hardware elements and their relationships. CIM is a vendor- and system-independent information model that attempts to unify and extend previous management models, such as those used by SNMP (simple network management protocol), DMI (distributed management interface), and CMIP (common management information protocol). Based on an object-oriented modeling approach, CIM describes the elements of IT systems using object classification, and specifies relationships among objects using object-oriented concepts, such as inheritance and dependencies. The goal of CIM is to model all aspects of the managed environment, including systems, devices, networks, operating systems, applications, and even users.

Each logical and physical element in CIM is represented as an instance of a class that contains various properties of that element. In addition, relationships between elements are modeled using a special type of class called *association*. The inheritance relation (a class X is a subclass of a class Y) is intrinsic to the object oriented model. An *association* is a new class whose instances have references to the class instances that are in some way related. The name of a reference in an association class defines the role that the class instance referred to plays in the relationship represented by the association class. For example, the association `Component` has two references PartComponent and GroupComponent to ManagedElement. This association can be used to represent the relationship between say a server and a networking card in that server.

B. CIM Policy Model

The CIM Policy model is an information model defined by the DMTF Policy Working Group for the CIM schema. The classes comprising the Policy Model serve as an extensible class hierarchy for defining policy objects that can be used by application developers, network administrators, and policy administrators to represent policy rules of the form: if ⟨condition(s)⟩ then ⟨action(s)⟩. Policy rules may be aggregated into policy groups, and these may be nested to represent a hierarchy of policies. The CIM Policy model has been influenced by its precursors, the Policy Core Information Model (PCIM) and PCIM Extensions (PCIMe) defined jointly by the Internet Engineering Task Force (IETF) Policy Framework Work Group and the DMTF Policy Working Group [5][6].

The main high level class in the CIM Policy model is the Policy class. The Policy class is a parent class for the PolicySet class, which is an abstract class representing a set of policies. The members of a policy set have a common set of policy roles that are defined via the PolicySetInRoleCollection association. The PolicyRule and PolicyGroup classes are subclasses of PolicySet. Stand-alone policies are instances of the PolicyRule class. The PolicyGroup class allows PolicyRules and/or PolicyGroups to be aggregated in a single container. A PolicyGroup can therefore nest other PolicyGroups, and there is no restriction on the depth of the nesting in sibling PolicyGroups.

The condition part of a policy rule can be specified via a PolicyConditionInPolicyRule association with the PolicyCondition class. The action part of a policy rule can be specified via a PolicyActionInPolicyRule association with the PolicyAction class. Conditions and actions can also be aggregated in CompoundPolicyCondition and CompoundPolicyAction classes. Two important points about the CIM Policy Model are that: (1) it represents a fairly comprehensive structure for describing policies; and, (2) it is an abstract model, and a language implementation does not have to mirror its structure. We will discuss more about the second point in later sections of this paper.

C. Other Languages

This section reviews several well-known policy languages. Our intent is to provide a brief background to the reader and provide information relevant to subsequent discussions in this paper.

PDL [5] from Bell Labs is one of the early policy languages to specify *event-condition-action* style policy rules (i.e., upon occurrence of an *event*, if a *condition* is true, then execute an *action*). It can also specify the generation of a new event when a condition is true (upon occurrence of some other event) to support complex event interactions. PDL (and all the other policy languages that we will discuss in this section) supports declarative policy rule semantics. In PDL, by using regular expressions, multiple events can be composed to define a complex event. PDL has the limitation that rules cannot be grouped in any form, i.e. all policies have a flat structure.

The OASIS eXentible Access Control Markup Language (XACML) [4] provides a means for writing access control policies. In addition to the syntax and semantics for access control policies, XACML also prescribes a standard format for access-control requests and responses. An XACML rule has a *condition-effect* structure, which is similar to that of PDL, except that *effect* can take only two values: "Permit" or "Deny." An XACML rule also has a *target* component that

defines the set of resources, subjects, actions, and environments to which the rule applies. A significant difference with PDL is that policy authors can specify intricate orders of rule evaluations. In PDL the order of the rules is irrelevant. In addition, XACML policies can be organized in hierarchical policy groups.

Ponder [2] is a general policy language that can define both access control rules like XACML, called authorization rules, and general management rules *a la* PDL, called obligations. For example, one can write an authorization policy rule with *subject, target, action* arguments to specify that a *subject* is authorized (or not permitted) to perform certain *action* on *target* objects. Similarly, Ponder can be used to write delegation rules (in which a subject temporarily grants access rights to others), information filtering rules (similar to the condition part in a PDL policy), and refrainment rules (indicating that a subject should refrain from performing certain actions). Obligations are rules that must be enforced after a particular event occurs. Ponder supports simple composition of events. Like XACML, policies can be organized into groups.

The CIM Query Language (CQL) [6] is a DMTF preliminary standard that facilitates writing queries for extracting data from a CIM data management infrastructure. CQL closely mimics the *select-from-where* SQL query syntax. The key difference is that a CQL query is applied to CIM properties (for SELECT clauses) and CIM classes (for FROM clauses). The CQL preliminary standard document suggests a mechanism to use CQL for the specification of CIM policy rules. To specify a CIM-style policy rule (as defined by the CIM Policy Model), CQL queries are used to specify both the condition part and the action part of the rule. For example, in a policy that allocates more space to a disk low on free space, the condition query first selects the disks with low space, and then the query action invokes a method to increase space to those selected disks. This makes the definition of actions very awkward since one needs to generate a relational table that encodes the action call, and then process the CQL query action with completely different semantics than that for the condition. Even for the specification of a query condition, CQL imposes a complicated syntax since CIM is based on an object model and CQL assumes a relational model, thus just to compare the values of two object properties CQL requires a query that interprets the objects as relations.

The Autonomic Computing Policy Language (ACPL) from IBM [3] is an attempt to support distributed IT systems management by extending the PCIM model [9]. In PCIM, each policy rule can specify a *condition* and an *action* (i.e., if the *condition* is true, then execute an *action*). In addition, each policy rule also has a *role* that specifies devices to which the policy rule should be applied, and an integer *priority* that specifies the relative priority of the policy rule in a policy group. Unlike most other policy languages, ACPL allows the user to extend the language syntax by defining new operations.

Note that CIM-SPL is not a generic policy language in the style of, for example, DEN-ng [10]. CIM-SPL is a *CIM compliant* policy language that exploits, and is constrained by, the stardard features of CIM. CIM-SPL incorporates many features of, and lessons learned from the implementation and use of, prior policy languages that could be incorporated while being compatible with CIM. We will highlight some of these features, and lessons learned, in Section VI.

III. CIM-SPL Policy rules

The *Condition-Action* structure of policy rules has the attractiveness of providing an intuitive way of describing the effect of the rule on the behavior of systems governed by these kinds of policy specifications:

> *if the system is in a state where the Condition is true then execute the Action*

Our goal is to provide more than an intuitive description of policies. We want to provide a full operational model of how policies can be evaluated. Let us consider the following example:

> *If the file system is 85% full then expand the storage pool by 25%.*

There are two elementary questions that arise from this policy:

1) What kind of data is available to determine that the file system is 85% full and how can we access it?
2) What kind of operations are available to manipulate the storage pool and how can we access them?

In the context of CIM there are natural answers to these questions. The data available is contained in the CIM data structures and it can be accessed using a CIM Object Manager (CIMOM). The operations are methods defined in CIM Classes and methods defined for manipulating CIM Objects (e.g., methods for creating a new CIM instance or for setting the value of a property). These operations correspond respectively to the extrinsic and the intrinsic operations described in the Web-Based Enterprise Management (WBEM) standards [11] and can be accessed using a WBEM implementation.

Returning to our example, a file system can be represented in CIM as an instance of the CIM_LocalFileSystem class which has, among others, two properties, AvailableSpace and FileSystemSize. These properties can be used to find the percentage of space that is in use. In the CIM model of the system that contains the file system, there must also be an object that is an instance of the CIM_StorageConfigurationService class. This object can be accessed by traversing CIM associations starting at the CIM_LocalFileSystem object. Then, we can use the CreateOrModifyElementFromStoragePool method from the CIM_StorageConfigurationService object to expand the storage pool.[2]

We now give a general overview of the syntax of CIM-SPL policy rules. A CIM-SPL policy rule follows the following schema:

import ⟨MOF Name⟩::⟨CIM Class Name⟩:⟨Condition⟩
policy{

[2]The name of the CIM classes and their properties is not that important. They are used to demonstrate how the CIM model can be used to evaluate and execute policies, but readers interested in the details of these classes can consult [12].

```
declaration { ⟨constant and macros⟩ }
condition { ⟨Java-like boolean expression⟩ }
decision { ⟨action workflow⟩ }
}
```

Each CIM-SPL policy is written under the scope of a single CIM Object referred to as the *anchor object* of the policy. All other CIM Objects referenced by a CIM-SPL policy should be accessible by traversing CIM associations starting from the anchor object of the policy. In Section VI-E, we will provide the rationale for this design decision.

The CIM Class specification in the import statement unambiguously defines the class of the anchor object for the policy. CIM uses the Managed Object Format (MOF) to publish class schemas. The import statement will have the name of a MOF file followed by the name of a CIM class described in that MOF file. There is an optional Condition that any anchor object must satisfy before a policy is evaluated on the object[3]. This condition is a simple conjunction of equalities and inequalities expressed in terms of the properties of the class. All unqualified identifiers referenced in the rest of the policy refer to properties of the anchor object. To refer to the anchor object itself we either use the reserved identifier self or we use the name of the CIM class directly. Other objects can be accessed by traversing associations. We will describe these traversals later in this section. The **declaration** section is optional and defines macros and constants to simplify notation in the condition, in case the long CIM names make it cumbersome to write simple expressions. The **condition** and **decision** sections correspond respectively to the if and then clauses of a policy rule and are elaborated below.

A. The Condition Section

The condition section of a policy rule contains a boolean expression similar to a Java boolean expression. We allow the standard boolean operators AND, OR and NOT. We also allow the comparison of arithmetic expressions using $<$, $>$, $==$, $\neq$, etc. In these comparisons, we allow arithmetic expressions involving operations such as +, -, etc. We can also compare strings and provide many predefined string and calendar operations. In general, sub-expressions of any CIM intrinsic type can be part of a condition, as long as the overall condition is a boolean expression. The following is an example of a condition that uses two properties from the CIM_LocalFileSystem class.

```
import CIM_X_XX_XXX::CIM_LocalFileSystem
policy{
    condition { AvailableSpace/FileSystemSize < 0.25 }
    decision { ... }
    }
```

Since CIM_LocalFileSystem is specified in the import statement of this policy, the anchor object for it should be an instance of CIM_LocalFileSystem. In other words, the policy server will evaluate this policy against an anchor object that is an instance of CIM_LocalFileSystem. The policy condition would be true if the available space provided by the AvailableSpace property of the anchor local file system divided by the total file system size provided by the FileSystemSize property of the same file system is smaller than 0.25.

[3]In Section VI-F, we provide the rationale for specifying conditions in two places: first in the import statement, then in the condition clause.

B. The Decision Section

As we mentioned earlier, an action in a decision will most likely be a method invocation on an object. The following is an example of such an invocation:

```
import CIM_X_XX_XXX::CIM_StorageConfigurationService
policy{
    condition { ... }
    decision { self.CreateReplica(...) }
    }
```

CreateReplica is a method of the CIM_StorageConfigurationService class.

Another type of action is one that sets the values of properties by using an intrinsic operator. The following example sets the value of the PortType property of an FCPort object:

```
import CIM_X_XX_XXX::CIM_FCPort
policy{
    condition {... }
    decision { Self.set(PortType = 1) }
    }
```

In addition to the basic action invocations, CIM-SPL allows composition of basic actions into complex actions. We have borrowed the decision structure from the Ponder policy language [2]. In Ponder, basic actions can be combined using two types of control structures:

1) Serial execution (two models) – In a sequence of two actions, execute the second action only if the first action has been completed successfully or only if the first action has failed.
2) Concurrent execution (two models) – In a block of actions, execute all actions concurrently or execute at least one action in a block.

Finally an action in CIM-SPL can also be the invocation of another policy. This is usually referred to as cascaded policies. The invocation is simply done by using a reference to a CIM Policy object in place of a basic action in the decision. The call must have as its argument a collection of objects. There will be an evaluation of the specified policy for each object, using the object as an anchor object. The class of these objects must match the class specified in the import statement of the invoked policy.

C. Data Types and Operators

CIM-SPL supports all intrinsic data types[4] defined by the CIM Meta Schema and one-dimensional arrays of an intrinsic type. CIM-SPL has all conventional arithmetic, boolean, and casting operators along with operators for string and date-time manipulation. CIM-SPL also has operations that apply

[4]The CIM Meta Schema defines data types for signed and unsigned 8-, 16-, 32-, and 64-bit integers, IEEE 4- and 8-byte floating point numbers, 16 bit UCS-2 character, UCS-2 string, boolean, date-time, and strongly typed reference to a CIM Object.

to whole arrays, such finding the minimum and maximum elements of an array, checking if a given value appears or does not appear in the array, and computing averages, medians, etc. of elements in an array.

A unique feature of CIM-SPL is that it has built-in operators that allow traversal of CIM associations. Such traversal is often required during a policy evaluation. One such operator is the **collect** operator with the following signature:

collect(⟨RefExpression⟩, ⟨association⟩, ⟨role⟩, ⟨resultRole⟩,
⟨CIM class⟩, ⟨expression⟩)

As an example, assume that the variable mySwitch has a reference to a SAN switch, and we want to collect references to all fibre channel ports in that switch in an array of references. We can use the **collect** operator as follows:

collect(mySwitch, CIM_SystemDevice,
GroupComponent, PartComponent,
CIM_FCPort, true)

In the CIM model, a SAN switch and a fibre channel port in that switch are associated with each other through an instance of the association CIM_SystemDevice. The CIM_SystemDevice association has two properties, GroupComponent which is a reference to a CIM_System (in this case, the SAN switch) and PartComponent which is a reference to a CIM_LogicalDevice (in this case, the fibre channel port). Thus, the first argument of the **collect** operator is the root of the traversal, and the second argument is the traversed association. The third argument is the name of the property that holds a reference to the first argument, and the fourth argument is the name of the property that holds a reference to the other end of the association traversal. Thus, the **collect** expression given above will collect all instances of CIM_SystemDevice that have a reference to the mySwitch object in the property GroupComponent. It will then collect references pointed to by the PartComponent property in all such instances in an array. The fifth argument of the **collect** operator specifies the class of the collected references, and can be null if there is no ambiguity concerning the possible classes of the collected references[5]. Finally, the last argument specifies a boolean expression that must be satisfied by objects referenced by the elements of the resulting array. The last argument acts as a filter to exclude certain references from being returned. For example, if we were only interested in ports of a switch that have unknown type, then we could use the following expression[6]:

collect(Self, CIM_SystemDevice,
CIM_GroupComponent, CIM_PartComponent,
CIM_FCPort, PortType == 0)

By composing several collect operations, we can traverse multiple associations and collect system components that are related to a single root instance by complex relationships.

[5]A director class switch may have its ports arranged in multiple blades. For such a switch, both the blades and the ports would be associated with the switch by a CIM_SystemDevice association. Therefore, we explicitly specify that we want to only collect instances of CIM_FCPort.

[6]The port type unknown is mapped to the unsigned integer 0.

IV. Policy groups

Effective policy management requires hierarchical organization of policies. A policy hierarchy may be constructed for many reasons: (a) to group policies by functionality; (b) to group policies by means of the components and subcomponents to which they apply; (c) to group policies by their relative importance; (d) to enforce a priority order, etc. In the CIM Policy Model policies are organized into groups. A policy group consists of policy rules and other policy groups. In CIM-SPL, the definition of a policy group follows the following schema.

import ⟨MOF Name⟩::⟨CIM Class Name⟩:⟨Condition⟩
strategy ⟨Execution Strategy⟩
declaration{ . . . }
policy{ . . . }:⟨Priority⟩
. . .
policy{ . . . }:⟨Priority⟩
policyGroup:⟨Association Class⟩
(⟨Property One⟩, ⟨Property Two⟩)
{ . . . }:⟨Priority⟩
⋮
policyGroup:⟨Association Class⟩
(⟨Property One⟩, ⟨Property Two⟩)
{ . . . }:⟨Priority⟩

A policy group within a policy group has an optional association specification that consists of an association class name and two properties of this class. We say that two managed elements E_1 and E_2 are *associated* with each other through an association specification assc(π_1,π_2) if there exists an instance of assc such that its property π_1 references E_1 and its property π_2 references E_2. Below, we describe how these arguments are used. The import statement in the policy group plays the same role as the import statement in a single policy rule. The import statement for a policy group indicates the class of the anchor object for policy rules and groups inside the group. Consider a policy group $\mathcal{P}$ whose constituent policy rules and policy groups are given by P_1, P_2, ..., P_n. For each anchor object A for which the policy group $\mathcal{P}$ needs to be evaluated, the evaluation of $\mathcal{P}$ proceeds as follows:

For each P_i in $\mathcal{P}$:

1) If P_i is a policy rule then the rule is evaluated with A as the anchor object.
2) If P_i is a policy group, a new set of anchor objects S is created as follows:
   a) If the policy group P_i has the optional association specification assc(π_1, π_2) then the set S has all CIM managed elements E_i's such that A and E_i are "associated" with each other through the association assc(π_1, π_2).
   b) If the optional association is not specified then by default, the association class is assumed to be the association CIM_Component, the first property is assumed to be the Property GroupComponent and the second property is assumed to be the Property PartComponent. As a consequence, the

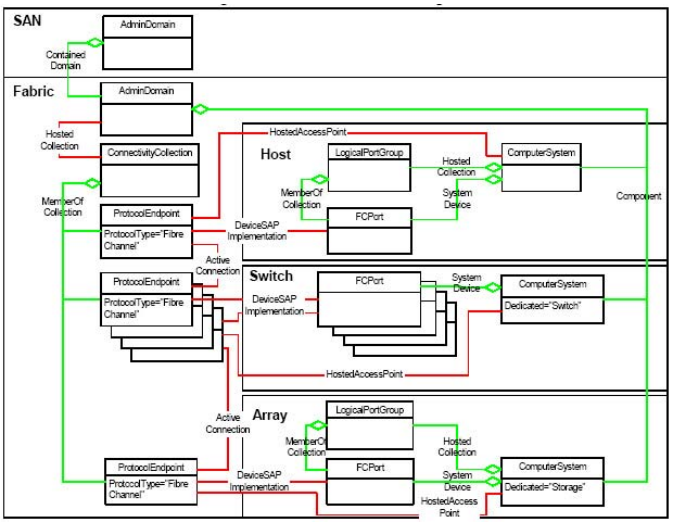

Fig. 1. SNIA spec

set S consists of all Components of the managed element A.

For each object O in S, we evaluate P_i using O as the anchor object.

This simple but powerful traversal of associations is a unique feature of CIM-SPL that significantly simplifies the management of policies in CIM. Let's take as an example the section of the SNIA specification [12] shown in Figure 1. The figure shows a SAN Fabric which has Host, Switch, and Array instances (distinguished by the value of the property Dedicated) as its components. We can have a policy group for a SAN fabric comprised of two policy subgroups, one for Switch instances and the other for Array instances. Inside the policy group for the Switch instances, we can have a policy group for fibre channel ports (FCPort). Schematically the policy group for the SAN fabric will look as follows:

```
import SAN xxx::CIM_AdminDomain
strategy Execute_All_Applicable
policyGroup {
   import SW yyy::CIM_ComputerSystem:Dedicated == "switch"
   ...
   policyGroup:CIM_SystemDevice(GroupComponent,PartComponent) {
      import FCP xxx::CIM_FCPort
      ...
      }
   }
policyGroup {
   import SG yyy::CIM_ComputerSystem:Dedicated == "storage"
   ...
   }
```

When this policy group is evaluated for a SAN fabric, all switches in the fabric (components of type CIM_ComputerSystem with the property Dedicated set to "switch") will have the first policy subgroup evaluated on them. Similarly, all storage arrays in the fabric (also components of type CIM_ComputerSystem but with the property Dedicated set to "storage") will have the second policy subgroup evaluated on them. For each switch in the fabric, all fibre channel ports (CIM_FCPort instances that are reached by traversing the association CIM_SystemDevice from a switch), will have the innermost policy group evaluated on them. The idea here is that when specifying a policy group for a managed element, certain policies will only be applicable to a particular component of the managed element; or, certain policies within a policy group may only be applicable to managed elements that are associated in a specific manner with the managed element. Such policies can be conveniently collected within a subgroup to ease the specification and evaluation of the policy group. In Section VI, we elaborate further on how such hierarchical specification of the policies along with the concept of an anchor object can help in efficiently gathering data required for policy evaluation.

We have not mentioned the effect of the strategy statement and priorities. The strategy statement works in coordination with priorities. Priorities are positive integers assigned to each policy rule and policy group. All priorities must be different. The CIM Policy Model defines two evaluation strategies: Execute_All_Applicable and Execute_First_Applicable. A policy rule is applicable if its condition evaluates to true. A policy group is applicable if at least one of its internal policy groups or policy rules is applicable. Under the Execute_All_Applicable strategy, all the policy rules and all the policy groups in a policy group are evaluated regardless of their priorities, and actions specified by all applicable policy rules are executed.

If the evaluation strategy is Execute_First_Applicable then the evaluation proceeds in the descending order of priority of constituent policy rules and policy groups, and the evaluation stops as soon as a policy rule or a policy group is found applicable. We note that in either case if a policy group is selected for evaluation, then the evaluation will follow the strategy declared internally for the group.

V. Example

The following example is based on the process for increasing the size of a file system as defined in the SNI Storage Management Standard [12]. This policy can be triggered periodically and if the file system is more than 85% full, it will take the necessary actions to expand the storage pool by 25%. We have already mentioned that using the AvailableSpace and the FileSystemSize properties of a LocalFileSystem, we can detect how full the file system is. To expand the storage pool we need two things: one, we need to determine the storage pool of the file system to which to allocate more space; and, two, we need to determine the storage configuration service of the computer system where the file system is located. We can get to the computer system from the local file system object by traversing the CIM_HostedFileSystem association. From the computer system we can get to the storage configuration service by traversing the CIM_HostedService association. These are the first two macros in the declaration section of the

policy below. To find the storage pool we need first to find out the logical disk of the file system, and from the logical disk, find the storage pool by traversing the CIM_ResidesOnExtend association. These are the next two macros in the declaration. The last macro gets a parameter that needs to be passed to the operation that expands the pool. This parameter describes the expected capabilities of the allocated space.

```
import CIM_X_XX_XXXX::CIM_LocalFileSystem;
strategy Execute_All_Applicable;
policy {
 declaration {
   computer_system =
     collect(self, CIM_HostedFileSystem, PartComponent
             GroupComponent, null, true)[0];
   storage_config_service =
     collect(computer_system, CIM_HostedService,
             Antecedent, Dependent,
             CIM_StorageConfigurationService, true)[0];
   logical_disk =
     collect(self, CIM_ResidesOnExtent,
             Dependent, Antecedent, null, true)[0];
   storage_pool =
     collect(logical_disk, CIM_AllocatedFromStoragePool,
             Dependent, Antecedent, null, true)[0];
   fs_goal =
     collect(self, CIM_ElementSettingData, ManagedElement,
             SettingData, CIM_FileSystemSetting, true)[0];
   }
 condition {
   (AvailableSpace / FileSystemSize) < 0.15
   }
 decision {
   storage_conf_service.
   CreateOrModifyElementFromStoragePool
             ("LogicalDisk", /* ElementType Volume */ 8,
             null, fs_goal,
             1.25 * FileSystemSize,
             storage_pool, logical_disk)
   | DoSomethingOnFailure()
   }
}:1;
```

The full definition of the method CreateOrModifyElement-FromStoragePool can be found in [12]. We bring to the reader's attention the third line of the attributes for this method where the new size is passed. We also point to the conditional operator "|" in the decision. This operator means that if the first action fails then the second is executed. Later in Section VI, we will discuss which workflows are allowed by CIM-SPL and why.

VI. Design Principles and Constraints

A. Compliance to CIM and CIM Policy Models

There are two aspects of our design choice to make CIM-SPL compliant with CIM. First, CIM-SPL should naturally support specifying policy rules using CIM constructs. To achieve this goal, we designed various language features, such as the import statement, collection operators, and the notion of a policy group. The role of an **import** statement is to specify which CIM object and which CIM version is relevant to a particular policy rule. With this information an implementation can retrieve data to be evaluated from a CIM server, and can facilitate error checking – e.g. the Status field in CIM_FileSystem is deprecated in CIM version 2.7.0 and erroneously referring to that field in a policy rule can be detected. We provide means to traverse CIM associations via **collect** operators. Finally, we support the notion of a policy group reflecting the logical hierarchy of managed elements in the CIM model. For example, the nested structure between Fabrics, Switches, and FCPorts in the CIM model denoted via aggregation can be represented as nested policy groups in the CIM-SPL.

The second aspect of compliance is consistency with the CIM Policy Model. For example, the policy rule structure and the policy group structure in CIM-SPL must be consistent with CIM_PolicyRule and CIM_PolicyGroup of the CIM Policy Model. One ramification of this compliance is that it rules out direct support for various other types of policies, e.g. those based on decision trees, which do not conform to the CIM Policy Model[7].

It is worth noting that being consistent with the CIM Policy Model does not mean that the syntax of our language should exactly mirror the class structure of the CIM Policy Model. This is because the model is a logical abstraction of the real world; it is not a blue print for implementing a policy system. Thus not all constructs in the information model should be taken literally. For example, the CIM_PolicySet abstract class in the CIM Policy Model does not need to be specified in an actual rendering of a policy because its semantic is covered by the concept of policy rules and policy groups.

B. Declarative

When we embarked on the design of a policy language for CIM, our goal was to come up with a language that was *declarative*. If we want the ability to analyze, and get a good grasp on generic ways to debug polices, we need to stay away from imperative programming language constructs such as destructive assignments and explicit loops[8]. The main effect on CIM-SPL of being declarative is that the evaluation of conditions has no side effects. Hence, when selecting a policy to evaluate from a policy group based on priorities, the order of evaluation of conditions is irrelevant. This, for example, allows any compiler to pre-fetch any data that might be reused in several policy groups and rules. It also simplifies the combination of policies. In addition, we are able to guarantee termination of policy evaluation by simply timing out actions that take too long to respond[9], i.e., all condition evaluations are finite. This design decision will allow us to use analysis algorithms for finding potential conflicts and for ratifying policies [1]. It will also allow automatic priority assignments based on relative ordering [1]. This constraint does not mean that no iterations are possible. CIM-SPL provides built-in operations that allow the creation of a collection of objects, and operations that iterate over the objects in a collection.

[7] Arguably, allowing other types of policies such as decision trees can enhance the policy systems. However, the benefits have to be weighed against the complexity, and the simple policy rule structure supported by the CIM Policy Model has proved to be sufficient for many applications domains.

[8] For example, try to analyze Perl scripts, a favorite of many for policy implementations.

[9] Note that actions are methods defined outside the control of the policy.

Similar operations can also be applied to CIM properties that are arrays.

C. Evaluation Trigger

The informal semantics of Condition-Action policy rules is: when the condition of a rule is true, then execute the corresponding action. In most policy systems, the condition of a policy rule is not evaluated continuously[10], it is triggered by an *evaluation trigger*. A key decision in the design of CIM-SPL was not to specify what triggers the evaluation of a policy (rule or group) in the policy itself. It is envisioned that when a policy is installed in a policy server, as a part of the installation, an evaluation trigger for the policy will be specified. The evaluation trigger could be specified as an event or as a pattern of events that occurs in the system (e.g., installation of new hardware, arrival of a special packet in the network, etc); it can be specified by a time schedule (e.g., every Friday at 3 AM); or, it can be an explicit request for policy evaluation and guidance from a managed element or a system administrator.

Many policy-based management systems exclusively use events (e.g., arrival of a special network packet or start-up of a device) as evaluation triggers. For example, PDL and Ponder explicitly include events that trigger evaluation of a policy as part of the policy specification. Arguably, using events as an evaluation trigger is sufficiently generic and covers the examples of evaluation triggers given above—for example, if a policy should be triggered when a complex pattern of events is observed (e.g., more than 10% failure of ping requests over the last 100 attempts), then a composite event that captures the complex pattern of events can be defined and the composite event can be used as the event that triggers the evaluation of the policy. If a policy should be evaluated every Friday at 3 AM, then an event can be defined that occurs every Friday at 3 AM.

While an event trigger mechanism is sufficiently generic, we believe that if events are the only way in which a policy can trigger, and the events are specified within the policy, then we will run into implementation oddities in many application domains. For example, events imply an asynchronous interaction style in which devices in a distributed system generate events, a policy server subscribes to the events relevant for its policies, and upon receiving a relevant event, the server evaluates corresponding policies and executes actions. This asynchronous interaction style is unnatural for scenarios where a "policy-aware" device is coded to seek policy guidance before proceeding further on a process or workflow. In such a scenario, a synchronous request-response type of interaction with the policy server may be more appropriate, and economical.

Another problem with specifying events within a policy is the variety of events present in different management domains. For example, in a load balancer, located in front of a server-farm, the event that requires the evaluation of a policy may be the initial TCP SYN packet [13]. In a storage area network failure recovery policies are triggered when a failure event is detected. The frequency of these two types of events is probably several orders of magnitude apart. There are also differences in how to monitor the state changes that cause events in different managed elements. Some managed elements may automatically report events, e.g. by means of SNMP traps, but in other cases a polling mechanism checking the status of an element at regular intervals is the only mechanism to detect changes. The implementation mechanism will rely on the support offered by the underlying managed elements. Furthermore, in many situations the event of interest is the result of sophisticated compositions of more basic events over time. For example, detecting failures or system performance degradation may require the correlation of events occurring in several places in the system. There are research groups dedicated to developing event correlation languages and systems (see, for example, [14] and the references therein). Given the multiplicity of environments where CIM is expected to be used, we decided to separate any event specification and evaluation from the policy specification and evaluation.

The advantage of this design is that when a policy is installed, the context of the particular environment can be taken into consideration to select the most appropriate event correlation engine or invocation method to drive policies.

CIM has defined an event model [15] that can be used to trigger CIM-SPL policies (while still keeping the event specification separate, as an installation time activity). In CIM one can define events, called indications, that are generated when a change occurs in a managed object. However, the language to specify how to select the event is a parameter left unspecified. It would be interesting to explore the possibility of standardizing an event correlation specification language or languages in CIM. Some issues regarding this kind of specification are discussed in [16].

D. Action Specification

The choices for specification of the action part of a policy rule range from defining a single atomic action to providing a complex workflow [17]. Our choice of allowing simple workflows is based on the principle that policies guide system behavior; they do not prescribe how to implement that behavior [16].

Note that a single atomic action could encapsulate a complex workflow. For example, the action part of the policy given in Section V specifies a single atomic action that creates or modifies a volume on a storage pool. A storage management system would typically implement this action as a complex workflow. Thus CIM-SPL does not restrict the policy author to specifying only "simple" actions.

On the other hand, limiting decision specification to a single atomic action would be too restrictive and policy authors would need to frequently extend standard CIM profiles of IT systems just to write meaningful policies. As a compromise, CIM-SPL allows four basic compositions of actions: (a) a sequence of two action blocks such that the second action

[10] A policy system that uses database triggers as implementation mechanisms can arguably evaluate the condition continuously.

block is only executed after the first action block has finished successfully; (b) a sequence of two action blocks such that the second action block is only executed if the first one fails; (c) a sequence of two action blocks such that both action blocks can be executed in parallel and the composite action is successful only when both action blocks are successful; and, (d) a sequence of two action blocks such that both action blocks can be executed in parallel and the composite action is successfully completed as soon as one of the action blocks is successfully completed.

These basic action compositions allow policy authors to write meaningful policies without having to extend CIM models of the managed elements. For example, as shown in the previous section, the composition (b) can be used for logging, tracing, and error-handling. Compositions (a) and (c) can be used when a the system needs to execute several actions in a serial or parallel manner respectively. Composition (d) can be used when several possible courses of actions are available and the policy author wants to specify multiple options.

E. Anchor Object

As noted earlier, each CIM-SPL policy is written under the scope of a single CIM Object called the *anchor object*. Note that the anchor object itself can be composite, e.g., a host with many Host Bus Adapter (HBA) cards in it. If a policy involves more than one CIM Object, then we require that the policy be written against a single CIM Object that represents the relationship between these objects. For example, a policy that constrains how two routers may be connected should be written against the link between the two routers. The principle here is the following: if there is a policy that references two objects that do not have a CIM association, then either the association should exist and is missing, or there is no single entity that has jurisdiction over both objects, and thus there is no entity able to author such a policy. The concept of anchor object is very useful when evaluating and analyzing nested policy subgroups. In CIM-SPL semantics, the outermost policy group is applicable to a single anchor object. An inner subgroup is applicable to a set of anchor objects[11] reached by traversing an association from the anchor object of the outer subgroup. Thus recursively we can build a tree-like evaluation structure that is easier to evaluate and analyze (for example, for potential conflicts with other policies). We note that the notion of an anchor object may not be suitable for every policy system.

F. Data Gathering for Policy Evaluation

1) Data gathering mechanism: Certain domain-specific policy languages assume a model of how the data required for policy evaluation will be gathered. For example, XACML prescribes a specific syntax for access requests and requires that data relevant for evaluating an access policy be present within the access request. Such assumptions on the data availability would not work well for a general purpose systems management policy language.

Our basic principle is that policy specification should be independent of how the data required for policy evaluation is gathered. Instead, when a policy is installed or activated in a policy server, the policy server should identify (by parsing the policy) what data is required for policy evaluation, and prompt the installer for mechanisms for gathering that data. There are myriad possibilities for what these data gathering mechanisms may be: data may be supplied by the evaluation trigger, the server may use SNMP to gather data on-demand from devices, or servers could query a configuration management database (CMDB), etc.

2) Prefetching and caching data: In a round of policy evaluations, a policy may be evaluated for several thousand devices. Components of a policy (for example, when evaluating a policy group) may be evaluated for a subsystem of the device under consideration, e.g., a storage-area-network (SAN) policy for a host may specify a sub-policy that needs to be evaluated for all Host Bus Adapters (HBAs) on that host. Thus the amount of data needed for a round of policy evaluation could be quite large. In case the data is not readily available at the policy server (e.g., it needs to be fetched from a CMDB, or directly from devices), a round of policy evaluations may spend a large amount of time in gathering data.

In many such instances, it is possible to pre-fetch and cache at least part of the data required for policy evaluation at the policy server, thus expediting policy evaluation. Depending on how frequently and when data changes, it may even be possible to cache data for multiple rounds of policy evaluation and reduce data-gathering overhead.

To help with data pre-fetching and caching, CIM-SPL specifies conditions in two places: the first condition (optional) is included in the import statement of a policy rule or policy group, and the second condition is specified as the condition of the policy rule. It is expected that the optional condition included in the import statement would only include properties that do not change, or change infrequently. For the file system example considered earlier, the policy author may want to exclude a LocalFileSystem whose ReadOnly attribute is set to true from the data-gathering efforts. In CIM-SPL, the policy author can specify this in the import statement as follows:

import CIM_· · · ::CIM_LocalFileSystem:(ReadOnly==false);

Due to this hint, an implementation may exclude all read-only file systems from the data-gathering activity, and save unnecessary cycles spent in gathering data from (a potentially large number of archival) file systems whose size does not change. Other examples are: excluding devices made by a particular vendor, or devices manufactured before a certain date, etc. Also note that in the file system example, the implementation may cache the FileSystemSize, leaving only the AvailableSpace to be gathered at policy evaluation time.

To the best of our knowledge, CIM-SPL is the first policy language to focus on the efforts of data gathering by breaking the condition explicitly into two parts. In particular, in CQL,

[11]The inner subgroup can be applied to each anchor object in this set in parallel.

the WHERE clause specifies the condition in totality making it much harder to exclude devices from data gathering.

VII. Conclusion

We have provided an introduction to the DMTF proposed standard CIM-SPL, a policy language that complies with the CIM Policy model and fully incorporates CIM constructs. We provided the rationale for many design choices in CIM-SPL and compared these choices with those taken in other well-known policy languages. The value proposition for a policy based management can be fully realized when policies can be analyzed to draw meaningful conclusions, assuming that their implementation is correct. Policies may be analyzed to gauge the impact of changing one or more existing policies, introducing a new policy, or to see if policies prescribe system behavior for a certain range of operating environment characteristics. To enable such analysis, CIM-SPL is a declarative policy language.

A key design decision for CIM-SPL was to separate event monitoring, data gathering, policy evaluation, and workflow execution concerns. It is our experience that while these concerns are intertwined one concern should not dictate how the other is specified or implemented. Indeed, there are several research and standards groups that work, for example, exclusively on event and workflow specifications. The richness and variety of such specifications convinced us that this design decision was the right choice for a policy language.

There are some issues that we did not discuss fully in this paper. We will briefly mention two of these issues here. First, each policy has a life-cycle that starts when a policy is first drafted, and ends when a policy is de-activated and is archived (for future auditing). It is tempting to provide constructs that would help in life-cycle management chores, e.g., providing a version field (and other meta-data). In this regard, we view policies *at par* with software. The policy life-cycle can be managed just as the software life-cycle is managed, i.e. by developing programming environments.

Second, using policy based management in large distributed environments often invokes concerns of synchronization of policy execution. If different parts of a distributed environment were to use different versions of management policies, then the results could be unpredictable. When and how to synchronize policies is an important issue, however, we believe that this falls outside the scope of a policy language discussion.

Although the ultimate validation of a language will be demonstrated by the degree of its adoption, there are several early indications that endorse the usability of CIM-SPL. First, most of the condition expressions in CIM-SPL comes from the Autonomic Computing language ACPL [3] that has been developed by IBM to enable self-managing computing systems. An implementation of a policy system incorporating ACPL has been extensibly tested and applied in various domains in distributed system management. For example, the concept of association traversal has been used in a policy-based configuration checker for storage area network management [18]. Second, CIM-SPL was reviewed by the members of the DMTF policy group during a period of about eight months to make sure it fit the CIM's needs. The value of the work is also reflected in an effort that has commenced on an open source implementation of a CIM-SPL policy engine for the OpenPegasus CIMOM [19]. The preliminary standard of CIM-SPL has been approved by the DMTF and we expect this will lead to another round of introspection of, and debate about, our design choices.

Acknowledgement

We thank Ric Telford and David Kaminsky for supporting this work. Andrea Westerinen helped us preparing the initial draft of CIM-SPL and joined our submission to the DMTF. Kamesh Aiyer did an extensive review of the draft, and provided valuable feedback. Numerous discussions held in the DMTF Policy Group and the IBM Policy Working Group helped us in forming our thoughts. We also thank the Tivoli team that implemented ACPL for germinating many ideas explored in this paper. Research was partially sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation heron.

References

[1] Agrawal, D., Giles, J., Lee, K.W., Lobo, J.: Policy ratification. In: 6th IEEE Policy Workshop. (2005) 223–232
[2] Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder policy specification language. In Lobo, J., Lupu, E., Sloman, M., eds.: Proc. of the 2nd Int. Workshop on Policies for Distributed Systems and Networks. Volume LCNS 1995. (2001) 18–38
[3] I.B.M.: Autonomic computing policy language. http://dl.alphaworks.ibm.com/technologies/pmac/acpl.pdf (2005)
[4] OASIS: Extensible Access Control Markup Language. (2004)
[5] Lobo, J., Bhatia, R., Naqvi, S.: A policy description language. In: Proc. of AAAI, Orlando, FL (1999)
[6] DMTF: CIM Query Language Specification. Version 1.0.0h edn. (2006)
[7] DMTF: CIM Simplified Policy Language (CIM-SPL). V 1.4.5 edn. (2006) http://www.dmtf.org/apps/org/workgroup/policy/.
[8] Masullo, M., Calo, S.B.: Policy management: An architecture and approach. In: Proceedings of the IEEE First International Workshop on Systems Management, Los Angeles, Ca., U.S., IEEE Computer Society (1993)
[9] Moore, B., Ellesson, E., Strassner, J., Westerinen, A.: Policy core information model (PCIM) version 1 specification (2001) Request for Comment 3060, Network Working Group.
[10] Strassner, J.: Den-ng: achieving business-driven network management. In: NOMS, IEEE/IFIP (2002) 753–766
[11] DMTF: Specification for CIM Operations over HTTP. Version 1.2 edn. (2004)
[12] SNIA: SNIA Standard - Storage Management. Version smi-s 1.0.0, rev 2 edn. (2005)
[13] Olshefski, D., Nieh, J.: Understanding the management of client perceived response time. SIGMETRICS Perform. Eval. Rev. **34** (2006) 240–251
[14] Carzaniga, A., Rosenblum, D.S., Wolf, A.: Design and evaluation of a wide-area event notification service. ACM Trans. Comput. Syst. **19** (2001) 332–383
[15] DMTF: CIM Event Model White Paper. Version 2.7 edn. (2003)
[16] Calo, S., Lobo, J.: A basis for comparing characteristics of policy systems. In: 7th IEEE Policy Workshop, Canada (2006) 183–192
[17] van der Aalst, W., Hofstede, A.: Yawl: Yet another workflow language (2002)
[18] Agrawal, D., Giles, J., Lee, K.W., Voruganti, K., Filali-Adib, K.: Policy-based validation of SAN configuration. In: IEEE Policy 2004. (2004)
[19] The Open Group: OpenPagasus: C++ CIM/WBEM manageability service broker. http://www.openpegasus.org (2006)

View publication stats