SANchk: SQL-Based SAN Configuration Checking

Eray Gençay, Carsten Sinz, Wolfgang Küchlin, and Thorsten Schäfer

***Abstract*—Storage Area Networks (SANs) connect groups of storage devices to servers over fast interconnects. An important challenge lies in managing the complexity of the resulting massive SAN configurations. Policy-based validation using new logical frameworks has been proposed earlier as a solution to this configuration problem. SANchk offers a new solution that uses standard technologies such as SQL, XML, and Java, to implement a rule-based configuration checker. SANchk works as a light-weight extension to the relational databases of storage management systems; current support includes IBM's TPC and the open source Aperi storage manager. Some five dozen best practices rules for SAN configuration are implemented in SANchk, many of them with configurable parameters. Empirical results with several commercial SANs show that the approach is viable in practice.**

***Index Terms*—storage area networks, network fault diagnosis, knowledge based systems, relational databases, decision support systems, query languages**

I. Introduction

STORAGE Area Networks (SANs) provide centralized pools of storage resources to any number of 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.

Figure 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.

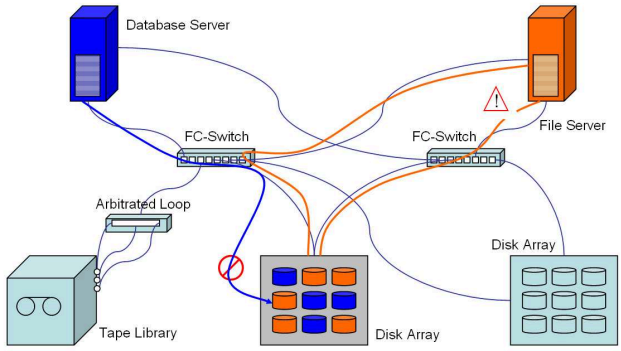

Fig. 1. Some configuration problems in a typical SAN: an invalid access to a LUN and an interrupted connection.

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.

In the remainder of the paper, we first place the SAN configuration checking problem and our SANchk approach into a wider context, providing examples from checking configurations of motor-cars and of the Apache web server software. In Section III, we give a detailed account of the architecture of SANchk. In Section IV, we discuss the power of SQL-based policy formulation by giving some example policies for SAN configuration and their implementations and by showing how all policy types presented by Agrawal et al. [1] can be formulated as SQL queries. In Section V,

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 VI our port to the new open source Aperi Storage Manager. The Aperi plug-in also provides an interactive GUI for editing the rule-sets. Finally, we present our conclusion in Section VIII.

II. 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.

A. General Scenario

The general setting with a rule-based approach is depicted in Figure 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.

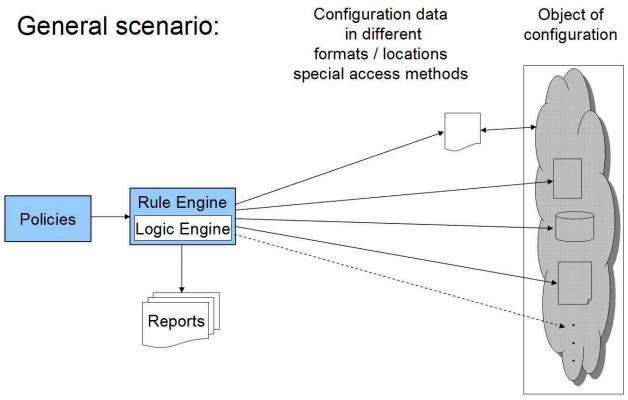

Fig. 2. General setting.

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. [2] 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. Our earlier CIMchk system 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.

B. 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)* [3] 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, our CIMchk system [4] 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 our own *CIM Constraint Language (CCL)*, whose variables may be references into the CIM model. The overall system architecture of CIMchk is depicted in Figure 3.

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 Apache Web-server is also reflected by the large number of books about this Web-server (see, e.g. [5]), 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

1. $\exists^{=1}\text{ServerProperties.ServerRoot}$
2. $[\text{ServerConfiguration}](\text{ServerProperties.MinSpareServer} < \text{ServerProperties.MaxSpareServer}) \wedge$
$[\text{ServerConfiguration}](\text{ServerProperties.MaxSpareServer} > 1)$
3. $|\text{HostProperties.ServerName}| = |\text{HostConfiguration.Name}|$
4. $[\text{HostProperties}]\neg\text{isPrefixOf}(\text{HostProperties.DocumentRoot}, \text{HostProperties.ErrorLog})$
5. $[\text{HostConfiguration}]\text{HostProperties}.\langle\text{HostAddress}, \text{HostPort}\rangle \subseteq \text{ListenSetting}.\langle\text{ListenAddress}, \text{ListenPort}\rangle$
6. $\exists\text{ServerProperties.ConfigName} \wedge \exists\text{ServerProperties.PidFile}$

Fig. 4. Formalization of some consistency properties in $\mathcal{CCL}$.

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 [6], 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.

Our CIM constraint language, $\mathcal{CCL}$ [4], is partly influenced by Description Logic [7] and partly resembles variable-free predicate logic. $\mathcal{CCL}$ consists of three kinds of expressions:

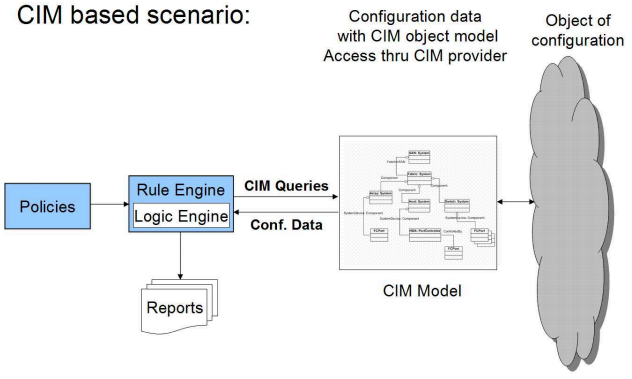

Fig. 3. CIM-based setting.

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 4, we give formal variants of part of the specification stated in natural language above, as well as some examples taken from the Apache documentation [6]. 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 [6]).
6) A configuration name and PID file must be specified for the Web-server.

C. 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 [8]. 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)* [9]. 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)* [10] 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) [11] 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 5 depicts the overall SQL-based scenario; a more detailed representation of the SANchk architecture is given in Figure 6.

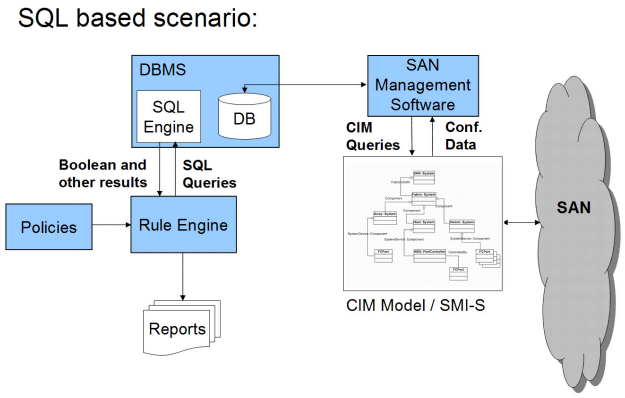

Fig. 5. SQL/DBMS based setting.

D. 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.

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.

III. Architecture of SANchk

A. Overview

Figure 6 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.

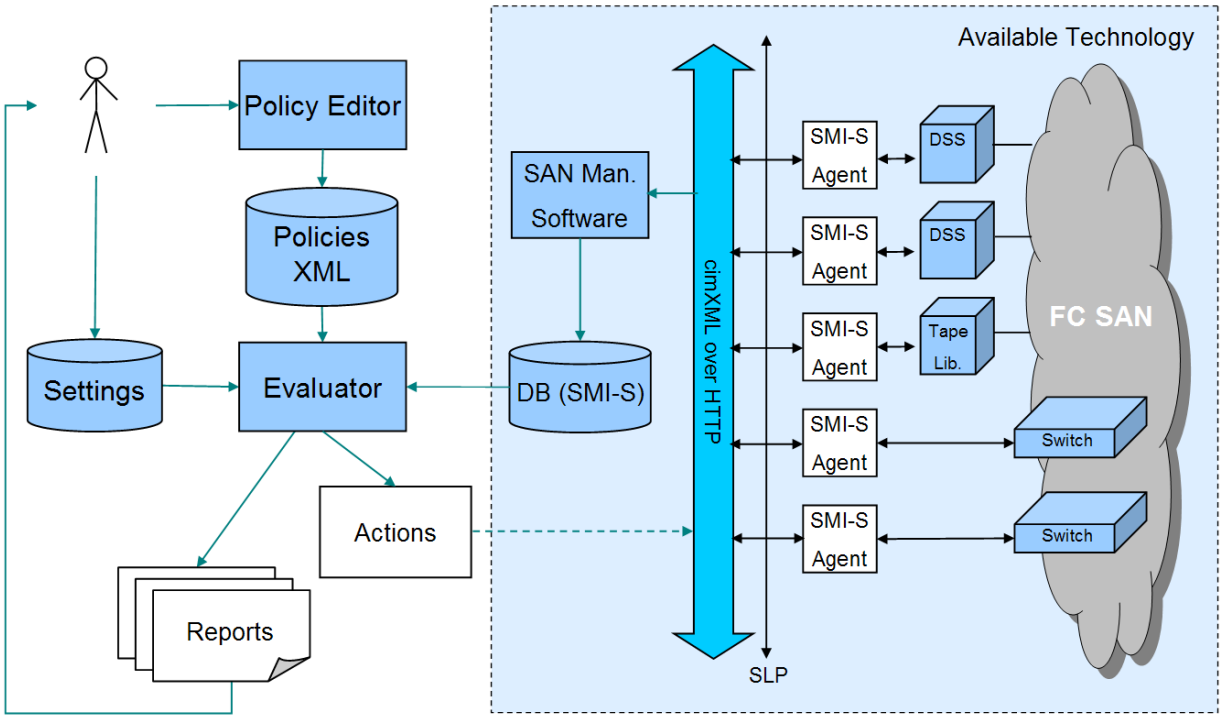

Fig. 6. SANchk system architecture.

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.

The following components make up the checking system:

- Policy Editor:
  We define entire policies in the XML schema that we specify in Section III.D. 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. E.g., local administrators 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 VI).
- 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.

B. 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.

C. Data Related Aspects of the System

The checking system in Figure 6 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.

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 7 [12], the SMI-S CIM class hierachy 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.

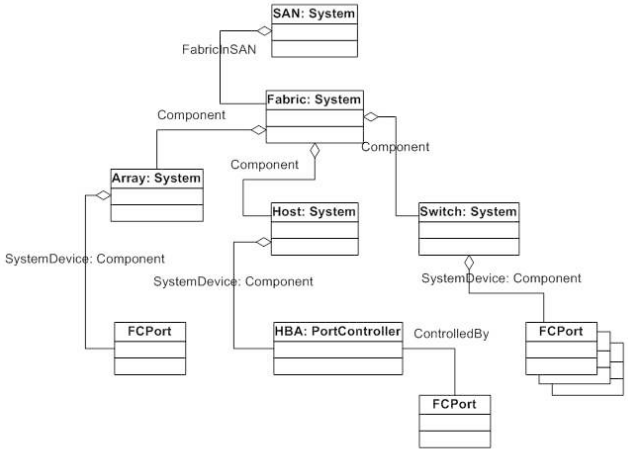

Fig. 7. The SMI-S model.

D. 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 8 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. Altough not implemented yet, *Priority* may be

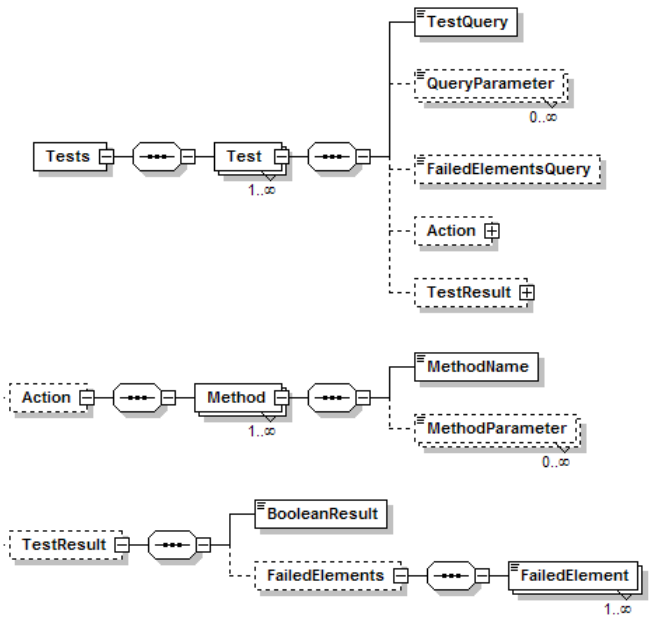

Fig. 8. XML schema diagram for test cases.

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 *StandardSQL* stands for the queries that can be directly sent to the SQL engine, *PreparedStatement* 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*, *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 E.

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*.

E. 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. 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 III.D. 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. [1] 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.

IV. Policy Implementation with SQL

In this section, we show how the "Collection Policies" of [1] 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.

A. Implementation Examples for "Collection Policies"

We represent implementation examples for the "Collection Policies" that were defined in [1] 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 [1] 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 $p_1, ..., p_m$.

1) Cartesian Property [1]: 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>
```

2) Graph [1]: 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>
```

3) Exclusion [1]: Given sets of values $A_{1,1}, \ldots, A_{1,m}$ and sets of values $A_{2,1}, \ldots, A_{2,m}$ for attributes $p_1, \ldots, p_m$, 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 libray 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>
```

4) Many-to-One [1]: 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>
```

5) One-to-One [1]: 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>
```

B. 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>
```

V. Empirical Data

A. 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.

B. Initial Tests

We conducted tests initially with four small but real systems (one of them is a demo system for IBM customers). Table I 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 I. 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.

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

TABLE I
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	-

C. Test on a Production SAN in Industry

Recently, 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.

VI. 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 [13] and present the integration of SANchk into Aperi.

A. 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

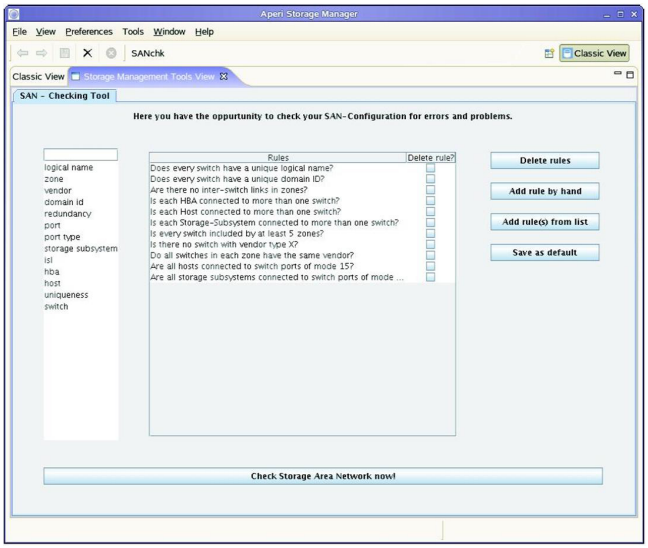

Fig. 9. SANchk GUI in the Aperi Storage Manager.

discovery of a SAN and its components, management of SAN components, retrieval and modification of their configuration, tracking storage capacity and band width, 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.

B. SANchk as Aperi Plugins

There are different ways to let application components communicate in Aperi [14].

- 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 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. Service 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 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.

VII. Related Work

This paper is an extended version of the article [15]. Here, we additionally present the integration of the solution into the open source SAN management software Aperi, give new test results on a large productive SAN environment from the industry, extend the list of the best practices rules, and put the solution into the context of configuration checking of complex systems in general.

Policy based validation of SAN configuration has been proposed previously by Agrawal et al. (2004) [1]. 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.

There is also work on frameworks for policy based storage management. For example, Devarakonda et al. (2002) proposed a policy based storage management framework [16]. 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.

A general framework for policy-based management of networked systems is *Policy Management for Autonomic Computing (PMAC)* by IBM [17]. 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 [18]. PMAC provides modules for policy creation, policy storage, policy evaluation, and policy enforcement as well as a policy ratification module.

Agrawal et al. [19] 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.

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)* [20], *Trust Policy Language (TPL)* [21] and *Enterprise Privacy Authorization Language (EPAL)* [22]. XACML enables defining distributed access control policies that can be shared across different applications. 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 *CIM Constraint Language ($\mathcal{CCL}$)*, *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 [23]. We developed the $\mathcal{CCL}$ previously to define constraints on the Apache Web-Server configuration data formally [4]. In this paper, we reuse this application of $\mathcal{CCL}$ as an example for a CIM-based configuration checking scenario. CIM-SPL allows to define policies in *condition-action* form and renders the policy model of the DMTF fully incorporating the CIM constructs [24].

An object-oriented and general purpose policy language is Ponder [25]. Ponder is designed for specifying management and security policies for distributed systems. Ponder can define authorization policies, obligation policies, refrain policies, delegation policies, and meta-policies. 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. Obligation policies state which actions must be taken by whom on which condition. Refrain policies define which action should be omitted by which subject. Delegation policies assign rights to others. And finally, meta-policies are used to check if conflicts exist between different policies.

VIII. Conclusion

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. [1] 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. The XML schema can also be used in other policy validation domains, in which a relational database serves as data source.

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.

In the future, we plan to extend SANchk to validate and analyze SAN configuration data in relation with *Service Level Agreements (SLAs)* and *Information Lifecycle Management (ILM)*. Our concept is in principle applicable to other domains such as host configuration, provided that the configuration data are stored in a relational database.

APPENDIX A
BEST PRACTICES RULES IMPLEMENTED IN SANCHK

A. Redundancy

1) Each HBA should be connected to multiple switches.
2) Each host should be connected to multiple switches.
3) Each storage subsystem should be connected to multiple switches.
4) Each HBA should be connected to multiple fabrics.
5) Each host should be connected to multiple fabrics.
6) Each storage subsystem should be connected to multiple fabrics.
7) Each disk group should have a redundant RAID level.
8) There should be at least 2 logical data paths from host to storage. [1]
9) There should be no more than 4 logical data paths from host to storage. [1]
10) There should be no more than 8 physical (useable) paths from host to storage.

B. Zoning

1) Every zone should include hosts with only one OS type.
2) Every zone should have either disk subsystems or tape libraries. [1]
3) All switches in each zone should have the same vendor.
4) Every switch should be included by at least k zones.
5) Every port should be included by at most k zones. [1]
6) Every port should be included by at least k zones.
7) In each zone, at most one tape library should exist.
8) There should be no inter-switch links in zones.
9) There should be no more than 5 server HBAs in each zone.
10) There should be at least 1 server HBA in each zone.
11) There should be at least 2 members in each zone.
12) Hosts from vendor X should not be in the same zone with storage subsystems from vendor Y.
13) Storage subsystems from vendor X should not be in the same zone with storage subsystems from vendor Y.
14) Switches from vendor X should not be in the same zone with storage subsystems from vendor Y.
15) Switches from vendor X should not be in the same zone with hosts from vendor Y.
16) Every fabric should have at least k zones. [1]
17) Every fabric should have at most k zones. [1]

C. Uniqueness (One-to-one rules)

1) Every switch should have a unique logical name.
2) Every switch should have a unique domain ID. [1]

D. Fabrics

1) There should be fewer than 5 E-ports in each fabric.
2) Each fabric should include fewer than k ports of type Y. [1]
3) Each fabric should have at least one active zone. [1]

E. HBAs [1]

1) All HBAs in a host should be from the same vendor.
2) Every HBA with vendor X and model Y should have a firmware version higher than or equal to k.

F. Firmware level (cf. [1])

1) All HBAs with vendor X and model Y should have the same firmware level.
2) All physical volumes with vendor X and model Y should have the same firmware level.
3) All PEs (physical entities) with vendor X and model Y should have the same firmware level.
4) All switches with vendor X and model Y should have the same firmware level.

G. Vendor exclusion (cf. [1])

1) There should not be an HBA with vendor X in the network.
2) There should not be a host with vendor X in the network.
3) There should not be a PE with vendor X in the network.
4) There should not be a physical package with vendor X in the network.
5) There should not be a physical volume with vendor X in the network.
6) There should not be a storage subsystem with vendor X in the network.
7) There should not be a switch with vendor X in the network.
8) There should not be a tape frame with vendor X in the network.
9) There should not be a backend controller with vendor X in the network.

H. Serial number (cf. [1])

1) Every HBA with vendor X and model Y should have a serial number in the range between m and n.
2) Every PE with vendor X and model Y should have a serial number in the range between m and n.
3) Every physical volume with vendor X and model Y should have a serial number in the range between m and n.
4) Every storage subsystem with vendor X and model Y should have a serial number in the range between m and n.
5) Every storage volume with subsystem id X and pool id Y should have a serial number in the range between m and n.
6) Every switch with vendor X and model Y should have a serial number in the range between m and n.
7) Every switch blade with vendor X and model Y should have a serial number in the range between m and n.
8) Every tape frame with vendor X and model Y should have a serial number in the range between m and n.

I. Connection restrictions (cf. [1])

1) Physical volumes from vendor X and with model Y should not be connected to a storage subsystem with operation system Z.
2) HBAs from vendor X and with model Y should not be connected to a host with operating system Z.
3) Backend controllers from vendor X should not be referred by a storage subsystem with operating system Z.
4) Switches from vendor X and with model Y should not be connected to a host with operating system Z.
5) Switches from vendor X and with model Y should not be connected to a storage subsystem with operating system Z.

J. Ports

1) The minimum capacity for all storage device ports is k Gbits. [1]
2) The maximum capacity for all storage device ports is k Gbits. [1]
3) All hosts should be connected to switch ports of mode X.
4) All storage subsystems should be connected to switch ports of mode X.
5) E-ports should be connected to E-ports.
6) G-ports in a fabric should not be connected to other ports.
7) NL-ports should be connected to ports of type NL, F, F/NL or FL.
8) F-ports should be connected to N-ports.

K. Connections

1) There should be at least k connections from a storage subsystem to a fabric.
2) A storage subsystem should be seen at most by k hosts.

L. Capacity

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

ACKNOWLEDGMENT

The research work for this paper made possible through funding by IBM Germany Development GmbH through their Center for Advanced Studies (CAS). We thank Benjamin Marstaller for the integration of SANchk into the Aperi Storage Manager. We thank IZB Informatik Zentrum, Germany, for kindly providing access to one of their configuration databases and allowing the publication of its test results. We are also grateful to the storage experts at IBM Deutschland GmbH, Mainz for their technical support in storage related subjects.

REFERENCES

[1] D. Agrawal, J. Giles, K.-W. Lee, K. Voruganti, and K. Filali-Adib, "Policy-based validation of SAN configuration," in *IEEE Policy 2004*. IEEE Computer Society, 2004, pp. 77–86. [Online]. Available: http:// csdl.computer.org/ comp/ proceedings/ policy/ 2004/ 2141/ 00/ 21410077abs.htm.

[2] C. Sinz, A. Kaiser, and W. Küchlin, "Formal methods for the validation of automotive product configuration data," *Artif. Intell. Eng. Des. Anal. Manuf.*, vol. 17, no. 1, pp. 75–97, 2003.

[3] DMTF Policy Working Group, "CIM Policy Model Whitepaper CIM Version 2.7." 2003. [Online]. Available: http:// www.dmtf.org /standards/ documents/ CIM/ DSP0108.pdf.

[4] C. Sinz, A. Khosravizadeh, W. Küchlin, and V. Mihajlovski, "Verifying CIM models of apache web-server configurations," in *QSIC*. IEEE Computer Society, 2003, pp. 290–297. [Online]. Available: http:// csdl.computer.org/ comp/ proceedings/ qsic/ 2003/ 2015/ 00/ 20150290abs.htm.

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

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

[7] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, Eds., *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[8] SNIA, "SNIA Storage Management Initiative Specification Version 1.0.1," 2003. [Online]. Available: http:// www.snia.org/ tech_activities/ smi_spec_pr/ spec/ SMIS_v101.pdf.

[9] DMTF, "Web-Based Enterprise Management (WBEM)," 2003. [Online]. Available: http:// www.dmtf.org /standards /wbem.

[10] IETF, "Service Location Protocol, Version 2 (SLP)," 1999. [Online]. Available: http:// tools.ietf.org/ html/ rfc2608.

[11] IBM Corporation, "IBM TotalStorage Productivity Center (TPC)," 2005. [Online]. Available: http:// www-03.ibm.com/ servers/ storage/ center/.

[12] J. Crandall, "Managing Fabrics using CIM - The Evolution," 2002. [Online]. Available: http:// www.snia.org/ tech_activities/ SMI/ cim/ CIM_Demo_White_Paper.pdf.

[13] The Eclipse Foundation, "Aperi Storage Management Project," 2008. [Online]. Available: http:// www.eclipse.org/ aperi/.

[14] B. Marstaller, "Integration einer SAN Validierungslösung in Aperi," Master's thesis, Eberhard Karls Universität, Tübingen, Germany, 2007. [Online]. Available: http://www-sr.informatik.uni-tuebingen.de/~gencay/marstaller.html

[15] E. Gençay, W. Küchlin, and T. Schäfer, "SANchk: An SQL-based validation system for SAN configuration," in *IM '07. 10th IFIP/IEEE International Symposium on Integrated Network Management*, 2007, pp. 333–342.

[16] M. V. Devarakonda, J. P. Gelb, A. Saha, and J. P. Strickland, "A policy-based storage management framework," in *POLICY*. IEEE Computer Society, 2002, pp. 232–235. [Online]. Available: http:// computer.org/ proceedings/ policy/ 1611/ 16110232abs.htm.

[17] D. Agrawal, K.-W. Lee, and J. Lobo, "Policy-based management of networked computing systems," in *Communications Magazine, IEEE, Volume 43, Issue 10*. IEEE Computer Society, 2005, pp. 69–75.

[18] D. Agrawal, J. Giles, K.-W. Lee, and J. Lobo, "Autonomic Computing Expression Language," 2005. [Online]. Available: http:// www-128.ibm.com/ developerworks/ edu/ ac-dw-ac-acel-i.html.

[19] D. Agrawal, S. Calo, J. Giles, K.-W. Lee, and D. Verma, "Policy management for networked systems and applications," in *Proceedings of Ninth IFIP/IEEE International Symposium on Integrated Network Management (IM 2005)*, 2005.

[20] OASIS, "eXtensible Access Control Markup Language (XACML) Version 1.0," 2003. [Online]. Available: http:// www.oasis-open.org/ committees/ download.php/ 2406/ oasis-xacml-1.0.pdf.

[21] A. Herzberg, Y. Mass, J. Mihaeli, D. Naor, and Y. Ravid, "Access control meets public key infrastructure, or: Assigning roles to strangers," in *IEEE Symposium on Security and Privacy*, 2000, pp. 2–14. [Online]. Available: http:// computer.org/ proceedings/ s%26p/ 0665/ 06650002abs.htm.

[22] M. Schunter, P. Ashley, S. Hada, G. Karjoth, and C. Powers, "Enterprise Privacy Authorization Language (EPAL 1.1)," 2003. [Online]. Available: http:// www.zurich.ibm.com/ security/ enterprise-privacy/ epal/ Specification/ index.html.

[23] J. Lobo, R. Bhatia, and S. A. Naqvi, "A policy description language," in *AAAI/IAAI*, 1999, pp. 291–298.

[24] D. Agrawal, S. B. Calo, K.-W. Lee, and J. Lobo, "Issues in designing a policy language for distributed management of it infrastructures," in *Integrated Network Management*, 2007, pp. 30–39.

[25] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," in *Proceedings of Policy 2001: Workshop on Policies for Distributed Systems and Networks*, Bristol, UK, Jan. 2001, pp. 17–28.

Eray Gençay studied Business Informatics at Marmara University in Istanbul, Turkey, and Reutlingen University of Applied Sciences, where he obtained an MS degree in Business Information Management. He is currently a PhD candidate in the Symbolic Computation Group at the University of Tübingen. His research is supported by a grant from IBM's Center for Advanced Studies in Böblingen, Germany.

Carsten Sinz studied Computer Science at the University of Tübingen, Germany, where he also obtained his PhD. He is recently a postdoctoral researcher in the Symbolic Computation Group at the University of Tübingen. From 2005 – 2006 he was a member of the Institute for Formal Models and Verification at the Johannes-Kepler University of Linz, Austria. His research interests include SAT-Solving, Formal Verification, and Product Configuration.

Wolfgang Küchlin studied Computer Science at the University of Karlsruhe, Germany and obtained his PhD in Computer Science from ETH Zürich. From 1987 – 1992 he was an Assistant Professor at Ohio State University and since 1992 is Professor of Symbolic Computation at the University of Tübingen. From 1999 – 2003 he was Vice Chair of ACM SIGSAM. He has major interests in symbolic methods for configuration checking and in parallel symbolic computation.

Thorsten Schäfer studied Electrical Engineering at the University of Applied Sciences of Bingen, Germany and at the University of Bedfordshire, UK. In 1997 he started with IBM in the area of HDD storage technology and currently leads a team that develops storage management software. Thorsten is especially interested in the SNIA storage management standard SMI-S and the configuration checking of enterprise sized storage networks.

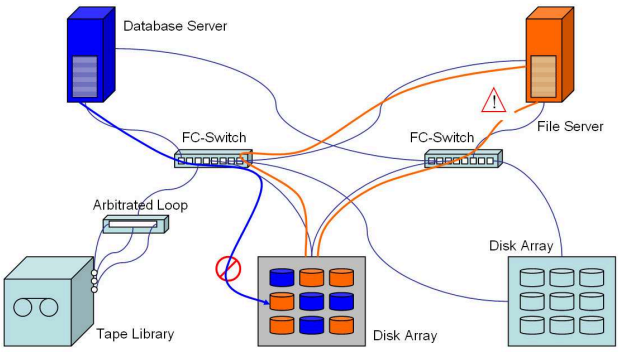

Figure 1: Some configuration problems in a typical SAN: an invalid access to a LUN and an interrupted connection.

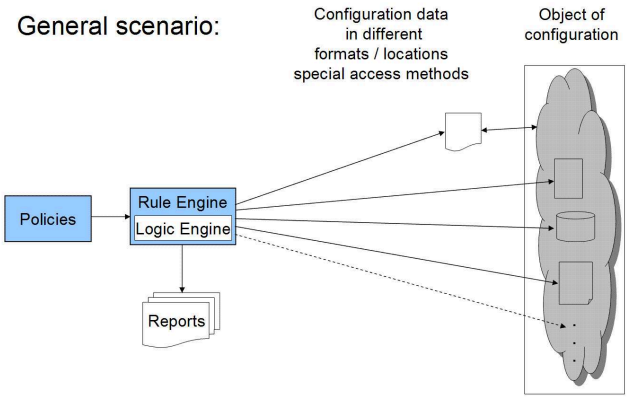

Figure 2: General setting.

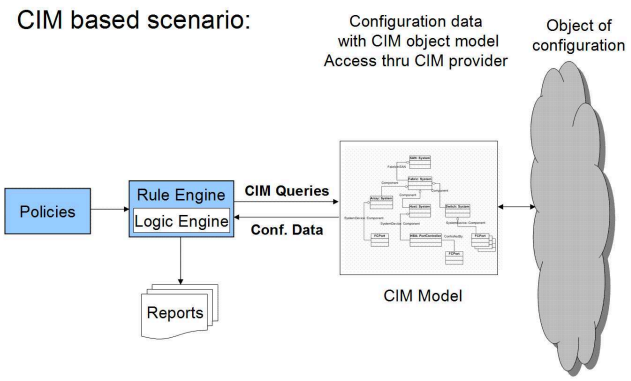

Figure 3: CIM-based setting.

1. $\exists^{=1}$ServerProperties.ServerRoot
2. [ServerConfiguration](ServerProperties.MinSpareServer $<$ ServerProperties.MaxSpareServer)$\wedge$
[ServerConfiguration](ServerProperties.MaxSpareServer > 1)
3. |HostProperties.ServerName| = |HostConfiguration.Name|
4. [HostProperties]$\neg$isPrefixOf(HostProperties.DocumentRoot, HostProperties.ErrorLog)
5. [HostConfiguration]HostProperties.$\langle$HostAddress, HostPort$\rangle$ $\subseteq$ ListenSetting.$\langle$ListenAddress, ListenPort$\rangle$
6. $\exists$ServerProperties.ConfigName $\wedge$ $\exists$ServerProperties.PidFile

Figure 4: Formalization of some consistency properties in $\mathcal{CCL}$.

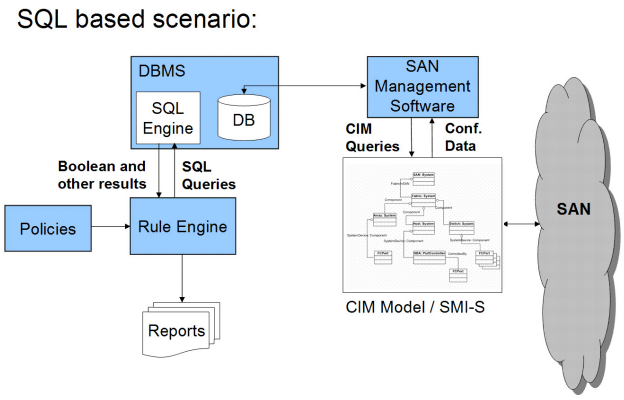

Figure 5: SQL/DBMS based setting.

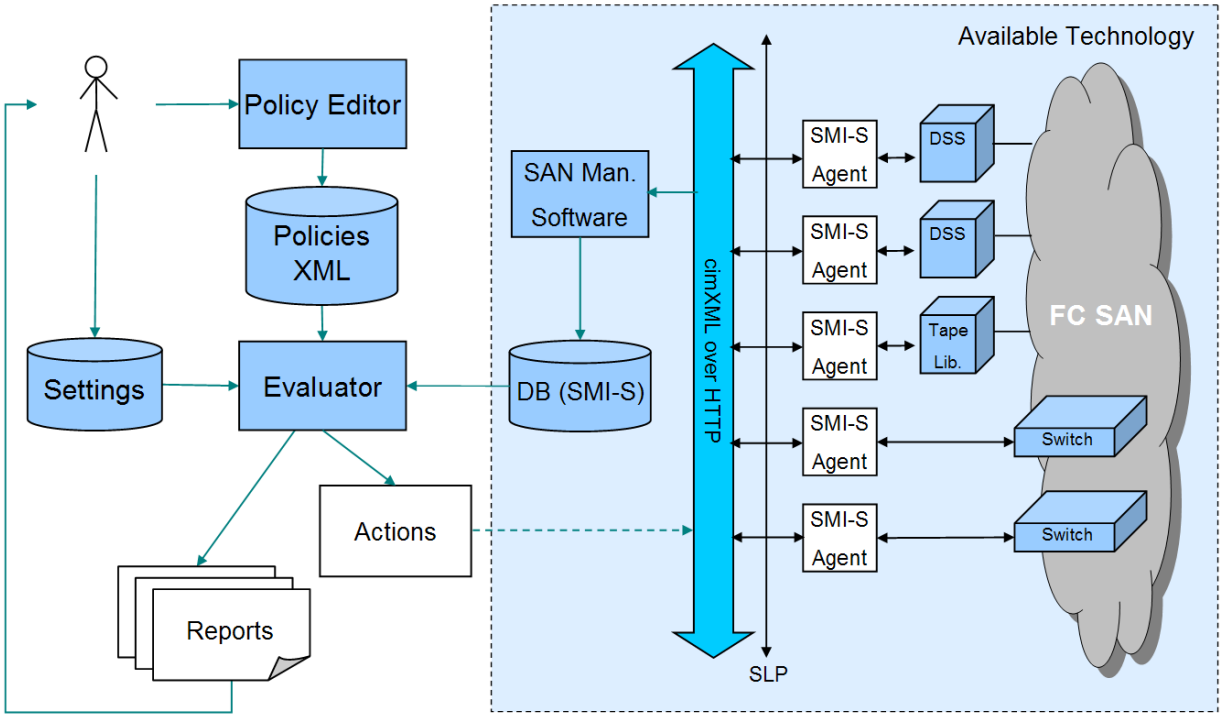

Figure 6: SANchk system architecture.

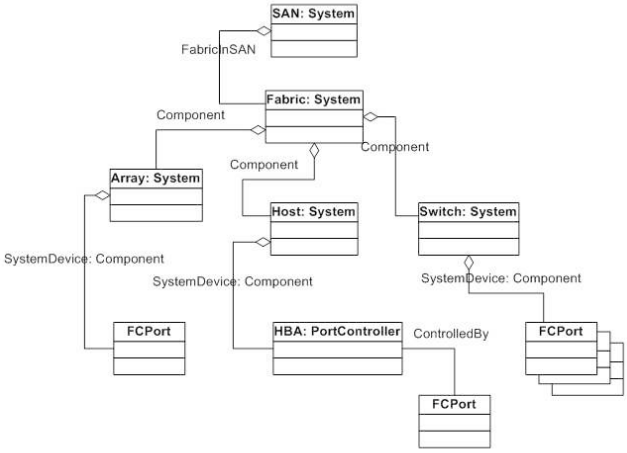

Figure 7: The SMI-S model.

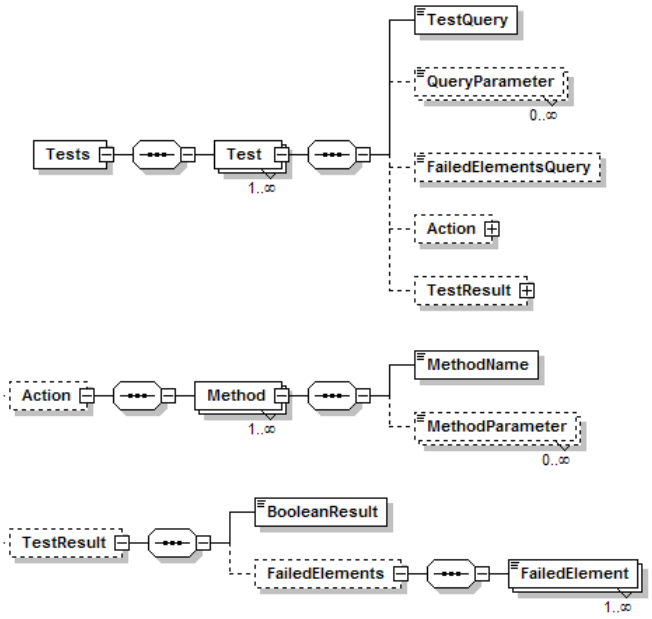

Figure 8: XML schema diagram for test cases.

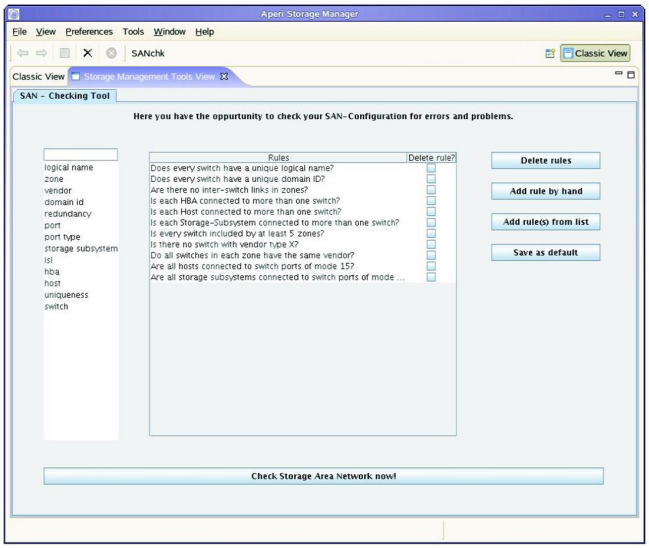

Figure 9: SANchk GUI in the Aperi Storage Manager.