

# Policy-Based Validation of SAN Configuration

Dakshi Agrawal\* James Giles\* Kang-Won Lee\* Kaladhar Voruganti† Khalid Filali-Adib‡  
\* IBM T. J. Watson Research Center † IBM Almaden Research ‡ IBM Austin Research  
{agrawal,gilesjam,kangwon,kaladhar,filali}@us.ibm.com

## Abstract

*Historically, storage has been directly connected to servers for fast local access and easy configuration. In recent years, storage area networks (SANs) have defined an alternative storage paradigm that allows storage to be shared among servers using fast interconnects. One of the key challenges of SAN management is the large number of configuration problems that are encountered in a typical SAN deployment. These configuration problems can be addressed by SAN management software. However, hard-coding the SAN configuration rules into the management software is not a viable option since it is not possible to easily modify or replace old configuration rules and specify new policies and guidelines. In this paper, we propose a novel policy-based SAN configuration validation system that can be used to specify, store, and evaluate configuration policies for SANs. We also introduce five new operators for collection policies that are useful for evaluating a wide variety of practical SAN configuration policies found in practice. The policy-based SAN configuration checking approach proposed in this paper is discussed within the context of device interoperability constraints. However, this approach is extensible as it can also be used to enforce performance, reliability, and security-related configuration constraints.*

## 1 Introduction

Historically, storage has been connected to servers in a direct attached storage paradigm. However, there is a limit to the amount of storage that can be added to a particular server, implying that increasing storage sometimes requires adding servers even if extra computing power is not needed. Another limitation of direct attached storage is that it is difficult to share storage among multiple servers because each server has to route its data requests for remote storage through the remote server associated with that particular storage. Storage area networks (SANs) define an alternative storage paradigm that allows storage to be shared

among servers such that storage purchasing and server purchasing are decoupled. The dominant SAN protocol, Fiber Channel (FC), does not suffer from the distance and device connectivity limitations of the parallel SCSI transport protocol used for direct attached storage, and it is much more mature than competing network attached storage protocols such as the iSCSI transport protocol.

However, one of the key challenges of FC SAN management is the large number of configuration problems that are encountered when (a) adding/removing devices in an existing SAN, (b) provisioning new storage, (c) upgrading firmware or device drivers, and (d) trying to ensure a certain level of reliability and security in the SAN. System administrators spend many hours troubleshooting SAN configuration problems.

SAN configuration problems are typically caused by one or more of the following reasons:

- Absence of proper device drivers with the correct level of functionality.
- Connection of incompatible devices.
- Incorrect zoning of devices.
- Configuration of the SAN that does not satisfy the reliability, performance, and security requirements.

The above SAN configuration problems can be avoided by using SAN management software. SAN management software queries the SAN devices (both hardware and software components) to determine their status, configuration, and properties. Constraints can be encoded into the SAN management software that will allow it to detect configuration problems.

Hard-coding the SAN configuration constraints into the management software, however, is not desirable due to the following reasons:

- Newer versions of SAN software and hardware components have different interoperability track records. Thus, management software must be able to handle changes in existing SAN configuration rules.

- Changing business guidelines may require activation of new rules or deactivation of existing rules.
- Different actions may need to be taken when a particular rule is violated.
- In systems with thousands of rules, it is difficult for a human to determine conflicts between rules, assign priorities, and specify the sequence of execution of the rules.

Thus, it is necessary to enhance existing SAN management programs with policy management functionality to address these shortcomings of manual or hard-coded SAN configuration checking. Furthermore, it is also necessary for SAN management software to download the latest SAN configuration policies from a centralized repository (similar to anti-virus scan downloads that are located at a remote site). The policies defined in the centralized repository can be informed by input from field technicians, interoperability lab reports, and manufacturer product specifications.

In this paper, we propose a framework for policy-enabled SAN management software to address the SAN configuration checking problems. In the proposed framework, the configuration of a SAN is scanned periodically, or whenever a hardware or software component change occurs, or when the system administrator explicitly invokes a checking process. Alternatively, a detailed plan of a to-be-deployed SAN can be used as input so that the architect of a SAN may discover configuration errors during the design phase before actually deploying the plan. This configuration is dumped into a SAN Database in a canonical data format for consumption by the Policy Event Generator. Based on the changes in the SAN configuration, the Policy Event Generator creates new SAN policy events to trigger the Policy Evaluation process. If, during the evaluation process, a configuration violation has been detected, the corresponding actions to correct the error will be invoked.

This paper presents a design of the proposed SAN management architecture detailing the role of each component and the interactions between the modules. We also show that the composite nature of the SAN structure and the corresponding policy scopes can be exploited to build an efficient evaluator. Furthermore, we introduce five new operators for collection policies that are useful for evaluating a wide variety of practical SAN configuration policies defined by field experts. While in this paper we only address the issue of checking device interoperability constraints, our approach can easily be extended to enforce performance, reliability, and security related configuration constraints.

The remainder of this paper is organized as follows: Section 2 provides a brief introduction to SANs and presents a few example policies. Section 3 describes the overall architecture of the proposed SAN management system. Section

4 presents the policy model and collection policy operators. Section 5 presents details for components of the architecture. Section 6 presents work related to our paper. Section 7 discusses our conclusions and future work. Finally the appendix presents the 64 SAN configuration policies that we collected from SAN administrators and used in our design and prototype implementation.

## 2 Background

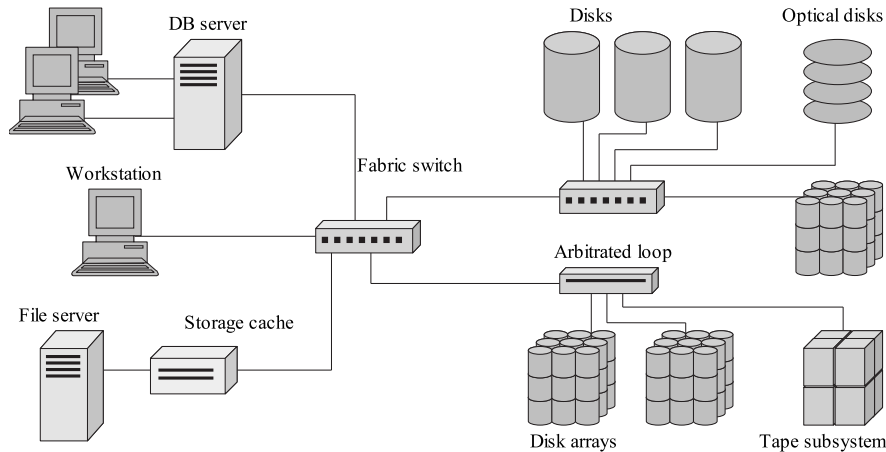
This section provides a simple introduction to storage area networks and illustrates some example policies used for configuration validation of SANs. Storage area networks provide interconnection between servers and consolidated storage devices. Current storage area networks are predominantly based on the Fiber Channel (FC) protocol, which provides reliable transport of storage I/O at gigabit speeds.<sup>1</sup>

Figure 1 shows a typical SAN configuration consisting of hosts (servers) and various types of storage devices. Hosts access the fabric via host bus adapters (HBAs), which play a role similar to network adapters and allow hosts to connect to the storage network. Similarly, storage devices are connected to the fabric via storage controllers. The fabric can include one or more of the following components: point-to-point link, arbitrated loop, and fabric switch. Point-to-point link provides a dedicated connection between two end-points (hosts, storage devices, or switches). Arbitrated loop is a shared media where multiple end-points connect in a logical ring topology. The operation of the loop is similar to that of the token ring; an end-point must first acquire a token before it sends data. The fabric switch provides data transport at the switched bandwidth of 100MBps per port. Additionally the fabric switch provides configuration capabilities including various types of zoning. By setting up zones, a SAN administrator can define an exclusive group of ports so that only permitted hosts can access certain storage devices.

Below we present some sample SAN configuration policies, which represent best practices for interoperability, performance, and security. In the appendix, we provide a comprehensive list of SAN configuration policies that have been defined by field experts.

- The same HBA cannot be used to access both tape and disk devices.
- No arbitrated loops are allowed in the FC fabric.
- The fabric cannot contain HBAs having a version below a certain level.

<sup>1</sup>Depending on types, the raw data rate of FC is either 1 or 2 Gbps. After the 8b/10b data encoding the actual data bandwidth of a 1 Gbps FC is about 100MBps.



**Figure 1. A typical storage area network configuration**

- A host needs to have at least two disjoint paths to each storage device for reliability.
- The number of hops between the host and the storage device should be no more than  $N$ .
- All HBAs in a host logical partition must be of the same type.
- There cannot be more than 3 servers in an arbitrated loop.

As one can see, these configuration rules are easy to understand and describe in plain English. However, SANs are complex systems, consisting of multiple servers each with multiple HBAs, servers and HBAs with many software and firmware components, storage devices of multiple varieties with many different configuration parameters, and a SAN fabric consisting of various types of fabric switches, point-to-point links, and arbitrated loops. Because of this inherent complexity, manually validating these rules is a nightmare to the SAN administrator for even a modest-scale storage network. Furthermore, manually fixing the configuration problem is an error-prone process during which new errors may be introduced. The objective of this paper is to provide a policy framework that can automate the process of validating and enforcing the policies of the above type so that the SAN management can be greatly simplified.

### 3 Architecture Overview

In this section we provide an overview of the proposed SAN configuration management architecture as illustrated in Figure 2.

The SAN configuration management system is used to check the configuration of a SAN, which can either be a

physically deployed SAN or planned SAN. For a physically deployed SAN, management agents query the SAN fabric and store the configuration information in a SAN Database using a format consistent with the Storage Management Initiative Standard (SMI-S) representation. For a planned SAN, the SAN planning software exports the designed configuration to a SAN Database.

The SAN configuration management system has a Policy Specification Tool that is used by a system administrator to author system configuration policies. The graphical user interface allows the author to build policies by combining system attributes with functions provided by the policy language. The policies in the SAN manager are specified in terms of a four-tuple: scope, condition, action, and priority, and are similar to those used by the PCIM specification [13]. Scope is used to group policies into sets that are applicable to particular system components, such as ports or HBAs, and identifies the set of attributes that may be included in the policy condition. Condition is a logical expression written on the attributes which determines if a policy is applicable for particular values of the attributes. Action describes the steps that should be taken by the system if the policy is applicable. Priority describes the relative importance of a policy (higher priority policies are chosen over lower priority policies when both the policies are applicable). The Policy Specification Tool stores authored policies in a policy repository.

The two types of policy repositories used by the SAN Manager are the Remote Centralized Policy Database and the Local Policy Database. The Remote Centralized Policy Database is a central repository that stores the configuration policies for many SAN managers. A distribution mechanism pushes policy updates from the Remote Centralized Policy Database to each SAN manager's Local Pol-

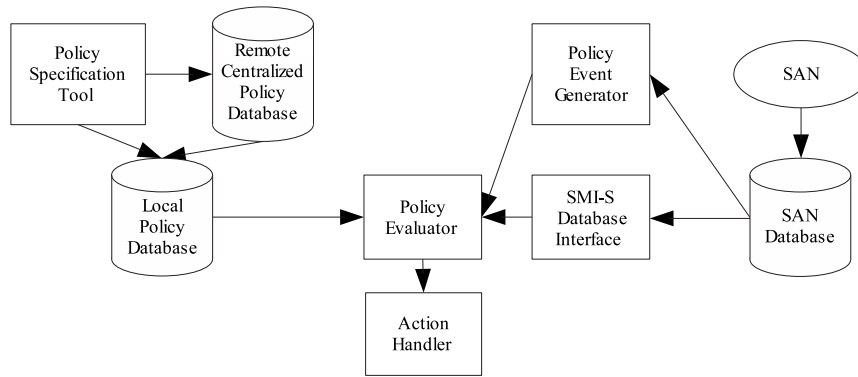


Figure 2. SAN management system architecture

icy Database. The Local Policy Database is co-located with an instance of a SAN manager and stores the configuration policies for that SAN manager. It can receive policies from a Remote Centralized Policy Database, or policies can be defined directly for the Local Policy Database by a system administrator using a Policy Specification Tool.

The Policy Event Generator component monitors changes to the SAN configuration by listening for changes in the SAN Database. When there is a change to an element in the SAN, the Policy Event Generator triggers an event and identifies the policy scopes that are applicable for the event. For example, a change to Fiber Channel port could trigger an event that would be relevant to port and host policies. The Policy Event Generator passes the applicable scopes to the Policy Evaluator for further processing. The Policy Event Generator can also be used by a system administrator to trigger configuration checking for subsets of the SAN, or the Policy Event Generator can be used by additional components to trigger configuration checking.

The Policy Evaluator component takes a list of scopes from the Policy Event Generator and selects the policies that are applicable to the scopes. To evaluate the selected policies, the Policy Evaluator identifies the SAN attributes referenced by the policies and requests those attributes from the SMI-S Database Interface. The gathered attributes are used to compute the condition of each of the selected policies. Among policies for which the condition evaluates to true, if any, the highest priority policy is selected. The action for the selected policy is passed to the Action Handler.

The Action Handler component takes actions passed from the Policy Evaluator, and invokes the specified actions. For example, the action can specify that an e-mail message be sent to a system administrator if the SAN configuration violates certain constraints. An action may also specify a function that can rectify a system configuration problem, such as performing a rezoning action when there is a zone configuration problem.

The SMI-S Database Interface collects system configuration attributes from the SAN Database or directly from the physically deployed system. It provides an abstraction for accessing system data so that the Policy Evaluator does not need to know the specifics of the SAN Database or how to query the SAN fabric.

## 4 Data and Policy Model

The typical size of SAN configuration data can easily run into several megabytes. Furthermore, there could be hundreds of configuration policies in the policy repository. A good organization of SAN configuration data and policies is critical for the good performance of the validation system.

### 4.1 Policy Scope Taxonomy

The first step toward data and policy modeling is to collect typical configuration policies and analyze them for the attributes that they validate. Using such policies, we modeled data for SAN components and major concepts such as host, switch, storage device, host bus adapter (HBA), port-controller, port, link, fabric, and zone. Our data models were inspired by the SMI-S standards, and they represent a subset necessary for the policy evaluation. For each given component type, we divide corresponding data into two parts: data describing *intrinsic properties* and data describing *associated components*. For example, for an HBA, its serial number, state, id, model name, vendor name, installed software, and hardware characteristics are defined to be its intrinsic properties. On the other hand, the ports contained in it, and the host in which the HBA is contained are defined to be the components associated with an HBA. This distinction of data for a component type turns out to be quite useful, since a close inspection of configuration policies reveals that there are three types of policies:

- **Intra-component Policies:** policies that validate constraints on the intrinsic properties of a single SAN component. An example policy is “An HBA from vendor  $X$  should have firmware level greater than  $Y$ ”. This policy may have been created because firmware levels less than or equal to  $Y$  for HBAs of vendor  $X$  are known to have a bug leading to sporadic data losses.
- **Inter-component Policies:** policies that validate intrinsic properties of one SAN component with respect to the intrinsic properties of another SAN component. An example of policy is “A host with operating system AIX version  $X$  should not have an HBA from vendor  $Y$  with firmware level greater than  $Z$ . This policy may be in force if the OS has only been certified up to a certain HBA firmware level.
- **Collection Policies:** policies that validate properties of a group of SAN components. An example policy is “All HBAs in a host should be from the same manufacturer”. This policy may be enforced for reliability and interoperability reasons.

Based on our analysis of SAN components and policies we developed a policy scope taxonomy. In this classification, policies are first divided on the basis of the type of component they validate. Thus, a policy validating intra-HBA constraints would fall under the scope “HBAPolicies”. A policy prescribing interoperability constraints between host OS and HBA firmware level would carry two scopes “HBAPolicies” and “HostPolicies”. A policy mandating uniformity among HBAs of the same host would fall under “HBAPolicies”. For each component type, the policies are further subdivided based on the three type of policies described above. Thus, all “HBAPolicies” may be divided into the following scopes, for instance:

- **IntraHBAPolicies.** Policies that validate constraints on the intrinsic properties of a single HBA.
- **InterHostHBAPolicies.** Policies that validate constraints between a host and a resident HBA.
- **InterHBAPortPolicies.** Policies that validate constraints between an HBA and a resident port.
- **HostHBACollectionPolicies.** Policies that validate constraints among all HBAs of a host.
- **ZoneHBACollectionPolicies.** Policies that validate constraints among all HBAs of a zone.
- **GeneralHBACollectionPolicies.** Policies that validate constraints among all HBAs in a SAN.

The advantage of making these policy subgroups will become clear in the next section when we describe architecture components in more detail. Essentially, this scope taxonomy allows us to build efficient policy evaluation for a given SAN event.

## 4.2 Collection Policies

In general, policies can be classified on the basis of the condition part of their four-tuple. At the first look, it may seem the conditions in the SAN configuration policies presented in the appendix do not present a structure for parsing and evaluation. However, more insight can be gained by expanding the Boolean expression representing the condition of a policy as an expression tree. We found that the expression tree of policy conditions mostly contains basic operators such as logical composite operators (AND, OR, NOT) and comparison operators ( $>$ ,  $<$ ,  $\geq$ ,  $\leq$ ,  $==$ ) with the five notable exceptions. All five exceptions are operations on a collection of elements. In the following, we describe these new operations.

Assume that a collection  $C$  has elements  $O_1, O_2, \dots, O_n$ . Further assume that each element in  $C$  has  $m$  attributes  $p_1, \dots, p_m$ . Now consider the following five operations on the collection  $C$ .

- **Cartesian Property:** Given sets of values  $A_1, \dots, A_m$  for attributes  $p_1, \dots, p_m$  respectively, return all elements  $O_i$  in  $C$  that satisfy the condition  $(O_i.p_1 \in A_1) \wedge \dots \wedge (O_i.p_m \in A_m)$ .  
– *Example Policy:* All HBAs of type Emulex 9002 must have firmware level of either 3.81a, 3.81b, 3.82, or 3.84.
- **Graph:** Given a directed graph  $G = (E, C)$  (graph with elements in  $C$  as its vertices, and directed edges in  $E$  between them), and two elements  $O_i$  and  $O_j$ , return all directed paths between  $O_i$  and  $O_j$ .  
– *Example Policy:* All hosts must have more than  $k$  paths to target.
- **Exclusion:** Given sets of values  $A_{1,1}, \dots, A_{1,m}$  and sets of values  $A_{2,1}, \dots, A_{2,m}$ , for attributes  $p_1, \dots, p_m$  respectively, return all elements  $O_i$  in  $C$  that satisfy the condition  $(O_i.p_1 \in A_{1,1}) \wedge \dots \wedge (O_i.p_m \in A_{1,m})$  while another element  $O_j$ ,  $j \neq i$ , simultaneously satisfies  $(O_j.p_1 \in A_{2,1}) \wedge \dots \wedge (O_j.p_m \in A_{2,m})$ .  
– *Example Policy:* Tape drives should not exist in a zone if it contains disk drives.

Cartesian	(1) (2) (4) (5) (6) (7) (11) (12) (15) (18) (19) (20) (21) (22) (23) (24) (25) (26) (27) (30) (31) (32) (33) (34) (35) (36) (37) (38) (40) (41) (42) (43) (47) (48) (49) (51) (52) (53) (54) (56) (57) (58) (60) (61) (62) (63)
Graph	(3) (15) (16) (17) (22) (23) (26) (34) (35) (51) (54) (57) (60) (61) (62)
Exclusion	(8) (28) (29) (50)
Many-to-One	(9) (10) (14) (39) (44) (45) (46) (59) (64)
One-to-One	(55)

**Figure 3. Mapping of the SAN configuration policies to collection operators**

- Many-to-One:** The value of an attribute  $p_i$ ,  $1 \leq i \leq m$  should be the same for all elements in  $C$ . If this is not the case, then return subsets of  $C$  constructed by partitioning  $C$  according to the values of  $p_i$ .
  - *Example Policy:* Only one host type should exist in a zone.
- One-to-One:** The value of an attribute  $p_i$  should be different for all elements in  $C$ . If this is not the case, then return subsets of  $C$  constructed by partitioning  $C$  according to the values of  $p_i$ .
  - *Example Policy:* No two devices in the system can have the same WWN (World-Wide Name).

In the appendix, we list 64 types of actual SAN configuration policies collected from field experts. Figure 3 shows how each policy can be modeled and expressed using the above five collection operators. The figure shows that the Cartesian Property operator is most popularly used and therefore the policy evaluator must implement it in an efficient manner.

## 5 Architecture Details

This section now describes the details of each component of the architecture.

### 5.1 Local Policy Repository

The design of the local policy repository has been driven by two critical requirements: First, the repository should be able to quickly retrieve all policies relevant to a Policy Request and pass it to the Policy Evaluator. Second, the repository should provide standard version control features: the ability to maintain multiple versions of policy documents,

the ability to roll-back to an older version of a policy document, and the ability to maintain a coherent set of policies to the Policy Evaluator in the presence of concurrent modifications to the policy documents.

These requirements are met by implementing the policy repository in two parts: a persistent part that stores multiple versions of policy documents and resides in a concurrent version system, and a policy cache that stores the currently enforced policy document in a compiled format and resides in the random-access memory of the SAN management system. The persistent part maintains a history of policy documents and a coherent set of policies in the presence of multiple authors trying to modify the policy document. The cache stores policies in a hash table, hashed by their policy scope. If the policy scopes are chosen carefully, then each Policy Event only triggers policies corresponding to a small number of scopes, and the policy evaluator can query the cache and efficiently retrieve all policies in a compiled format.

Upon receiving a new policy document, the persistent part of the policy repository validates the syntax of the new policy document. If the document passes validation checks, then it notifies the policy cache. The policy cache retrieves the most recent policy document, compiles it to an expression tree suitable for policy evaluation and stores policies contained in the document in a temporary hash table. Once the compilation is finished, it switches to the newly compiled policies.

### 5.2 Policy Evaluator

The Policy Evaluator operates on the boolean expressions that represent the policy conditions. Boolean expressions are compiled into the form of an expression tree, with the nodes being the operators and the children nodes being the operands. The operators include logical operators and comparison operators, as well as the five collection operators discussed in the previous sections. The evaluator computes the value of the expression in a depth-first traversal order. The Policy Evaluator is optimized for performance by caching sub-expressions that appear more than once in the expression trees. Such sub-expressions representing the policy conditions are evaluated only once, with the results being cached to avoid repeated evaluations.

The policy evaluator obtains the data needed to evaluate the expression tree from the SMI-S Data Layer which is described in subsection 5.4. When a request for policy evaluation of a given scope is triggered, the attribute values for the corresponding objects are obtained by the SMI-S Data Layer and are cached so that they do not have to be fetched during evaluation of the tree for policies corresponding to the scope and its sub-scopes.

### 5.3 Policy Request Generator

The Policy Request Generator is responsible for sending properly formatted requests for configuration checks to the Policy Evaluator. The Request Generator sends a request based on input from two sources: the user interface where a user can ask the request generator to run a specific configuration check, or the SAN configuration database, where a stored procedure can send a notification every time a configuration change takes places. Typically, a user asks to run a specific configuration check for routine maintenance or for trouble-shooting a configuration problem. Alternatively, some typical actions that would trigger a notification by the SAN configuration database to the Policy Request Generator are as follows:

- Addition/removal/failure of SAN components.
- Firmware upgrade/downgrade of SAN component.
- Zoning operations.
- Software configuration changes.

Once the Policy Request Generator receives a triggering event, it examines the event to see if a new request for configuration checking should be generated. For example, the configuration of a SAN may change during business hours to provide more resources to business customers. The Policy Request Generator in our architecture filters out such triggering events. The filtering of events in the request generator is governed by meta-policies that are input by the system administrator. These meta-policies are stored in a local policy repository under a scope.

If the trigger event qualifies, the Policy Request Generator generates a corresponding request for configuration checking. The request for configuration checking includes a policy scope to indicate all the policies that need to be checked, and the identification of affected system component. For example, if the triggering event is an upgrade of the firmware of an HBA, then the attached scope would include HBAPolicies (policies checking configuration of a single HBA) and the request for checking configuration policies would carry the globally unique (GUID) of the affected HBA.

### 5.4 SMI-S Data Layer

Depending upon the scope of the involved policies, the policy evaluator obtains the necessary policy data from the storage management initiative standard (SMI-S) data layer. SMI-S is a SNIA standard that uses DMTF CIM modeling technology to model both physical storage devices such as switches, storage arrays and hosts, and also logical storage components such as volumes, extents, LUNs etc. Since the

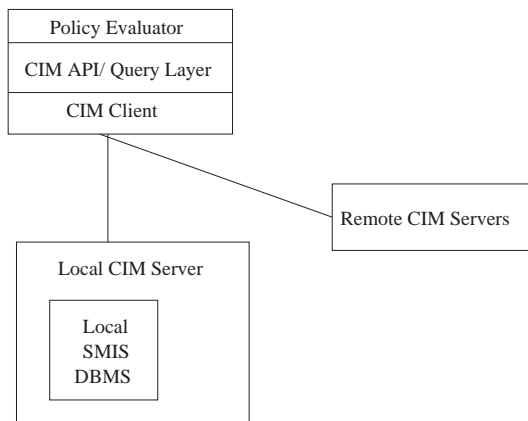


Figure 4. SMI-S Data Layer

SMI-S standard is being supported by most storage vendors, use of SMI-S entities and attributes in the policy framework to represent storage resources allows the policy framework described in this paper to be used in a vendor-neutral manner. As shown in Figure 4, the policy evaluator layer invokes the CIM-SMI-S APIs that could potentially be accessing data from a database management system (DBMS) under the covers, or could be directly contacting the storage devices. In either case, the SMI-S data layer maintains a cache of objects at the policy manager location to ensure that the same data is not retrieved multiple times if it is needed by multiple policies.

When the SMI-S data layer is utilizing a DBMS, it relies on the DBMS cache manager to provide the caching functionality and the CIM API/Query language layers leverage the SQL querying capability of the underlying DBMS to query the SMI-S database. The policy evaluator layer can potentially directly access (instead of CIM APIs or query language) the data from the DBMS but then the code will not be portable for cases where the data is being retrieved from remote storage devices and not from a local DBMS. As described above, use of the CIM API layer insulates the policy management software from the location details of where the information is stored (i.e. data could be located at remote devices).

Figure 5 shows the entities of the SMI-S schema that map to some of the key storage devices. ComputerSystem, PhysicalPackage, SoftwareElement, and FCPort are some of the SMI-S classes that are used in the SAN configuration checker. The ComputerSystem class represents a switch, a storage array, or a host. The PhysicalPackage class represents the hardware characteristics (manufacturer, model number) of a ComputerSystem, while the SoftwareElement class corresponds to software characteristics (firmware level, OS level, device driver level) of a ComputerSystem. It is important to note that the ComputerSystem



Systems like SAN Architect [11] from EMC and BrightStor SAN Designer [7] from Computer Associates also provide support for SAN configuration checking. However, to our knowledge these products do not utilize a policy-management framework similar to the one presented in this paper. Most of the current storage management software products contain only very primitive support for policy-based storage management because it is difficult for them to keep up with changing workloads and system configurations. Thus, new techniques are required to make it easier to express transformation from high level goals into low level system actions. Similarly, sophisticated policy conflict resolution methods need to be developed to detect both static and dynamic policy conflicts. Currently, there is support only for very rudimentary policy conflict detection mechanisms.

The SAN configuration checking framework presented in this paper can use any one of the existing policy specification mechanisms. That is, this approach is compatible with DMTF CIM Policy [10], IETF PCIM [13], and Web Services policy specification [5] approaches. Similarly, this framework can utilize any one of the many different policy execution engines such as the ABLE [3] or the Hypercube [2] policy evaluation engines. Finally, the configuration checking framework presented in this paper uses the SNIA SMI-S model by default, but it can also be extended to work with other proprietary data models.

## 7 Conclusion

In recent years, policy-based management and configuration validation has been proposed for various complex systems. In this paper, we have presented an architecture for a policy-based SAN configuration manager that can be used to specify, store, and evaluate interoperability policies for SANs. By explicitly using policies to drive the system rather than hard-coded configuration rules, the system can more easily adapt to changes in SAN software and hardware components and actions can be customized or extended as the needs of the SAN manager change. Furthermore, the SAN management system can profit from policy-specific functions such as efficient evaluation, validation, and conflict resolution. The generic policy components of the proposed architecture including the Policy Evaluator, Policy Specification Tool, Policy Database, and Action Handler can be used in many different scenarios and application domains. We are in the process of building policy-based design and planning tools, auditing tools, and configuration tools based on these generic policy components for a wide variety of application domains such as databases, networks, and messaging systems.

## References

- [1] M. Bearden, S. Garg, and W. J. Lee. Integrating Goal Specification in Policy Based Management. In *Proc. of IEEE Policy 2001*, January 2001.
- [2] P. Bhattacharya, S. Kamat, R. Rajan, and S. Sarkar. Search tree for policy based packet classification in communication networks. U.S. Patent 6,587,466, July 2003.
- [3] J. P. Bigus, D. A. Scholsnagle, J. R. Pilgrim, W. N. Mills III, and Y. Diao. ABLE: A toolkit for building multiagent autonomous systems. *IBM Systems Journal*, 41(3), April 2002.
- [4] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. IETF RFC 2475, December 1998.
- [5] D. Box, F. Curbera, M. Hondo, C. Kaler, D. Langworthy, A. Nadalin, N. Nagaratnam, M. Nottingham, C. von Riegen, and J. Shewchuk. Web Services Policy Framework (WS-Policy) version 1.1. <http://ifr.sap.com/ws-policy/ws-policy.pdf>, May 2003.
- [6] R. Chadha, Y.-H. Cheng, T. Cheng, S. Gadgil, A. Hafid, K. Kim, G. Levin, N. Natarajan, K. Parmeswaran, A. Poylisher, and J. Unger. PECAN: Policy-enabled configuration across networks. In *Proc. of IEEE Policy 2003*, June 2003.
- [7] Computer Associates Corporation. Brightstor SAN designer. <http://www3.ca.com/Solutions/Product.asp?ID=4590>.
- [8] M. Devarakonda, J. Gelb, A. Saha, and J. Strickland. A policy-based storage management framework. In *Proc. of IEEE Policy 2002*, June 2002.
- [9] M. Devarakonda, A. Segal, and D. Chess. Toolkit-based approach to supporting storage policy templates. In *Proc. of IEEE Policy 2003*, June 2003.
- [10] DMTF. CIM Policy Model White Paper CIM Version 2.7. <http://www.dmtf.org/standards/documents/CIM/DSP0108.pdf>, June 2003.
- [11] EMC Corporation. SAN architect. [http://www.emc.com/products/software/san\\_architect.jsp](http://www.emc.com/products/software/san_architect.jsp).
- [12] Z. Fu, S. F. Wu, H. Huang, K. Loh, F. Gong, I. Baldine, and C. Xu. A policy-based storage management framework. In *Proc. of IEEE Policy 2001*, pages 39–56, January 2001.
- [13] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen. Policy core information model (PCIM) – version 1 specification. IETF RFC 3060, February 2001.
- [14] A. Sahai, S. Singhal, R. Joshi, and V. Machiraju. Automated Policy-Based Resource Management in Utility Computing. In *Proc. of IEEE/IFIP*, 2004.
- [15] D. Verma, M. Beigi, and R. Jennings. Policy Based SLA Management in Enterprise Networks. In *Proc. of IEEE Policy 2001*, pages 137–152, January 2001.

## Appendix

In this appendix, we provide a list of 64 general SAN configuration policies that we used for designing the extensions of the policy evaluation primitives.

1. HBA Model and Firmware Level: Every HBA that has a vendor name  $X$  and that has a model  $Y$  must have a firmware level either  $n_1$ ,  $n_2$ , or  $n_3$ .
2. Hardware Constraints: All host systems must be from vendor  $X$ .
3. Firmware Level: Firmware level of  $X$  must be upgraded to a version of at least  $n$  or higher.
4. Number of Ports per Zone: There must be no less than  $M$  ports and no more than  $N$  ports in a zone.
5. Number of Zones per Fabric: There must be no less than  $M$  zones and no more than  $N$  zones in a fabric.
6. Hard Zoning: All zones must be defined using domain or WWN.
7. Soft Zoning: All devices connected to an interconnect device must support name service to do soft zoning.
8. Device Selection in a Zone: Devices of type  $X$  and devices of type  $Y$  are not allowed in a same zone.
9. Host Selection per Zone: No two different host types should exist in a same zone.
10. HBA Selection per Zone: No two host HBAs should exist in a same zone.
11. Number of Zones per Port: No port should be a member of more than  $N$  zones.
12. Port Status in Active Zone: If a port has a good status, it must be in at least  $M$  zones and at most  $N$  zones.
13. Number of Ports in Fabric: The number of ports of type  $X$  in the fabric is less than  $N$ .
14. Domain IDs: All domain IDs throughout all monitored FC switches must be unique.
15. Equal Link Speed: FC link speed must be equal for all HBAs in system  $X$  connected to device type  $Y$ .
16. Logical Multiple Paths: A host can logically see multiple paths to target.
17. Physical Multiple Paths: A host physically has multiple paths to target.
18. System Type: All systems connected to SAN  $X$  must be of type  $Y$  or  $Z$ .
19. Port Type/System Type: All devices of type  $X$  must be connected to switch port of mode  $Y$ .
20. Tape and Port Loop Mode: All tape connections are loop.
21. G-Port Policy: No G-Port/In-Sync connections are allowed in the fabric.
22. Device Hop Constraints: There are no more than  $N$  hops between devices  $X$  and  $Y$ .
23. Switch Hop Constraints: There are no more than  $N$  hops between switches.
24. Environmental thresholding: Certain environmental parameters have to be within a specified range. For example, the room temperature must be between  $M$  and  $N$  degrees.
25. Host Hardware Profile: All fabric host hardware matches a user-defined profile.
26. Redundant Paths: There should be at least  $M$  and at most  $N$  paths to a device.
27. Active Zones: There should be no active zones having only a single port.
28. Tape/Disk and Zone: No tapes and disk should exist in a same zone.
29. HBA with Tape/Disk Devices: A host bus adapter cannot be used to access both tape and disk devices.
30. Tape Recovery Protocol: The tape recovery protocol should be enabled/disabled on HBA  $X$  in location  $Y$ .
31. Class of Service: All devices must run in class of service  $N$ , where  $N = 2, 3$ .
32. Loop Disk: No loop disk is allowed in the fabric.
33. Optic Mode: If optic link is in single mode, the BB.Credit<sup>2</sup> must be between  $M$  and  $N$ .
34. Connection to a Director Class Switch: No end-point devices can connect directly to a director class switch except for device  $X$ .
35. Connection to a Director Class Switch: No edge switches can connect to a director class switch unless the switch vendors are the same.
36. Link Speed: A link capable of  $X$  speed has been set to  $Y$  speed.
37. Port Mode: The device connection method is set to loop instead of FC port.
38. BB.Credits: A device running in mode  $E$  should have BB.credits between values  $M$  and  $N$ .
39. HBAs and LPAR: All HBA cards must be of the same type within an LPAR (logical partitioning).
40. Number of LUNs: A storage Port should have minimum/maximum number of LUNs.
41. Port Capacity: A storage Port should have a minimum/maximum capacity of  $k$  GB.
42. Device Type and Vendor: Device type  $X$  of Vendor type  $Y$  is not allowed.
43. Open System: An ESS array is not available to open systems if an iSeries system is configured to array.
44. HBA and Vendor: All the HBAs in a system should be from the same vendor.
45. Storage Type: The system should not have mixed storage types such as SSA, FC, and SCSI parallel.
46. Firmware Level: All firmware levels must be equal for device type  $X$ .
47. OS and Device types: Devices of type  $T$  should not be attached to an OS of type  $X$ .
48. OS Levels and Patches: A desired OS level and a list of the patches must have been install.
49. File System Types: Only a certain type of file systems can be on the system.
50. Interconnects: Copper and optical interconnects are not allowed on same device or system.
51. Active LUNs: All HBA devices must have at least one active LUN associated with it.
52. HBA and LUN: Each HBA on a host system has active LUNs associated with them.
53. Optics and Vendor: Optics must be of type  $X$  and at version  $Y$  in a device type  $Z$  if the vendor is  $Q$ .
54. LUN Masking: Proper LUN masking should have been enabled for a storage device that is accessed by multiple systems.
55. WWN Uniqueness: No two devices in the system can have the same WWN (World-Wide Name).
56. Serial Number Range: A certain serial number range for component  $X$  is not allowed.
57. Logical Paths: No less than two and no greater than four logical paths are allowed from endpoint  $A$  to endpoint  $B$ .
58. Switch Interoperability: Only the switches that are inter-operable can work together in the system.
59. Active Zone: Every fabric must have at least one active zone.
60. Core-Edge Policy: All edge switches must be connected to one or more core switches.
61. Redundant Paths: There must exist redundant SAN paths between host  $H$  and endpoint  $D$ .
62. Tape Device: Tape devices of type  $X$  may have at most one connection to any host.
63. No Sharing: Each zone should only contain the ports from a single host.
64. Alias: All hosts in the system must have at most  $X$  aliases.

---

<sup>2</sup>Buffer-to-buffer credit is used for flow control by FC classes 2 and 3.