Configuration Checking and Design Optimization of Storage Area Networks

Dissertation

of the Faculty of Information and Cognition Sciences
at the Eberhard Karls University of Tübingen
to achieve the grade of doctor rerum naturalium

Eray Gençay

born in Balıkesir, Turkey

Tübingen
2009

Konfigurationsvalidierung und Entwurfsoptimierung von Speichernetzen

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität zu Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von

Eray Gençay (M.Sc.)

aus Balıkesir, Türkei

Tübingen
2009

Tag der mündlichen Qualifikation:	21. Juli 2009
Dekan:	Prof. Dr.-Ing. O. Kohlbacher
1. Berichterstatter:	Prof. Dr. sc. techn. W. Küchlin
2. Berichterstatter:	Prof. Dr. T. Grust

babamın anısına

(in memory of my father)

Contents

Acknowledgments **ix**

1 Introduction **1**
1.1 Motivation and Problem Description 1
1.2 Thesis Organization 3

2 Storage Area Networks **5**
2.1 Components of Storage Area Networks 5
2.1.1 Host Layer 5
2.1.2 Fabric Layer 6
2.1.3 Storage Layer 9
2.2 Protocols 12
2.2.1 FC-AL 12
2.2.2 FC-SW 13
2.2.3 SCSI 13
2.3 SAN Topologies 13
2.3.1 Point-to-point Topology 13
2.3.2 Arbitrated Loop Topology 14
2.3.3 Switched Fabric Topology 16
2.4 Management of Storage Area Networks 20
2.4.1 Zoning 20
2.4.2 Best Practices 20

2.4.3 SAN Management Frameworks 21
2.4.4 Service Level Agreements 22

3 SANchk SAN Configuration Checker 23
3.1 Scenarios for Configuration Checking 25
3.1.1 General Scenario 25
3.1.2 CIM-Based Scenario 26
3.1.3 SQL-Based Scenario 28
3.1.4 Comparison of CIMchk and SANchk 29
3.2 Architecture of SANchk 30
3.2.1 Overview 30
3.2.2 Evaluator: Boolean Tests as SQL Fragments 33
3.2.3 Data Related Aspects of the System 34
3.2.4 An XML Format for SAN Configuration Policies 34
3.2.5 Action Handling 37
3.3 Policy Implementation with SQL 38
3.3.1 Implementation Examples for "Collection Policies" 38
3.3.2 An Example for Action Handling 43
3.4 Empirical Data 45
3.4.1 Implementation Details 45
3.4.2 Initial Tests 47
3.4.3 Test on a Production SAN in Industry 48
3.5 Aperi Integration 48
3.5.1 Aperi Project 48
3.5.2 SANchk as Aperi Plugins 49
3.6 Related Work 52
3.6.1 Policy Based Validation of SAN Configuration by Agrawal et al. . . 52
3.6.2 CIM Constraint Language ($\mathcal{CCL}$) by Sinz et al. 54
3.6.3 Policy Management for Autonomic Computing (PMAC) by IBM . . 57
3.6.4 Policy Middleware Architecture for Managing IT Systems by Agrawal et al. 63
3.6.5 eXtensible Access Markup Language (XACML) 63
3.6.6 Ponder 68
3.6.7 Other Related Work 73

4 SANopt SAN Design Optimizer **75**
4.0.8 Methods and Technologies . . . 76
4.1 SAN Storage Assignment Problem . . . 77
4.1.1 Assignment of Applications to Storage Devices . . . 77
4.1.2 Constraint Blocks . . . 78
4.1.3 Optimization Problem . . . 79
4.1.4 Test of Preconditions . . . 81
4.1.5 Example . . . 81
4.1.6 Empirical Results . . . 82
4.2 SAN Connection Problem . . . 83
4.2.1 Encoding as a Pseudo-Boolean Problem . . . 85
4.2.2 Example . . . 87
4.3 Implementation in a SAN Management Framework . . . 89
4.4 Related Work . . . 91
4.4.1 Appia: Automatic Storage Area Network Fabric Design . . . 91
4.4.2 Using a Genetic Algorithm to Design and Improve SAN Architectures 93
4.4.3 An Ant Inspired Technique for SAN Design . . . 94
4.4.4 Chameleon . . . 95
4.4.5 Polus . . . 97
4.4.6 Other Related Work . . . 99

5 Conclusion **101**
5.1 Results . . . 101
5.2 Publications . . . 102

A Best Practices Rules Implemented **103**
A.1 Redundancy . . . 103
A.2 Zoning . . . 107
A.3 Uniqueness . . . 111
A.4 Fabrics . . . 112
A.5 HBAs . . . 113
A.6 Firmware level . . . 113
A.7 Vendor exclusion . . . 114
A.8 Serial number . . . 116
A.9 Connection restrictions . . . 118

A.10 Ports 120
A.11 Connections 122
A.12 Capacity 123

Bibliography **125**

Zusammenfassung **131**

Abstract **133**

Curriculum Vitae **135**

Acknowledgments

First of all, I would like to thank my supervisor, Prof. Dr. Wolfgang Küchlin, and Dr. Carsten Sinz for their guidance during my study years. The definition of the problems that were subject to this work was based on their previous work and on suggestions of the experts at IBM Storage Division in Mainz, Germany. Without the grant of the Center for Advanced Studies at IBM Böblingen, this work would not have been possible. I am also very grateful to my previous project leader at IBM Mainz, Thorsten Schäfer, and his successor for the last months of the project, Christoph Reichert, for their technical and organizational support as my advisors at IBM in Mainz, Germany. Last but not least, I thank my colleagues in the Symbolic Computation Group of the Wilhelm Schickard Institute for their valuable comments.

Eray Gençay
Tübingen
July 17, 2009

Chapter 1

Introduction

An idea is salvation by imagination.

Frank Lloyd Wright

1.1 Motivation and Problem Description

Storage Area Networks (SANs) connect servers to a separate network of mass storage devices over fast interconnects. Thus, storage resources can be provided and managed independent of compute servers (hosts), and they can be assigned to hosts in a flexible and scalable way. The flexibility in assignments enables a higher utilization of storage devices, since storage resources can now be shared between hosts. Storage size can also grow independent of the number of servers, thus providing scalability.

Another advantage of SANs in comparison to the former storage technologies is their speed. The SAN interconnects like Fibre Channel (FC) switches or Host Bus Adapters (HBAs) owe their high speed to the fibre optic transmission technology. The Fibre Channel protocol enables long distance transmission of data with high bandwidth, so that geographically distributed storage networks or disaster recovery solutions are feasible.

However, the SAN paradigm also provides some challenges. The combination of the diversity and the interoperability of the devices with the scalability of the network makes SAN configuration complex and thus error-prone.

In a large SAN, it is almost impossible for an administrator to check the SAN manually for configuration problems. Therefore, software systems for the validation of SAN configuration are needed. The realization of these kinds of systems requires a common

and structured representation of the SAN configuration. Otherwise, a configuration management system for SAN must be able to communicate through the proprietary interface of every single device. This requirement was fulfilled with the development of an object-oriented model named Storage Management Initiative - Specification (SMI-S) that standardizes the management interfaces of storage related devices and technologies [50]. SMI-S is developed by the Storage Network Industry Association (SNIA), which was established by storage industry vendors. SMI-S is based on the standard Web-based Enterprise Management (WBEM) from the Distributed Management Task Force (DMTF) [20]. WBEM has been developed for the web-based management of enterprise information system infrastructures. WBEM consists of three main components. It uses the Common Information Model (CIM) [21] to model the management data, the xmlCIM encoding to represent the CIM classes and objects in XML format, and HTTP for the transport of CIM operations. SNIA extended the CIM model with SAN related classes and combined WBEM with the Service Location Protocol (SLP) to enable the automatic detection of the devices in the network. Using the above standards, the development of central management software for SANs became possible, and several applications are now available. SAN management applications usually discover the devices in a network periodically, collect the configuration data, and store them in their databases. The configuration data are accessed indirectly via a database due to performance restrictions.

In this dissertation, we use the database of the SAN management software IBM TotalStorage Productivity Center (TPC) [32] to access the configuration data that we validate. The configuration validation problem that is described above is addressed in Chapter 3. We define policies for SAN configuration according to technical restrictions and best practices rules of experts. We then encode our policy rules as SQL fragments which we can evaluate using the SQL engine of the underlying database management system, providing executable checks on the configuration data. Each rule is embedded in a test case, defined by an XML schema, which combines the check with an explanation component and an action component. These components retrieve information about the SAN components that cause the check to fail and provide methods that we invoke on a success or a failure of the test, respectively. Our approach is light-weight in that no extra logic and logical execution engine is needed on top of SQL and the database management system of TPC.

Another problem that is caused by the high complexity of a SAN is to find an optimal SAN design. Human SAN experts usually build a SAN topology following some rules of thumb. These rules lead often to a reliable SAN, but they do not necessarily minimize the

total cost of the network, or provide a better topology to meet the Service Level Agreements (SLAs). In this dissertation, we also consider the problem of configuring a SAN in an optimal way, while additionally taking a number of SLAs into account. This problem is addressed in Chapter 4. Our primary concern is not the minimization of hardware cost, but to maximize the flexibility to accommodate changing SLA requirements in the future. This is because the most critical cost factor associated with a SAN is not raw hardware but operation downtime, e. g. for reconfiguration. In fact, downtime of business critical SANs is simply not an option beyond maybe one hour of scheduled maintenance once a year. Whenever a new or changed SLA cannot be accommodated because reconfiguration is too costly, it must be accommodated by purchasing additional new hardware. Therefore it is of prime importance for a new configuration to avoid future performance bottlenecks which may necessitate major reconfigurations.

For a given set of hardware devices, we attempt to achieve a uniformly high proportion of free resources at each device, grouped by device types (e. g. X free storage capacity on each storage device and Y unused ports in each switch). In this way, we can increase the probability that the QoS requirements of all applications are still met, even if some of the applications demand more resources than planned, because bottlenecks are avoided. This in turn will result in a decrease of SLA violations and SLA penalties, respectively. We formulate SAN design problem as two Pseudo-Boolean problems.

1.2 Thesis Organization

The remainder of the thesis is organized as follows.

Chapter 2 serves as an introduction to the storage area networks (SANs) for the readers who are not familiar with the subject. In the first three sections, we introduce some of the most important SAN components, protocols that are used with SANs, and some SAN topologies. Then, we outline the management of SANs in the last section of this chapter by discussing the switch zoning, best practices rules for configuration, SAN management frameworks, and service level agreements (SLAs).

In Chapter 3, we first place the SAN configuration checking problem and our SANchk approach into a wider context, providing examples from checking configurations of motorcars and of the Apache web server software. In Section 3.2, we give a detailed account of the architecture of SANchk. In Section 3.3, we discuss the power of SQL-based policy formulation by giving some example policies for SAN configuration and their implementa-

tions and by showing how all policy types presented by Agrawal et al. [5] can be formulated as SQL queries. In Section 3.4, we provide empirical results from checking several SAN configurations. While SANchk was originally developed on top of IBM's TPC software, we present in Section 3.5 our port to the open source Aperi Storage Manager. The Aperi plug-in also provides an interactive GUI for editing the rule-sets.

Chapter 4 is organized as follows: Section 4.1 gives a formal definition of the first part of our SAN design problem, namely the problem of assignments of applications to storage devices, and demonstrates it with an example. Section 4.2 defines the second problem, the connection problem. Section 4.3 discusses the integration of the solution into a SAN management framework on the example of Aperi. In Section 4.4, we present work related to our paper.

In Chapter 5, we present our results and give an account of the publications that derived from this dissertation.

Finally, we give a list of best practices rules for SAN configuration in Appendix A.

Chapter 2

Storage Area Networks

This chapter intends to serve as an introduction to the storage area networks in order to facilitate the understanding of the technical parts of this dissertation. The information given in this chapter originates mainly from the storage experts at IBM Storage Division in Mainz, Germany, and from the books on storage area networks by Poelker et al. [42], and by Troppens et al. [51].

2.1 Components of Storage Area Networks

The components that are used in building a SAN can be grouped into three layers: host layer, fabric layer, and storage layer. In the following, we introduce some of the important SAN components at each of these layers.

2.1.1 Host Layer

Host layer provides the infrastructure for servers to enable them to communicate using the Fibre Channel protocol. The SAN components in the host layer include the host bus adapter (HBA), the Gigabit Interface Converter (GBIC), and the host bus adapter drivers.

Host Bus Adapters (HBA)

In order to be able to connect a server to a fabric in the SAN, an HBA is mounted to one of the slots of the server. The HBAs are intelligent devices that let the server's operating system interact with the storage devices. The operating system of the server

can communicate with the HBA using its driver for that operating system. The drivers are usually written by the manufacturer of the particular HBA. In addition to the device driver used by the operating system, a piece of software called firmware is contained by the HBA in its BIOS chip. The BIOS helps transport the data to the gigabit link module (GLM), which is attached to the HBA. A GLM serializes the data into a bit stream and transmits it as light pulses into the fiber optical cable. The GLMs are also referred to as GBICs, since they are mostly integrated into the HBAs.

The Gigabit Interface Converter (GBIC)

The Gigabit Interface Converter (GBIC) is the physical counterpart of the logical port. GBICs are used by switches and hubs to convert bit streams into light pulses, so that they can be transmitted over the fiber optical cables.

The GBICs are differentiated by the light wavelength that they are processing with. There are two sorts of GBICs: short wave and long wave. The wavelength of a short wave GBIC is between 780nm and 850nm. Short wave GBICs can connect devices in distances between 0.5 m and 500 m. Long wave GBICs operate with a wavelength at 1300nm and can connect devices in distances between 2 m and 10 km.

The GBICs produce light pulses using a very small part called laser or light-emitting diode (LED). The GBICs are designed for full-duplex data transmission, as they have separate connections for transmitting and receiving data. Accordingly, the fiber optical cables contain also two strands of glass fiber, which are connected to their counterparts in the GBICs.

The connector on the GBIC where the cable is plugged in can be either of type SC (Subscriber connector) or LC (Lucent connector). SC connectors are the first generation connectors and can transmit data on 1Gbit speed. Whereas the LC connectors can operate on a transfer rate from 2Gbit up to 10Gbit. LC connectors are smaller than SC connectors. It is possible to build switches with more ports using LC connectors.

2.1.2 Fabric Layer

The fabric layer provides for the interconnection between the devices in the host layer and storage layer. The SAN components in the fabric layer include SAN hubs, SAN switches, data routers and cables.

SAN Hubs

A SAN hub acts architecturally like the other hubs as a loop of wire with ports connected to it. In difference to the other hubs, SAN hubs have GBICs as their ports and use the FC-AL (Fibre Channel Arbitrated Loop) dialect of the Fibre Channel protocol for the communication. The connection that is made with a SAN hub is called Fibre Channel loop. The devices that are connected to the hub must take a turn to transmit their data. Since it is a single loop of wire, and the total bandwidth is used by the transmission, only one packet can be sent at a time. The total bandwidth of a SAN hub is 100MB/s.

SAN hubs can have 4, 8, or 12 ports. Hubs can be cascaded to obtain larger loops up to the limit of 128 ports.

Due to disadvantages like port limitation, the occurrence of congestion at an increased number of devices, or the obligatory stopping of all devices to attach a new device, SAN hubs are rarely used in SANs. The SAN hubs are today mostly used to share tape devices which are used to back up data. Operations on a tape drive are usually long, continuous, and not performance critical, so that the limitations of hubs are not relevant in this case.

SAN Switches

SAN switches connect all of their ports with each other like other switches. That enables different pairs of devices attached to the switch to communicate at the same time. SAN switches have GBICs as their ports and use the FC-SW (Fibre Channel Switched) dialect of the Fibre Channel protocol for the communication.

When one or more switches are connected and used in a SAN, a fabric is created. In case of connecting multiple switches with each other, one switch takes over the management of the whole. A network of interconnected switches is called switched fabric. A fabric can be extended up to the limit of 239 switches. The collection of all of the single fabrics in the entire SAN is called a SAN fabric. SANs usually contain more than one fabric to avoid a single point of failure. A problem in the managing switch in a fabric can paralyze the whole fabric, as they are managed together.

SAN switches can be divided into modular class switches (standard) and director class switches (enterprise). Modular class switches are smaller than the director class switches and usually have 8, 16, or thirty-two ports. Modular switches can be used to build small SANs. To scale the SAN, larger fabrics can be created interconnecting several modular switches. In this way, also the availability is increased, as the single points of failure are decreased. In case of the failure of a switch in a fabric, the data-paths can be moved to

other switches in the fabric automatically.

Director class switches differ from modular class switches in their lager size and in their very high reliability. Most director class switches are built for zero downtime and can be maintained without taking them offline. Since they mostly consist of redundant parts, they do not need to be powered off when a part is replaced.

Director class switches are scalable in that they use port blades. Port blades are mounted into the director frame. In case of a failure of a blade, it can be replaced without having to power off the director. The intelligence needed by a director class switch comes from processor blades. A director has two processor blades for redundancy. On a failure of one of the processor blades, the other processor blade takes over automatically.

Another difference of director class switches in comparison with modular class switches is that they have very low latency. Director class switches are used mostly in large SANs or in SANs that do not tolerate latency. In large SANs, director class switches are mostly connected directly to the storage devices as core switches. Then, they are connected to the hosts over modular class switches. This makes sense, since the director class switches are much faster and more reliable than the modular class switches. By creating multiple paths between a host and a director class switch using multiple modular class switches, the availability can also be improved at the server side.

Data Routers

Data routers are intelligent bridging devices to integrate SCSI devices into a SAN. Older SCSI devices can be connected over a data router to a Fibre Channel switch for example. Data routers have usually one Fibre Channel interface and one or more SCSI interfaces.

Cabling and Ports

In the early times of Fibre Channel, copper cables were used for interconnecting the devices. Then, fiber optical cables began to be used to increase the speed and to avoid distance limitations of copper cables.

There are mainly three types of fiber optic cables that are used in SANs: the 50μm[1] multimode cable, the 9μm single-mode cable, and the 62.5μm multimode cable.

The main difference between multimode and single-mode environments is the wavelength at which light is emitted. In multimode environments, GBICs use a light-emitting

[1] A micrometer: one millionth of a meter.

diode (LED) that produces light at a wavelength in the range of 850nm. In single-mode environments, a powerful laser is used to produce light that is emitted at a wavelength in the range of 1300nm. Since the LED is not as powerful as laser, multimode cables are safer to use, but let the light reach smaller distances. Another difference between multimode and single-mode environments is the diameter of the glass core, which is called numerical aperture. The cables in multimode environments have a larger diameter glass core. The smaller the glass core diameter, the longer distances can be reached, since the light will be reflected inside the cable at a smaller angle. Obviously, the fastest cable variant is the 9μm single-mode cable, then the 50μm multimode cable, and the slowest variant is the 62.5μm multimode cable. Not to intermix the cable type in the same SAN is accepted as a best practice. The most used variant is the 50μm multimode cable. The 62.5μm multimode cables were mostly used in TCP/IP networks. The 9μm single-mode cables are used when the network connects distant locations. The selection of the cable type sets also the requirements for the GBICs in the interconnector devices and GLMs in the HBAs.

Cables are connected to the devices through ports. A port is a logical name for a GBIC or GLM depending whether the device is an interconnector or an HBA. The SAN ports have modes of operation, after which they are named. The ports change their modes of operation automatically depending on the device that is connected to it. Storage device ports or server ports are called N_Ports (Node Ports). In case that they are connected to a hub, they become NL_Ports (Node-to-Loop). Hub ports are called L_Ports (Loop Ports). Switch ports are called G_Ports (Global Ports) when no device is plugged into them. If a switch port is connected to a server port, it changes its mode of operation to an F_Port (Fabric Port). Switch ports that are connected to Hub ports are called FL_Ports (Fabric-to-Loop Ports). A switch port that is connected to another switch port is called either an E_Port (Expansion Port) or a T_Port (Trunk Port). The first connection creates an inter-switch link, whereas the latter connection creates a trunk (logically aggregated data links) between two switches.

2.1.3 Storage Layer

The storage layer is the place where the data is eventually stored. In the storage layer of a SAN, storage arrays and tape devices are usually to find.

Storage Arrays

Storage arrays contain a number of disks that are arrayed together to build larger disks. Storage arrays are diversified depending on their sizes and features for example. Most commonly, they are classified by size into monolithic arrays and modular arrays.

Monolithic arrays comprise duplicate parts, so that they have high availability. Even changing parts on a monolithic array can be done without having to power it off. A feature called Phone Home lets monolithic arrays perform a self-diagnosis and call the manufacturer in case of a failure. Monolithic arrays usually require raised floor, conditioned air and power, and multiple large-amperage three-phase electrical connectors. Monolithic arrays are mostly used in data centers with mainframe computers. Because of that, they are also called as Enterprise arrays. The disks in a monolithic array are assembled inside the array frame. Monolithic arrays are also called cache centric arrays, since the disks are connected to numerous internal controllers through large amounts of cache memory.

Modular arrays are comparatively smaller than monolithic arrays. The main difference between modular arrays and monolithic arrays however is that the modular arrays cannot connect to mainframe computers. Because of that, they are also called midrange or departmental storage subsystems. Modular arrays have usually two controller shelves to ensure availability in case of a failure of one of the controllers. The controllers are separated from the disks and connected to them through SCSI or Fibre Channel cables depending on the disk type. Modular arrays can be improved adding disk shelves for capacity and controller shelves for performance.

Data is processed by the port logic when it enters a storage array. The port logic includes the GBIC (GLM) and a small processor unit. Data enters a storage array first at a GBIC of a controller. The GBIC converts the light pulses into digital data. The small processor unit in the port logic is an application-specific integrated circuit (ASIC) processor that has a set of instructions to assemble the data and communicate with other devices in the SAN using FC-SW or FC-AL protocols. After having packed the data, the port logic sends it off to the cache memory of the storage array. If the data arrives the cache memory safely, the server application that sent the data is acknowledged, so that it can continue with its tasks.

The data is sent after that through the internal data paths of the storage array to a RAID parity generator (RAID controller) and finally to the disk drives. The internal data paths inside a storage array can be based either on a switch architecture or on a bus based architecture. The switch architecture provides advantages like simultaneous handling of

multiple servers, better I/O performance, and scalability. Redundant Array of Inexpensive Disks (RAID) helps combine a number of disks into one logical disk and generates parity information to regenerate the data if a disk fails.

The disks are connected to the storage array over a SCSI bus or fibre cables depending on their type. This decides also about the protocol that is to be used communicating with the disks.

Redundant Array of Inexpensive Disks (RAID)

Redundant Array of Inexpensive Disks (RAID), also called Redundant Array of Independent Disks, is a method to represent a number of drives as a single logical drive to a server. The logical drive is called a LUN (Logical Unit Number). Depending on the RAID type used, the data can be distributed to several disks to increase availability, or it can be written redundantly onto different disks to gain better performance. Also a compromise can be made using a RAID type that combines these methods.

There are different types of RAID, which are numbered from 0 to 5. RAID levels 0, 1, and 5 are the most commonly used.

RAID 0: RAID 0 is also called disk striping. In RAID 0, the data is distributed onto all the disks in the RAID set. RAID 0 provides only improvement in performance. Since no parity information is generated, RAID 0 does not affect the availability of the data. RAID 0 requires at least two physical disks.

RAID 1: RAID 1 is also called disk mirroring. All the disks in the RAID set contain the same data. The same data is written simultaneously to the each of the disks. RAID 1 provides the availability, since the data is stored redundantly. In case of a failure of a disk, another disk can take over. RAID 1 also improves the performance of read operations, but it worsens the performance of write operations, because the data must be written redundantly. For RAID 1, a minimum of two separate physical disks is required.

RAID 1+0: RAID 1+0 is also called RAID 10. RAID 1+0 provides a combination of RAID 1 and RAID 0. The data is accordingly either first mirrored and then striped, or the other way around. For RAID 1+0, a minimum of four physical disks is required.

RAID 3: RAID 3 spreads the data across several disks like RAID 0. In difference, RAID 3 stores also parity information on a physically separate disk. RAID 3 is currently not used,

because of its bad performance with frequent data operations. Though, it can be used with applications that operate with long, sequential data transfers. For RAID 3, a minimum of three physical disks is required.

RAID 5: RAID 5 combines disk striping with parity. The data is distributed onto all of the disks in the RAID set. The parity information for every chunk of data on each disk is stored along with the data onto a disk other than the disk having the original chunk of data. RAID 5 is the most commonly used RAID type, since it provides a balance between performance and availability. RAID 5 requires a minimum of three physical disks.

Adaptive RAID: In Adaptive RAID, the RAID controller chooses between the methods RAID 3 and RAID 5 depending on the type of the data.

Logical Unit Numbers (LUNs)

After creating a RAID set using one of RAID types, a Logical Unit Number (LUN) can be created from the whole or a slice of the RAID set, which would be called a partition.

Slicing a RAID set and so creating LUNs in various sizes provides a way for flexible assignment of storage space to the servers.

2.2 Protocols

2.2.1 FC-AL

The Fibre Channel-Arbitrated Loop protocol specifies the communication between two devices that are connected to each other within a Fibre Channel loop. In a Fibre Channel loop, the connection is made through a hub. Ports in a hub are connected by an internal logical loop of cable. Conversation on a hub can not occur simultaneously. The FC-AL protocol is used to decide which device may use the loop at which time. The FC-AL protocol is also used in the internal communication of some of the storage arrays. A maximum of 128 devices can be connected to a Fibre Channel loop. Every device in the loop is addressed by a loop ID, which helps determine the priority of the device in the communication.

2.2.2 FC-SW

The Fibre Channel-Switched protocol specifies the communication between two devices that are connected to each other through a Fibre Channel switch. Switch ports are connected over a backplane that connects them to each other directly unlike a hub. Thus, pairs of devices can communicate at the same time through a Fibre Channel switch. By connecting Fibre Channel switches together, a Fibre Channel-switched fabric is created. Devices in a fabric are addressed with a World Wide Name (WWN). A WWN is a unique identifier that is assigned to an HBA at the factory. Every device is registered into the name server of the switch with its WWN and the corresponding port number. The FC-SW protocol uses the name server while controlling the communication.

2.2.3 SCSI

The Small Computer System Interface (SCSI) protocol specifies the communication between a computer application and the disk storage devices that are used by this application. The SCSI protocol is placed on top of one of the Fibre Channel protocols. Thus, it is ensured that the Fibre Channel is backward-compatible with the applications that use the SCSI protocol to communicate with the storage disks.

SCSI is used in a Fibre Channel network serially instead of in parallel as it is used inside a server internally. This is needed because the data must be transmitted over a Fibre Channel cable as light pulses. However, it is still possible to integrate old devices with parallel SCSI interfaces into a SAN using a data router.

2.3 SAN Topologies

2.3.1 Point-to-point Topology

In point-to-point topology, a server is connected directly to a storage device. Since there is no fabric layer between the storage and the server, this kind of connectivity can actually hardly be considered a SAN.

The most important benefit of the point-to-point topology is the high data transmission speed due to the fibre optical cables and Fibre Channel protocol. Fibre Channel supports also connections up to ten kilometers without the involvement of repeaters. However, the scalability of the point-to-point topology is limited by the number of ports in the controller of the connected storage device. Monolithic storage arrays e. g. can have 64 Fibre Channel

ports. The point-to-point topology has also the disadvantage that a single server is attached to each port at the storage device, so that the bandwidth is not used in an efficient way.

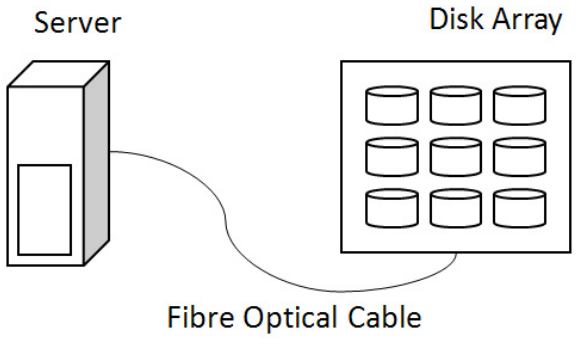

Figure 2.1: *Point to point topology.*

2.3.2 Arbitrated Loop Topology

An arbitrated loop topology is created when the storage devices and servers are connected to each other using a Fibre Channel hub or hubs. Hub ports, and therefore the devices connected to them, are connected to each other on a loop of wire. The devices connected to the loop arbitrate sequentially for access to the loop. The arbitration is controlled by means of the FC-AL (Fibre Channel-Arbitrated Loop) protocol. Devices that are added to the loop are automatically assigned an address according to the FC-AL protocol.

An arbitrated loop can be expanded to up to 128 ports by connecting another hub to it according to the FC-AL protocol. However, most hub vendors limit the number of the hubs to be cascaded to three because of performance concerns. Since the maximal bandwidth is shared by the servers in the loop, every server added to the network decreases the bandwidth available for any one server on average. Hubs can be connected to build a loop of hubs. This enhances the resilience of the network, since the communication between the devices would be possible even after a broken link. The cost of this enhancement is the dedicated ports between the hubs that tie the other hubs to a loop.

Arbitrated loop topology has some drawbacks compared to fabric topology: the limited scalability, lower aggregate bandwidth, or the absence of services like aliasing, routing, name server, and zoning. However, arbitrated loops can still be found in practice because

of their cost advantage; disk drives in a disk subsystem or tape drives in a tape library are preferably interconnected by arbitrated loops.

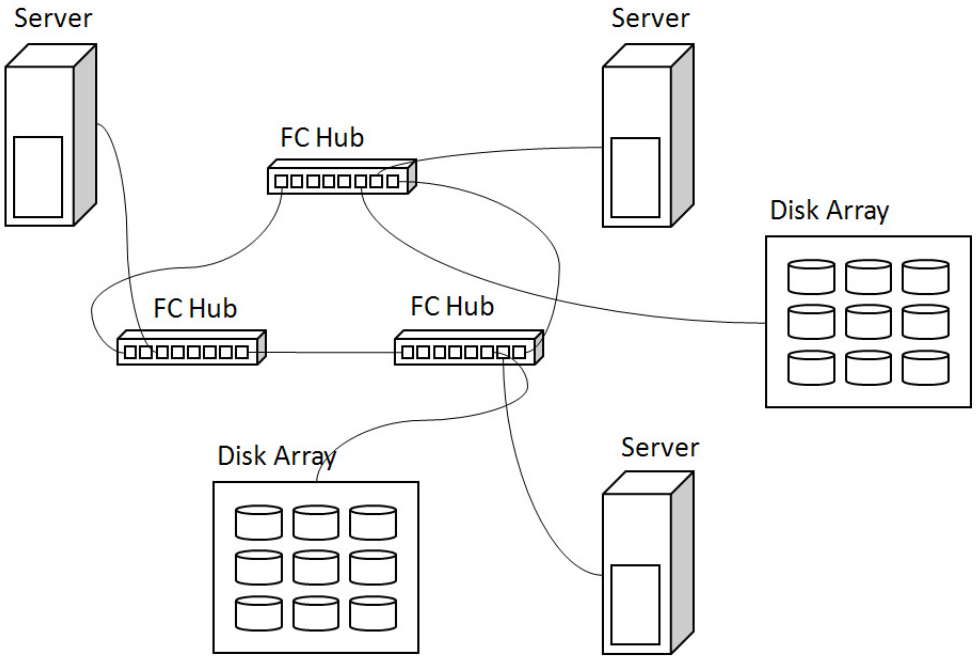

Figure 2.2: *Arbitrated loop topology with a loop of hubs.*

2.3.3 Switched Fabric Topology

A switched fabric is created when a group of switches are tied together through inter-switch links. While an arbitrated loop can contain a maximum of 128 devices, a switched fabric can include theoretically tens of millions of servers and storage devices. Switched fabrics can be organized in many different topologies. Each topology affects the resiliency and the performance of the network in a different way.

The most simple switched fabric topology can be built by using two switches to connect a server to a storage device. Such a network provides the smallest SAN with No Single Points of Failure (NSPOF) and is called a SAN cell. The switches should not be connected to each other, since this would remove the fabric redundance. Servers should have two HBAs in order to connect to two switches.

Similar to the hubs, switches can also be connected into a loop of switches. Two loops of switches are needed to keep the SAN still fabric redundant. One thing to consider is the number of the hops from the server to the storage device. A hop is a node in the network the data must go through. Each hop adds about one millisecond latency. Most storage experts do not recommend more than four hops between a server and a storage device. A loop of switches provides resiliency while only a limited number of ports must be reserved for inter-switch communication.

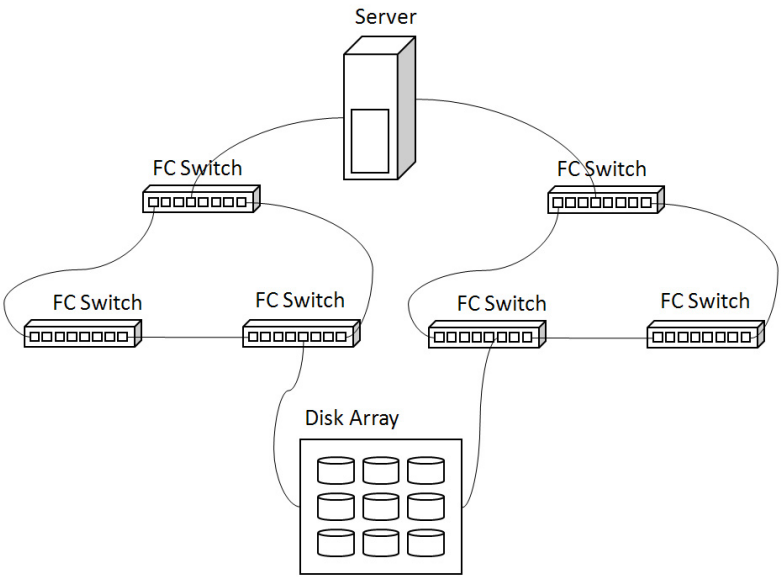

Figure 2.3: *A loop of switches.*

The resiliency of the network can be enhanced in that all the switches in each fabric are connected to each other. The topology created this way is called a meshed fabric topology. A meshed fabric requires at least four switches for each fabric. The meshed fabric topology provides very high resiliency, but needs also a high number of ports dedicated to inter-switch communication.

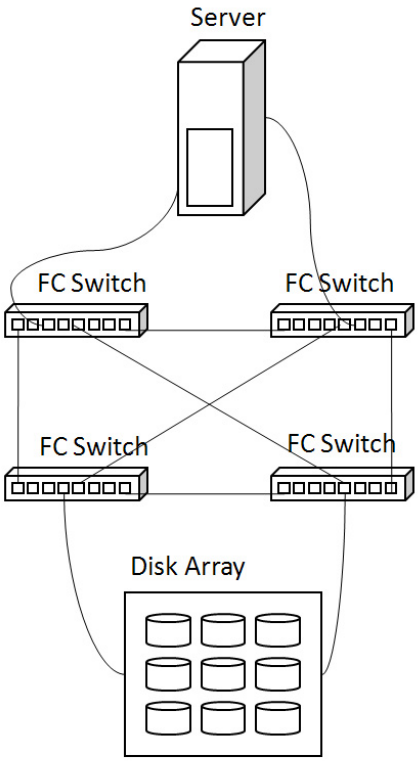

Figure 2.4: *A meshed fabric.*

Connecting the switches in a star topology is a way to avoid the overuse of the inter switch links as in the meshed fabric topology. A single switch links all the other switches together in the star topology. The ports of the core switch and one port for each switch at the edge are used for the inter-switch communication. One problem with the star topology is that the core switch causes a Single Point of Failure. The topology can be modified in that the edge switches are connected together to create a ring. This would increase the resiliency in case of a failure of the core switch, but the number of the hops on the ring would be still a latency problem.

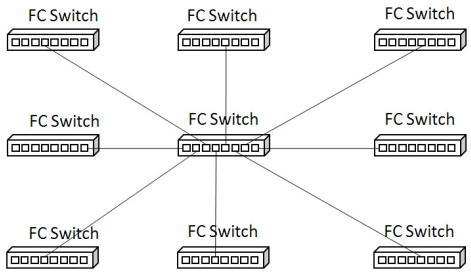

Figure 2.5: *Star fabric topology.*

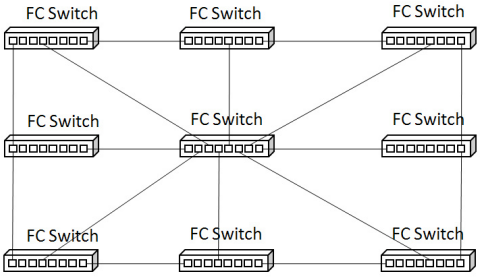

Figure 2.6: *Star ring fabric topology.*

An enhanced version of the star topology is the core-edge topology where two switches are at the core. Every switch at the edge is connected to both of the core switches, so that the redundancy at the core is ensured. The edge switches are divided into two groups, one for the storage devices and the other for the servers. The core-edge topology provides a good balance between the usable ports and the ports dedicated to inter-switch communication.

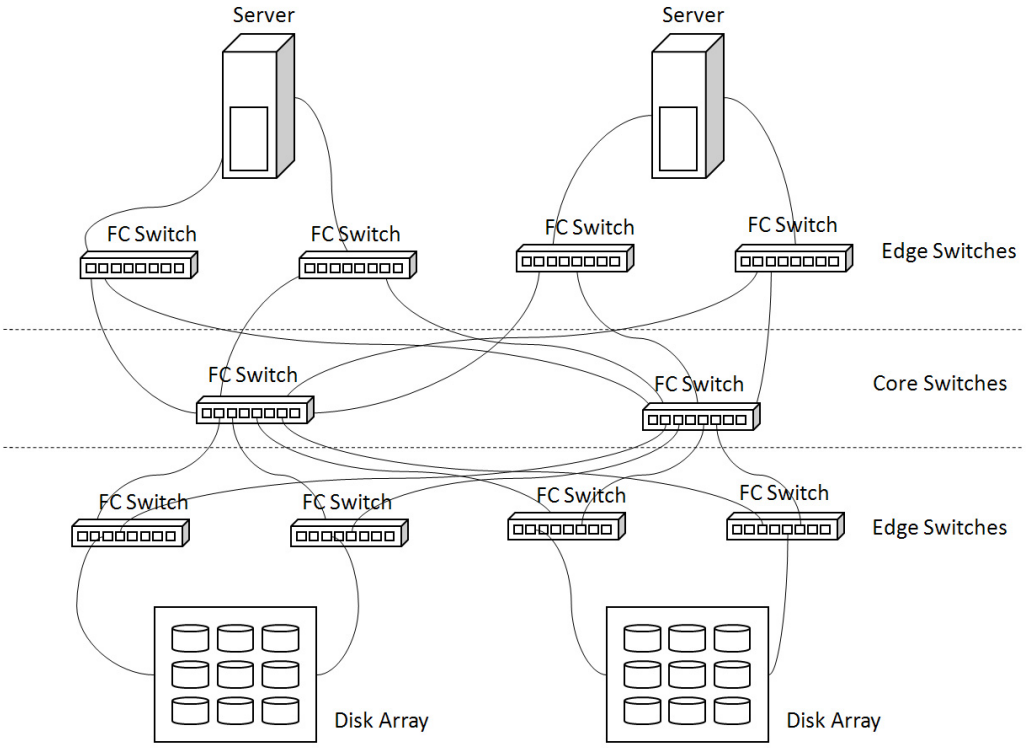

Figure 2.7: *Core edge topology.*

2.4 Management of Storage Area Networks

2.4.1 Zoning

Zoning is a method to isolate parts of a SAN from each other. A zone is a group of switch ports or device addresses. Only the devices in the same zone are allowed to communicate with each other. There are usually a number of zones in a SAN. Zones can be grouped into a zone configuration. Zone configurations must be activated and only one zone configuration can be active at a time.

Zoning is used to enhance the security in the network. Among its other uses, it can be used to prevent servers having different operating systems from accessing the disks of each other. Another possible use would be separating the connections for the web server from the other parts of the network.

There are two types of zoning according to the way they are created. Grouping devices into a zone using their World Wide Names is called soft zoning, whereas using physical switch ports for grouping is called hard zoning.

Soft zoning

In soft zoning, the devices are grouped into a zone set by means of their World Wide Names (WWNs). World Wide Names are 64-bit addresses that are assigned to the devices that are connected to a fabric.

Soft zoning is more flexible than hard zoning, since moving devices does not require also a change of the zoning configuration.

Hard zoning

In hard zoning, physical switch ports in a switched fabric are grouped into a zone set. Hard zoning is not as flexible as soft zoning, but it provides better security.

Aliases can be created and used instead of port names and WWNs to make them more understandable for the SAN administrators.

2.4.2 Best Practices

There is no standard design for SANs that would satisfy the needs of everyone. However, there are some guidelines that are emerged through the experience. These guidelines are called best practices.

Best practice rules can be universally valid, but there are also some that are only valid according to the objectives of a particular SAN design. These customizable rules usually have parameters that can be adjusted. An example for an universally valid rule would be that the zone aliases should be named uniquely throughout the network, whereas a rule about the maximum number of zones allowed in a fabric would be a rule that depends on the size of the network.

2.4.3 SAN Management Frameworks

SANs comprise a great diversity of components, which can come from different vendors. Moreover, SANs are highly scalable networks which can grow rapidly. These two facts make SANs difficult to manage for a human administrator without management tools. The realization of these kinds of systems requires a common and structured representation of the SAN configuration. Otherwise, a configuration management system for SAN must be able to communicate through the proprietary interface of every single device. This requirement was fulfilled with the development of an object-oriented model named Storage Management Initiative - Specification (SMI-S) [50] that standardizes the management interfaces of storage related devices and technologies. SMI-S is developed by the Storage Network Industry Association (SNIA), which was established by storage industry vendors. SMI-S is based on the standard Web-based Enterprise Management (WBEM) from the Distributed Management Task Force (DMTF). WBEM has been developed for the web-based management of enterprise information system infrastructures. WBEM consists of three main components. It uses the Common Information Model (CIM) [21] to model the management data, the xmlCIM encoding to represent the CIM classes and objects in XML format, and HTTP for the transport of CIM operations. SNIA extended the CIM model with SAN related classes and combined WBEM with the Service Location Protocol (SLP) to enable the automatic detection of the devices in the network. Using the standards above, the development of central management software for SANs became possible, and several applications are now available. SAN management applications usually discover the devices in a network periodically, collect the configuration data, and store them in their databases. The configuration data are accessed indirectly via a database due to performance restrictions.

2.4.4 Service Level Agreements

SAN services are provided by data centers that run large SAN installations. Many organizations prefer to outsource the management of their data to reduce their costs, and to focus on their own business. A data center must maintain certain service levels for the applications that it hosts. Not a complete SAN, but the data-paths that are used by an application are subjected to an SLA.

A service level is mostly defined by some criteria like availability, performance, or disaster recovery. Availability is affected e. g. if there are any single points of failure in the paths from an application to the storage devices that it uses. Another factor for the availability would be the allowed downtime per year. Five minutes downtime per year is accepted as high availability, whereas one hour of downtime per year would mean low availability.

The actual performance of a connection depends on many factors in a SAN in a production environment. I/O operations per second, bandwidth, and latency are some of the metrics that can be used to measure the performance of a data path. To find the right network design to meet the requirements for a data path is a challenging task, since the network paths, including the switch ports, are shared.

In case of breaching an SLA, a monetary penalty can be assessed, or depending on the agreement, the customer will not have to pay for the service during the time the SLA was breached. In either way, it means for the provider an increase of costs.

Chapter 3

SANchk SAN Configuration Checker

Storage Area Networks (SANs) provide centralized pools of storage resources for servers. SAN technology has been made possible by fast network technology such as 10 Gb/s Fibre Channel (most recently also 10 Gb/s Ethernet) and high-speed switches and host-bus adapters. Thus, storage resources can be provided and managed independent of compute servers (hosts), and they can be assigned to hosts in a flexible and scalable way. The flexibility in assignments enables a much higher utilization of storage devices, since storage resources can now be shared between hosts. Storage size can also grow independent of the number of servers, providing scalability. The Fibre Channel (FC) protocol also carries SCSI commands over long distances, so that geographically distributed storage networks or disaster recovery solutions are now feasible.

However, the SAN paradigm also provides some challenges. The combination of the diversity and the interoperability of the devices with the scalability of the network makes SAN configuration error-prone. In a large SAN, it is increasingly difficult for an administrator to check the SAN manually for configuration problems. Our SANchk system provides assistance to the administrator by evaluating a set of configuration rules over the relational configuration database provided by common SAN management systems [26] [29].

Figure 3.1 shows some typical dependencies in a SAN with minimal dimensions. In the figure, a database server and a file server are connected through redundant and disjoint paths over two switches to two disk arrays and a tape library. Both servers use the disk array in the middle and possess their own disks which together form logical volumes (Logical Unit Number: LUN). In order to prevent a server from accessing a disk belonging to another server, methods like *zoning* and *LUN masking* are used. By means of zoning,

SAN members can be grouped to isolate them from other devices in the network. LUN masking helps assign LUNs to servers. A configuration fault in zoning or LUN masking can cause overwriting of data. Another example of a configuration issue is *multipathing*. There must always exist redundant paths in a SAN between storage devices and servers, so that in case of a failure in a path there is still a spare connection. Connections between incompatible devices or firmware levels that are required for interoperability are further examples for configuration restrictions in a SAN.

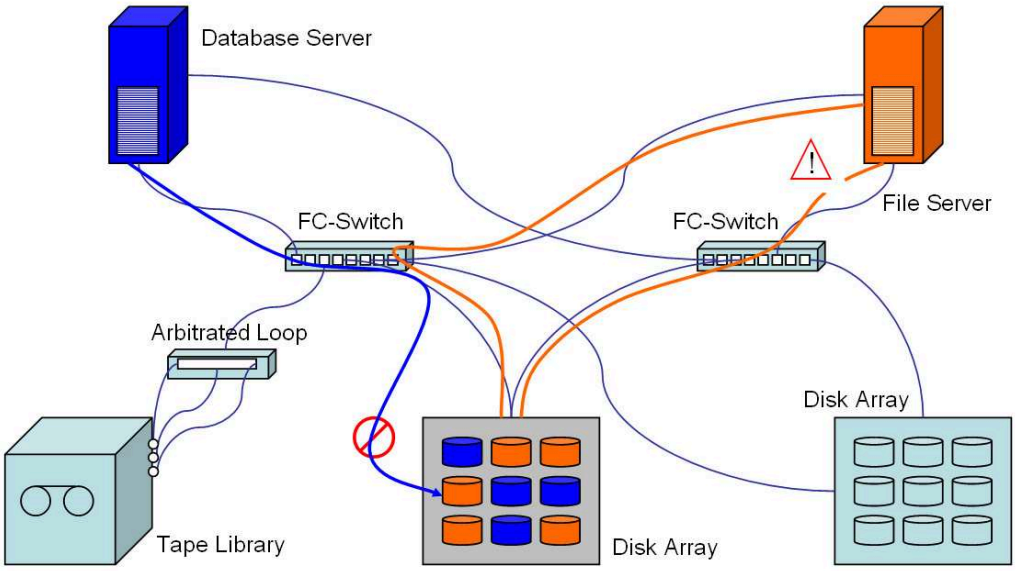

Figure 3.1: *Some configuration problems in a typical SAN: an invalid access to a LUN and an interrupted connection.*

SAN management is so complex that it is commonly performed using storage management systems which gather configuration and performance data in relational databases, using a standardized interface. Our approach to SAN configuration checking is based on collections of best-practices rules which are checked directly on configuration databases with the help of standard relational database technology. In contrast to other approaches, we use SQL to define our policy rules as executable checks on these configuration data. Each rule is embedded in a test case, defined by an XML schema, which combines each check with an explanation component and an action component. These components retrieve information about the SAN components that cause the check to fail and provide methods that will be invoked on a success or a failure of the check.

3.1 Scenarios for Configuration Checking

Configuring complex systems correctly is a daunting task, and all but impossible to achieve "by hand." However, writing a configuration checker is also difficult: First, the checks must somehow be programmed into the system, with facilities to edit, add or delete individual checks, and second, the checker must be able to access the configuration parameters of each device in the network. In the following, we discuss three architectural settings (scenarios) for rule-based configuration checking, which we have investigated in our research.

3.1.1 General Scenario

The general setting with a rule-based approach is depicted in Figure 3.2. Configuration parameters may be hidden deep inside a managed device, and the device itself may be hidden inside a network. Access software must be written for each device, plus device discovery software for the network.

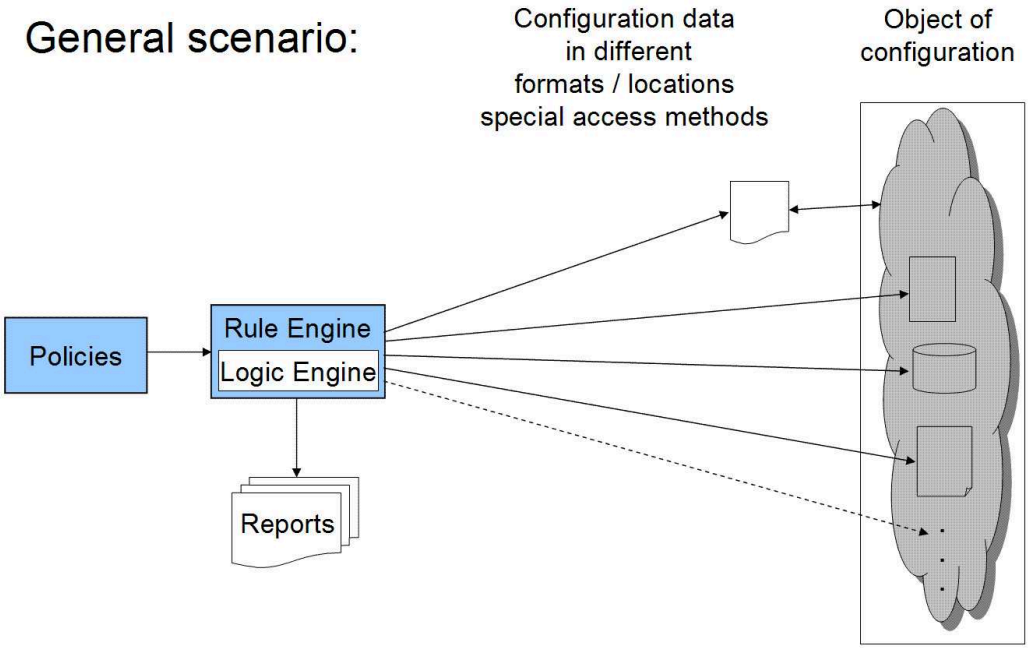

Figure 3.2: *General setting.*

Configuration information may also be kept in external documents. In the motor-car industry, the creation and maintenance of documents pertaining to car configuration is

called (Electronic) Product Data Management (EPDM). The *product overview* (PO) describes which sales options may be combined in which ways into manufacturable cars, while the *bill of materials* (BoM) lists the parts that are needed to build all correctly configured vehicles. In the German motor-car industry, variants of Boolean Algebra are commonly used to denote the configuration constraints in the PO document and the conditions attached to the parts in the BoM which decide whether a part is used in a certain car configuration. Sinz et al. [48] have investigated the application of Boolean satisfiability solving techniques (SAT-solving) to the problem of detecting flaws in the configuration constraints themselves, rather than in individual cars.

In the following, we discuss two solutions without external documents, both using rule-based checks executed by a suitable interpreter. The CIMchk system by Sinz et al. solves the access problem by operating on top of a CIM (Common Information Model) system abstraction layer, while the SANchk system operates on top of a relational database which in turn is fed in part by underlying CIM providers (agents) operating on the system components.

3.1.2 CIM-Based Scenario

In the case where an external description is not available and configuration data must be retrieved from the system itself, it is now common to realize a system abstraction layer by means of data provider agents running on the individual devices. One popular abstraction layer is the object-oriented *Common Information Model (CIM)* [21] whose agents are called CIMOMs (CIM Object Managers). The CIMOM is provided by the manufacturer, and the rule engine can run CIM queries on the system's CIM abstraction layer without knowing the real location of the configuration data.

As an example of a direct CIM-based approach, the CIMchk system [49] is concerned with the case of software system configuration on the example of the Apache web server. An experimental CIMOM produced by IBM Germany Development solves the access problem to configuration data. The CIMOM permits well structured access to configuration information by exporting an object oriented model of Apache's internals. The configuration checks are written as rules in the *CIM Constraint Language ($\mathcal{CCL}$)*, whose variables may be references into the CIM model. The overall system architecture of CIMchk is depicted in Figure 3.3. Section 4.4 provides more detailed information on the CIMchk.

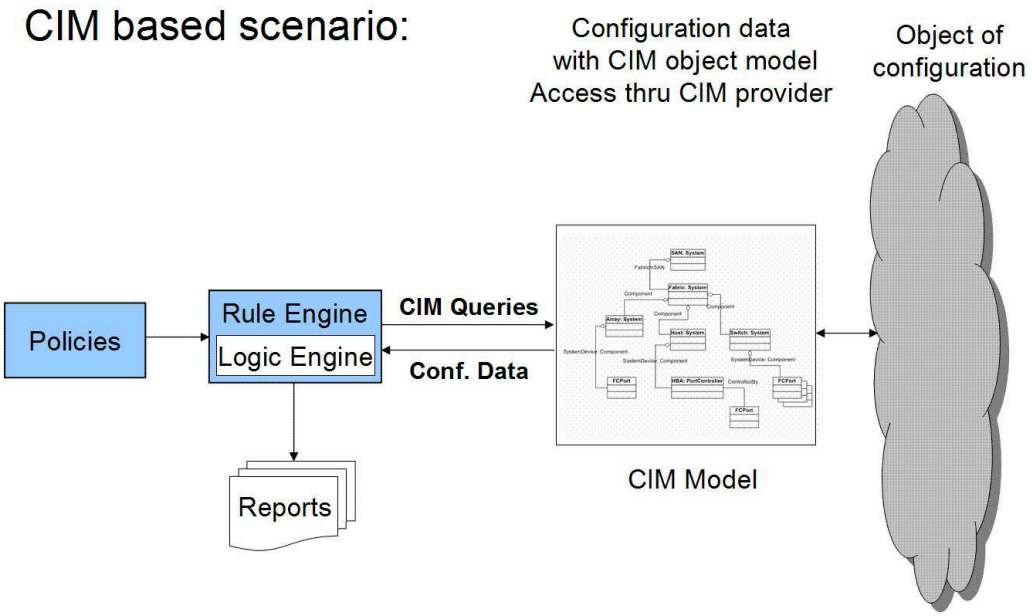

Figure 3.3: *CIM-based setting.*

3.1.3 SQL-Based Scenario

In the case of SANs, configuration management is even more complex, because we must now deal with a multitude of heterogeneous networked devices. For this reason industrial SAN management systems have been developed, and again a common and structured representation of the SAN's internal system parameters was required, for otherwise the system would have to communicate through the proprietary interface of every single device. This requirement was fulfilled with the development of an object-oriented model named *Storage Management Initiative - Specification (SMI-S)* that standardizes the management interfaces of storage related devices and technologies [50]. SMI-S is developed by the *Storage Network Industry Association (SNIA)*, which was established by storage industry vendors. SMI-S is based on the standard *Web-based Enterprise Management (WBEM)* from the *Distributed Management Task Force (DMTF)* [20]. WBEM has been developed for the web-based management of enterprise information system infrastructures. WBEM consists of three main components. It uses CIM to model the management data, the *xmlCIM* encoding to represent the CIM classes and objects in XML format, and HTTP for the transport of CIM operations. SNIA extended the CIM model with SAN related classes and combined WBEM with the *Service Location Protocol (SLP)* to enable the automatic detection of the devices in the network.

Using the standards, the development of central management software for SANs became possible, and several applications such as IBM's TPC (TotalStorage Productivity Center) [32] are now available. SAN management applications usually discover the devices in a network periodically, collect the configuration data using various device agents (perhaps given as CIMOMs), and store them in their databases. The configuration data are accessed indirectly via a database, thus shielding the SAN from untimely CIM queries.

Consequently, SANchk cannot use the CIMchk approach which requires the execution of CIM queries at checking time. However, a relational model is now available through a modern relational database management system (DBMS) which provides SQL execution and user defined (Java) functions, amongst other features. The key insight underlying SANchk is that rules can now be composed from SQL and Java snippets, and that the DBMS can be re-used in lieu of a special logic engine. Each SANchk rule is a package of SQL fragments, supplied parameters, and Java classes which is fed to the database by the SANchk engine which manages the execution. The rules also contain methods that the engine invokes on a success or a failure of the test, respectively. This approach is light-weight in that no extra logic and logical execution engine is needed on top of SQL

and the database management system. Figure 3.4 depicts the overall SQL-based scenario; a more detailed representation of the SANchk architecture is given in Figure 3.5.

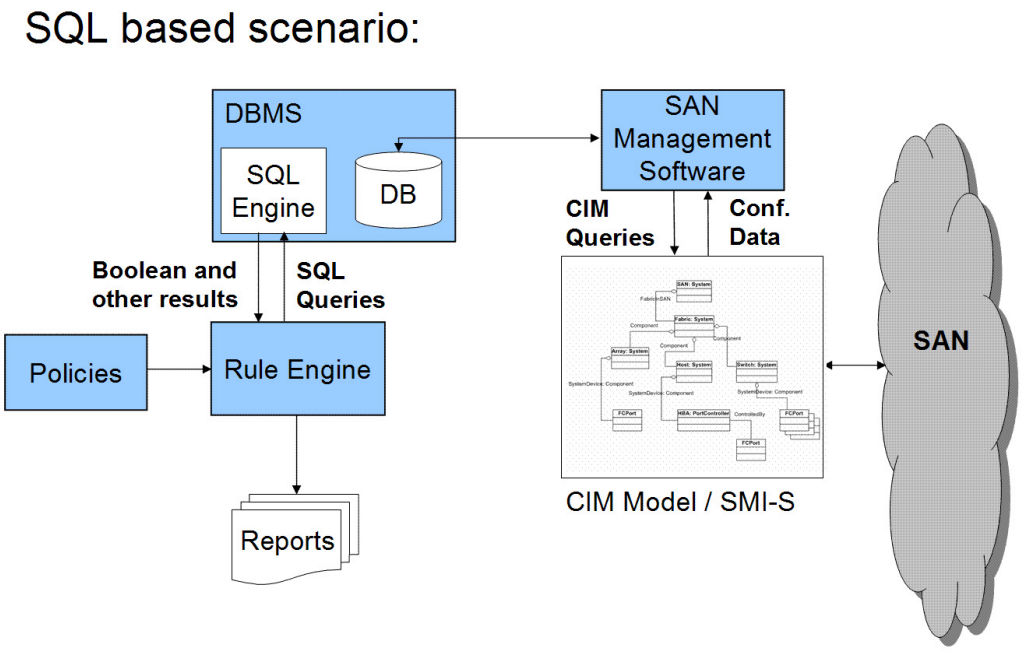

Figure 3.4: *SQL/DBMS based setting.*

3.1.4 Comparison of CIMchk and SANchk

CIMchk defines a new logical language, requires its own evaluator and needs to be understood by anyone writing new checks. CIMchk also queries current configuration data from the managed system by means of a CIM agent (CIMOM) while the system is in production.

SANchk uses well-known languages like SQL and XML. It needs only a light-weight rule engine which passes rule condition and explanation query statements to a standard relational DBMS. Most system administrators will know SQL, and most DBMS will execute it fairly efficiently. SANchk uses configuration data that is stored periodically in a relational database, parts of which may be stale at the time of execution.

SANchk has the advantage of using the features of the underlying database management system, but works with relatively older configuration data. CIMchk works with current configuration data, but it can cause additional workload as it sends queries to the managed system in production time. Considering the differences of these two approaches,

the approach to apply should be chosen according to the configuration checking domain.

While Apache configuration can be checked with CIMchk without jeopardizing system performance in real time by sending queries to a single CIMOM, it is impossible to execute CIM queries on a SAN while it is running close to capacity. For SANs, any practical solution must use the database, while for Apache a database is not needed.

3.2 Architecture of SANchk

3.2.1 Overview

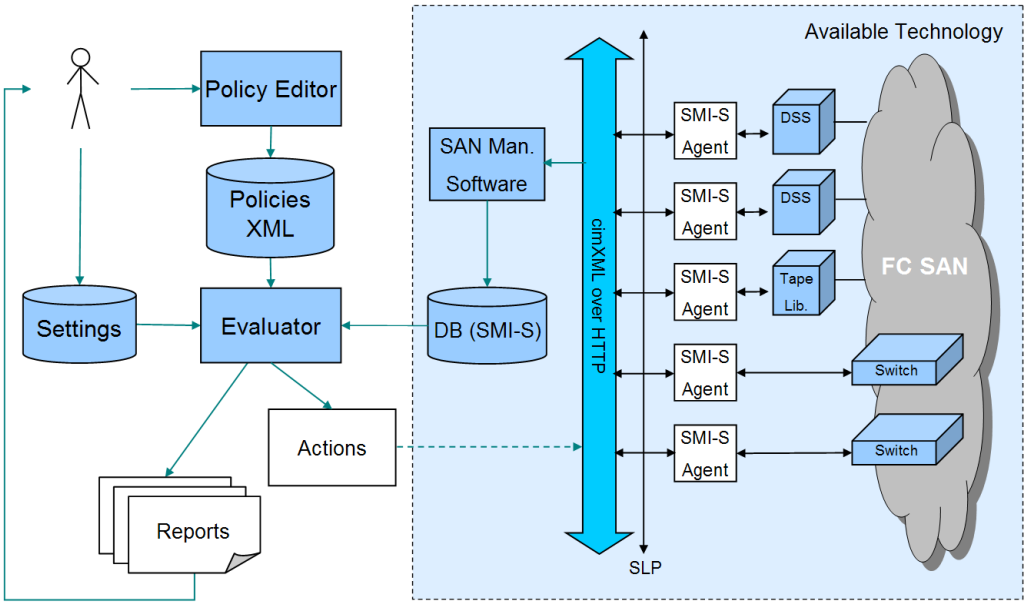

Figure 3.5: *SANchk system architecture.*

Figure 3.5 shows the architectural overview of our configuration checking system.

The checking system has interfaces to an SMI-S compliant SAN infrastructure, so that for every device in the network a corresponding SMI-S agent is provided. SMI-S agents implement management interfaces for the devices that are specified in the SMI-S standard. SMI-S agents are automatically discovered by a client application, in this case the SAN management software, by means of the Service Location Protocol (SLP). The communication between the SMI-S clients and the agents is realized using the *CIM-XML* protocol over HTTP, as specified in the WBEM standard.

Due to performance reasons, SAN management applications usually do not retrieve the configuration information from the CIM objects every time they need them. Instead, they discover the network and fetch the configuration data periodically and store it in a relational database.

Our checking system has two interfaces to this existing infrastructure. The first interface is the connection to the database that contains the configuration data, and the second interface is the direct access to the SMI-S agents using the SMI-S infrastructure, to enable actions to modify the configuration of devices or to use SMI-S services.

Both interfaces can be seen as loose and flexible connections. The only precondition for the checking system to work is the availability of a relational database that contains the configuration data. Instead of using the SMI-S standard, the configuration information could be theoretically collected also by means of proprietary management interfaces for the devices. This alternative way could also be considered for the second interface. The actions modifying the configuration of devices or starting storage management tasks can also be realized using non-SMI-S APIs.

The following components make up the checking system:

- Policy Editor:

  We define entire policies in the XML schema that we specify in Section 3.2.4. In this schema, policy conditions are expressed in SQL syntax and are embedded in an XML element. We assume that among all formalisms, SQL is the one most likely to be known by SAN administrators.

  Most configuration rules reflect best-practice configuration knowledge by SAN experts, so these rules will remain stable once they are coded in optimized SQL by experts. Changes and adaptations made by local administrators will mostly concern parameter settings in these rules. Local administrators, for example, may define the values for X and Y in the rule: "All firmware levels in switches of type X must be greater than Y." This is easily achieved by adding a GUI, and no further SQL coding is needed.

  There are several ways to realize the policy editor. The first way is the utilization of a standard XML editor that uses the corresponding XML schema to provide help writing the policies. Another way is to define a non-markup language that maps to the XML format. In this case, a translator between the new language and the XML format is needed. The third way is the development of a visual editor for the policies,

which we did for Aperi (Section 3.5).

- Settings Data:

  The system uses two types of settings, the settings that guarantee the portability of the system, and the settings that are used in reporting. An example for the first type of settings could be the database connection information. The settings of the second type include the path information for the XSL files that are used for the formatting of the test results for example. We designed an XML schema also for the settings that contains key-value pairs.

- Policies in XML:

  An XML file serves as the policy repository in the system. Also the results of an evaluation process are kept in this file, since the policies defined in the file are extended with their evaluation results.

- Evaluator:

  The Evaluator is the system component that processes the policies and evaluates their conditions. It has interfaces to three information sources. The first source is the XML file containing the policies. The Evaluator first checks the syntax of this file and parses it using a DOM parser and extracts the query information in SQL syntax. The condition in the SQL syntax is then completed by the Evaluator as explained in the next section. The second source that the Evaluator uses is the database. The Evaluator sends the valid SQL statement to the SQL engine through JDBC to evaluate it, and it receives the result in return. The last source is the settings file. The Evaluator uses this file to look up data that is to be embedded in code to avoid hard coding system dependent configuration.

- Action Handling:

  If a configuration rule is not satisfied, an optional action may be executed. As actions, arbitrary Java methods can be specified. To realize the action handling mechanism in the system, the Java Reflection API is used, which provides methods for analyzing and using Java classes that are unknown at compile time. Thus, we can call Java methods whose names appear textually in the XML rule file. We also use a WBEM API that provides access methods for SMI-S infrastructures. Thus, actions can be implemented that modify the configuration or use storage management

services. Generally, autonomic reconfiguration is considered by SAN administrators as too risky. So far, we have implemented action handling in the rules for notification purposes only.

- Reporting:

  The source for the reports is the policy XML file that is complemented with the result elements after the evaluation. We implemented the report generator using XSL transformation, so that the processed XML file is formatted using a common XSLT API. The report generator takes the evaluated XML policy file and generates from this an HTML file that represents the checking results in an HTML form. Using HTML forms, results can be used interactively. The faulty configuration data can be modified and submitted directly from the form to storage management services, so that they can be used in predefined actions that can be selected on the HTML form.

3.2.2 Evaluator: Boolean Tests as SQL Fragments

SQL as a query language is designed to retrieve data from a relational database according to the statement that is sent to the SQL engine. However, we need to define and evaluate comparisons in SQL in such a way that the result is a simple boolean value only. To do this, we employ a trick: we created an auxiliary table with only one row and one field that holds a token CHAR value.

We defined the following SQL fragment as a prefix to our comparisons in SQL syntax, so that the concatenation of them can be sent to the SQL engine as a correct SQL statement that returns boolean values.

```
select count(*)
from sanchk.sanchk_aux where
```

Since we have only one row in the table, the statement can return only one of the values 0 or 1 corresponding to the boolean comparison in the WHERE clause of the statement. Thus, the boolean checks can be left to the SQL engine.

Using SQL in defining configuration policies also has the advantage that the whole spectrum of the language elements including the operators, functions and predicates in SQL is available for the use in policies. If, despite that variety, new functions are needed, most database management systems provide the infrastructure to define additional functions. For example, we implemented a User Defined Function based on a Java method to compare software versions.

3.2.3 Data Related Aspects of the System

The checking system in Figure 3.5 requires the existence of a database that contains the SAN configuration data. Alternatively, the data could be obtained directly from an SMI-S infrastructure using a WBEM/CIM API or the *CIM Query Language (CQL)*. Considering the dependencies between the devices, multiple queries must be processed to obtain the data needed for the evaluation of a policy. For performance reasons, we keep CIM queries separate from policy checks using a database, at the price of reduced portability. In this way, we also want to explore the suitability of SQL as a query language to implement the policy checks.

We use the database of the SAN management software TotalStorage Productivity Center (TPC) by IBM as the data source for the checking system. TPC obtains the configuration data using the SMI-S standard. It discovers periodically or upon request the devices in the network and fetches the data through the responsible agents and stores them in its relational database. The agents use the CIM classes to obtain the configuration data. In Figure 3.6 [14], the SMI-S CIM class hierarchy is represented. In the model, a SAN aggregates fabrics. In turn, the fabrics aggregate interconnect elements like switches or platforms like hosts or arrays, which in turn aggregate ports or HBAs.

3.2.4 An XML Format for SAN Configuration Policies

We use XML to define test cases in a semi-formal way designed to be intuitively understandable for users, so that using the system would not require a long learning phase.

Figure 3.7 presents the XML schema diagram for the test cases. The root element of our XML schema is the *Tests* element which can contain one or more *Test* elements. A *Test* element describes a test case and can be interpreted as a single configuration policy. It has the attributes *Description*, *Severity*, *Priority*, *Category* and *IsActive*. *Description* is a verbal explanation of the test case. *Severity* helps assign the test case to the stages of severity like *Error* or *Warning*. To allow that some policies can be deactivated in certain cases, we also have the attribute *IsActive*, which holds a boolean value. *Priority* can have integer numbers to specify the importance of a test case. Although not implemented yet, *Priority* may be used in consistency checking of policies to choose the right proceeding order if several policies affect the evaluation of each other through their actions.

TestQuery is the formal part of the *Test* element. It contains the SQL fragment to evaluate and the type of the query, which can be *StandardSQL* or *PreparedStatement*. While

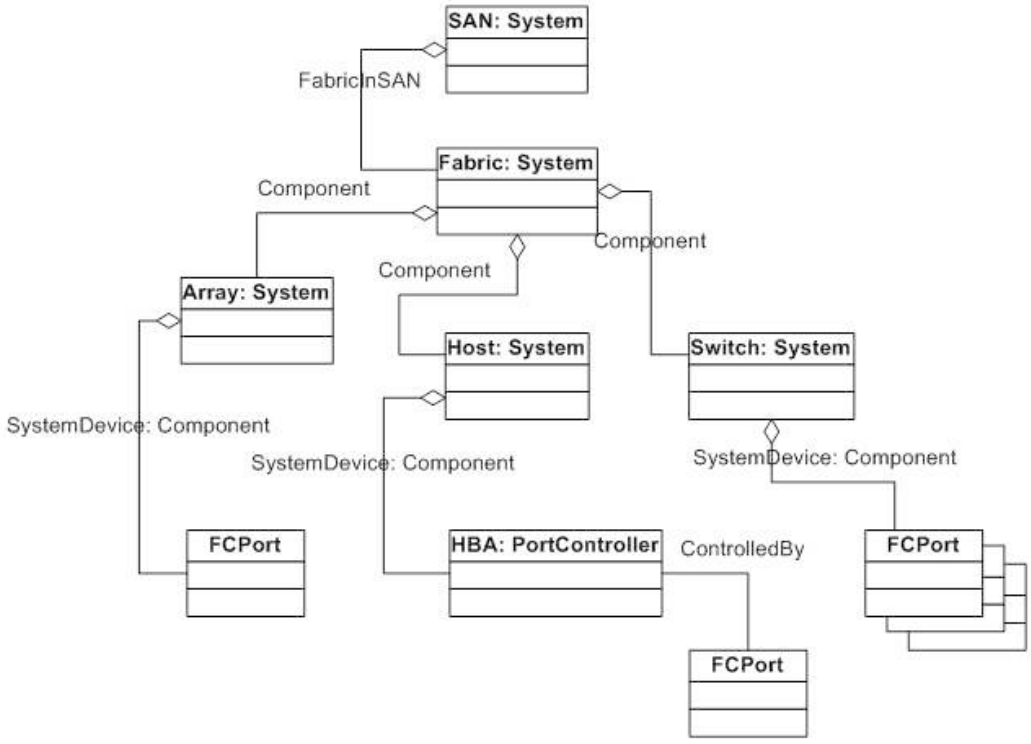

Figure 3.6: *The SMI-S model.*

StandardSQL stands for the queries that can be directly sent to the SQL engine, *Prepared-Statement* indicates a query that should be interpreted as a JDBC PreparedStatement. In this case, values that are held in *QueryParameter* elements will be inserted into the SQL fragment in *TestQuery* to replace the question marks as placeholders.

Since a boolean result alone does not help solve the configuration problem, we introduced an element called *FailedElementsQuery* that contains the query to get information about components that caused the test to fail. Just like *TestQuery*, *FailedElementsQuery* can also have parameters. The result of the query in the element *FailedElementsQuery* can then be used as the parameter values for the methods in the *Action* element.

The *Action* element contains *Method* elements with Java methods to invoke according to the result of the *TestQuery*. This functionality is realized through the attribute *InvokedOn* of the *Method* element that can have one of the values *Failure* or *Success*. The element *MethodName* contains the Java method name as its text value and its return type in the attribute *ReturnType*. The *MethodParameter* element has the attributes *ParamType*,

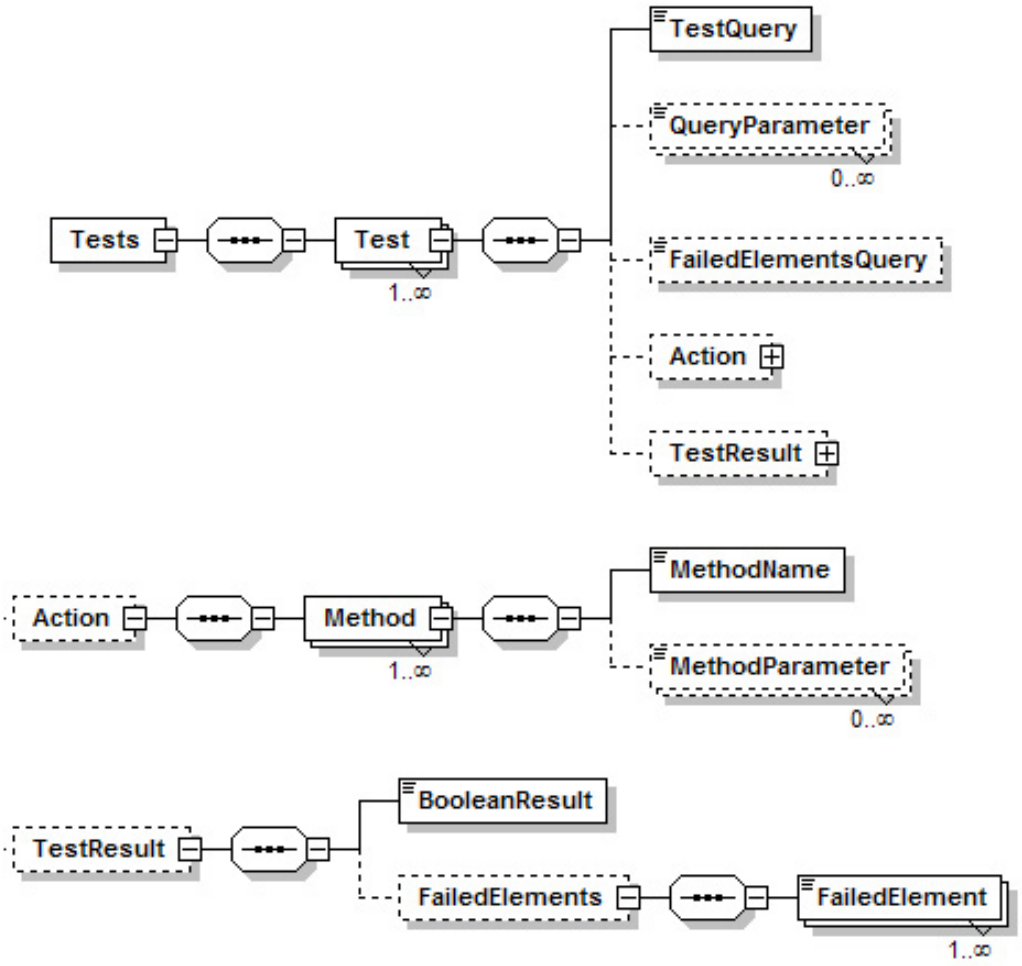

Figure 3.7: *XML schema diagram for test cases.*

JavaType and *ParamIndex*. *ParamType* can have the value *FixedValue* or *FailedElements*. In the first case, the parameter value as the text content of *MethodParameter* is to be used. In the second case, the values in the *FailedElement* elements are packed into a Java array and applied with the method. We use the *Java Reflection API* to invoke the Java methods described in the *Action* element. A more detailed explanation of the action handling mechanism can be found in Section 3.2.5.

TestResult is an optional element that appears only after the evaluation. It contains the elements *BooleanResult* and *FailedElements*. *BooleanResult* holds as its element value one of the values 0 or 1 according to the evaluation result. The element *FailedElements* contains one or more *FailedElement*'s that hold the result of the query defined in the

element *FailedElementsQuery*.

3.2.5 Action Handling

We use the Java Reflection API to implement the action handling mechanism in the system architecture. In this way, it is possible to integrate a very wide range of applications with such a checking system. The availability of diverse APIs and the language capabilities of Java like JNI to integrate C code make this approach attractive.

The Java Reflection API helps represent the classes, interfaces, and objects in the Java Virtual Machine (JVM). By means of this API, applications can analyze and use objects from classes that were unknown at compilation time. The Java Reflection API *java.lang.reflect* has been a part of the Java Development Kit since JDK 1.1 , but already since JDK 1.0 a class named *Class* has been provided to represent the primitive Java types.

We needed the reflection functionality to realize a flexible action handling component that can directly invoke Java class methods. We can thus call Java methods with parameters containing the configuration information returned dynamically from the policy evaluation. This means that the rule parameter does not have to be recompiled if the rules change (rules are data).

We integrated the Java Reflection API into the system through the XML format that we presented in Section 3.2.4. After an evaluation, we dynamically invoke the Java methods that are identified with their signatures and parameters in the *Method* elements. The location of the classes that are referenced must have been added to the class path of the system, so that the location of invocation does not matter. The classes and methods need not be known prior to the invocation.

Agrawal et al. [5] define three types of actions. The actions of the first type are the actions that modify the current configuration of the SAN using a management interface, which can be SMI-S based or not. Changing the members of a zone or changing the RAID level of certain disks are examples of these kinds of actions. Currently, established WBEM APIs are available for Java, so that these kinds of actions can be easily defined in our system. The second type of actions are actions that are triggered by a workflow manager. An example for this type of actions could be shutting down or restarting a device due to some conditions. State management controls like these have been added to CIM 2.8 by the DMTF and they are also implemented in the current WBEM APIs. Actions of the third type are needed when informing the administrators about relevant events in the system. Sending an e-mail to notify an administrator about a resource reaching its capacity would

be an example to this type of actions. Obviously, actions of this type can be defined in the system without a problem.

3.3 Policy Implementation with SQL

In this section, we show how the "Collection Policies" of [5] can be implemented in SQL. Besides the basic comparison operators like = (equal), < (less than), > (greater than), >= (greater than or equal), <= (less than or equal), <> (different), SQL also provides keywords like BETWEEN (range check), IN (inclusion check), ALL (all values satisfy the comparison), SOME/ANY (at least one of the values satisfies the comparison) to formulate quantified comparisons.

The GROUP BY clause is used with column functions and helps return the results of a query for each group. We use this clause for example to formulate comparisons for each device of a certain type in a SAN.

The keyword DISTINCT helps check whether a certain column contains unique data.

EXISTS is used to formulate existence restrictions.

Logical operators like AND, OR and NOT enable the formulation of compound comparisons.

In the following, we give for every type of "Collection Policy" a corresponding example implementation with SQL.

3.3.1 Implementation Examples for "Collection Policies"

We represent implementation examples for the "Collection Policies" that were defined in [5] as exceptional policy types, since they cannot be expressed using only basic operators like the logical operators (AND, OR, NOT) and the comparison operators ($>$, $<$, $\geq$, $\leq$, $==$). In each case, we cite the definitions of the policy types from [5] and give an example policy and its implementation.

In the following definitions, C represents a collection that has the elements $O_1, O_2, ..., O_n$. Each of these elements has in turn m attributes $p_1, ..., p_m$.

Cartesian Property

Given sets of values $A_1, \ldots, A_m$ for attributes $p_1, \ldots, p_m$, respectively, return all elements O_i in C that satisfy the condition $(O_i.p_1 \in A_1) \wedge \ldots \wedge (O_i.p_m \in A_m)$.

Example: Every HBA with vendor_id 0 and model_id 1 should have one of the firmware versions: 2.02.00, 2.02.01 or 2.02.02.

We use the table T_RES_HBA to implement this policy. This table contains the configuration data about the Host Bus Adapters (HBA) that are discovered in the network. In the SQL fragment below, we use the negation of the SQL reserved word IN as a set operator to express the condition in the policy. In this way, we can check the cartesian product of the firmware version attributes of all HBAs in the system with the firmware versions demanded by the policy.

```
<Test
  Name="hbaFirmwareVersionCheck"
  Description="Every HBA with vendor_id 0 and model_id 1
               should have one of the firmware versions:
               2.02.00, 2.02.01 or 2.02.02."
  Severity="Warning"
  Priority="350"
  IsActive="1">
  <TestQuery QueryType="StandardSQL">
    0 = (SELECT COUNT(*) FROM tpc.t_res_hba
         WHERE vendor_id = 0 AND model_id = 1
         AND firmware_version
           NOT IN {'2.02.00','2.02.01','2.02.02'})
  </TestQuery>
</Test>
```

Graph

Given a directed graph $G = (E, C)$ (with elements in C as vertices, and directed edges in E), and two elements O_i and O_j in C, return all directed paths between O_i and O_j.

Example: All HBAs should be connected to multiple switches.

The policy below checks if the requirement of having multiple connections from the host HBAs to different switches is satisfied. In this way, it can be guaranteed that there is still a path available even if a switch fails.

To implement this policy, we use two views and a table. The view T_VIEW_ZONE2MEMBER_ENT_CON_TO_SWITCH represents the membership relation between the zones and the devices in the SAN. The second view T_VIEW_NODE2COMPUTER contains the information about the relation between the nodes and computers in the SAN. A node is interpreted as a connection interface at the hosts, so that this table also holds the data about the HBAs in the system that are used by hosts. In the SQL fragment below, we group the switches by HBAs connected to them and check if their number is greater than 1.

```
<Test
  Name="HBAConToMultipleSwitches"
  Description="All HBAs should be connected to multiple
               switches."
  Severity="Warning"
  Priority="150"
  IsActive="1">
  <TestQuery QueryType="StandardSQL">
    1 < ALL(SELECT COUNT(DISTINCT b.switch_id)
            FROM tpc.t_res_node2hba A,
                 tpc.t_view_zone2member_ent_con_to_switch B,
                 tpc.t_view_node2computer C
            WHERE C.host_id = B.entity_id
            AND A.node_id = C.node_id
            GROUP BY A.hba_id)
  </TestQuery>
</Test>
```

Exclusion

Given sets of values $A_{1,1}, \ldots, A_{1,m}$ and sets of values $A_{2,1}, \ldots, A_{2,m}$ for attributes p_1, ..., p_m, respectively, return all elements O_i in C that satisfy the condition $(O_i.p_1 \in A_{1,1}) \wedge \ldots \wedge (O_i.p_m \in A_{1,m})$ while another element O_j, $j \neq i$, simultaneously satisfies $(O_j.p_1 \in A_{2,1}) \wedge \ldots \wedge (O_j.p_m \in A_{2,m})$.

Example: Every zone should have either disk subsystems or tape libraries.

The connections to a tape library and a disk subsystem are to be separated by means of zoning. Tape libraries start long I/O processes that should not be interrupted, because

after an interruption of the transmission a restart of the process is required. Without the isolation of the connections to a tape library and a disk subsystem, this kind of problem may occur. The following policy checks whether zones with tape libraries and disk subsystems exist in the SAN.

To implement this policy, we use the view T_VIEW_ZONE2MEMBER_ENT_CON_TO_SWITCH in the TPC database. As we want to compare the data in the same column, we must use the table twice. In SQL, the result sets can be named with aliases and used again in queries like tables. We join these tables so that a cartesian product of them is produced. After this step, we can check if a matching of a tape library and disk subsystem is available.

```
<Test
  Name="zonesWithTapeLibrariesAndDiskSubsystems"
  Description="Every zone should have either
               disk subsystems or tape libraries."
  Severity="Error"
  Priority="60"
  IsActive="1">
  <TestQuery QueryType="StandardSQL">
    0 = (SELECT COUNT(*) FROM
          (SELECT DISTINCT prefix_id, zone_id
           FROM tpc.t_view_zone2member_ent_con_to_switch)
           AS res1,
          (SELECT DISTINCT prefix_id, zone_id
           FROM tpc.t_view_zone2member_ent_con_to_switch)
           AS res2
         WHERE res1.zone_id = res2.zone_id
         AND res1.prefix_id = 'tapelibrary:'
         AND res2.prefix_id = 'subsystem:')
  </TestQuery>
</Test>
```

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: There should be only one OS type in each zone.

The policy example below checks whether hosts with different operating systems exist in the same zone or not. Since most operating systems use the resources aggressively and do not care much for the other operating systems that would use the same resources, it is highly recommended to separate their storage spaces. This is handled in SANs with zoning. The example below demonstrates a minimal and valid test case containing only the child element TestQuery.

We use in the SQL fragment the SQL quantified predicate ALL to apply the condition for each zone in the SAN. To express this policy in SQL, we needed to use the view T_VIEW_ZONE2MEMBER_ENT_CON_TO_SWITCH and the table T_RES_HOST that is holding information about the servers in the SAN. In the SQL fragment below, we count the operating system types for each zone and check if the number is less than 2.

```
<Test
  Name="differentOSTypesInZones"
  Description="There should be only one OS type in each
               zone."
  Severity="Error"
  Priority="100"
  IsActive="1">
  <TestQuery QueryType="StandardSQL">
    2 >
    ALL(
      SELECT ostype_count
      FROM(SELECT COUNT(distinct os_type) ostype_count,
                  zone_id
           FROM tpc.t_view_zone2member_ent_con_to_switch A,
                tpc.t_res_host B
           WHERE A.entity_id = B.computer_id
           AND A.prefix_id = 'server:'
           GROUP BY A.zone_id, B.os_type)
      AS res)
  </TestQuery>
```

```
</Test>
```

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: Every switch should have a unique logical name.

To implement this policy, we used the TPC table T_RES_SWITCH that contains the data about the switches discovered in the SAN. Checking whether a table column holds unique data, or in other words, whether the column data form a set, can be checked by means of the SQL reserved word DISTINCT. Using DISTINCT in a SELECT statement, duplicate data can be removed from the result set. Comparing the result sets from the same query, once with the DISTINCT keyword and once without, we can find out whether a column contains a set.

```
<Test
  Name="uniqueSwitchLogicalNames"
  Description="Every switch should have a unique
               logical name."
  Severity="Warning"
  Priority="600"
  IsActive="1">
  <TestQuery QueryType="StandardSQL">
    (SELECT COUNT(logical_name) FROM TPC.T_RES_SWITCH) =
    (SELECT COUNT(DISTINCT logical_name)
     FROM TPC.T_RES_SWITCH)
  </TestQuery>
</Test>
```

3.3.2 An Example for Action Handling

In the example below, we check if there is only one tape library in each zone. Tape libraries occupy network connections for long time intervals. In addition to that, if the

connection to the tape library is interrupted during the transmission, it must rewind and start transmitting again from the beginning. Thus, having more than one tape library in a zone would be an invitation for this kind of problem.

In *TestQuery*, we count the tape libraries for each zone in the network and check if their number is less than two for each. In the *FailedElementsQuery*, we reuse the SQL fragment in *TestQuery*. In contrast to the *TestQuery*, *FailedElementsQuery* is a valid SQL statement that can be directly sent to the SQL engine. Since we want to retrieve the information about configuration entities that cause the test condition to fail, we reverse the condition in *TestQuery*.

In the *Action* part, we demonstrate a notification by e-mail about the zones in the SAN that include multiple tape libraries.

The element *Method* describes a Java method call by giving the signature of the method and its parameters. In our example, the method `sendWarningMail` is called with two arguments: The first one is the email address which is explicitly given as the content of the element *MethodParameter*. `ParamType="FixedValue"` indicates that the parameter value is given as a constant. The second one is the result of the SQL query given in the element *FailedElementsQuery*. To specify this, `ParamType="FailedElements"` is used.

```
<Test
  Name="notMoreThanOneTapeLibInZones"
  Description="There should be only one tape library
               in each zone."
  Severity="Warning"
  Priority="300"
  IsActive="1">
  <TestQuery QueryType="StandardSQL">
    2 > ALL(SELECT COUNT(entity_id)
      FROM tpc.t_view_zone2member_ent_con_to_switch
      WHERE prefix_id = 'tapelibrary:'
      GROUP BY zone_id)
  </TestQuery>
  <FailedElementsQuery
    Description="Zones with more than one tape library.">
    SELECT zone_id, count_entities
    FROM (SELECT COUNT(entity_id) count_entities
```

```
          FROM tpc.t_view_zone2member_ent_con_to_switch
          WHERE prefix_id = 'tapelibrary:'
          GROUP BY zone_id))
    AS res
    WHERE count_entities > 1
  </FailedElementsQuery>
  <Action>
    <Method InvokedOn="Failure">
      <MethodName ReturnType="void">
        de.uni_tuebingen.sanchk.sendWarningMail
      </MethodName>
      <MethodParameter ParamType="FixedValue"
                       JavaType="String"
                       ParamIndex="0">
        adm@foo.de
      </MethodParameter>
      <MethodParameter ParamType="FailedElements"
                       JavaType="String" ParamIndex="1"/>
    </Method>
  </Action>
</Test>
```

3.4 Empirical Data

3.4.1 Implementation Details

We developed a first prototype of our SANchk configuration checking tool in about 800 lines of Java code and about 170 lines of SQL for 27 unparameterized rules (shown in Appendix A) that we used for our initial tests. When compiled, the class files except for the libraries for database connectivity and XML parsing amount to nearly 34 KB. We used DB2 libraries for the database connectivity and the Apache Project's Xerces Java Parser for XML parsing, which occupy 1.02 MB and 1.39 MB on disk respectively.

Figure 3.8 shows the package diagram of our prototype. The diagram represents the dependencies between the packages. The package *apps* is the starting point for the execution. This package invokes methods in package *eval* to evaluate the SAN configuration.

Then, the transformation methods in package *transform* are called to create HTML reports from particular result XML files and XSL files for data representation. The package *reflect* contains classes that use the methods from the *java.reflect* package. The package *data* uses DB2 library functions and provides auxiliary methods for other classes in the system. In analogy to that, package *util* contains classes helping in XML parsing. The package *conf* is concerned with the settings of the tool, such as the JDBC driver, or XSL files to use in reporting.

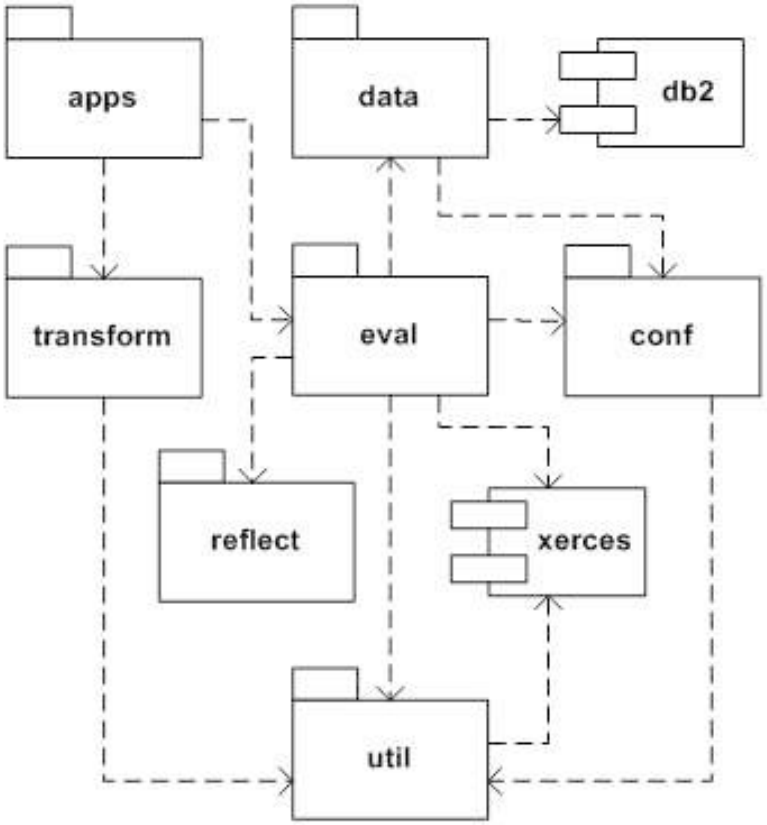

Figure 3.8: *Package diagram.*

3.4.2 Initial Tests

We conducted tests initially with four small but real systems (one of them is a demo system for IBM customers). Table 3.1 contains the results of the tests and information about the systems that were tested. In the tests, we first used a set of 27 unparameterized rules, and then all of the rules in Appendix A. We detected a total of 13 configuration problems in these 4 systems. This number does not include problems found by the rules with parameters, as parameter settings often reflect soft constraints. The problems types that we detected were as follows: In System 0 and System 3, we had problems of type redundancy. In System 1, it was detected that the physical links limit was exceeded beside some redundancy problems. Finally, an excess of logical paths limit and a violation of a port-zone relation rule were detected in System 2. The execution times can be seen in Table 3.1. The testing time does not increase proportionally to the number of the rules, indicating that a noticeable amount of time is needed for initialization.

We did not record the elapsed time for System 3 since that database was accessible only through an operating system virtualization software. Therefore, the performance results for System 3 are not comparable to the others. Database sizes do not reflect system complexities and vary considerably because they include performance statistics gathered by TPC.

Table 3.1: *Test Results*

SAN	System 0	System 1	System 2	System 3
Hosts	8	7	4	6
Storage subsystems	2	2	5	5
Switches	12	12	2	4
DB size (MB)	779	2856	2959	151
Detected problems	2	5	1	5
Testing time (s) with 27 rules	62	114	40	-
Testing time (s) with 71 rules	76	155	61	-

For the tests in Table 3.1, an Intel® Pentium® 4 with 3.00 GHz and 1 GB RAM was used.

3.4.3 Test on a Production SAN in Industry

We also had the opportunity to check the configuration of a very well managed SAN at the data center of IZB Informatik Zentrum, a major customer of IBM in Germany.

The SAN hosts data from a large number of banks. It contains 21 storage subsystems, 4 directory class switches, 18024 data paths, 1635 storage volumes, 649 zones and 1900 ports.

This was the first test-run of SANchk at a major site, so that the results should be viewed as preliminary.

Testing time was 258 seconds excluding the rule K.2 in Appendix A. If this rule was included, testing time was increased to 8079 seconds. In contrast to the other rules, this rule mines implicit data from the database where it is spread over a number of tables. This result led us to provide feedback to the designers of the database.

Overall, SANchk did not find any real configuration problems. The reported problems are false positives of the following 3 categories: (a) Rule parameters not tailored to the specific installation. We did not have the time to work closely with the customer in order to obtain realistic parameters for the rules. Parameters generally reflected our own guesses or were set with exemplary values. (b) Misinterpretation of database entries. For example, "IBM" and "IBM Corp." were interpreted by SANchk as different vendors. (c) Rules no longer applicable to the modern hardware of IZB, which is more interoperable.

Nevertheless, the results give an indication of SANchk's performance on a large real system and of the types of reports one can expect. In addition, we got feedback concerning the database, which can be improved to better support the tests.

3.5 Aperi Integration

SANchk was developed initially as a stand-alone tool. In a subproject, SANchk was transformed into several plugins to enable it to work with Aperi. At the end, the plugins were donated to the Aperi project. In the following, we give an overview of the Aperi project [24] and present the integration of SANchk into Aperi.

3.5.1 Aperi Project

Aperi is an open source project at the Eclipse Foundation, which aims to provide a storage management framework based on open standards. The functionalities of Aperi include

discovery of a SAN and its components, management of SAN components, retrieval and modification of their configuration, tracking storage capacity and bandwidth, and management of availability of resources.

Aperi uses Equinox, the reference implementation of the *"R4 core framework specification"* from the standard *Open Services Gateway initiative (OSGi)*, and the *Rich Client Platform (RCP)*, which contains the main part of the Eclipse framework. By the means of Equinox, plugins can be directly integrated into Eclipse and so also into Aperi, to extend the framework.

Aperi is comprised of several major components that communicate with each other: Two servers (data server and device server), an RCP-GUI, a database as repository for Aperi, and Aperi host agent that runs on hosts and collects data. The data server is the central point of the interaction between the components, since it has connections to other components and to the database. All components of Aperi are based on the OSGi framework and the GUI is based on the Rich Client Platform. Aperi is developed with Eclipse. Hence, the features of OSGi and RCP can be used during the implementation.

As a result of using the OSGi framework, Aperi provides for each component its own Extension Points or permits the use of Extension Points of Eclipse.

The communication with the devices can be carried out in Aperi using its own agents or SNMP and CIM agents.

3.5.2 SANchk as Aperi Plugins

There are different ways to let application components communicate in Aperi [36].

- Apache SOAP with XML-RPC:

  One of the ideas was to use an interface in Aperi that allows the use of XML-RPC functionality. The Aperi SOAP package enables sending and receiving XML-RPC messages. However, RPC messages are sent to a receiver, so that it executes a procedure. To send a response, the receiver should also send a new XML-RPC message. Considering the implementation need for the management of the communication both for the SANchk GUI and the SANchk application, the approach seems to be costly.

- A single plugin in the Aperi GUI:

  Another way would be to develop a single plugin for SANchk that contains both SANchk application and its GUI. Such a plugin could then be integrated into the

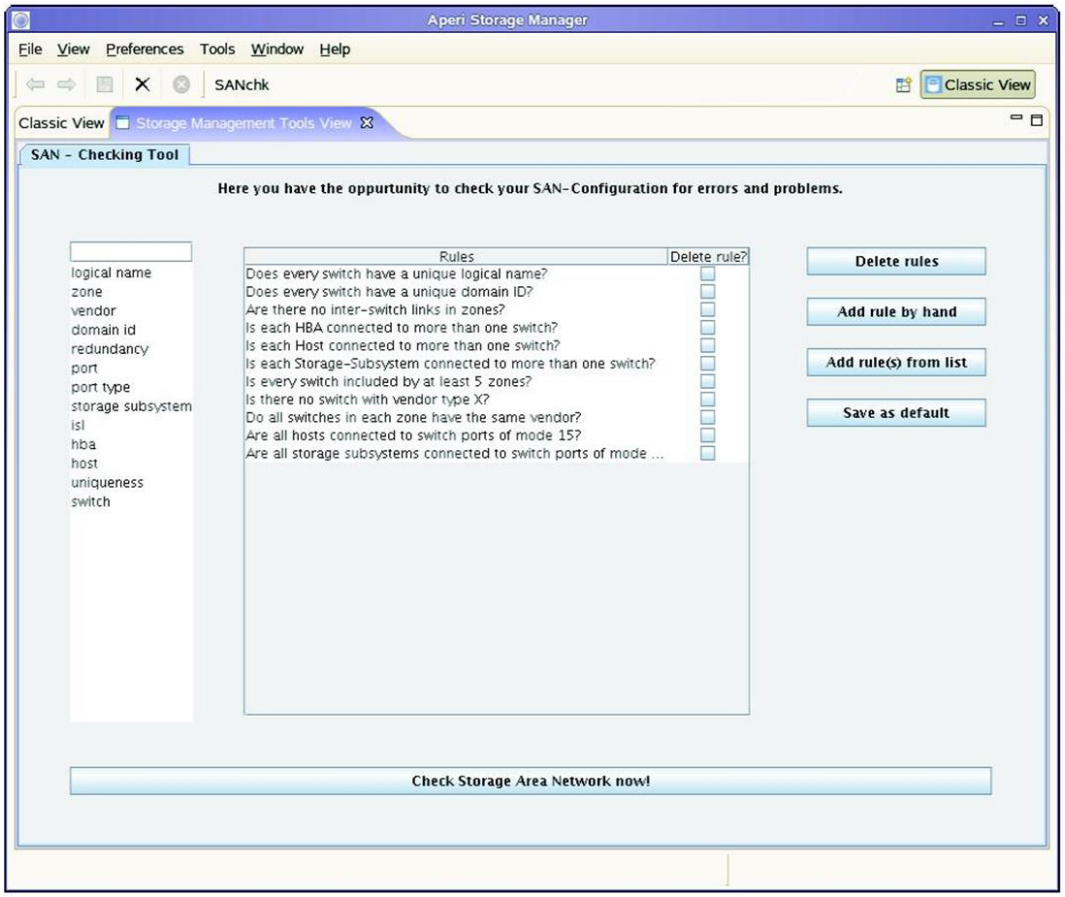

Figure 3.9: *SANchk GUI in the Aperi Storage Manager.*

Aperi GUI, and so the communication with other components of Aperi would not be necessary. This would be easy to do, but the architectural concept of Aperi would be undermined. Moreover, it would not be possible to use the infrastructure of Aperi for the database connections, so that the SANchk plugin would have to contain the database connection information and libraries of the corresponding DBMS.

- Aperi's own request-response architecture:

  Aperi possesses its own request-response architecture, whose center is in the data server. To provide this functionality, Aperi uses the Service Provider, Request Handler, Request and Response objects. In the data server, there are different Service Providers that can be reused like a Service Provider for Request objects that are sent to the data server by Aperi GUI. These Service Providers can be used from the outside by means of an extension point that is defined in the data server. Ser-

vice Providers are responsible for the thread management of Request and Response objects.

In order to process own Request and Response objects, a Request Handler must be implemented in the plugin. Then, this Request Handler must be bound through an Extension Point to a Service Provider. One can also implement his own Service Providers. To be able to use his own Request Handler, one must also implement his own Request and Response objects.

In our implementation, we followed this approach, since it does not cause the problems that other approaches do.

In the following, we present three plugins that were developed to integrate SANchk into Aperi.

- de.uni_tuebingen.informatik.smt

  This plugin contains the SANchk application and two classes for the communication with the GUI plugin. The interface for the communication between the GUI and the plugin itself is a Request Handler (SmtHandler) which is placed in this plugin. The class SmtHandler accepts Request objects from the GUI component, recognizes the command, and returns a Response object. This plugin is loaded by the data server as an OSGi bundle.

- de.uni_tuebingen.informatik.smt.gui

  In addition to the whole GUI for SANchk, this plugin contains also the components that are necessary for its integration into Aperi. An extra Eclipse view, auxiliary classes to organize the GUI in several tabs (to integrate additional tools) are such components. This plugin is loaded by invoking the SANchk in Aperi GUI. Figure 3.9 shows the SANchk GUI.

- de.uni_tuebingen.informatik.smt.common

  In this plugin, classes are contained which are commonly used by the GUI plugin and the main plugin. For example, it contains a class defining what a Request object should contain. It also contains a class for each type of Response object.

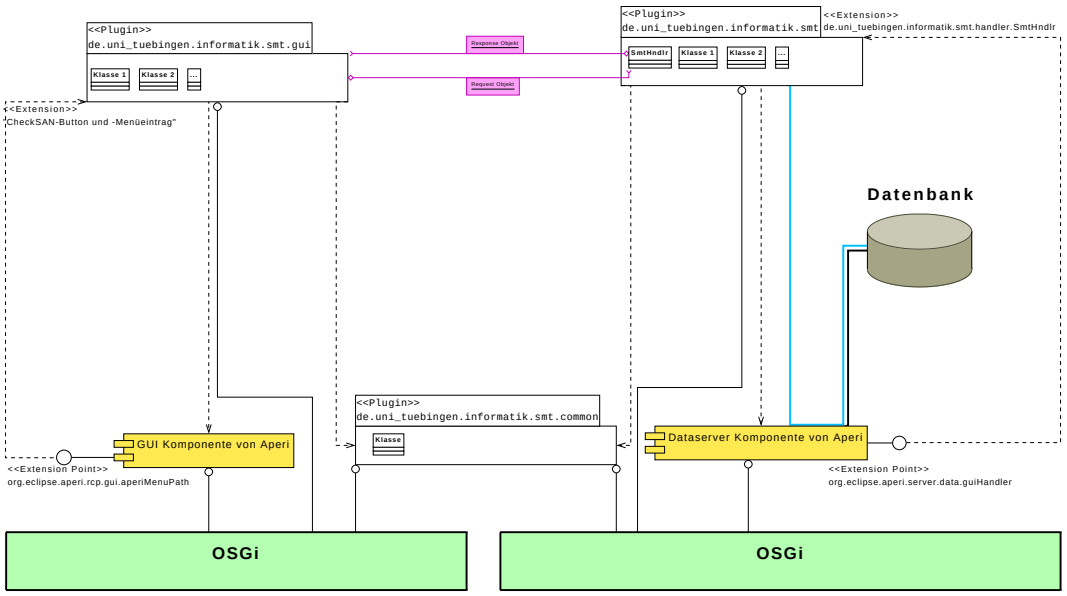

Figure 3.10: *Interaction of plugins, database, data server and RCP GUI of Aperi.*

3.6 Related Work

3.6.1 Policy Based Validation of SAN Configuration by Agrawal et al.

Policy based validation of SAN configuration has been proposed previously by Agrawal et al. (2004) [5]. They proposed a SAN configuration validation system and introduced five collection policy types that are needed to check SAN configuration policies in addition to the basic logical and comparative operations. We use these policy type definitions and show that they can be formulated as SQL queries by our method. We also show how evaluation and action handling mechanisms of such a system can be realized using well-known software technologies.

Agrawal et al. propose a SAN configuration management architecture that assumes a database to hold the SAN configuration data. The configuration database should be filled in either using the SMI-S representation of the SAN in case of a deployed SAN, or by a SAN planning software in case of a planned SAN.

The SAN configuration management system provides a Policy Specification Tool that should be used by a system administrator to write configuration policies. The administrator creates policies by combining system attributes with functions provided by the policy

language using a graphical user interface. The policy language allows to define configuration policies in a four-tuple: scope, condition, action, and priority. Scope specifies which components a policy is applicable for, and identifies the attributes that can be used in the condition part of the policy. Condition is a logical expression that compares specified values with the values of the attributes. Action describes the tasks that should be done if the logical expression in the Condition is satisfied. Priority states the relative importance of the policy. In cases where multiple policies are applicable, the policy with the highest priority is used. Policies are stored in a policy repository by the Policy Specification Tool.

Two types of repositories are available: the Remote Centralized Policy Database and the Local Policy Database. The Remote Centralized Policy Database serves as a central repository for many SAN managers. Policy updates are pushed from the Remote Centralized Database to the Local Repositories of each SAN manager by a distribution mechanism. The architecture allows to receive policies from a Remote Centralized Policy Database or to define policies for the Local Policy Database using a Policy Specification Tool.

A component called the Policy Event Generator is designed to monitor changes to the SAN configuration, and trigger an event on a change while identifying the policy scopes that are applicable for the event. The Policy Event Generator feeds the Policy Evaluator with the applicable scopes, so that they can be processed further. The Policy Event Generator can be used by a system administrator to trigger a configuration checking for subsets of the SAN.

The Policy Evaluator receives a list of scopes from the Policy Event Generator and selects the policies that are applicable to the scopes. In the next step, the Policy Evaluator requests the values of the attributes that are found in the policies from the SMI-S Database Interface. The SMI-S Database Interface is an abstraction layer for the configuration data retrieval. It collects the configuration data either from the SAN Database or directly from the physically deployed SAN. A cache mechanism is used to avoid retrieving the same configuration data multiple times. The Policy Evaluator computes the condition of each selected policy using the attribute values. The policy with the highest priority whose condition was satisfied is then selected and its action is sent to the Action Handler.

Eventually, the Action Handler invokes the specified actions that were passed by the Policy Evaluator. The actions may reconfigure the SAN, e. g. by changing the zone configuration, or just notify the administrator about a problem via e-mail.

Agrawal et al. modeled SAN configuration data inspired by the SMI-S standards. They divide the configuration data for a given component type into data describing intrinsic

properties and data describing associated components. The serial number, state, vendor name, or model name of an HBA for example would be its intrinsic properties, while the ports connected to it, or the host in that the HBA is connected are defined to be the the components associated with an HBA. Based on this distinction of configuration data, they define three types of policies.

- Intra-component Policies:

  Intra-component Policies are policies that define their conditions on the intrinsic properties of a SAN component. An example policy would be "An HBA from vendor X should have firmware level greater than Y."

- Inter-component Policies:

  Inter-component Policies are policies that define their conditions on the intrinsic properties of a SAN component with respect to the intrinsic properties of another SAN component. An example policy would be "A host with operating system AIX version X should not have an HBA from vendor Y with firmware level greater than Z."

- Collection Policies:

  Collection Policies are policies that define their condition on the properties of a group of SAN components. An example policy would be "All HBAs in a host should be from the same manufacturer."

  Collection Policies can be further divided into five types: cartesian property, graph, exclusion, many-to-one, one-to-one. These policy types have been discussed previously.

3.6.2 CIM Constraint Language ($\mathcal{CCL}$) by Sinz et al.

In an Apache configuration file there can be more than 200 different so-called *directives*. A directive is the Apache synonym for the textual representation of a configuration option. The Apache Web-server is designed as a modular program, and can thus be extended by about 40 different modules, each of them providing additional configuration options or directives. For any module's directives to get activated, the related module has to be loaded. Moreover, some modules and directives are obsolete and should not be used anymore, e. g. for security reasons. The complexity of the configuration process of the

1. $\exists^{=1}$ServerProperties.ServerRoot
2. [ServerConfiguration](ServerProperties.MinSpareServer < ServerProperties.MaxSpareServer)∧ [ServerConfiguration](ServerProperties.MaxSpareServer > 1)
3. |HostProperties.ServerName| = |HostConfiguration.Name|
4. [HostProperties]¬isPrefixOf(HostProperties.DocumentRoot, HostProperties.ErrorLog)
5. [HostConfiguration]HostProperties.⟨HostAddress, HostPort⟩ ⊆ ListenSetting.⟨ListenAddress, ListenPort⟩
6. ∃ServerProperties.ConfigName ∧ ∃ServerProperties.PidFile

Figure 3.11: *Formalization of some Consistency Properties in $\mathcal{CCL}$.*

Apache Web-server is also reflected by the large number of books about this Web-server (see, e. g. [34]), and the considerable space that these books spend on configuration issues and the related question of security.

Some examples may illustrate the kind of conditions we have to deal with:

- The *ServerRoot* directive must be specified exactly once in the server configuration.
- Apache allows for setting a minimum and maximum number of spare servers via directives *MinSpareServers* and *MaxSpareServers*. We require *MaxSpareServers* > *MinSpareServers* and *MinSpareServers* > 1.
- When several virtual hosts are running on the same server, each of them must have its own unique *ServerName*.
- For security and privacy reasons it is strongly recommended that the log files of all virtual hosts are not visible to the outside world. For example, the *ErrorLog* file should not be located in *DocumentRoot*.
- All virtual servers should have their own log files.

Some of these constraints may be hard constraints, in the sense that they are indispensable for a correct functioning of the Web-server. Other constraints may recommend sensible values (soft constraints) that are appropriate for most Web-server installations but that are not enforced. Additionally, there may be site-specific local constraints that reflect the company's (or site maintainer's) security policy, user accessibility rights and other features. Part of such constraints can stem from the Apache documentation itself [8], others may have to be collected and specified by Web-server administrators or other personnel.

Given a powerful and generally applicable system model like CIM, new perspectives on verification tasks arise, concerning, e. g. consistency of site-specific policy rules, checking of individual configurations, or computation of implied constraints. Combining CIM's powerful data model with the flexibility and generality of a formal constraint-based expert system seems particularly promising.

A suitable language to formulate the constraints has to reflect both CIM peculiarities (handling of classes, instances, properties and the structural relations between them) as well as basic logical concepts known from, e. g. Boolean logic and other general non-logical concepts such as arithmetic or string processing.

The CIM constraint language, $\mathcal{CCL}$ [49], is partly influenced by Description Logic [10] and partly resembles variable-free predicate logic. $\mathcal{CCL}$ consists of three kinds of expressions: v-expressions, a-expressions and f-expressions. V-expressions represent arbitrary finite sets of property values (numbers, strings,...), a-expressions are the atomic propositions of our language, and f-expressions constitute formulae. These expressions are recursively defined as follows:

v-expressions (denoted by s,t,...)**:**

- $C.P$ where C is a class name and P a property name.
- $C.\langle P_1, \ldots, P_k \rangle$ where C is a class name and $P_1, \ldots, P_k$ are property names.
- v where v is an arbitrary property value constant (string, number, etc).
- $f(s_1, \ldots, s_k)$ where f is a k-ary (interpreted) function and $s_1, \ldots, s_k$ are v-expressions.

a-expressions:

- $R(s_1, \ldots, s_k)$ where R is a k-ary (interpreted) predicate and $s_1, \ldots, s_k$ are v-expressions.
- $\exists^{\geq n} C$ where n is a natural number and C a class name.
- $\exists^{\geq n} C.P$ where n is a natural number, C a class name and P a property name.

f-expressions (denoted by F,G,...)**:**

- Boolean logic expressions built from connectives $\wedge$, $\vee$, $\neg$, true, false, $\Rightarrow$, $\Leftrightarrow$, and auxiliary symbols (and), using a-expressions as atoms.
- $[C]F$ where C is a class name and F an f-expression.

In Figure 3.11, formal variants of part of the specification stated in natural language above are given, as well as some examples taken from the Apache documentation [8]. These are to be understood as follows:

1. Property *ServerRoot* is defined exactly once.

2. For each server configuration, the *MinSpareServer* and *MaxSpareServer* properties are set as mentioned above. Here, we also used the comparison operators $<$ and $>$.

3. Each virtual host has its own unique server name. Here, we used an additional unary function, $|\cdot|$, computing the cardinality of its argument set, and the key property *Name* of CIM class *HostConfiguration*.

4. The error log should not be stored in directory *DocumentRoot* or a subdirectory thereof. Here, we used a binary predicate on sets of strings (*isPrefixOf*), returning true if all elements of the first set are prefixes of all elements of the second set. The context operator assures that these sets are singletons.

5. The address/port pair of each virtual host must be an address/port the Web-server is listening to (see [8]).

6. A configuration name and PID file must be specified for the Web-server.

3.6.3 Policy Management for Autonomic Computing (PMAC) by IBM

A general framework for policy-based management of networked systems is *Policy Management for Autonomic Computing (PMAC)* by IBM [6]. PMAC policies have the form *event–condition–action* and are expressed in the *Autonomic Computing Expression Language (ACEL)*, which is an XML-based and strongly-typed expression language with its own function set [3]. PMAC provides modules for policy creation, policy storage, policy evaluation, and policy enforcement as well as a policy ratification module.

The policy information model used in PMAC is inspired by the CIM (Common Information Model) policy model. The CIM policy model enables a unified and consistent representation of policies on the element that can be described by the information model including the configuration data.

The PMAC policies comprise four components: conditions, actions, priority, and role. The conditions are Boolean expressions that will be evaluated to the values true or false. If the conditions for a policy are evaluated to true, the policy is called applicable. In case that a policy is applicable, the actions of the policy get executed. The priority is a non-negative integer and used to determine the policy to apply when several potentially

conflicting policies are applicable. The role defines the context in which the policy will be relevant. For example, a policy on configuration data of tape libraries can have the role "tape-library".

The PMAC platform is based on the IBM Autonomic Computing (AC) architecture. The AC architecture defines a framework for self-managing IT systems [13]. The AC architecture presents two key abstractions: an *autonomic manager* (AM) and a *managed resource* (MR). An AM monitors computing resources, analyzes the status of the resources, plans action for the resources, and executes actions. An MR is a computing system that is controlled and managed by the AM. The relationship between AM and MR is one-to-many. An AM can control one or more MRs, whereas an MR is controlled by exactly one AM. The communication between an AM and an MR is realized through MR's management interfaces using *sensors* and *effectors*. The sensors provides AM with the internal state of the MR, and the effectors let the AM invoke actions on the MR.

PMAC not only adopts the AC concepts but it also extends AC's bindings model which mainly is based on the web services. The AM in PMAC exposes a set of Java APIs so that the policy module is embedded as a library in applications. The AM can also be run in an application server as a stateless session Enterprise Java Bean (EJB) or as a web service to managed resources that are located remotely.

Figure 3.12 gives an architectural overview of PMAC. PMAC has three architectural layers: definition and persistence, policy evaluation and decision, and policy enforcement layers. The definition and persistence layer contains one or more policy definition tools (PDT) for policy authoring and a policy editor storage (PES) for policy deployment and persistence. In the policy evaluation and decision layer, PMAC provides autonomic managers (AM) for policy evaluation. Finally, managed resource-side components (MR libraries) for policy enforcement reside in the policy enforcement layer.

The policies are authored using a policy definition tool (PDT) and then stored in the policy editor storage (PES). PMAC allows the concurrent use of multiple policy definition tools by different policy authors. This makes the consistency checking of the policies at the policy editor storage (PES) necessary. PMAC uses as an XML-based policy language called the Autonomic Computing Policy Language (ACPL).

The policy editor storage (PES) component serves as a central repository where policies from different policy definition tools are stored. PES keeps the autonomic managers up-to-date with new policies according to their scope. The PES component stores policies either on a file system or in a relational database according to the configuration or application

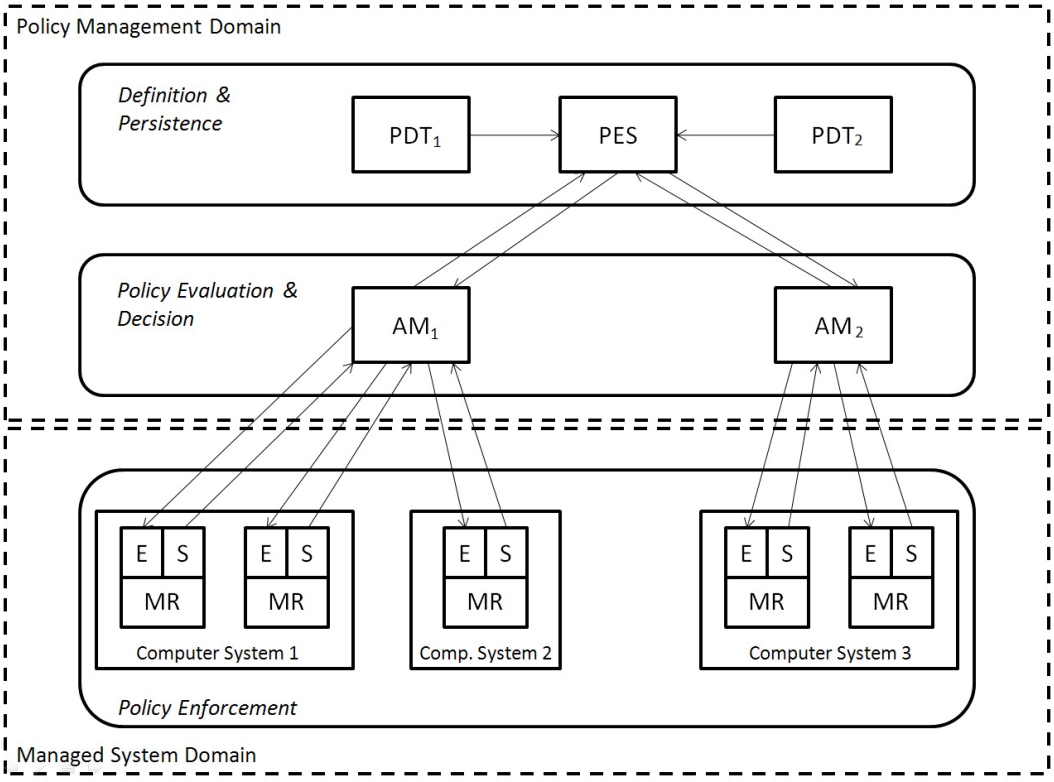

Figure 3.12: *PMAC architecture overview.*

requirements.

The autonomic manager component receives policies from the policy editor storage and registers managed resources that need its policy guidance. In order to determine if an MR needs policy guidance, the AM reads the state of the MR using the MR's *sensor* interface. After that, the AM evaluates relevant policies using the state of the MR and plans actions. The AM executes an action on the MR using the *effector* interface of the MR, or only return the result to the MR according to the result type of the policy evaluation.

Autonomic Computing Expression Language (ACEL)

The Autonomic Computing Expression Language (ACEL) is a part of ACPL that enables writing policy rules [3]. As a part of ACPL, ACEL is also a entirely XML-based and strongly-typed language.

ACEL uses nine primitive and two composite data types. The primitive data types that are based on the XML standard are as follows: Boolean, Short, Integer, Long, Float,

Double, String, Calendar, and URI. The composite types are CompositeData and Collection. A CompositeData corresponds to the XML complex type. A Collection object is an ordered collection of ACEL expressions. ACEL provides also a Reference element that contains a pointer or a handle to an expression defined elsewhere.

ACEL provides a rich set of functions. The functions in ACEL can be grouped by their return type. There are three main groups of functions in ACEL: numeric functions, Boolean functions, and String functions. Numeric functions include type cast functions (e. g. `ToInt`, `ToFloat`, `ToLong`, etc.), arithmetic operators (e. g. `Plus`, `Product`, `Remainder`, `Max`, `Log`, `Pow`, `Ceiling`, etc.), and calendar field extraction functions (e. g. `GetDayOfMonth`, `GetHour`, `GetYear`, etc.). Boolean functions include a type cast function (`ToBoolean`), logical functions (`And`, `Or`, `Not`, `Xor`), relational functions (e. g. `Greater`, `Equal`, `GreaterEqual`, etc.), string matching functions (`Begins`, `Contains`, `Ends`), calender comparison functions (`IsAfter`, `IsBefore`, `IsWithin`), and set functions (e. g. `Belongs`). String functions include type cast functions (`ToString`), substring extraction functions (e. g. `LeftSubstring`, `RightSubstring`, `ReplaceSubstring`, etc.), and case conversion functions (`ToUpper`, `ToLower`).

A condition in a PMAC policy is defined as an ACEL expression of type Boolean with variables corresponding to *sensor* names. In ACEL, variables are represented by the `PropertySensor` element. A `PropertySensor` element is embedded in a function element with its value and data type. The following example demonstrates an expression that multiplies a `Float` constant 3.14159 times the `PropertySensor` diameter.

```
<Product>
  <FloatConstant>
    <Value>3.14159</Value>
  </FloatConstant>
  <PropertySensor propertyName="diameter" />
</Product>
```

The XML representation of the ACPL policies is meant to be for internal processing, policy persistence, and deployment. A policy definition tool works with the XML representation of the policies. In case that a policy author does not use a policy definition tool, PMAC supports also a simple policy language (SPL). SPL provides a way to express a policy like in a programming language instead of the XML syntax. Consider the Boolean expression below for example:

```
<And>
  <Not>
    <Equal>
      <PropertySensor propertyName="NumberOfPorts" />
      <IntConstant>
        <Value>16</Value>
      </IntConstant>
    </Equal>
  </Not>
  <Equal>
    <PropertySensor propertyName="VendorId" />
    <IntConstant>
      <Value>5</Value>
    </IntConstant>
  <Equal>
    <PropertySensor propertyName="Type" />
    <StringConstant>
      <Value>Core Switch</Value>
    <StringConstant>
  </Equal>
</And>
```

The same expression can be written using SPL as follows:

```
(Sensor(NumberOfPorts) != 16) and (Sensor(VendorId) = 5) and
(Sensor(Type) = "Core Switch")
```

The PMAC implementation of SPL provides a subset of ACPL functionality. Policies written in SPL are not parsed and evaluated directly, instead they are translated to ACPL and parsed and evaluated as usual. Both the policies in SPL and their translated versions are stored in PES.

Policy Ratification

PMAC has a distributed architecture, where multiple policy authors can write policies independently. In such a system, the consistency of the policies must be ensured. Therefore, *policy ratification* mechanisms are implemented in PMAC to address this issue [4].

Policy ratification is the process of certifying a policy by analyzing its relationships with other policies in the system before the policy is activated or ratified. PMAC provides a set of general operations that are used for policy ratification: dominance check, conflict check, and coverage check.

Dominance check: A policy x is dominated by a group of policies $Y = \{y_1, \cdots, y_n\} (n \geq 1)$ when the addition of x does not affect the behavior of the system governed by Y. An example of this relationship can be seen between the following policies: "password length ≥ 6" and "password length ≥ 8". The first policy is dominated by the second, since it is subsumed by the latter. The ratification operation for this example would be to determine that $(p.length \geq 8) \Rightarrow (p.length \geq 6)$.

Conflict check: Conflicting policies are policies that specify goals that cannot be satisfied simultaneously. For example, the policies "password length > 8" and "$4 \leq$ password length ≤ 8" cannot be met at the same time. Regarding the configuration policies, two policies conflict when they specify different configuration values: "disk quota = 2 GB" and "disk quota = 1 GB". In the event-condition-action policy model, a conflict between two policies can also occur when the conditions of the policies can be satisfied at the same time while having incompatible actions. Conflicting policies would be resolved by human administrators by marking policies inactive or by setting the relative priorities of the policies.

Coverage check: Administrators may want to know if policies have been explicitly defined for a certain range of input parameters. In the event-condition-action model, the administrator may want to ensure that at least one has a true condition regardless of the values of input parameters. The ratification operation in this case is to find out if a set of Boolean expressions *implies* another Boolean expression, where the second expression represents the value space that is to be covered.

Consistent Priority Assignment

Policy systems usually assign an integer value to each policy, so that the policy with the highest priority is executed when several policies apply. Manual assignment of priorities is infeasible when the number of policies increases. In case that the assignment of priorities is not done with care, problems may arise when a new policy is to be inserted. The conflict

resolution module of PMAC automatically assigns priority values to the new policies by adjusting the priority values of the related policies, when given only the *relative* priority of a new policy. To accomplish this, PMAC uses an algorithm that is a modified version of an *order-maintenance* algorithm in a list. On average, the amortized reassignment of priorities is guaranteed to be done in constant time.

3.6.4 Policy Middleware Architecture for Managing IT Systems by Agrawal et al.

Agrawal et al. [1] present a policy middleware architecture for managing IT systems and applications that are distributed over multiple networks and administrative domains. The architecture provides means for a platform-neutral and extensible specification of policies, the local ratification of policies, and the transformation of the policies in order to match the local environments.

3.6.5 eXtensible Access Markup Language (XACML)

XACML is a declarative access control policy language implemented in XML [40]. XACML also provides a processing model that describes how to interpret the policies. XACML is a replacement of IBM's XML Access Control Language (XACL) [30]. XACML was standardized in 2003 by the Organization for the Advancement of Structured Information Standards (OASIS). The members of the OASIS and the developers of XACML are among others the companies Sun Microsystems, Entrust, IBM, and OpenNetwork. Since 2005, XACML Version 2.0 was announced as an OASIS standard [41].

According to the OASIS XACML Technical Committee, XACML is developed to standardize the use of declarative policy to control access to resources. XACML provides a core schema and corresponding namespace for the expression of authorization policies in XML. The resources that are subject to XACML policies are also defined in XML.

XACML enables the use of arbitrary attributes in policies, role-based access control, security labels, time/date-based policies, "deny" policies, and dynamic policies. XACML provides support for defining distributed access control policies that can be shared across different applications.

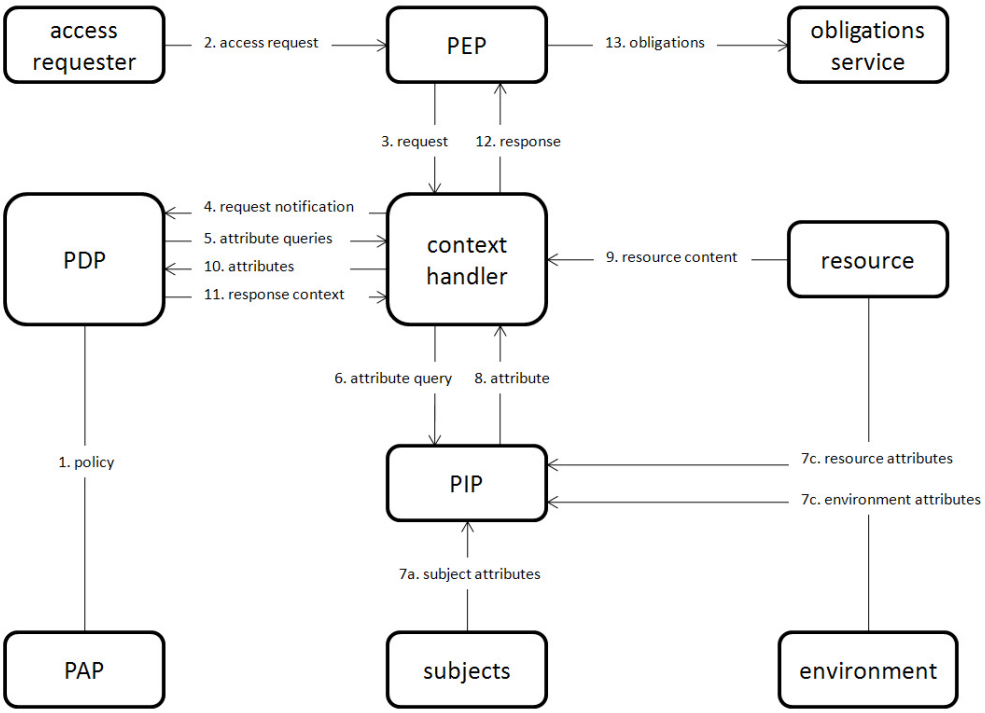

Figure 3.13: *XACML data flow diagram.*

Architecture of XACML

The key components of XACML architecture are policy administration point (PAP), policy decision point (PDP), policy enforcement point (PEP), and policy information point (PIP).

The policy administration point (PAP) creates policies or policy sets and makes them available for PDP. A policy set can be a set of policies, a set of other policy sets, or a policy-combining algorithm. The policy decision point (PDP) evaluates applicable policies and renders authorization decisions. The policy enforcement point (PEP) performs access control by making decision requests and and enforcing authorization decisions. Policy information point (PIP) acts as a source of attribute values. The access requester sends requests for access to the PEP. The PEP sends the request for access to the context handler in its native request format, optionally including attributes of the subjects, resource and action. The context handler constructs an XACML request context. Subject, resource and environment attributes may be requested from a PIP. The PIP obtains the requested attributes and returns them to the context handler. Optionally, the context handler includes

the resource in the context. The context handler sends a decision request, including the target, to the PDP. The PDP identifies the applicable policy and retrieves the required attributes and (optionally) the resource from the context handler. The PDP evaluates the policy and returns the response context (including the authorization decision) to the context handler. Then, the context handler translates the response context to the native response format of the PEP, and returns the response to the PEP. The PEP fulfills the obligations. If access is permitted, then the PEP permits access to the resource; otherwise, it denies access.

A rule is the main object of a policy. It includes three main parts: the target, the result (effect), and the condition or conditions. The constituents of a rule target are subject, resource, action elements, and optionally the environment. Using this information and the conditions, the PDP will return a Boolean value.

A policy has one or more rules, a rule-combining algorithm, and optionally obligations. The constituents of a policy, a policy set, or a rule include a target. The four attributes of the target specify the expression of these constituents.

The rule-combining algorithm specifies how the rule are combined within a policy. Multiple rules in a policy may check the same attribute. In case that not all of the rules are applicable, the algorithm decides about the result.

In comparison with a policy, a policy set includes one or more policies instead of rules and a policy-combining algorithm instead of a rule-combining algorithm.

An example policy rule in XACML that allows patients access to their medical records is given below [40].

```
<?xml version="1.0" encoding="UTF-8"?>
<Rule
  xmlns="urn:oasis:names:tc:xacml:1.0:policy"
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xmlns:ctx="urn:oasis:names:tc:xacml:1.0:context"
  xmlns:md="http://www.medico.com/schemas/record.xsd"
  RuleId="urn:oasis:names:tc:xacml:examples:ruleid:1"
  Effect="Permit">
  <Description>
    A person may read any medical record in the
    http://www.medico.com/schemas/record.xsd namespace
    for which he or she is a designated patient
```

```
</Description>
<Target>
<Subjects>
  <AnySubject/>
</Subjects>
<Resources>
  <Resource>
    <!-- match document target namespace -->
    <ResourceMatch
      MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
      <AttributeValue
        DataType=http://www.w3.org/2001/XMLSchema#string>
        http://www.medico.com/schemas/record.xsd
      </AttributeValue>
      <ResourceAttributeDesignator AttributeId=
        "urn:oasis:names:tc:xacml:1.0:resource:target-namespace"
        DataType=http://www.w3.org/2001/XMLSchema#string/>
    </ResourceMatch>
    <!-- match requested xml element -->
    <ResourceMatch
      MatchId="urn:oasis:names:tc:xacml:1.0:function:xpath-node-match">
      <AttributeValue
        DataType=http://www.w3.org/2001/XMLSchema#string>/md:record
      </AttributeValue>
      <ResourceAttributeDesignator AttributeId=
        "urn:oasis:names:tc:xacml:1.0:resource:xpath"
        Data Type=http://www.w3.org/2001/XMLSchema#string/>
    </ResourceMatch>
  </Resource>
</Resources>
<Actions>
  <Action>
    <ActionMatch
      MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
```

```
        <AttributeValue
          DataType=http://www.w3.org/2001/XMLSchema#string>read
        </AttributeValue>
      <ActionAttributeDesignator AttributeId=
        "urn:oasis:names:tc:xacml:1.0:action:action-id"
        DataType=http://www.w3.org/2001/XMLSchema#string/>
      </ActionMatch>
    </Action>
  </Actions>
  </Target>
  <!-- compare policy number in the document with
  policy-number attribute -->
  <Condition
    FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
  <Apply
    FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
    <!-- policy-number attribute -->
    <SubjectAttributeDesignator AttributeId=
      "urn:oasis:names:tc:xacml:1.0:examples:attribute:policy-number"
      DataType=http://www.w3.org/2001/XMLSchema#string/>
  </Apply>
  <Apply
    FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
    <!-- policy number in the document -->
    <AttributeSelector RequestContextPath=
      "//md:record/md:patient/md:patient-number/text()"
      DataType=http://www.w3.org/2001/XMLSchema#string>
    </AttributeSelector>
  </Apply>
  </Condition>
</Rule>
```

When the rule evaluates to "True", it emits the value of the `Effect` attribute. A rule target defines a set of decision requests that are applicable to the rule. The Subjects element may contain either a disjunctive sequence of Subject elements or AnySubject

element. The AnySubject element is a special element that matches any subject in the request context. The Resources element may contain either a disjunctive sequence of Resource elements or AnyResource element. The Resource element encloses the conjunctive sequence of ResourceMatch elements. The ResourceMatch element compares its first and second child elements according to the matching function. A match is positive if the value of the first argument matches any of the values selected by the second argument. This match compares the target namespace of the requested document with the value of "ttp://www.medico.com/scema.records.xsd". The MatchId attribute names the matching function. The ResourceMatch compares the results of two XPath expressions. The first XPath expression is /md:record and the second XPath expression is the location path to the requested xml element. The "xpath-node-match" function evaluates to "True" if the requested XML element is below the /md:record element. MatchId attribute names the matching function. The md prefix is resolved using a standard namespace declaration. The Actions element may contain either a disjunctive sequence of Action elements or an AnyAction element. The Action element contains a conjunctive sequence of ActionMatch elements. The ActionMatch element compares its first and second child elements according to the matching function. Match is positive if the value of the first argument matches any of the values selected by the second argument. In this case, the value of the action-id action attribute in the request context is compared with the value "read". The MatchId attribute names the matching function. A condition must evaluate to "True" for the rule to be applicable. This condition evaluates the truth of the statement: the patient-number subject attribute is equal to the patient-number in the XML document. The FunctionId attribute of the <Condition> element names the function to be used for comparison. In this case, comparison is done with rn:oasis:names:tc:xacml:1.0:fnction:string-equal;. This function takes two arguments of the "ttp://www.w3.org/2001/XMLScema#string" data-type. Functions can take other functions as arguments. The Apply element encodes the function call with the FunctionId attribute naming the function. Functions can take other functions as arguments. The Apply element encodes function call with the FunctionId attribute naming the function.

3.6.6 Ponder

Ponder is a declarative, object-oriented language developed at Imperial College, London. It is designed to specify both security and management policies [16] [15]. Ponder authorization policies can be implemented using various access control mechanisms for firewalls, operating

systems, databases and Java.

Key concepts of Ponder include domains to group objects to which policies apply, roles to group policies that specify the rights of a position in an organization, relationships to define interactions between roles, and management structures to define a configuration of roles and relationships belonging to an organizational unit such as department [33].

Domains

Domains provide a means to group the objects according to geographical boundaries, object type, responsibility and authority or to provide human managers a better overview of the system.

Membership of an object in a domain is not implicitly specified by its attributes, but defined explicitly. Objects are assigned to a domain through references and not capsulated by a domain. The concept is similar to a file system directory that holds files. A domain can be a member of another domain. In that case, the member domain is called a *sub-domain*. Members of a sub-domain are not direct members of the parent domain, as the files in a sub-directory are not direct members of the parent directory. A member of a domain can also be a member of another domain i.e. domains can overlap. Domains serve as an abstraction layer between the objects and policies, so that the addition or the removal of an object from a domain do not imply a change of the policies. Domains have been implemented as directories in an extended LDAP Service.

Ponder Primitive Policies

Ponder provides *authorization policies* to define what activities a member of the subject domain can perform on the set of objects in the target domain. Authorization policies are access control policies that express if a person, a group or a process is allowed to execute a certain method of an object. Authorization policies are divided into positive and negative authorization policies. A positive authorization policy specifies the actions that subjects are permitted to perform on target objects, while a negative authorization policy defines the actions that subjects are forbidden to perform on target objects.

Ponder enables the reuse of policies by means of policy types. Any policy element can be passed to an instance of a policy type as a formal parameter. The definition of a policy type and its two instances are given in the example below:

```
type auth+ PolicyOpsT(subject s, target <PolicyT> t) {
```

```
                action load(), remove(), enable(), disable();}

  inst auth+ switchPolicyOps=PolicyOpsT(/NetworkAdmins, /NRegion/switches);
  inst auth+ routersPolicyOps=PolicyOpsT(/QoSAdmins, /NRegion/routers);
```

In the example, two policy instances are created from a `PolicyOpsT` type. The first policy instance allows members of `/NetworksAdmins` to load, remove, enable, or disable objects of type `PolicyT` within the `/NRegion/switches`. The second policy instance allows members of `/QoSAdmins` to perform the same operations on the same object type in the `/NRegion/routers`.

It is also possible to create policies directly without using a policy type. A negative authorization policy is given below as an example for this:

```
inst auth- /negativeAuth/testRouters{
  subject /testEngineers/trainee;
  action performance_test();
  target <routerT>/routers;
  when time.between("0900","1700")
}
```

The example policy forbids trainee test engineers to perform performance tests on routers between the hours 0900 and 1700. The policy is stored within the `negativeAuth` domain.

Ponder also provides a number of other access control policies: *Information filtering* policies are used to transform input or output parameters in an action. Information filtering policies can be compared to the database concept "view". Some users may be granted the privilege to see only a restricted view of a table. An example of an information filtering policy would be a policy that permits a payroll clerk to read personnel records of employees below a particular level. *Delegation* policies permit subjects having privileges that base on an existing authorization policy to grant them to grantees to perform an action on their behalf. For example, a subject might grant read privilege on a file to a printer spooler in order to print a file. *Refrain* policies specify actions that a subject must not perform on target objects even though they may actually be permitted to perform the action.

Ponder primitive policies include also *obligation* policies. Obligation policies are event-triggered condition-action rules that specify actions subject must perform on objects in the target domain. An example obligation policy is given in the following:

```
inst oblig loginFailure{
  on 3*loginfail(userid);
  subject s = /NRegion/SecAdmin;
  target <userT> t = /NRegion/users^{userid};
  do t.disable() -> s.log(userid);
}
```

The obligation policy in the example is triggered by three consecutive `loginfail` events with the same `userid` and specifies that the `NRegion` security administrator must disable the user with `userid` in the `/NRegion/users` domain and then log the `userid` using a local operation of the subject. The operator "->" is a separator between operations in a sequence.

Ponder Composite Policies

Ponder composite policies provide a means to facilitate policy management in large, complex enterprises. Policies can be grouped and structured to reflect the organizational structure of an enterprise using Ponder composite policies. They help the administrators get a better overview of the policy system. Roles, relationships, and management structures can be named as concepts that Ponder provides for this.

A *role* is a group of policies that have the same subject. Roles are generally used to specify the privileges pertaining to a organizational position. Roles are defined implicitly. An example role definition is given below:

```
type role ServiceEngineer (CallsDB callsDb) {
  inst oblig serviceComplaint {
    on customerComplaint(mobileNo);
    do t.checkSubscriberInfo(mobileNo, userid) ->
       t.checkPhoneCallList(mobileNo) ->
       investigate_complaint(userid);
    target t = callsDb; // calls register }
  inst oblig dectivateAccount { . . . }
  inst auth+ serviceActionsAuth { . . . }
  // other policies
}
```

In the example, a service engineer role in a mobile telecommunications service is modeled by the role type `ServiceEngineer`. A service engineer responds to customer complaints and service requests. The role type is parameterized with the customer calls database. The database holds the data of the subscribers and their calls. The role type `ServiceEngineer` contains a series of different policies. The obligation policy `serviceComplaint` is triggered by a `customerComplaint` event with cell phone number of the customer as its attribute. After the policy is triggered, the subject must perform a sequence of actions to check the subscriber related data and investigate the complaint.

A *relationship* groups the policies that define the rights and duties of roles towards each other. An example relationship type definition is given below:

```
type rel ReportingT(ProjectManagerT pm, SecretaryT secr){
  inst oblig reportWeekly{
    on timer.day("monday");
    subject secr;
    target pm;
    do mailReport();
  }
}
```

In the example, the relationship type `ReportingT` is defined between the role types `ProjectManagerT` and `SecretaryT`. According to the obligation policy `reportWeekly`, the subject `secr` of the role type `SecretaryT` must mail a report to the target `pm` of the role type `ProjectManagerT` every Monday.

Ponder also provides the composite policy *management structure*. A management structure groups roles, instances, and nested management structures to model an organizational unit in a large enterprise. For example, a management structure type would be used to define a branch in a bank or a department at a university and then could be instantiated for a particular branch or department. An example management structure type definition is given below:

```
type mstruct BranchT(...){
  inst role projectManager = projectManagerT(...);
       role projectContact = SecretaryT(...);
       role softDeveloper = SoftDeveloperT(...);
```

```
  inst rel supervise = SupervisionT(projectManager, softDeveloper);
       rel report = ReportingT(projectContact, projectManager);
}
inst mstruct branchA = BranchT(...);
     mstruct branchB = BranchT(...);
```

In the management structure type, three role instances and two relation instances are defined. The relation instances govern the interactions between these roles. After that, two instances of the management structure type are created.

Ponder also enables policy consistency checking by providing *meta-policies*. Meta-policies are used to check if conflicts exist between different policies.

3.6.7 Other Related Work

We define policies in our system in XML and the policy conditions in SQL. Some examples of XML-based policy languages are *eXtensible Access Markup Language (XACML)* [39], *Trust Policy Language (TPL)* [31] and *Enterprise Privacy Authorization Language (EPAL)* [45]. TPL is used to map predefined business roles automatically to the users in the Internet. EPAL was developed to enable writing enterprise privacy policies to manage data handling according to fine-grained rights.

The languages *Policy Description Language (PDL)*, and *Simple Policy Language for CIM (CIM-SPL)* have been used for configuration checking. PDL is a language developed to enable the specification of goal-oriented policies in system management [35]. CIM-SPL allows to define policies in *condition-action* form and renders the policy model of the DMTF fully incorporating the CIM constructs [2].

There is also work on frameworks for policy based storage management. For example, Devarakonda et al. (2002) proposed a policy based storage management framework [17]. In their framework, they used collections of logical attributes describing application, data and storage levels. The high-level policies that describe the required QoS for the storage system are mapped to low-level actions using connectors.

As representative for work on formal verification of (semi-formal) UML-diagrams, we want to mention Dupuy-Chessa and du Bousquet's validation of UML models [23] and Meyer and Souquières' formalization based on the specification language B [37]. In contrast to their work, we do not use a powerful specification language using full predicate logic, but restrict our specification language to a variable-free logic that resembles Description

Logic [9], which potentially offers advantages for automated theorem proving tasks. Dong *et al.* present an approach to specify Semantic Web Services using Z in order to find errors in the ontology [22]. Work on validation and integrity checking of XML data can also be found in the literature [38].

Chapter 4

SANopt SAN Design Optimizer

SAN configurations have to fulfill Service Level Agreements (SLAs), which express performance requirements for the SAN. For example, an SLA may guarantee certain throughput rates or storage capacities for a host application. Many SLAs reflect data flow requirements that originate from the applications running on the servers. While designing a SAN, these requirements should be considered besides "best-practices" constraints like redundancy rules, and technical constraints like the limits of the resources, e. g. the number of ports of a device.

SANchk already checks whether a given SAN configuration respects a number of best practices rules. In this chapter, we consider the problem of configuring a SAN in an optimal way, while additionally taking a number of SLAs into account. Our primary concern is not to minimize hardware cost but to maximize the flexibility to accommodate changing SLA requirements in the future. This is because the most critical cost factor associated with a SAN is not raw hardware but operation downtime, e. g. for reconfiguration. In fact, downtime of business critical SANs is simply not an option beyond maybe one hour of scheduled maintenance once a year. Whenever a new or changed SLA cannot be accommodated because reconfiguration is too costly, it must be accommodated by purchasing additional new hardware. Therefore it is of prime importance for a new configuration to avoid future performance bottlenecks which may necessitate major reconfigurations.

Algorithms for the SAN design problem have been mostly developed to minimize the provisioning cost while meeting the QoS requirements and other constraints. In contrast, we attempt to achieve a flexible design by the optimization goal of the most evenly balanced distribution of workloads at all devices. To be more precise, for a given set of hardware

devices, we attempt to achieve a uniformly high proportion of free resources at each device, grouped by device types (e. g. X% free storage capacity on each storage device and Y% unused ports in each switch). In this way, we can increase the probability that the QoS requirements of all applications are still met, even if some of the applications demand more resources than planned, because bottlenecks are avoided. This in turn will result in a decrease of SLA violations and SLA penalties, respectively.

We formulate the SAN design problem as two Pseudo-Boolean problems [28] [27]. In general, both the Pseudo-Boolean decision and optimization problems are NP-hard [25].

4.0.8 Methods and Technologies

We use Pseudo-Boolean logic [12] (also known as 0-1 integer (linear) programming [46]) to formulate our constraint and optimization problems. Pseudo-Boolean logic is a specialization of linear integer programming, in which all variables (unknowns) have Boolean domains $B = \{0, 1\}$.[1] Thus, a Pseudo-Boolean problem consists of a set of constraints $C = \{c_1, \ldots, c_k\}$, where each constraint is of the form

$$a_1 \cdot x_1 + \cdots + a_n \cdot x_n \leq b \ .$$

The coefficients a_i and b are arbitrary integer numbers and the x_i are unknowns that can take values out of the Boolean domain B.

Pseudo-Boolean problems come in two flavors, either as decision problems or as optimization problems. For a Pseudo-Boolean decision problem, it is only asked whether the constraint set C possesses any solution. To obtain an optimization problem, the constraint set is extended by an optimization goal function of the form

$$\min \quad a_1 \cdot x_1 + \cdots + a_n \cdot x_n \ ,$$

which expresses that the linear constraint $\sum_1^n a_i \cdot x_i$ should be minimized (maximization is equally well possible). A solution then consists of a variable assignment $\alpha : \{x_1, \ldots, x_n\} \rightarrow B$, for which the goal function takes on a value as small as possible while all constraints of the set C are still satisfied.

The domains of the unknowns can be extended to integer ranges by using a bit-vector encoding. Thus, for example, to encode a variable y that can take values from the domain

[1] Often 0 is identified with *false* and 1 with *true* in this setting.

$D = \{0, 1, 2, 3\}$, we use two Boolean variables y_0 and y_1 and interpret the value of variable y as $y_0 + 2 \cdot y_1$.

In general, both the Pseudo-Boolean decision and optimization problems are NP-hard [25]. But in practice, it often turns out that encodings of real-world problems can be solved much faster than one could expect from the theoretical complexity.

4.1 SAN Storage Assignment Problem

In the storage assignment problem, we have as input the applications with their requirements (throughput and storage space), the information which host is serving which applications, and data about the storage devices' capacities (throughput, storage space). We want to find out, which applications should use which storage devices in order to get a distribution of the workloads on the devices as evenly as possible.

In order to formulate this problem, we use an algorithm that generates all the constraints that are needed, as well as the optimization goal function.

4.1.1 Assignment of Applications to Storage Devices

Let $D = \{d_1, \ldots, d_n\}$ be the set of storage devices, and $H = \{h_1, \ldots, h_m\}$ be the set of hosts. Storage devices $d_i \in D$ are modelled as pairs $d_i = (c_i, t_i)$, where c_i stands for the provided storage capacity and t_i for the provided throughput capacity of device i. In analogy, for all hosts $h_i \in H$, h_i is a pair $h_i = (c'_i, t'_i)$ consisting of the need for storage capacity c'_i of host i and its throughput requirement t'_i. To express that there is an assignment between a host and a storage device, we define a set $X = \{x_{i,j} \in \{0, 1\} \mid 1 \leq i \leq m, 1 \leq j \leq n\}$ with $x_{i,j} = 1$ iff there is a host i that is assigned to storage device j. Obviously, $|X| = |H \times D| = m \cdot n$.

Applications and hosts

Let $A_i = \{a_1, \ldots, a_k\}$ be the set of applications that run on host i. Applications a are pairs (c', t') consisting of storage capacity requirement and throughput requirement. In our formalization, instead of mapping applications to storage devices, we use a simplifying trick: We represent each of the applications on a host as a "pseudo host". Thus, if there are k applications on host i, we replace host i by k pseudo hosts. In order to make this simplification step work, we have to conduct an additional trivial pre-check to see if the

host can serve all the applications running on it with the desired throughput capacity. Our optimization then produces an assignment of pseudo hosts to storage devices. Since we know which application is running on which host, we can find out in a next step, which physical links are needed between hosts and storage devices.

In our approach, we define an application as an atomic entity that occupies an indivisible block on the storage device. Hence, we would represent a DBMS that uses different storage spaces for its table data and logging information—having different requirements for each of them—as two different applications in our model.

4.1.2 Constraint Blocks

In the following, we formulate some constraints (in Pseudo-Boolean logic) to ensure that we obtain valid assignments to the variables in the set X. We use $I_H = \{1, \ldots, m\}$ and $I_D = \{1, \ldots, n\}$ as index sets for hosts and storage devices, respectively.

Exactly one connection to a storage device for each pseudo host

Every pseudo host (application) should be served by exactly one storage device at the end of the computation: $\forall i \in I_H : \sum_{k=1}^{n} x_{i,k} = 1$.
Example: three hosts, three storage devices:

$$x_{1,1} + x_{2,1} + x_{3,1} \geq 1$$
$$x_{2,1} + x_{2,2} + x_{2,3} \geq 1$$
$$x_{3,1} + x_{3,2} + x_{3,3} \geq 1$$

Only one storage device per pseudo host (i.e. per application)

Every single application should be served by only one storage device. This can be formulated formally as follows: $\forall i \in I_H \forall j, k \in I_D$ with $j \neq k : x_{i,j} + x_{i,k} \leq 1$.

This constraint block contains $m \cdot \frac{n(n-1)}{2}$ inequalities.

Capacity constraints

It has to be ensured that the storage capacities of the storage devices are not exceeded: $\forall j \in I_D : \sum_{k=1}^{m} c'_k x_{k,j} \leq c_j$, i.e., the sum of the storage space requirements of all hosts that are assigned to a storage device d_j does not exceed the storage capacity c_j of the device.

We obtain from this a constraint block with exactly n additional inequalities, where n is the number of storage devices.

Throughput constraints

This constraint block ensures that the port speeds provided by the storage devices are not exceeded: $\forall j \in I_D : \sum_{k=1}^{m} t'_k x_{k,j} \leq t_j$.

The throughput constraints extend the inequality system by n additional inequalities.

Altogether, $m + m \cdot \frac{n(n-1)}{2} + n + n = m(1 + \frac{n(n-1)}{2}) + 2n$ inequalities, i.e. $\mathcal{O}(n^2 m)$, are to be formulated.

4.1.3 Optimization Problem

So far, we can find valid assignments between hosts and storage devices. The next step is the definition of a goal function for the optimization. We want the workloads for storage devices to be balanced as much as possible, so that the free resources on devices are maximized proportional to their capacities. This kind of optimization has advantages like the increase of flexibility in the choice of resources, since fewer devices would be working with their full capacity. Another advantage is that the probability of violating service level agreements would also be decreased. Since we maximize the unused part of the devices' resources, it is less likely that the SLAs are violated, even if some of the applications behave unexpectedly.

Scaling of inequalities

In order to achieve an equal balance of the throughputs, we first have to scale all throughput constraints such that their right hand sides become equal. By doing this, we can generate constraints that limit the throughput for each storage device to a constant factor below what is maximally possible. Scaling also serves to obtain integer coefficients, as in our case the constraint solver can only handle such constraints.

Using scaling factors s_j^* $(1 \leq j \leq n)$ for our throughput constraints, we obtain: $\forall j \in I_D : s_j^* \cdot \sum_{k=1}^{m} t'_k x_{k,j} \leq s_j^* \cdot t_j$. We define the scaling factors by $s_j^* = 1/t_j \cdot \prod_{i=1}^{n} t_i$ to achieve equal right hand sides $t_{\text{norm}}^* = \prod_{i=1}^{n} t_i$ of the throughput inequalities. We might need an additional scaling factor s_I (now the same for all inequalities) to convert the coefficients to integers, but we will not consider this in our further discussion, and assume that after scaling with the factors s_j^* all coefficients are integers.

Obviously, one would like to make the number t^*_{norm} as small as possible in order to reduce the time the solver needs to process the problem. To achieve this, we divide all inequalities by the greatest common divisor q of all occurring coefficients.

We compute the GCD of the set $Q = \{s^*_j \cdot t'_k \mid 1 \leq k \leq m, 1 \leq j \leq n\} \cup \{t^*_{\text{norm}}\}$. Having computed the divisor q, we divide all throughput constraints by this number. We call the resulting scaling factors s_j, i.e. $s_j = s^*_j/q$. Similarly, we call the resulting common right hand side of the inequalities t_{norm}, i.e. $t_{\text{norm}} = t^*_{\text{norm}}/q$.

Auxiliary variables

To transform the satisfiability problem we have obtained so far into an optimization problem, we introduce a set of auxiliary variables $L = \{l_0, \ldots, l_p\}$. The auxiliary variables are the binary representation of a number $l = l_0 + 2 \cdot l_1 + \cdots + 2^p \cdot l_p$. These auxiliary variables are added to the scaled throughput capacity inequalities: $\forall j \in I_D : s_j \cdot \sum_{k=1}^{m} t'_k x_{k,j} + \sum_{r=0}^{p} 2^r l_r \leq t_{\text{norm}}$.

The auxiliary variables in an inequality symbolize the unused resources on the device. The idea now is that we maximize the value of l. To ensure the proportionality, we must first bring the values on the right hand side of the inequalities to the same value and then add the auxiliary variables.

Since $l_i \in \{0, 1\}$ for all auxiliary variables $l_i \in L$, we still have a Pseudo-Boolean problem. The size of the set L, i.e. how large the index p has to be chosen, depends on the right hand side of the inequalities: we have to make sure, that l can cover the whole range from 0 up to t_{norm}. From the fact that the numbers l_i are the binary representation of l, we obtain that $p = \lceil \log_2(t_{\text{norm}} + 1) \rceil$.

Objective function

The objective function of the optimization problem maximizes the sum of the auxiliary variables, formally: max $\sum_{r=0}^{p} 2^r l_r$.

It should be remarked that the values on the right hand sides of the constraints still get potentially very big if large prime numbers are met among the coefficients of the values on the right hand sides. The number of the necessary auxiliary variables depends logarithmically on the size of the normalized right hand side. Thus, a big value on the right hand side of the constraint system increases the solving time very quickly. To achieve better performance at the cost of having a suboptimal solution instead of the optimal one, some big prime numbers in the coefficients of the value on the right hand side can be slightly

increased or decreased to solve them in smaller prime numbers. Another way would be to round the right hand sides considering the necessary precision of the parameters.

4.1.4 Test of Preconditions

Before generating the constraint system, we check some trivial preconditions, whose unsatisfiability implies the unsatifiability of the whole constraint system due to an invalid configuration of hosts, applications or storage devices. Checking these preconditions, invalid inputs can be avoided and with them time expensive building and solving of a constraint system.

Throughput requirements

If the sum of the throughput requirements of the applications that are resident on a host is greater than the throughput rate that is supplied by the host bus adapter of that host, then it is impossible to find a satisfying solution. Formally: Let $T_{A_i} = \{t_1, \ldots, t_s\}$ be the set of the throughput requirements of the applications on host i and $t_{\text{sum},i}$ the sum of the throughput rates of the ports of the host. If we have $\sum_{t_i \in T_{A_i}} t_i > t_{\text{sum},i}$, the configuration is invalid.

Storage space requirements

If the storage space requirement of a single host or a single application is greater than the greatest available capacity on the side of the storage devices, the configuration is unsatisfiable. Formally: Let C_{A_i} be in analogy to 1) the set of storage space requirements of the applications that are resident on host i, and $c_{\max}$ the maximal storage capacity among the storage devices. Then the configuration is unsatisfiable, if we have $\exists c_k \in C_{A_i}, i \in I_H$: $c_k > c_{\max}$.

4.1.5 Example

In the following, we demonstrate a small example with three applications and three storage devices. $A = \{a_1, a_2, a_3\}$ is the application set, where $a_i = (t'_i, c'_i)$ is a pair consisting of the throughput and capacity requirements for the application. The applications are supposed to have the following settings: $t'_1 = 0.1$, $t'_2 = 1.0$, $t'_3 = 0.5$, $c'_1 = 50$, $c'_2 = 100$, $c'_3 = 70$.

The storage devices are represented as a set $D = \{d_1, d_2, d_3\}$ that contains tuples with the provided throughput and storage capacities of the storage devices: $d_i = (t_i, c_i)$. The

storage devices have the following settings: $t_1 = 2.0$, $t_2 = 1.0$, $t_3 = 1.0$, $c_1 = 200$, $c_2 = 120$, $c_3 = 300$.

Now, the constraints are as follows:

Exactly one connection to a storage device for each pseudo host

$x_{1,1} + x_{1,2} + x_{1,3} = 1$, $x_{2,1} + x_{2,2} + x_{2,3} = 1$, $x_{3,1} + x_{3,2} + x_{3,3} = 1$.

Capacity constraints

$50 \cdot x_{1,1} + 100 \cdot x_{2,1} + 70 \cdot x_{3,1} \leq 200, 50 \cdot x_{1,2} + 100 \cdot x_{2,2} + 70 \cdot x_{3,2} \leq 120, 50 \cdot x_{1,3} + 100 \cdot x_{2,3} + 70 \cdot x_{3,3} \leq 300$.

Throughput constraints

$0.1 \cdot x_{1,1} + 1 \cdot x_{2,1} + 0.5 \cdot x_{3,1} \leq 2, 0.1 \cdot x_{1,2} + 1 \cdot x_{2,2} + 0.5 \cdot x_{3,2} \leq 1, 0.1 \cdot x_{1,3} + 1 \cdot x_{2,3} + 0.5 \cdot x_{3,3} \leq 1$.

Optimization function

To build the optimization function, we must first scale the throughput constraints to obtain a common right hand side and integer coefficients (in our example, we have to scale the constraints to a common right hand side of 20). Then, we calculate how many auxiliary variables we need, and add them with their corresponding coefficients to every single throughput constraint. The optimization function is to maximize this sum. To cover a range of 0 to 20, we need five auxiliary variables, and thus $l = l_0 + 2l_1 + 4l_2 + 8l_3 + 16l_4 + 32l_5$. The optimization function is then to maximize l.

After solving the problem by the solver OPBDP [11], an implementation of an implicit enumeration algorithm for solving (non-)linear Pseudo-Boolean optimization problems with integer coefficients, the following variables were fixed to 1: l_1, l_3, $x_{1,2}$, $x_{2,1}$, $x_{3,3}$. The resulting SAN storage assignment is shown below.

4.1.6 Empirical Results

In our initial tests, it took less than one minute to calculate problems of size up to 15 hosts and 15 storage devices with real device attribute data and manually generated QoS requirements information. From size 16 x 16 on, the measured run times were at least 10 minutes. Our test results are shown in Table 4.1.

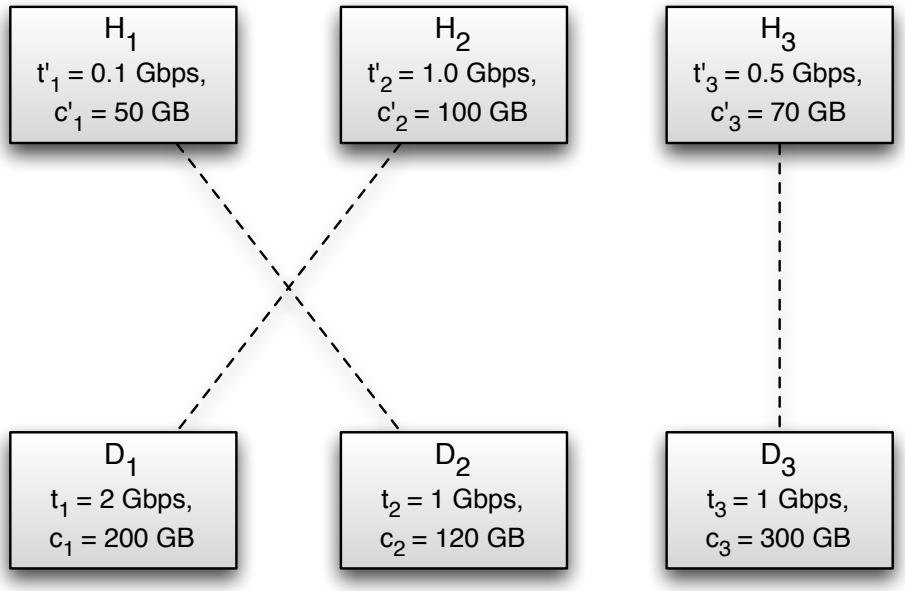

Figure 4.1: *An example configuration with three hosts and three storage devices, and their computed assignments.*

Table 4.1: *Solving times for problems up to size 15 x 15.*

Hosts	Storage Devices	Mean Time (s)
10	10	0.3
11	11	0.3
12	12	0.5
13	13	14.3
14	14	12.7
15	15	51.2

4.2 SAN Connection Problem

The second SAN configuration problem we consider deals with how to lay out connections in a SAN, which link hosts, switches, and storage devices. Certain criteria (SLAs) have to be met in order to obtain a correct configuration. For example, it is typically required that all paths from hosts to storage devices are redundant.

We assume that the assignment from hosts to storage devices is already given (e. g. computed by the algorithm we have given in Section 4.1), i.e. we already know which

paths are needed and their required throughput.

We assume sets of hosts $H = \{h_1, \ldots, h_k\}$, switches $S = \{s_1, \ldots, s_l\}$ and storage devices $D = \{d_1, \ldots, d_m\}$. By $\mathcal{D} = H \cup S \cup D$ we denote the set of all SAN devices. Moreover, we assume a set of paths $P = \{p_1, \ldots, p_n\}$ which are to be routed through the SAN. Each path $p \in P$ connects a host $h(p) \in H$ to a storage device $d(p) \in D$, but the individual hops are not known. Redundancy is modeled by having the same host and storage device connected by more than one path. By $\text{thr}(p)$ we denote the required throughput of a path $p \in P$. Similarly, $\text{thr}(x)$ denotes the maximal throughput for a single port of device $x \in \mathcal{D}$ (we assume the throughput to be the same for all ports of a device). By $\text{ports}(x)$ we denote the number of available ports on device x. To characterize a path, we use predicates $\text{conn}(x, y, p)$, denoting that device x is (directly) connected to device y on path p, and functions $\text{mult}(x, y, p)$ to denote the required multiplicity of the link between x and y on path p, if it is realized[2]. Note that $\text{mult}(x, y, p)$ can be computed in advance by $\text{mult}(x, y, p) = \lceil \text{thr}(p) / \min\{\text{thr}(x), \text{thr}(y)\} \rceil$. We will also use auxiliary predicates $x \in p$ to denote that device x occurs on path p. Note that $\text{conn}(x, y, p)$ implies $x \in p \wedge y \in p$.

Now we can specify the constraints for a correctly configured SAN network:

1. Each $p \in P$ must be a valid path in the network, connecting device $h(p)$ with device $d(p)$, i.e. there must be a sequence $s_1, \ldots, s_t$ of switches, such that the predicates $\text{conn}(h(p), s_1, p)$, $\text{conn}(s_t, d(p), p)$, and $\text{conn}(s_i, s_{i+1}, p)$ for all $1 \leq i < t$ hold.

2. Redundant paths must not use the same switches. I.e., if paths p_1 and p_2 are redundant—which we will denote by $\text{red}(p_1, p_2)$—then $s \in p_1$ implies $s \notin p_2$ for all $s \in S$.

3. For each path p, the throughput requirement must be satisfied, i.e. $\text{mult}(x, y, p) \cdot \min\{\text{thr}(x), \text{thr}(y)\} \geq \text{thr}(p)$ must hold for all x, y, for which $\text{conn}(x, y, p)$ is true. This constraint always holds if we use the definition for $\text{mult}(x, y, p)$ as given above.

4. The number of ports of each device must be sufficient: $\sum_{p \in P, y \in \text{out}(x), z \in \text{in}(x)} \text{mult}(x, y, p) + \text{mult}(z, x, p) \leq \text{ports}(x)$ for all $x \in \mathcal{D}$, where $\text{out}(x) = \{u \in \mathcal{D} \mid \text{conn}(x, u, p)\}$ and $\text{in}(x) = \{u \in \mathcal{D} \mid \text{conn}(u, x, p)\}$.

As our optimization goal we have chosen to minimize the fraction of ports that are used on each device. This goal ascertains that the load on all devices is equally balanced. It can

[2]We use the predicates $\text{conn}(x, y, p)$ always in such a way that x is closer to the "host side" and y is closer to the "storage device side".

be expressed as $\min \max_{x \in \mathcal{D}} \left\{ \frac{\text{ports\_used}(x)}{\text{ports}(x)} \right\}$, where ports_used$(x)$ is the expression on the left hand side of the inequality in constraint 4. To convert this expression to a Pseudo-Boolean optimization goal, we use the same scaling trick that we already used in Section 4.1.3, i.e. we scale the inequalities in constraint 4 such that they possess a common right hand side, and then introduce a slack variable $l = \sum_{i=0}^{p} 2^i l_i$ on the left hand side of each inequality. Afterwards we maximize the slack variable. We want to conclude this section with two remarks: First, the maximal path length t (in number of switches) is typically quite low, e. g. 3, which facilitates the encoding of paths. And second, SAN devices often put a limit on the maximal trunk width (i.e. the number of "parallel" links between two devices). This may be specified by an additional restriction similar to constraint 4. All these constraints can be converted to Pseudo-Boolean logic and handled by a standard Pseudo-Boolean solver.

4.2.1 Encoding as a Pseudo-Boolean Problem

We will now show in detail how the constraints given in the last section can be encoded as a Pseudo-Boolean optimization problem. This is done in order to use an off-the-shelf solver to check the constraints and compute an optimal solution.

First, we encode paths: Let $t_{\max}$ be the maximal number of switches that shall occur on a path. We encode each path p as a sequence $(u_p, v_{p,1}, \ldots, v_{p,t_{\max}}, w_p)$ of binary numbers, where $v_{p,i}$ is the number of the i-th switch on path p ($1 \leq v_{p,i} \leq l$), $u_p + 1$ is the index of the host ($0 \leq u_p < k$) and $w_p + 1$ is the number of the storage device ($0 \leq w_p < m$) of path p. A value $v_{p,i}$ of zero indicates the end of a path. For each path we thus need $t_{\max} \cdot \text{ld}(l+1) + \text{ld}(k) + \text{ld}(m)$ bits to encode it.[3] Note that the values of u_p and w_p are fixed by the SAN specification, whereas the values of the $v_{p,i}$ are undetermined. We derive predicates red(p_1, p_2) and ports_suff as follows:

$$\text{red}(p_1,p_2) \quad \Leftrightarrow \bigwedge_{1 \leq i \leq t_{\max}} \Big(v_{p_1,i} \neq 0 \Rightarrow \bigwedge_{1 \leq j \leq t_{\max}} (v_{p_2,j} \neq 0 \Rightarrow v_{p_1,i} \neq v_{p_2,j}) \Big)$$

$$\text{ports\_suff} \quad \Leftrightarrow \bigwedge_{1 \leq i \leq k} \text{ports\_suff}_{\text{H}}(h_i) \quad \wedge \bigwedge_{1 \leq i \leq l} \text{ports\_suff}_{\text{S}}(s_i) \quad \wedge \bigwedge_{1 \leq i \leq m} \text{ports\_suff}_{\text{D}}(d_i)$$

[3] By ld(x) we denote the *logarithmus dualis* of x.

$$\text{ports\_suff}_{\text{H}}(h) \Leftrightarrow \sum_{\substack{p\in P, s\in S \\ h(p)=h}} \text{mult}(h,s,p)\cdot(v_{p,1}=s) \le \text{ports}(h)$$

$$\begin{aligned}\text{ports\_suff}_{\text{S}}(s) \Leftrightarrow \sum_{p\in P}\Big(&\text{mult}(h(p),s,p)\cdot(v_{p,1}=s)\\ &+\sum_{s'\in S, s'\neq s}\big(\text{mult}(s,s',p)\cdot\textstyle\bigvee_{1\le i<t_{\max}}(v_{p,i}=s\ \wedge\ v_{p,i+1}=s')\\ &\quad+\text{mult}(s',s,p)\cdot\textstyle\bigvee_{1<i\le t_{\max}}(v_{p,i}=s\ \wedge\ v_{p,i-1}=s')\big)\\ &\quad+\text{mult}(s,d(p),p)\cdot L(s,p)\Big)\ \le \text{ports}(s)\end{aligned}$$

$$\text{ports\_suff}_{\text{D}}(d) \Leftrightarrow \sum_{\substack{p\in P, s\in S \\ d(p)=d}} \text{mult}(s,d,p)\cdot L(s,p) \le \text{ports}(d)$$

where $L(s,p)$ is true, if s is the last switch on path p:

$$L(s,p) \Leftrightarrow \bigvee_{1\le t\le t_{\max}}\Big((v_{p,t}=s)\ \wedge \bigwedge_{t<t'\le t_{\max}}(v_{p,t'}=0)\Big)$$

Moreover, we have to restrict the values $v_{p,i}$ to admissible values in the range $[0,l]$, as long as $l+1$ is not a power of 2. This can be achieved easily by further Boolean logic constraints.

The equations $v_{p,i}=x$ are translated as follows:

$$v_{p,i}=x \quad\Leftrightarrow\quad \bigwedge_j\Big(\text{bit}_j(v_{p,i})\Leftrightarrow\text{bit}_j(x)\Big)$$

(with obvious simplifications if x is a constant), where $\text{bit}_j(x)$ denotes the value of bit j in the binary representation of x.

As a further constraint, we might add that on each path at least one switch is used. This excludes degenerate cases, where hosts are directly connected to storage devices, which is typically not considered as a good SAN design. To achieve this, we simply add the constraint $v_{p,1}\neq 0$ for each path p.

The whole set of Pseudo-Boolean constraints for the SAN connection problem then consists of the set of redundancy constraints $\text{red}(p_1,p_2)$ for all paths that should be redundant, plus the constraints generated by ports_suff, plus the additional definitions for the $v_{p,i}$, $L(s,p)$, and C_k.

Our encoding as given does not allow sharing of links, so far. However, by changing the

definition of mult(x, y, p), such that it would also allow "fractions of ports" to be allocated, sharing of links could be formalized.

Another optimization goal, slightly different from the one given above, would be to balance the throughput of the used ports of each switch as evenly as possible. By this, smaller changes in the throughput requirements would not lead to the necessity of reconfiguring the SAN. Such an optimization goal could also be realized by allowing fractions of ports to be virtually allocated.

4.2.2 Example

Consider the example shown in Fig. 4.2. Here we have three hosts, two switches and two storage devices. The number of ports of each device is shown in the figure, as well as the throughput of each port. We assume that three paths have to be laid out: The first path should connect H_1 with D_1 with a throughput of 4 Gbps. This path is also required to be redundant. This is why we obtain two paths, p_1 and p_2, for this connection. The second path (p_3) has to connect H_2 with D_2 with a maximal throughput of 2 Gbps, and the third and last path (p_4) has to link H_3 to D_2, also with 2 Gbps.

As there are at most two switches on each path, we have $t_{\max} = 2$. Thus, we get the following constraints for the redundancy of paths p_1 and p_2 (already in clausal form, and with the simplification $v_{p,1} \neq 0$ applied):

$$\begin{aligned} v_{p_1,1} &\neq v_{p_2,1} \\ v_{p_2,2} = 0 \vee v_{p_1,1} &\neq v_{p_2,2} \\ v_{p_1,2} = 0 \vee v_{p_1,2} &\neq v_{p_2,1} \\ v_{p_1,2} = 0 \vee v_{p_2,2} = 0 \vee v_{p_1,2} &\neq v_{p_2,2} \end{aligned}$$

The constraints expressing that there are enough ports on each device (ports_suff) are as follows. First, for the hosts $H_1, \ldots, H_3$:

$$\begin{aligned} 2 \cdot e_{p_1,1,S_1} + 2 \cdot e_{p_1,1,S_2} + 2 \cdot e_{p_2,1,S_1} + 2 \cdot e_{p_2,1,S_2} &\leq 4 \\ 1 \cdot e_{p_3,1,S_1} + 1 \cdot e_{p_3,1,S_2} &\leq 2 \\ 2 \cdot e_{p_4,1,S_1} + 2 \cdot e_{p_4,1,S_2} &\leq 2 \end{aligned}$$

Here we have introduced auxiliary variables $e_{p,t,s}$ for the predicates $v_{p,t} = s$. Due to space

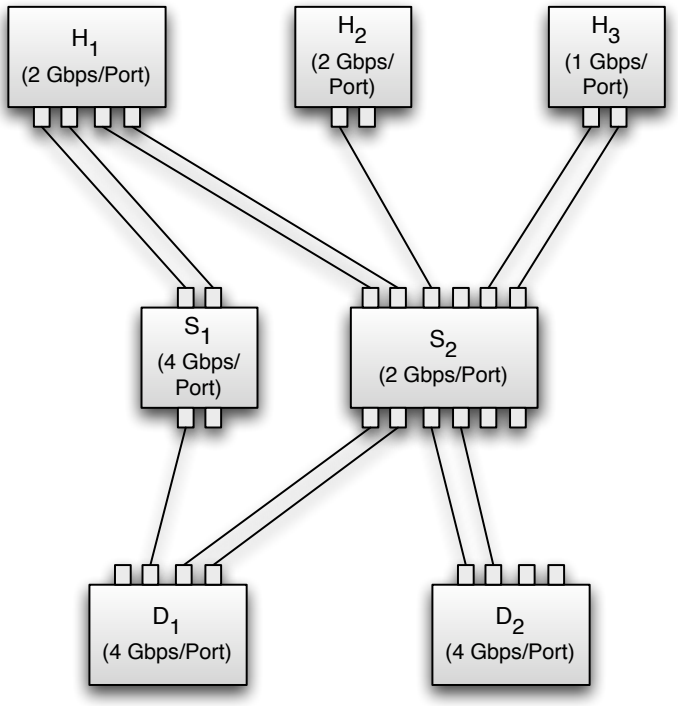

Figure 4.2: *Example configuration of a small SAN with three paths (paths p_1 and p_2 are separated because of a redundancy SLA). A solution computed by our algorithm is also given.*

limitations, we only give the port-constraint for switch S_1, but not for S_2:

$$\begin{aligned} 2 \cdot e_{p_1,1,S_1} + 2 \cdot f_{p_1,S_1,S_2} + 2 \cdot b_{p_1,S_1,S_2} + 1 \cdot L(S_1,p_1) \\ + 2 \cdot e_{p_2,1,S_1} + 2 \cdot f_{p_2,S_1,S_2} + 2 \cdot b_{p_2,S_1,S_2} + 1 \cdot L(S_1,p_2) \\ + 1 \cdot e_{p_3,1,S_1} + 1 \cdot f_{p_3,S_1,S_2} + 1 \cdot b_{p_3,S_1,S_2} + 1 \cdot L(S_1,p_3) \\ + 2 \cdot e_{p_4,1,S_1} + 1 \cdot f_{p_4,S_1,S_2} + 1 \cdot b_{p_4,S_1,S_2} + 1 \cdot L(S_1,p_4) \leq 4 \end{aligned}$$

Here, $f_{p,s,s'}$ and $b_{p,s',s}$ are auxiliary variables defined by

$$\begin{aligned} f_{p,s,s'} &\Leftrightarrow \bigvee_{1 \leq i < t_{\max}} (e_{p,i,s} \wedge e_{p,i+1,s'}) \\ b_{p,s',s} &\Leftrightarrow \bigvee_{1 < i \leq t_{\max}} (e_{p,i,s} \wedge e_{p,i-1,s'}) \end{aligned}$$

Inequality constraints encoding that there are sufficient ports on the storage devices are similar to those for the hosts. We will skip them here, too, due to space limitations.

Finally, we have to add constraints for the equivalences $L(s,p)$, $e_{p,t,s}$, $f_{p,s,s'}$, $b_{p,s',s}$ and C_k, as well as the constraints $v_{p,1} > 0$ for all paths ($e_{p,t,s}$ is defined by the equivalences

of Eq. 4.2.1). Transforming these equivalences into Pseudo-Boolean constraints is accomplished in two steps: First, the equivalences are converted to CNF (conjunctive normal form). Then, each clause $C = (x_1 \vee \cdots \vee x_i \vee \neg y_1 \vee \cdots \vee \neg y_j)$ of the CNF—positive and negative literals are separated—is converted into an inequality constraint:

$$x_1 + \cdots + x_i + (1 - y_1) + \cdots + (1 - y_j) > 0 \ .$$

A solution to the whole constraint system (without the optimization constraints, yet) is shown in Fig. 4.2.

4.3 Implementation in a SAN Management Framework

Since SANs are managed mostly by an integrated and central management software, it is very likely that such a solution is to be implemented as part of a storage management software. An example of such a software framework is Aperi. Aperi is an open source project at the Eclipse Foundation. The project aims to provide a framework for SAN management based on open standards. In an earlier work, we integrated a solution (SANchk) for SAN configuration checking according to "best practices rules" into Aperi [26] [29]. An optimization tool with the function described above can be integrated into Aperi as plugins in analogy to that.

Aperi is based on Equinox and the Rich Client Platform. Equinox is the reference implementation of the "R4 core framework specification" from the standard Open Services Gateway initiative (OSGi). Using Equinox, plugins can be directly integrated into Eclipse and thus also into Aperi to extend the framework.

The major components of Aperi are two servers (data server and device server), an RCP-GUI, a database, and agents that run on hosts and collect data. Any extension to Aperi should extend some of these components. Since Aperi uses the OSGi framework, it provides for each component its own Extension Points.

Aperi has a request-response architecture, which uses Service Provider, Request Handler, Request and Response objects. An extension to the framework should also use these objects to accomplish the communication between the new plugin or plugins and the components of Aperi.

In Figure 4.3, we demonstrate, how the solution could be integrated into a storage

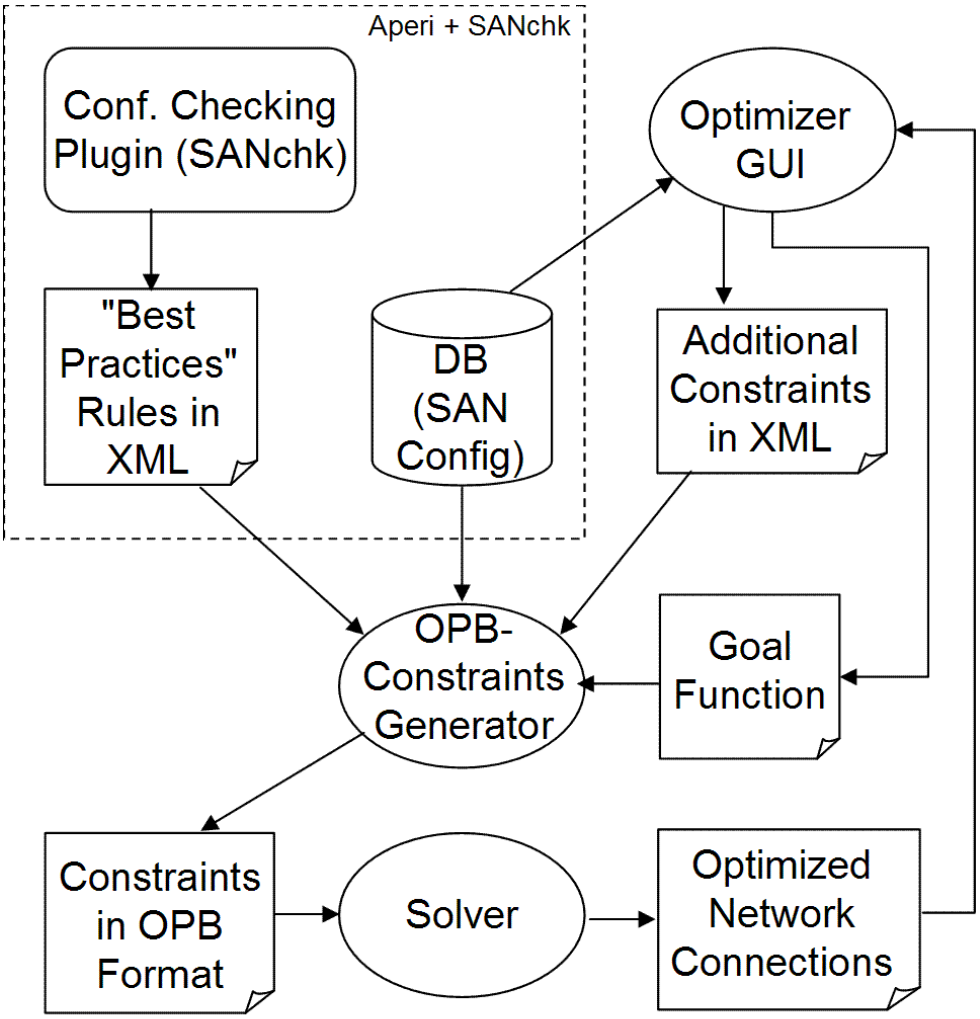

Figure 4.3: *Integration of the optimizer into Aperi.*

management framework on the example of Aperi with our configuration checking plugin SANchk. Except for some small improvements, Aperi and SANchk do not need to be modified. In order to enable SLA-based network design optimization, the Aperi database should be extended by tables for applications that are running on the hosts and their SLA requirements. On the SANchk side, attributes to the XML elements for rules and their rule parameters should be added, which indicate how a rule should be transformed into a constraint in OPB format.

The *Optimizer GUI* should help users add some custom constraints, modify settings and see results after computation. The GUI should show the current network design at

the start, so that users can exclude one part of the devices from the optimization, or can append a fictitious device that is planned to be procured. These actions on the GUI can than be transformed into constraints in OPB format and added to the constraint system. The goal function for the constraint system should also be specified on the GUI. Users should be able to choose among several different optimization goals.

The *OPB Constraints Generator* obtains input from several sources. It uses rules from the configuration checking plugin that are relevant for network design like redundancy rules, additional constraints that are generated by the GUI as mentioned above, the chosen goal function, and the information about the network components (existing hosts, storage devices, switches etc. with their capacities) to generate single OPB constraints. After the constraints in OPB format are generated, the problem should be solved by an off-the-shelf solver. The solver should have an API in Java or C in order to be able to work with Aperi. After the computation, the results should be reported to users on the GUI.

The functionality described above can be packed into two plugins, one for the GUI and the other for the application logic. The plugin for application logic should contain the OPB Constraints Generator and the code that handles the solver. It should also contain a Request Handler in order to accept Request objects from the GUI plugin. After processing the request, it should send back a Response object. The GUI plugin should contain beside the Optimizer GUI components also an extra Eclipse view and other auxiliary classes that are needed for the integration of the GUI into the Aperi GUI.

4.4 Related Work

Optimization problems related to SAN design have been treated previously in several publications ([56], [19], [18], [52], [54], [53]). They differ from our approach mainly in the definition of the optimization problem and its goal function, in the input parameters or methods used to solve the problem.

4.4.1 Appia: Automatic Storage Area Network Fabric Design

Ward et al. [56] present two different heuristic algorithms, namely *FlowMerge* and *Quick-Builder*, to design a SAN automatically.

The SAN design problem that Appia solves is described as follows: A set of hosts, a set of storage devices, and a set of requirements in the form of data flows between host device pairs are given. The goal is to build a SAN with the minimum-cost while meeting

the bandwidth requirements of the data flows. Moreover, the resulting fabric design is subjected to some constraint: Degree constraints i.e. link connections at a device should not exceed the number of ports available there, and bandwidth limitations of links and fabric nodes. It is considered that hubs and switches transmit data in different ways: while a hub sends a packet to all devices that are connected to it, a switch sends a packet only to its recipient.

The FlowMerge algorithm merges single flows of data that share a switch or hub in a set of flows in a recursive way. Beginning with a fully bipartite, directed graph (every host is connected with every storage device), the number of the edges are decreased incrementally while hub and switch nodes are added to the network. The basic building block of a FlowMerge fabric is a single-layer fabric. The algorithm produces first a Single-Layer FlowMerge and continues with a recursive procedure to create more layers if needed.

The FlowMerge algorithm begins with direct links between hosts and storage devices for each flow. This design usually does not meet the degree constraints, since the number of ports at devices are limited. The algorithm partitions the flow requirements into disjoint subsets. These subsets are called *flowsets*. At the beginning, each flow is in its own flowset. At each iteration, two flowsets are merged, so that a new, coarser partition is obtained. Flowsets to merge are selected to alleviate port violations at devices with most severe violations. FlowMerge continues with merging flowsets until either no two flowsets can be merged, or all port violations have been eliminated and no merger decreases the costs.

The algorithm is demonstrated in Figure 4.4.

The problem has 3 hosts and 3 devices, each with 2 ports, and a single type of switch available with 8 ports. Each of the eight flows in the problem has bandwidth 33 MB/s. Links and ports have bandwidth 100 MB/s. Six incremental steps and their corresponding designs are shown, beginning with the one that assigns each flow to its own link. In each design, hosts and devices with the highest port violation are circled. For example, in the first design, the highest port violation is one: there are two hosts and two devices each with three incident links and only two ports. Each design in the series reduces the port violation on one host or device from the previous design by merging two flowsets together. After four mergers, all port violations are eliminated. The last merge eliminates one fabric node and thereby reduces the cost of the fabric.

The QuickBuilder algorithm groups the ports that communicate with each other into port groups. In that way, the SAN design problem is partitioned into smaller design problems, since a port group can be treated as an independent design problem. In general,

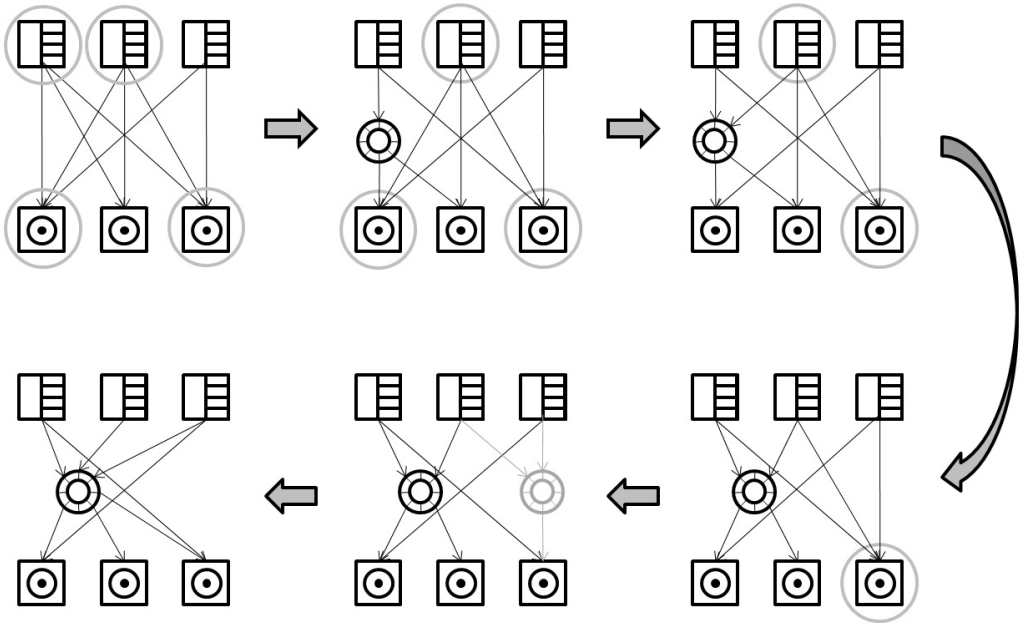

Figure 4.4: *Example application of Single-Layer FlowMerge.*

the fewer ports in a port group, the less fabric is needed to support its flows. The algorithm seeks a design with a finer decomposition of ports to decrease the cost of the fabric.

Appia was in the tests compared to the manually designed SANs able to find cheaper designs in a few minutes.

4.4.2 Using a Genetic Algorithm to Design and Improve SAN Architectures

Dicke et al. [19] use a genetic algorithm to both improve SAN designs that are developed with heuristic techniques and to create new designs. The method proved useful for small networks up to about 10 hosts and 10 storage devices.

SAN design problem is divided into two parts: the determination of the SAN topology, and the determination of the routing for the data flows. Using the algorithm, the topology is inferred indirectly from the determined routes of data flows. An encoded genome specifies the route that a particular flow should take when an arbitrarily large pool of fabric nodes is available for use.

The genome encoding that is used is limited to expression of single-layered networks only. The genome specifies for each flow which fabric node it should be routed through.

It can also alternatively specify that there will be a direct connection between the source and the destination.

The genomes are represented as arrays of integers indicating the number of fabric nodes that a flow should go through. The elements of the array stand for host-device pairs. Zero means a direct connection between a host and a device. Genomes are generated considering the bandwidth requirements of the flows. After the network has been built, the topology and routing of the flows are evaluated using a formula for the overall cost. The evaluation is followed by the creation of a new generation. Two genomes with better solutions are chosen. A single-point crossover operator is applied to each gene with a probability of 0.05. One of the resulting genomes is then chosen at random as the "child" and the mutation operator is applied to each gene with a probability of 0.01. In case that a mutation occurs at a particular locus, a random number is chosen between 0 and n, to represent a new route for a flow. A mutation is likely to decrease the number, so that a gene with the value 1 is likely to result in a direct connection. Successive pairs are chosen as "parents" and produce a "child" until the new population size is one less than the size of the old population. The last member of the new population is an exact copy of the most fit member of the previous population. This ensures that the fittest member is not lost through mutation or crossover.

In the tests, a genetic algorithm which encoded single layered networks could evolve buildable networks for small, 10 host by 10 device problems. Compared with the Appia algorithms, the solution usually cannot outperform the FlowMerge algorithm, but it outperforms the QuickBuilder algorithm about half of the time.

4.4.3 An Ant Inspired Technique for SAN Design

Another biologically inspired approach by Dicke et al. [18] applies the ant colony optimization algorithm to the network design problem. This algorithm is based on the path selection of ant species using pheromone trails. The ants deposit some amount of pheromone as they walk, so that shorter paths to the target get a higher concentration of pheromone. When an ant faces a choice of routes, it will take the path with a higher concentration of pheromone. If several paths have the same amount of pheromone, each of the equivalent paths is chosen with equal probability. Occasionally, an ant will ignore the pheromone concentration and choose the path to go randomly.

The ant colony rules can be also used to solve other problems that can be translated into a shortest path problem. The SAN design problem is transformed into a path optimization problem and adapt the ant colony rules to it. The transformation is accomplished as

follows:

A set F of flows and a set N of fabric nodes are assumed. A direct connection between a host and a device is also accepted as an element of N. The solutions are limited to single-layered networks. A *route* is an assignment of a flow $f \in F$ to a fabric node $n \in N$. A network topology will be inferred at the end from the chosen routes. In order to choose its path, a particular ant will iterate through the set of required flows ordered by decreasing bandwidth requirements, choosing a fabric node (or direct connection) for each of the flows to pass through. After the path is constructed, the available nodes for the next flows are restricted only to those node which will result in a feasible network. If there are no feasible network nodes available, the ant is terminated and ignored. The resulting network is then evaluated in terms of cost.

In the tests, it is shown that an appropriately configured ant colony optimization implementation can outperform Appia algorithms a majority of the time. Using ant colony optimization algorithms, storage area network designs that provide significant cost saving over Appia designed networks can be found. The solution has difficulty finding buildable solutions for large networks.

4.4.4 Chameleon

Uttamchandani et al. [52,54] present Chameleon, an adaptive arbitrator for shared storage resources. Chameleon relies on self-refining models and constrained optimization to adapt the configuration of a storage system to variable workloads on a dynamically changing device landscape.

Chameleon is a framework that provides predictable performance for multiple clients that share a common storage infrastructure. Multiple hosts are connected to storage devices in the back end via interconnection fabrics. Chameleon's main goal is to prevent SLA violations. Each workload j has an SLA_j associated with it, and uses a fixed set of components on its invocation path, such as controllers, logical volumes, switches, and logical units (LUNs). In order to meet the SLAs, Chameleon identifies and throttles workloads. It unthrottles workloads, when it detects unused bandwidth. Chameleon monitors and optionally delays I/O processed by the system. This is implemented in the prototype of Chameleon at each host.

The SLAs that are used in Chameleon are conditional: a workload will be guaranteed a specified upper bound on average I/O latency, as long as its I/O rate does not exceed a specified limit. An SLA violation occurs if the throughput (I/O rate) is below the limit,

and I/O latency exceeds its upper bound. Chameleon periodically evaluates the SLAs of the workloads. The periodic interval for SLA evaluation is set to 60 s. in the prototype of Chameleon. This time should be large enough to get better average values, but also small enough to have a reasonably responsive system.

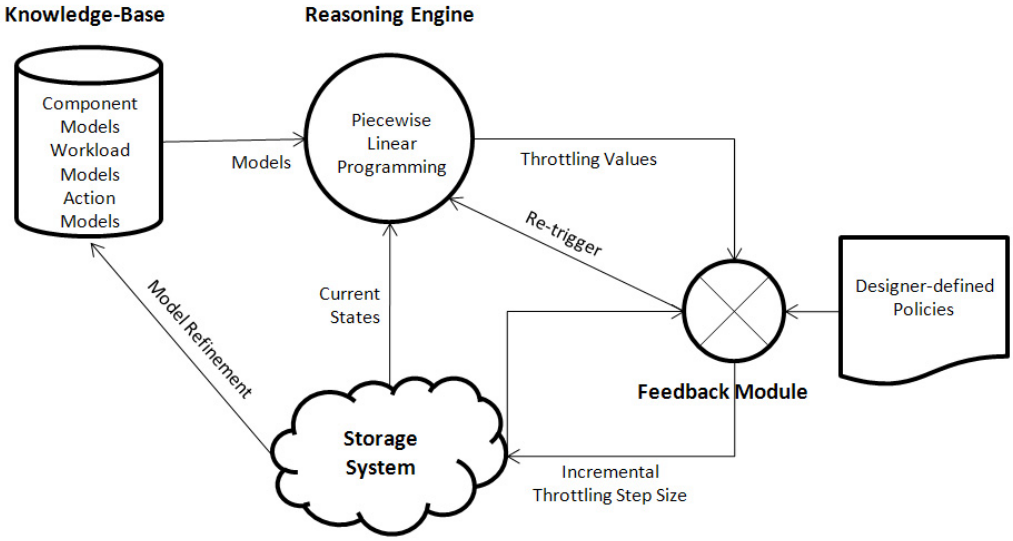

Figure 4.5: *Architecture of Chameleon*

Chameleon consists of four main parts as demonstrated in Figure 4.5:

- Knowledge base: Chameleon automatically builds internal *black-box models* of the system by taking periodic performance samples on the running system. The models are constructed using Support Vector Machines (SVM), a machine learning technique for regression. Performance data needed to build black-box models sources from metrics monitored from client hosts, data tallied by each component, and collected via proprietary interfaces for data collection, or via standard protocols such as SMI-S.

- Reasoning engine: Using the performance data from the knowledge base, the reasoning engine computes the allowed throughput for each workload stream. The reasoning engine is implemented as a constraint solver that analyzes all possible combinations of workload token rates and selects the one that optimizes an objective function defined by an administrator. Examples for the objective functions would be "minimize the number of workloads violating their SLA", or "ensure that highest priority workloads always meet their guarantees".

The reasoning engine is not only invoked upon SLA violations, but also periodically to unthrottle the workloads if the system resources are available.

- Designer-defined policies: A set of fixed heuristics as a fallback mechanism is specified by the system designer for coarse grained throttling control. Heuristics are needed whenever the predictions of the models are not reliable. This can occur during bootstrapping or after significant system changes such as hardware failures. Sample heuristics include "if system utilization is greater than 85%, start throttling workloads in the `lucky` region", or "if the workload-priority variance is less than 10%, uniformly throttle all workloads sharing the component". The heuristics can be expressed in a variety of ways such as Event-Condition-Action (ECA) rules or hard-wired code.

- Informed feedback module: The feedback module incrementally throttles workloads based on the decisions of either the reasoning engine or the system-designer heuristics. In case that the throttling is based on the decisions of the reasoning engine, throttling is applied at incremental steps whose size is proportional to the confidence value of the constraint solver; otherwise, the throttling step is a constant value.

In the tests, traces from production environments were replayed on a real storage system. It was found that Chameleon makes very accurate decisions for the workloads examined. The reaction time was in the 3-14 min. range.

4.4.5 Polus

A framework for storage QoS management is Polus [53], which uses a combination of rule-of-thumb specification mechanism, a reasoning engine, and a learning engine to generate code that maps the QoS goals to the low level system actions.

Polus addresses the problems of complexity and brittleness while designing robust storage management systems. Writing an event-condition-action (ECA) rule is in different ways a complex task. To accomplish this, an administrator should choose a combination of system parameters to observe from a large set of observables, determine appropriate threshold values after considering the interaction of a large set of system variables, and select a specific corrective action from a large set of competing options. Brittleness means the difficulty for vendors to provide prepackaged transformation code with their products because this code becomes brittle with respect to changing system configurations, user workloads, and department/business constraints.

The Polus framework assigns the task of writing policy mapping code to a combination of reasoning and learning engines instead of human administrators and experts. The system administrator should input his knowledge in the form of rules of thumb. For example, "To invoke Prefetch action *requires* memory", "Invoke Prefetch action *requires* the workload to be Sequential", "Prefetch action improves throughput." The system administrator does not have to quantify the threshold values for actions, observables, workload characteristics and resources, or spell-out the details of the action to be invoked.

The relationships defined in the form of rules of thumb are quantified by a learning engine. The system management actions, the state of the system resources and workload characteristics when a particular action was taken, and the current values of the observables are monitored and stored in the knowledge base. The learning engine predicts and quantifies the relationships based on the information in the knowledge base. For example, "Prefetching improves throughput when available memory is greater than 20 percent", and "Use prefetching when Sequential/Random ratio is greater than 0.4."

In case that a particular QoS goal is violated, the Polus reasoning engine is invoked. The reasoning engine uses the first order predicate calculus for logical computation. For example, that an action (x) can be invoked only if the resource required (*precondition*) for its invocation are available in the current-state (cs) is expressed as follows:

$$\forall x, invoke(x, cs) \Rightarrow (available(cs) > precondition(x))$$

The current-state is defined in terms of values of resources, workload characteristics, and observables. Using the current-state and the information from the knowledge base, the reasoning engine derives the actions to be invoked at run-time. For example, when the reasoning engine tries to invoke the prefetch action the *invoke* function is instantiated as $invoke(prefetch, cs)$ which will be true if

$$(available(cs) > precondition(prefetch))$$

The reasoning engine unifies cs with the values that were passed as input, and retrieves the information of the prefetch action from the knowledge base.

In the experiments, Polus was compared to a rule-based ECA system. The iterative convergence of the results of Polus to that of the ECA system was promising.

4.4.6 Other Related Work

A framework for automated storage management based on QoS specifications is Rome [57] by HP Laboratories. Rome allows to represent QoS goals, predictions, and observations and turns these into designs that correspond to the stated goals. Also a flexible attribute inheritance model is provided by Rome, so that attributes can be added and overridden, as needed.

Walker et al. [55] have a mixed-integer approach to the SAN design problem and also use the minimal provisioning cost as the optimization goal. They focus on the Core-Edge reference topology and provide two formulations for the SAN design problem. The size of the formulations is reduced by introducing generic component types and through a preprocessing method that removes suboptimal switches and links from consideration.

Singh et al. [47] have proposed a SAN FS planning tool that uses the notions of application templates and a planning engine to assist a system designer with the design process.

The following two works concentrate on the design of the storage devices according to the application workloads rather than the design of the whole SAN with its connections:

PulsatingStore [43] is an analytical framework that provides an automated storage management service for DBMS that balances the conflicting goals of performance guarantees and on-demand resource usage.

Anderson et al. [7] present the Disk Array Designer (DAD) that uses a generalized best-fit bin packing heuristic to design disk arrays according to both capacity and I/O performance demands for the application data.

Reiss and Kanungo [44] examine the problem of choosing a QoS level for each table or index in a service provider's backend databases to minimize the cost of provisioning storage while satisfying the application level SLAs. They define the problem formally and convert it first into a Boolean quadratic problem and then into a Boolean linear program. After that, they solve the problem with a publicly available solver. Their methodical approach is very similar to the one in our work, but the problem definition and optimization goal differ from ours.

Chapter 5

Conclusion

5.1 Results

With SANchk, we have demonstrated that a policy based configuration checking system for SAN can be realized using well-known technologies like SQL and XML. Our use of SQL for defining the policy conditions implies that the checking systems using data from relational databases can define their conditions directly in SQL. Specific to SAN configuration, we have shown that all the five types of the "Collection Policies" as defined by Agrawal et al. [5] can be defined in SQL by our method.

We have specified an XML schema to define configuration policies. The XML schema provides means for checking policy conditions, for tracking components that cause a condition to fail and for defining actions related to failed components.

We have used the Java Reflection API to realize a flexible action handling component that can directly invoke Java class methods. The ability to call Java methods with parameters was needed to define actions that use configuration information returned from the policy evaluation.

The test results show that configuration problems of even real and large industrial systems can be detected by SANchk in acceptable time. While much larger SAN systems exist, our checking tool can still be optimized and it can be ported to much larger (e. g. parallel) hardware, because it can immediately profit from the highly optimized database engines in existence.

Our concept is in principle applicable to other domains such as host configuration, network configuration, or database configuration, provided that the configuration data are

stored in a relational database.

With SANopt, we defined and formalized the SAN design problem to increase the flexibility of a SAN instead of to minimize the provisioning cost. The flexibility increases on the one hand its ability to meet the QoS requirements for the applications running on its hosts, even if some of the workloads behave themselves spontaneously irregularly, and on the other hand the ability of the network to grow without the need of major structural changes. We formalized the problem in two different parts and solved an example for the assignment problem with an off-the-shelf Pseudo-Boolean constraint solver.

The optimization model described is not exhaustive. It can and should be expanded by further QoS attributes and by more constraints of practical relevance to find solutions that better correspond to the real world problem. The problem model can be enhanced this way. Another issue that can be the subject of future work is enhancing the performance of the algorithms. As we stated before, better performance can be achieved at the cost of having a suboptimal solution instead of the optimal one, by slightly increasing or decreasing the value on the right hand side of the inequality system to solve them in smaller prime numbers. Another way would be to round the right hand sides considering the necessary precision of the parameters. The algorithms should be further tested with large real world examples.

5.2 Publications

- Eray Gençay, Carsten Sinz, Wolfgang Küchlin, and Thorsten Schäfer. SANchk: SQL-based SAN configuration checking. *IEEE Transactions on Network and Service Management*, 5(2):91-104, 2008.

- Eray Gençay, Carsten Sinz, and Wolfgang Küchlin. Towards SLA-based optimal workload distribution in SANs. In *IEEE/IFIP Network Operations and Management Symposium*, NOMS 2008, pages 755-758, 2008.

- Eray Gençay, Carsten Sinz, and Wolfgang Küchlin. SLA-based SAN design. In *Computer Communications Workshops, IEEE INFOCOM 2008*, pages 1-6, 2008.

- Eray Gençay, Wolfgang Küchlin, and Thorsten Schäfer. SANchk: An SQL-based validation system for SAN configuration. In *IEEE/IFIP Symposium on Integrated Network Management, IM 2007*, pages 333-342, 2007.

Appendix A

Best Practices Rules Implemented

A.1 Redundancy

1. Each HBA should be connected to multiple switches.

```
1 <
ALL(SELECT COUNT(DISTINCT b.switch_id)
    FROM tpc.t_res_node2hba A,
      tpc.t_view_zone2member_ent_con_to_switch B,
      tpc.t_view_node2computer C
    WHERE C.host_id = B.entity_id
    AND A.node_id = C.node_id
    GROUP BY A.hba_id)
```

2. Each host should be connected to multiple switches.

```
1 <
ALL(SELECT COUNT(DISTINCT b.switch_id)
    FROM tpc.t_res_host A,
      tpc.t_view_zone2member_ent_con_to_switch B
    WHERE A.computer_id = B.entity_id
    GROUP BY A.computer_id)
```

3. Each storage subsystem should be connected to multiple switches.

```
1 <
ALL(SELECT COUNT(DISTINCT b.switch_id)
    FROM tpc.t_res_storage_subsystem A,
      tpc.t_view_zone2member_ent_con_to_switch B
    WHERE A.subsystem_id = B.entity_id
    GROUP BY A.subsystem_id)
```

4. Each HBA should be connected to multiple fabrics.

```
1 <
ALL (SELECT COUNT(DISTINCT b.fabric_id)
     FROM tpc.t_res_node2hba A, tpc.t_view_zone2member B,
       tpc.t_view_node2computer C
     WHERE C.host_id = B.entity_id AND A.node_id = C.node_id
     GROUP BY A.hba_id)
```

5. Each host should be connected to multiple fabrics.

```
1 <
ALL(SELECT COUNT(DISTINCT b.fabric_id)
    FROM tpc.t_res_host A, tpc.t_view_zone2member B
    WHERE A.computer_id = B.entity_id
    GROUP BY A.computer_id)
```

6. Each storage subsystem should be connected to multiple fabrics.

```
1 <
ALL(SELECT COUNT(DISTINCT b.fabric_id)
    FROM tpc.t_res_storage_subsystem A,
      tpc.t_view_zone2member B
    WHERE A.subsystem_id = B.entity_id
    GROUP BY A.subsystem_id)
```

7. Each disk group should have a redundant RAID level.

```
1 >
ALL(SELECT COUNT(DISTINCT a.subsystem_id)
    FROM tpc.t_res_disk_group A
    WHERE A.raid_level LIKE '%0%'
    AND NOT A.raid_level LIKE '%1%'
    AND NOT A.raid_level LIKE '%2%'
    AND NOT A.raid_level LIKE '%3%'
    AND NOT A.raid_level LIKE '%4%'
    AND NOT A.raid_level LIKE '%5%'
    AND NOT A.raid_level LIKE '%6%'
    AND NOT A.raid_level LIKE '%7%'
    AND NOT A.raid_level LIKE '%8%'
    AND NOT A.raid_level LIKE '%9%')
```

8. There should be at least 2 logical data paths from host to storage. [5]

```
1 <
ALL(SELECT COUNT (DISTINCT b.zone_id)
    FROM tpc.t_res_storage_subsystem a,
      tpc.t_view_zone2member b,
      tpc.t_res_host c,
      tpc.t_view_zone2member d
    WHERE b.entity_id = c.computer_id
    AND d.entity_id = a.subsystem_id
    AND b.zone_id = d.zone_id
    AND b.fabric_id = d.fabric_id
    GROUP BY c.computer_id, a.subsystem_id)
```

9. There should be no more than 4 logical data paths from host to storage. [5]

```
5 >
ALL(SELECT COUNT(DISTINCT b.zone_id)
    FROM tpc.t_res_storage_subsystem a,
      tpc.t_view_zone2member b,
      tpc.t_res_host c,
```

```
      tpc.t_view_zone2member d
    WHERE b.entity_id = c.computer_id
    AND d.entity_id = a.subsystem_id
    AND b.zone_id = d.zone_id
    AND b.fabric_id = d.fabric_id
    GROUP BY c.computer_id, a.subsystem_id)
```

10. There should be no more than 8 physical (useable) paths from host to storage.

```
9 >
ALL(SELECT SUM(connections)
    FROM (SELECT DISTINCT g.computer_id, m.fabric_wwn,
             e.subsystem_id, COUNT(DISTINCT a.port_id) *
             COUNT(DISTINCT e.port_id) connections
          FROM tpc.t_view_host2port a, tpc.t_res_port2port b,
            tpc.t_res_switch2port c, tpc.t_res_fabric2switch d,
            tpc.t_view_subsystem2port e, tpc.t_view_port2zone f,
            tpc.t_res_host g, tpc.t_res_fabric2switch h,
            tpc.t_res_switch2port i, tpc.t_res_port2port j,
            tpc.t_view_zone2member k, tpc.t_view_zone2member l,
            tpc.t_res_zone m
          WHERE a.host_id = g.computer_id
          AND ((a.port_id = b.port_id2
          AND b.port_id1 = c.port_id)
          OR (a.port_id = b.port_id1
          AND b.port_id2 = c.port_id))
          AND c.switch_wwn = d.switch_wwn
          AND d.fabric_wwn = h.fabric_wwn
          AND h.switch_wwn = i.switch_wwn
          AND ((i.port_id = j.port_id1
          AND j.port_id2 = e.port_id)
          OR (i.port_id = j.port_id2
          AND j.port_id1 = e.port_id))
          AND a.host_id = k.entity_id
          AND e.subsystem_id = l.entity_id
```

```
            AND k.zone_id = l.zone_id
            AND m.fabric_wwn = d.fabric_wwn
            AND m.zone_id = l.zone_id
            GROUP BY g.computer_id, e.subsystem_id,m.fabric_wwn)
              AS RES
      GROUP BY computer_id, subsystem_id)
```

A.2 Zoning

1. Every zone should include hosts with only one OS type.

```
2 >
ALL(SELECT ostype_count
    FROM (SELECT COUNT(distinct os_type) ostype_count, zone_id
          FROM tpc.t_view_zone2member_ent_con_to_switch A,
            tpc.t_res_host B
          WHERE A.entity_id = B.computer_id
          AND A.prefix_id = 'server:'
          GROUP BY A.zone_id, B.os_type)
            AS res)
```

2. Every zone should have either disk subsystems or tape libraries. [5]

```
0 =
(SELECT COUNT(*)
 FROM (SELECT DISTINCT prefix_id, zone_id
       FROM tpc.t_view_zone2member_ent_con_to_switch) AS res1,
        (SELECT DISTINCT prefix_id, zone_id
         FROM tpc.t_view_zone2member_ent_con_to_switch) AS res2
 WHERE res1.zone_id = res2.zone_id
 AND res1.prefix_id = 'tapelibrary:'
 AND res2.prefix_id = 'subsystem:')
```

3. All switches in each zone should have the same vendor.

```
1 =
ALL(SELECT COUNT(DISTINCT vendor_id)
    FROM tpc.t_view_zone2member_ent_con_to_switch A,
      tpc.t_res_switch B
    WHERE A.switch_id = B.switch_id
    GROUP BY A.zone_id)
```

4. Every switch should be included by at least k zones.

```
? <
ALL(SELECT COUNT(DISTINCT zone_id)
    FROM tpc.t_view_zones_ent_con_to_switch
    GROUP BY switch_id)
```

5. Every port should be included by at most k zones. [5]

```
? >
ALL(SELECT COUNT(zone_id)
    FROM tpc.t_view_port2zone group by port_id)
```

6. Every port should be included by at least k zones.

```
? <=
ALL(SELECT COUNT(zone_id)
    FROM tpc.t_view_port2zone
    GROUP BY port_id)
```

7. In each zone, at most one tape library should exist.

```
2 >
ALL(SELECT COUNT(entity_id)
    FROM tpc.t_view_zone2member_ent_con_to_switch
    WHERE prefix_id = 'tapelibrary:'
    GROUP BY zone_id)
```

8. There should be no inter-switch links in zones.

```
0 =
(SELECT COUNT(*)
 FROM tpc.t_view_port2zone A, tpc.t_res_port B
 WHERE A.port_id = B.port_id
 AND B.type = 14)
```

9. There should be no more than 5 server HBAs in each zone.

```
? >
ALL(SELECT COUNT(B.hba_id)
    FROM tpc.t_view_zone2member_ent_con_to_switch A,
      tpc.t_view_node2computer B
    WHERE prefix_id = 'server:'
    AND A.entity_id = B.host_id
    GROUP BY zone_id)
```

10. There should be at least 1 server HBA in each zone.

```
? <
ALL(SELECT COUNT(B.hba_id)
    FROM tpc.t_view_zone2member_ent_con_to_switch A,
      tpc.t_view_node2computer B
    WHERE prefix_id = 'server:'
    AND A.entity_id = B.host_id
    GROUP BY zone_id)
```

11. There should be at least 2 members in each zone.

```
1 <
ALL(SELECT member_count
    FROM (SELECT COUNT(zone_member_id) member_count, zone_id
          FROM tpc.t_res_zone2member GROUP BY zone_id) AS res)
```

12. Hosts from vendor *X* should not be in the same zone with storage subsystems from vendor *Y*.

```
1 >
(SELECT COUNT(DISTINCT b.computer_id)
 FROM tpc.t_view_zone2member a, tpc.t_res_host b,
   tpc.t_res_storage_subsystem c, tpc.t_view_zone2member d
 WHERE b.vendor_id = ?
 AND c.vendor_id = ? AND b.computer_id = a.entity_id
 AND c.subsystem_id = d.entity_id AND a.zone_id = d.zone_id)
```

13. Storage subsystems from vendor *X* should not be in the same zone with storage subsystems from vendor *Y*.

```
1 >
(SELECT COUNT(DISTINCT b.subsystem_id)
 FROM tpc.t_view_zone2member a, tpc.t_res_storage_subsystem b,
  tpc.t_res_storage_subsystem c, tpc.t_view_zone2member d
 WHERE b.vendor_id = ? AND c.vendor_id = ?
 AND b.subsystem_id = a.entity_id
 AND c.subsystem_id = d.entity_id
 AND a.zone_id = d.zone_id)
```

14. Switches from vendor *X* should not be in the same zone with storage subsystems from vendor *Y*.

```
1 >
(SELECT COUNT(DISTINCT c.subsystem_id)
 FROM tpc.t_view_zone2member_ent_con_to_switch a,
   tpc.t_res_switch b, tpc.t_res_storage_subsystem c
 WHERE b.vendor_id = ? AND c.vendor_id = ?
 AND b.switch_id = a.switch_id
 AND c.subsystem_id = a.entity_id)
```

15. Switches from vendor *X* should not be in the same zone with hosts from vendor *Y*.

```
1 >
(SELECT COUNT(DISTINCT c.computer_id)
 FROM tpc.t_view_zone2member_ent_con_to_switch a,
   tpc.t_res_switch b, tpc.t_res_host c
 WHERE b.vendor_id = ? AND c.vendor_id = ?
 AND b.switch_id = a.switch_id
 AND c.computer_id = a.entity_id)
```

16. Every fabric should have at least k zones. [5]

```
? <
ALL(SELECT COUNT(DISTINCT zone_id)
    FROM tpc.t_view_zone2member
    GROUP BY fabric_id)
```

17. Every fabric should have at most k zones. [5]

```
? >
ALL(SELECT COUNT(DISTINCT zone_id)
    FROM tpc.t_view_zone2member
    GROUP BY fabric_id)
```

A.3 Uniqueness

1. Every switch should have a unique logical name.

```
(SELECT COUNT(logical_name) FROM TPC.T_RES_SWITCH) =
(SELECT COUNT(DISTINCT logical_name) FROM TPC.T_RES_SWITCH)
```

2. Every switch should have a unique domain ID. [5]

```
(SELECT COUNT(DISTINCT domain) FROM TPC.T_RES_SWITCH) =
(SELECT COUNT(domain) FROM TPC.T_RES_SWITCH)
```

A.4 Fabrics

1. There should be fewer than 5 E-ports in each fabric.

```
? >
ALL(SELECT COUNT(D.port_id)
    FROM tpc.t_res_fabric A, tpc.t_res_port2port B,
      tpc.t_view_port2port2fabric C, tpc.t_res_port D
    WHERE B.port2port_id = C.port2port_id
    AND A.fabric_wwn = C.fabric_wwn
    AND D.port_id IN (B.port_id2, B.port_id2)
    AND D.type = ?
    GROUP BY A.fabric_wwn)
```

2. Each fabric should include fewer than k ports of type Y. [5]

```
? >
ALL(SELECT COUNT(distinct port_id)
    FROM tpc.t_res_port A, tpc.t_view_port2port2fabric B,
      tpc.t_res_port2port C
    WHERE B.port2port_id = C.port2port_id
    AND A.port_id in (C.port_id1, C.port_id2)
    AND A.type = ?
    GROUP BY fabric_wwn)
```

3. Each fabric should have at least one active zone. [5]

```
0 <
ALL(SELECT COUNT (DISTINCT a.zone_id) zones
    FROM tpc.t_view_zone2member a,
      tpc.t_view_zone2member b, tpc.t_res_fabric c,
      tpc.t_res_host d, tpc.t_res_storage_subsystem e
    WHERE c.fabric_id = b.fabric_id
    AND c.fabric_id = a.fabric_id
    AND d.computer_id = a.entity_id
```

```
    AND e.subsystem_id = b.entity_id
    AND a.zone_id = b.zone_id
    GROUP BY a.fabric_id)
```

A.5 HBAs

1. All HBAs in a host should be from the same vendor. [5]

```
0 >
ALL(SELECT count(DISTINCT a.host_id)
    FROM tpc.t_res_hba a, tpc.t_res_hba b
    WHERE a.host_id = b.host_id
    AND NOT a.vendor_id = b.vendor_id
    GROUP BY a.host_id)
```

2. Every HBA with vendor X and model Y should have a firmware version higher than or equal to k. [5]

```
1 =
ALL(SELECT SANCHK.VERSIONAL_GTE(
      firmware_version, CAST(? AS VARCHAR(255)))
    FROM tpc.t_res_hba
    WHERE vendor_id = ? AND model_id = ?)
```

A.6 Firmware level

1. All HBAs with vendor X and model Y should have the same firmware level. (cf. [5])

```
2 >
(SELECT COUNT(DISTINCT firmware_version)
 FROM tpc.t_res_hba
 WHERE vendor_id = ? AND model_id = ?)
```

2. All physical volumes with vendor X and model Y should have the same firmware level. (cf. [5])

```
2 >
(SELECT COUNT(DISTINCT firmware_rev)
 FROM tpc.t_res_physical_volume
 WHERE vendor_id = ? AND model_id = ?)
```

3. All PEs (physical entities) with vendor X and model Y should have the same firmware level. (cf. [5])

```
2 >
(SELECT COUNT(DISTINCT firmware_revision)
 FROM tpc.t_res_phy_pe
 WHERE vendor_id = ?
 AND model_id = ?)
```

4. All switches with vendor X and model Y should have the same firmware level. (cf. [5])

```
2 >
(SELECT COUNT(DISTINCT version)
 FROM tpc.t_res_switch
 WHERE vendor_id = ?
 AND model_id = ?)
```

A.7 Vendor exclusion

1. There should not be an HBA with vendor X in the network. (cf. [5])

```
0 =
(SELECT COUNT(vendor_id)
 FROM tpc.t_res_hba
 WHERE vendor_id = ?)
```

2. There should not be a host with vendor X in the network. (cf. [5])

```
0 =
(SELECT COUNT(vendor_id)
 FROM tpc.t_res_host
 WHERE vendor_id = ?)
```

3. There should not be a PE with vendor X in the network. (cf. [5])

```
0 =
(SELECT COUNT(vendor_id)
 FROM tpc.t_res_phy_pe
 WHERE vendor_id = ?)
```

4. There should not be a physical package with vendor X in the network. (cf. [5])

```
0 =
(SELECT COUNT(vendor_id)
 FROM tpc.t_res_physical_package
 WHERE vendor_id = ?)
```

5. There should not be a physical volume with vendor X in the network. (cf. [5])

```
0 =
(SELECT COUNT(vendor_id)
 FROM tpc.t_res_physical_volume
 WHERE vendor_id = ?)
```

6. There should not be a storage subsystem with vendor X in the network. (cf. [5])

```
0 =
(SELECT COUNT(vendor_id)
 FROM tpc.t_res_storage_subsystem
 WHERE vendor_id = ?)
```

7. There should not be a switch with vendor X in the network. (cf. [5])

```
0 =
(SELECT COUNT(vendor_id)
 FROM tpc.t_res_switch
 WHERE vendor_id = ?)
```

8. There should not be a tape frame with vendor X in the network. (cf. [5])

```
0 =
(SELECT COUNT(vendor_id)
 FROM tpc.t_res_tape_frame
 WHERE vendor_id = ?)
```

9. There should not be a backend controller with vendor X in the network. (cf. [5])

```
0 =
(SELECT COUNT(vendor_id)
 FROM tpc.t_res_backend_controller
 WHERE vendor_id = ?)
```

A.8 Serial number

1. Every HBA with vendor X and model Y should have a serial number in the range between m and n. (cf. [5])

```
1 =
ALL(SELECT SANCHK.SER_NUM_RANGE(CAST(? AS VARCHAR(255)),
      CAST(? AS VARCHAR(255)), serial_number)
    FROM tpc.t_res_hba
    WHERE vendor_id = ? AND model_id = ?)
```

2. Every PE with vendor X and model Y should have a serial number in the range between m and n. (cf. [5])

```
1 =
ALL(SELECT SANCHK.SER_NUM_RANGE(CAST(? AS VARCHAR(255)),
      CAST(? AS VARCHAR(255)), serial_number)
    FROM tpc.t_res_phy_pe
    WHERE vendor_id = ? AND model_id = ?)
```

3. Every physical volume with vendor X and model Y should have a serial number in the range between m and n. (cf. [5])

```
1 =
ALL(SELECT SANCHK.SER_NUM_RANGE(CAST(? AS VARCHAR(255)),
      CAST(? AS VARCHAR(255)), serial_number)
    FROM tpc.t_res_physical_volume
    WHERE vendor_id = ? AND model_id = ?)
```

4. Every storage subsystem with vendor X and model Y should have a serial number in the range between m and n. (cf. [5])

```
1 =
ALL(SELECT SANCHK.SER_NUM_RANGE(CAST(? AS VARCHAR(255)),
      CAST(? AS VARCHAR(255)), serial_number)
    FROM tpc.t_res_storage_subsystem
    WHERE vendor_id = ? AND model_id = ?)
```

5. Every storage volume with subsystem id X and pool id Y should have a serial number in the range between m and n. (cf. [5])

```
1 =
ALL(SELECT SANCHK.SER_NUM_RANGE(CAST(? AS VARCHAR(255)),
      CAST(? AS VARCHAR(255)), serial_number)
    FROM tpc.t_res_storage_volume
    WHERE subsystem_id = ? AND pool_id = ?)
```

6. Every switch with vendor X and model Y should have a serial number in the range between m and n. (cf. [5])

```
1 = ALL(SELECT SANCHK.SER_NUM_RANGE(CAST(? AS VARCHAR(255)),
          CAST(? AS VARCHAR(255)), serial_number)
        FROM tpc.t_res_switch
        WHERE vendor_id = ? AND model_id = ?)
```

7. Every switch blade with vendor X and model Y should have a serial number in the range between m and n. (cf. [5])

```
1 =
ALL(SELECT SANCHK.SER_NUM_RANGE(CAST(? AS VARCHAR(255)),
      CAST(? AS VARCHAR(255)), serial_number)
    FROM tpc.t_res_switch_blade
    WHERE vendor_id = ? AND model_id = ?)
```

8. Every tape frame with vendor X and model Y should have a serial number in the range between m and n. (cf. [5])

```
1 =
ALL(SELECT SANCHK.SER_NUM_RANGE(CAST(? AS VARCHAR(255)),
      CAST(? AS VARCHAR(255)), serial_number)
    FROM tpc.t_res_tape_frame
    WHERE vendor_id = ? AND model_id = ?)
```

A.9 Connection restrictions

1. Physical volumes from vendor X and with model Y should not be connected to a storage subsystem with operation system Z. (cf. [5])

```
1 >
(SELECT COUNT (b.physical_volume_id)
 FROM tpc.t_res_storage_subsystem a, tpc.t_res_physical_volume b
 WHERE b.vendor_id = ? AND b.model_id = ?
 AND a.subsystem_id = b.subsystem_id
 AND a.os_type = ?)
```

2. HBAs from vendor X and with model Y should not be connected to a host with operating system Z. (cf. [5])

```
1 >
(SELECT COUNT (b.hba_id)
 FROM tpc.t_res_host a, tpc.t_res_hba b
 WHERE b.vendor_id = ? AND b.model_id = ?
 AND a.computer_id = b.host_id
 AND a.os_type = ?)
```

3. Backend controllers from vendor *X* should not be referred by a storage subsystem with operating system *Z*. (cf. [5])

```
1 >
(SELECT COUNT (b.backend_controller_id)
 FROM tpc.t_res_storage_subsystem a, tpc.t_res_backend_controller b
 WHERE b.vendor_id = ? AND a.subsystem_id = b.referenced_subsystem_id
 AND a.os_type = ?)
```

4. Switches from vendor *X* and with model *Y* should not be connected to a host with operating system *Z*. (cf. [5])

```
1 >
(SELECT COUNT (switchModelVendorPort.switch_id)
 FROM (SELECT a.host_id, a.port_id, b.os_type
       FROM tpc.t_view_host2port a, tpc.t_res_host b
       WHERE a.host_id = b.computer_id) AS  hostPortOSType,
       tpc.t_res_port2port p2p,
      (SELECT a.switch_id, a.model_id, a.vendor_id, b.port_id
       FROM tpc.t_res_switch a, tpc.t_res_switch2port b
       WHERE a.switch_wwn = b.switch_wwn) AS switchModelVendorPort
 WHERE hostPortOSType.port_id =  p2p.port_id2
 AND p2p.port_id1 =  switchModelVendorPort.port_id
 AND switchModelVendorPort.vendor_id = ?
 AND switchModelVendorPort.model_id = ?
 AND hostPortOSType.os_type = ?)
```

5. Switches from vendor *X* and with model *Y* should not be connected to a storage subsystem with operating system *Z*. (cf. [5])

```
1 >
(SELECT COUNT (switchModelVendorPort.switch_id)
 FROM (SELECT a.subsystem_id, a.port_id, b.os_type
       FROM tpc.t_view_subsystem2port a, tpc.t_res_storage_subsystem b
       WHERE a.subsystem_id = b.subsystem_id) AS  subsystemPortOSType,
```

```
       tpc.t_res_port2port p2p,
      (SELECT a.switch_id, a.model_id, a.vendor_id, b.port_id
       FROM tpc.t_res_switch a, tpc.t_res_switch2port b
       WHERE a.switch_wwn = b.switch_wwn) AS switchModelVendorPort
 WHERE subsystemPortOSType.port_id =  p2p.port_id2
 AND p2p.port_id1 =  switchModelVendorPort.port_id
 AND switchModelVendorPort.vendor_id = ?
 AND switchModelVendorPort.model_id = ?
 AND subsystemPortOSType.os_type = ?)
```

A.10 Ports

1. The minimum capacity for all storage device ports is k Gbits. [5]

```
0 <
(SELECT COUNT(DISTINCT b.port_speed)
 FROM tpc.t_res_storage_subsystem a, tpc.t_res_port b
 WHERE a.subsystem_id = b.subsystem_id
 AND b.port_speed  <= SANCHK.get_speed_in_bits(CAST(? AS INTEGER)))
```

2. The maximum capacity for all storage device ports is k Gbits. [5]

```
0 =
(SELECT COUNT(DISTINCT b.port_speed)
 FROM tpc.t_res_storage_subsystem a, tpc.t_res_port b
 WHERE a.subsystem_id = b.subsystem_id
 AND b.port_speed  >= SANCHK.get_speed_in_bits(CAST(? AS INTEGER)))
```

3. All hosts should be connected to switch ports of mode X.

```
? =
ALL(SELECT b.type
    FROM tpc.t_view_switch2host a, tpc.t_res_port b
    WHERE a.switch_port_id = b.port_id)
```

4. All storage subsystems should be connected to switch ports of mode *X*.

```
? =
ALL(SELECT b.type
    FROM tpc.t_view_switch2subsystem a, tpc.t_res_port b
    WHERE a.switch_port_id = b.port_id)
```

5. E-ports should be connected to E-ports.

```
0 =
(SELECT COUNT(*)
 FROM tpc.t_view_port2port
 WHERE sanchk.get_port_type(port_id1) <> sanchk.get_port_type(port_id2)
 AND 14 IN (sanchk.get_port_type(port_id1),
            sanchk.get_port_type(port_id2)))
```

6. G-ports in a fabric should not be connected to other ports.

```
1 >
ALL(SELECT COUNT(DISTINCT a.fabric_wwn)
    FROM tpc.t_res_fabric2switch a, tpc.t_res_switch2port b,
      tpc.t_view_port2port c, tpc.t_res_port d
    WHERE a.switch_wwn = b.switch_wwn
    AND b.port_id = c.port_id1
    AND b.port_id = d.port_id
    AND d.type = 18)
```

7. NL-ports should be connected to ports of type NL, F, F/NL or FL.

```
0 =
(SELECT COUNT(*)
 FROM tpc.t_view_port2port
 WHERE ((sanchk.get_port_type(port_id1) = 11
 AND NOT(sanchk.get_port_type(port_id2) IN (11,12,15,16))
 OR (sanchk.get_port_type(port_id2) = 11
 AND NOT(sanchk.get_port_type(port_id1) IN (11,12,15,16))))))
```

8. F-ports should be connected to N-ports.

```
0 =
(SELECT COUNT(*)
 FROM tpc.t_view_port2port
 WHERE (sanchk.get_port_type(port_id1) IN (5,10,11,12,13)
 AND NOT sanchk.get_port_type(port_id2) IN (5,8,9,12,15,16,18))
 OR (sanchk.get_port_type(port_id2) IN (5,10,11,12,13)
 AND NOT sanchk.get_port_type(port_id1) IN (5,8,9,15,12,16,18)))
```

A.11 Connections

1. There should be at least k connections from a storage subsystem to a fabric.

```
? >
ALL(SELECT MAX(connections)
    FROM (SELECT count (b.port_id) connections
          FROM tpc.t_res_storage_subsystem a, tpc.t_res_port b,
    tpc.t_res_fabric c, tpc.t_res_fabric2switch d,
  tpc.t_res_switch2port e, tpc.t_res_port2port f
          WHERE a.subsystem_id = b.subsystem_id
          AND b.port_id = f.port_id2
          AND f.port_id1 = e.port_id
          AND e.switch_wwn = d.switch_wwn
          AND d.fabric_wwn = c.fabric_wwn
          GROUP BY a.subsystem_id, c.fabric_wwn)
    AS res)
```

2. A storage subsystem should be seen at most by k hosts.

```
? >
ALL (SELECT MAX (hosts)
     FROM (SELECT count (DISTINCT a.host_id) hosts
           FROM tpc.t_view_host2fabric a, tpc.t_res_storage_subsystem b,
             tpc.t_view_zone2member c, tpc.t_view_zone2member d
```

```
            WHERE a.host_id = c.entity_id
            AND b.subsystem_id = d.entity_id
            AND a.fabric_id = d.fabric_id
            AND c.zone_id = d.zone_id
            GROUP BY b.subsystem_id)
        AS res)
```

A.12 Capacity

1. There should not be a storage volume that has less than k bytes capacity.

```
(SELECT COUNT(*) FROM tpc.t_res_storage_volume WHERE capacity > ?) > 0
```

Bibliography

[1] Dakshi Agrawal, Seraphin Calo, James Giles, Kang-Won Lee, and Dinesh Verma. Policy management for networked systems and applications. In *Proceedings of Ninth IFIP/IEEE International Symposium on Integrated Network Management (IM 2005)*, 2005.

[2] Dakshi Agrawal, Seraphin B. Calo, Kang-Won Lee, and Jorge Lobo. Issues in designing a policy language for distributed management of it infrastructures. In *Integrated Network Management*, pages 30–39, 2007.

[3] Dakshi Agrawal, James Giles, Kang-Won Lee, and Jorge Lobo. Autonomic Computing Expression Language, 2005.

[4] Dakshi Agrawal, James Giles, Kang-Won Lee, and Jorge Lobo. Policy ratification. In *IEEE International Workshop on Policies for Distributed Systems and Networks (Policy 2005)*, pages 223– 232, 2005.

[5] Dakshi Agrawal, James Giles, Kang-Won Lee, Kaladhar Voruganti, and Khalid Filali-Adib. Policy-based validation of SAN configuration. In *IEEE Policy 2004*, pages 77–86. IEEE Computer Society, 2004.

[6] Dakshi Agrawal, Kang-Won Lee, and Jorge Lobo. Policy-based management of networked computing systems. In *Communications Magazine, IEEE, Volume 43, Issue 10*, pages 69–75. IEEE Computer Society, 2005.

[7] Eric Anderson, Susan Spence, Ram Swaminathan, Mahesh Kallahalla, and Qian Wang. Quickly finding near-optimal storage designs. *ACM Trans. Comput. Syst.*, 23(4):337–374, 2005.

[8] The Apache Software Foundation. *Apache HTTP Server Version 1.3 Documentation*, 2002. `http://httpd.apache.org/docs`.

[9] F. Baader, D. McGuinness, P. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[10] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[11] Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Research Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, January 1995.

[12] Endre Boros and Peter L. Hammer. Pseudo-Boolean optimization. *Discrete Applied Mathematics*, 123(1-3):155–225, 2002.

[13] IBM Corporation. Autonomic computing: Creating self-managing computing systems.

[14] John Crandall. Managing Fabrics using CIM - The Evolution, 2002.

[15] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *POLICY*, pages 18–38, 2001.

[16] Nicodemos C. Damianou. *A Policy Framework for Management of Distributed Systems*. PhD thesis, Imperial College of Science, Technology and Medicine University of London, 2002.

[17] Murthy V. Devarakonda, Jack P. Gelb, Avi Saha, and Jimmy P. Strickland. A policy-based storage management framework. In *POLICY*, pages 232–235. IEEE Computer Society, 2002.

[18] Elizabeth Dicke, Andrew Byde, Dave Cliff, and Paul J. Layzell. An ant inspired technique for storage area network design. In *BioADIT*, pages 364–379, 2004.

[19] Elizabeth Dicke, Andrew Byde, Paul J. Layzell, and Dave Cliff. Using a genetic algorithm to design and improve storage area network architectures. In *GECCO (1)*, pages 1066–1077, 2004.

[20] DMTF. Web-Based Enterprise Management (WBEM), 2003.

[21] DMTF Policy Working Group. CIM Policy Model Whitepaper CIM Version 2.7., 2003.

[22] J.S. Dong, J. Sun, and H. Wang. Z Approach to Semantic Web. In *International Conference on Formal Engineering Methods (ICFEM'02)*, pages 156–167. Springer-Verlag, 2002.

[23] S. Dupuy-Chessa and L. du Bousquet. Validation of UML models thanks to Z and Lustre. In *Proc. of the Intl. Symp. on Formal Methods Europe (FME 2001)*, pages 254–258, Berlin, Germany, 2001. Springer-Verlag.

[24] Eclipse Foundation, Inc. *Aperi Storage Management Project*, 2006. `http://www.eclipse.org/aperi/`.

[25] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[26] Eray Gençay, Wolfgang Küchlin, and Thorsten Schäfer. SANchk: An SQL-based validation system for SAN configuration. In *IEEE/IFIP Symposium on Integrated Network Management, IM 2007*, pages 333–342, 2007.

[27] Eray Gençay, Carsten Sinz, and Wolfgang Küchlin. SLA-based SAN design. In *Computer Communications Workshops, IEEE INFOCOM 2008*, pages 1–6, 2008.

[28] Eray Gençay, Carsten Sinz, and Wolfgang Küchlin. Towards SLA-based optimal workload distribution in SANs. In *IEEE/IFIP Network Operations and Management Symposium, NOMS 2008*, pages 755–758, 2008.

[29] Eray Gençay, Carsten Sinz, Wolfgang Küchlin, and Thorsten Schäfer. SANchk: SQL-based SAN configuration checking. *IEEE Transactions on Network and Service Management*, 5(2):91–104, 2008.

[30] Satoshi Hada and Michiharu Kudo. *XML Access Control Language: Provisional Authorization for XML Documents*. Tokyo Research Laboratory, IBM Research, 2000. `http://www.trl.ibm.com/projects/xml/xacl/xacl-spec.html`.

[31] Amir Herzberg, Yosi Mass, Joris Mihaeli, Dalit Naor, and Yiftach Ravid. Access control meets public key infrastructure, or: Assigning roles to strangers. In *IEEE Symposium on Security and Privacy*, pages 2–14, 2000.

[32] IBM Corporation. IBM TotalStorage Productivity Center (TPC), 2005.

[33] Imperial College of Science, Technology and Medicine University of London. *The PONDER Policy Based Management Toolkit*, 2002. `http://www-dse.doc.ic.ac.uk/Research/policies/ponder/PonderSummary.pdf`.

[34] B. Laurie and P. Laurie. *Apache: The Definitive Guide (3rd Edition)*. O'Reilly & Associates, 2002.

[35] Jorge Lobo, Randeep Bhatia, and Shamim A. Naqvi. A policy description language. In *AAAI/IAAI*, pages 291–298, 1999.

[36] Benjamin Marstaller. Integration einer SAN Validierungslösung in Aperi. Master's thesis, Eberhard Karls Universität, Tübingen, Germany, 2007.

[37] E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In *Proc. of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, pages 875–896, Toulouse, France, 1999. Springer-Verlag.

[38] C. Nentwich, W. Emmerich, and A. Finkelstein. Static consistency checking for distributed specifications. In *Proc. of the 16th IEEE Intl. Conf. on Automated Software Engineering (ASE'01)*, pages 115–125, Coronado Bay, CA, 2001. IEEE Computer Society.

[39] OASIS. eXtensible Access Control Markup Language (XACML) Version 1.0, 2003.

[40] Organization for the Advancement of Structured Information Standards. *OASIS*, 1993. `http://www.oasis-open.org/who/`.

[41] Organization for the Advancement of Structured Information Standards. *OASIS Announcement: XACML 2.0 specification submitted for OASIS Standard*, 2004. `http://lists.oasis-open.org/archives/members/200412/msg00013.html`.

[42] Christopher Poelker and Alex Nikitin. *Storage Area Networks*. Wiley Publishing, Inc., 2003.

[43] Lin Qiao, Divyakant Agrawal, Amr El Abbadi, and Balakrishna R. Iyer. PULSATING-STORE: An analytic framework for automated storage management. In *ICDEW '05: Proceedings of the 21st International Conference on Data Engineering Workshops*, page 1213, Washington, DC, USA, 2005. IEEE Computer Society.

[44] Frederick R. Reiss and Tapas Kanungo. Satisfying database service level agreements while minimizing cost through storage QoS. In *SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing*, pages 13–21, Washington, DC, USA, 2005. IEEE Computer Society.

[45] Matthias Schunter, Paul Ashley, Satoshi Hada, Günter Karjoth, and Calvin Powers. Enterprise Privacy Authorization Language (EPAL 1.1), 2003.

[46] Alexander Shrijver. *Theory of Linear and Integer Programming*. Wiley, 1998.

[47] Aameek Singh, Kaladhar Voruganti, Sandeep Gopisetty, Aki Fleshler, Ramani Routray, and Chung hao Tan. SANFS Maestro: Resource planning for enterprise storage area network (SAN) file systems. In *CSREA EEE*, pages 32–38, 2005.

[48] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Formal methods for the validation of automotive product configuration data. *Artif. Intell. Eng. Des. Anal. Manuf.*, 17(1):75–97, 2003.

[49] Carsten Sinz, Amir Khosravizadeh, Wolfgang Küchlin, and Viktor Mihajlovski. Verifying CIM models of apache web-server configurations. In *QSIC*, pages 290–297. IEEE Computer Society, 2003.

[50] SNIA. SNIA Storage Management Initiative Specification Version 1.0.1, 2003.

[51] Ulf Troppens and Rainer Erkens. *Speichernetze*. punkt.verlag GmbH, 2003.

[52] Sandeep Uttamchandani, Guillermo A. Alvarez, and Gul Agha. DecisionQoS: An adaptive, self-evolving QoS arbitration module for storage systems. In *POLICY*, pages 67–76, 2004.

[53] Sandeep Uttamchandani, Kaladhar Voruganti, Sudarshan Srinivasan, John Palmer, and David Pease. Polus: Growing storage QoS management beyond a "4-year old kid". In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 31–44, Berkeley, CA, USA, 2004. USENIX Association.

[54] Sandeep Uttamchandani, Li Yin, Guillermo A. Alvarez, John Palmer, and Gul Agha. CHAMELEON: a self-evolving, fully-adaptive resource arbitrator for storage systems. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, pages 6–6, Berkeley, CA, USA, 2005. USENIX Association.

[55] Cameron Walker, Michael O'Sullivan, and Timothy Thompson. A mixed-integer approach to core-edge design of storage area networks. *Comput. Oper. Res.*, 34(10):2976–3000, 2007.

[56] Julie Ward, Michael O'Sullivan, Troy Shahoumian, and John Wilkes. Appia: Automatic storage area network fabric design. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 15, Berkeley, CA, USA, 2002. USENIX Association.

[57] John Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *IWQoS '01: Proceedings of the 9th International Workshop on Quality of Service*, pages 75–91, London, UK, 2001. Springer-Verlag.

Zusammenfassung

Die Speichernetze (Storage Area Networks - SANs) verbinden Gruppen von Speichergeräten zu den Servern über schnelle Verbindungsgeräte mit Hilfe der Protokolle wie Fibre Channel oder iSCSI, sodass Speicherressourcen den Servern in einer flexiblen und skalierbaren Weise zugeordnet werden können. Eine wichtige Herausforderung ist die Beherrschung der Komplexität der SAN-Konfiguration, die auf die hohe Skalierbarkeit des Netzes und auf die Zusammenschaltung der vielfältigen Geräte zurückzuführen ist. Policy-basierte Validierung wurde früher als eine Lösung für dieses Konfigurationsproblem vorgeschlagen. Mit SANchk wird eine leichtgewichtige SQL-basierte Lösung, in der vorhandene gutbekannte Technologien verwendet werden, vorgeschlagen, um ein solches System zu implementieren. Der Ansatz von SANchk basiert auf einer relationalen Datenbank, die die Konfigurationsdaten, die dem System durch eine WBEM-Standard-Schnittstelle entnommen worden sind, beinhaltet. Im Unterschied zu anderen Ansätzen benutzt SANchk SQL um Policy-Regeln and ausführbare Tests auf diese Konfigurationsdaten zu definieren.

Ein anderes Problem, das von der hohen Komplexität eines SANs verursacht wird, ist die Frage nach einem optimalen SAN-Entwurf. Menschliche SAN-Experten bilden eine SAN-Topologie meistens durch die Verwendung von Daumenregeln. Diese Regeln führen oft zu einem zuverlässigen SAN, aber sie minimieren nicht nötigerweise die totale Kosten des Netzwerks oder bieten eine bessere Topologie um die Service Level Agreements (SLAs) zu treffen. In dieser Dissertation betrachten wir auch das Problem des optimalen SAN-Entwurfs hinsichtlich der SLAs. Erst definieren wir einen Algorithmus für die Zuweisung der Speichergeräte zu den Anwendungen auf den SAN-Hosts. Dieser Algorithmus versucht die Auslastung der Speichergeräte möglichst anzugleichen. Unserer zweite Algorithmus nimmt diese Zuweisungen ein und berechnet die Datenpfade, die nötig sind, um die gewünschte Konfiguration zu erreichen, unter der

Berücksichtigung der Redundanzanforderungen. Auch dieser Algorithmus versucht die Auslastungen aller Verbindungen und Geräte anzugleichen. Folglich, unsere Netzwerkkonfigurationen respektieren alle SLAs und bieten Flexibilität für zukünftige Änderungen durch die Vermeidung der Engpässe an den Speichergeräten oder Switches. Wir erörtern auch die Integration unserer Lösung in die open-source SAN-Management-Software Aperi.

Abstract

Storage Area Networks (SANs) connect groups of storage devices to servers over fast interconnects, so that storage resources can be pooled and assigned to applications in a flexible and scalable way. An important challenge lies in managing the complexity of the resulting massive SAN configurations. Policy-based validation has been proposed earlier as a solution to this configuration problem. We propose a light-weight, SQL-based solution that uses existing well-known technologies to implement such a validation system.

Our approach is based on a relational database which stores configuration data extracted from the system via a WBEM standard interface. In contrast to other approaches, we use SQL to define our policy rules as executable checks on these configuration data.

Another problem that is caused by the high complexity of a SAN is to find an optimal SAN design. Human SAN experts usually build a SAN topology following some rules of thumb. These rules lead often to a reliable SAN, but they do not necessarily minimize the total cost of the network, or provide a better topology to meet the Service Level Agreements (SLAs). In this dissertation, we also consider the problem of designing a SAN in an optimal way, while additionally taking a number of SLAs into account. First, we give an algorithm for assigning storage devices to applications running on the SAN's hosts. This algorithm tries to balance the workload as evenly as possible over all storage devices. Our second algorithm takes these assignments and computes the interconnections (data paths) that are necessary to achieve the desired configuration while respecting redundancy (safety) requirements in the SLAs. Again, this algorithm tries to balance the workload of all connections and devices. Thus, our network configurations respect all SLAs and provide flexibility for future changes by avoiding bottlenecks on storage devices or switches. We also discuss integrating our solution with the open source SAN management software Aperi.