# Verifying CIM Models of Apache Web-Server Configurations

Carsten Sinz Amir Khosravizadeh Wolfgang Küchlin
Symb. Comput. Group, WSI for Computer Science
University of Tübingen, Germany
{sinz,khosravi,kuechlin}@informatik.uni-tuebingen.de

Viktor Mihajlovski
Linux Technology Center (LTC)
IBM Lab. Böblingen, Germany
mihajlov@de.ibm.com

## Abstract

*We show how configuration properties of the Apache Web-server can be formally verified, so that an installation is safe with respect to both universal and site specific local constraints. Our approach starts from an existing semi-formal component model of the Web-server in the Common Information Model (CIM) standard. Hence our approach is applicable also to the verification of other systems for which a CIM model exists.*

## 1. Introduction

For today's complex software systems, development does not end at the manufacturer's side but extends to the client's side, because standard software products must frequently be configured after delivery to meet particular needs or policies. Hence, software verification and validation are no longer confined to the traditional software development process but must extend to the configuration phase at the client site. For the particular example treated in this paper, a formal verification of a Web-server can not be considered complete if it does not extend to particular installations, i.e. configurations, of the product. A flawed Web-server configuration may cause malfunctions just as if the software itself were flawed, and may constitute a serious security breach, even if the product as such were "perfect."

Thus, the problem of verifying and validating particular system configurations is at least as important as that of traditional software verification. It is even more important in the sense that, in general, product (re-)configuration by the customer occurs much more frequently than product completion by the manufacturer, and because traditional software quality assurance methods hardly extend to the configuration phase at the client's site. For the Apache Web-server the Netcraft survey (http://www.netcraft.com/survey) has determined over 13,000,000 installations as of July 2003, all of which have been configured individually to meet each site's demands.

In this paper we attempt a first step towards an ideal scenario where a configurable software product is delivered together with an expert system containing configuration constraints, such that, based on formal methods, the client can

- check a concrete configuration against the manufacturer's set of universal formal configuration constraints;
- set up site specific local configuration constraints that can be checked automatically;
- suggest possible actions of repair to bring a flawed installation back into compliance with the established constraints;
- contain a constraint editor which helps both manufacturer and client in setting up non-contradictory constraints which are actually satisfiable.

The advantages of building on formal methods include that constraints can be concisely and precisely formulated and represented, that there are precise notions of when constraints are actually satisfied, that highly tuned implementations of general purpose reasoning methods such as Boolean satisfiability checking can be employed, and that abstract high-level reasoning methods such as first order resolution are available to reason about the consistency of constraints independent of concrete configurations.

However, it is not easy to bring formal methods to bear on practical application problems, because usually a significant gap must be bridged between the abstract formal method and the concrete application [16]. In particular, a formal model of the application (a *system theory* [17]) must be constructed, the constraints must be formulated as theorems of the system theory, and efficient reasoning methods must exist to formally prove the theorems. For something as complex as the Apache Web-server, building the system theory can be a daunting task in itself because it requires intimate knowledge of both the application and of formal logic. Moreover, the system theory must be kept consistent with a continuously changing application, because otherwise the theorems which are proved hold only of the model and not of the real application.

In our previous work, we could successfully bridge the formalization gap in two applications. One is an industrial information system used for the configuration of motor cars [10]. There, propositional logic formulae are used in rules that control order modification and checking. The other was the verification of an expert system that was part of a larger system management application [15, 16]. This expert system contained situation-action rules which we modeled using PDL (Propositional Dynamic Logic, [8]).

For our present work, we believe that an important part of our success is due to the use of a semi-formal intermediate model of the Apache Web-server formulated in terms of the Common Information Model (CIM) standard [3]. Starting from the CIM model of Apache, it was feasible to build a faithful formal model of constraints and to feed real system values via CIM-based software into the variables of our constraints formulae. From a practical point of view, the existence of CIM and its usefulness outside of formal verification is extremely important, because industry hardly ever builds abstract models for formal verification purposes only. In the case of the CIM standard, it has been developed and is being used to provide abstract system management interfaces to complex systems. A CIM interface presents an object-oriented view of the underlying system and provides abstract interfaces to retrieve and manipulate configuration data. Hence, the system manufacturer maintains a faithful CIM model of the implemented system independent of any formal verification, because it greatly aids in investigating and manipulating the configuration of a system without resorting to implementation details.

It is the core of our approach to hook into the CIM model and start the formal modeling from there. Using CIM software, we can feed real system values into our abstract constraints sets and we can even attempt to repair a configuration under the control of our expert system. Moreover, since our methodology is built on CIM, our work is not particular to the Apache Web-server but can be applied in principle to other systems for which a CIM model exists.

## 2. Common Information Model (CIM)

CIM is an object-oriented data model for system-management purposes that aims at unifying several special-purpose data models (such as DMI, SNMP, CMIP) into a single consistent model, and creating a general framework for construction of truly interoperable management applications. It was designed in the late 90s as an industry-wide standard and is maintained by the *Distributed Management Task Force (DTMF)*, of which all major soft- and hardware manufacturers are members. The goal is to provide a conceptual view of all (physical and logical) components of a system, regardless of manufacturer, architecture, or operating system. As an information model, CIM focuses on standardizing data-semantics and uniform interfaces, and is independent of any encoding and protocol considerations. The Web Based Enterprise Management (WBEM) Initiative is defining additional standards for CIM implementation interoperability (like operational semantics and communication protocols).

Any hardware or software system component is called a managed element and is represented as an instance of a CIM class. Instances contain *properties* (name/value pairs) describing units of data. All properties are accessible through uniform getter/setter-operations. Some of the properties may be declared as *key properties*, with the intended meaning that each CIM object is uniquely identifiable among its other class members by these properties.

A collection of class definitions that describe managed elements (in a particular environment) is called a *Schema*. Schemas have a framework character and are designed to be extensible. CIM Schemas are represented by UML (*Unified Modeling Language*) Diagrams, or MOF files. MOF (*Managed Object Format*) is a declarative language similar to CORBA's IDL (*Interface Definition Language*). All CIM Schemas have to satisfy special restrictions given by the CIM Meta-Schema. The most significant restriction is the consequent use of *association classes* to model relationships between objects. Thereby, composition or any kind of reference within a class is strictly avoided, thus preventing anomalies caused by complex class relations. Aggregations are just special association classes. Most of the association classes of the CIM core are abstract and thus have to be refined.

CIM Schemas belong to one of the levels *Core Model*, *Common Model*, and *Extension*, depending on their level of specificity: the Core Model contains only a small number of very general classes as an abstract description of components and relationships that are found in most environments. These classes are inherited by the more specific classes in the Common Model that includes a series of domain-specific, but platform-independent classes like *System* and *Network*. The most specific classes are gathered in Extension Schemas derived from the Common Model. Sophisticated design patterns like the *Composite Pattern* [7] are used in all layers. A main aspect in designing CIM Schemas is their real world usability, i.e. the completeness with respect to the use case scenarios.

The concrete information of the application is gathered and delivered by a set of *provider* applications to a CIM Object Manager (CIMOM), and can then be accessed and modified by client applications communicating with the CIMOM using standardized XML-mappings [4], utilizing HTTP as transport protocol.

The CIM classes relevant for our work are the *Configuration* and the *Setting* classes together with their associations, as defined by the CIM Core Schema. While there are already some small-scale examples in the literature of device configuration management using CIM, there has not been any real software example yet. The reason for this might be that real software configurations possess a considerably larger number of options and parameters, and therefore give rise to a much more complex CIM modeling effort (see the next section for an example).

The work presented in this paper results from a collaboration of our research group at the University of Tübingen with the IBM Linux Technology Center (LTC) Systems Management Group located at the IBM laboratory at Böblingen, Germany, which has developed the CIM model for (part of) the Apache Web-server configuration. The IBM LTC is maintaining the Open Source Project SBLIM that is providing CIM models and instrumentation for Linux Systems Management [9].

## 3. Apache Web-Server Configuration

### 3.1. Apache Configuration Basics

A device driver (e.g. for a printer) may have a handful of parameters specifying some options (paper format, resolution, etc). Highly developed software is much more flexible. 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. [11]), and the considerable part that these books spend on configuration issues and the related question of security.

The multiplicity of options and the evolutionary character of the application also result in complex interdependencies between the options: Apache allows for running several "*Virtual Hosts*" on the same server, which allows, e.g., running Web-servers for multiple companies on the same machine. Each of them may (or must) have its own *Document Root* (the place where the Web pages and other content is stored), its own *(Virtual) Address* and *Port* (to allow distinct access to each virtual host), and other resources and properties (e.g. log files).

The document root gives a directory in the file system, under which the document tree is stored. Any directory inside a document tree (i.e., any directory potentially visible to the outside world) may contain an *.htaccess* file in which the directory's owner specifies access limitations to that directory and its subdirectories as well as some indexing and representation policies. To this purpose, *.htaccess*-files contain additional directives. These may override the general policies specified in the main configuration file by the system administrator, or other directives in some parent directory. For security reasons and to prevent inconsiderate misconfigurations, an additional set of directives in the server's main configuration can specify which directives are allowed in which *.htaccess*-files. Now, if a directive in an *.htaccess*-file tries to modify file access in a way that is not allowed by the server's main configuration, Apache will not deliver any data from that directory and any of its subdirectories.

Most directives have a one line, parameter/value structure. However, there is also a small number of XML-alike directive pairs, making up scopes or contexts, like `<VirtualHost> </VirtualHost>` or `<Directory> </Directory>`. All directives enclosed between such a pair are said to belong to this context. In the Apache documentation all directives are accompanied by annotations determining in which scopes (or contexts) they may appear.

There are some obvious shortcomings in the original configuration data format as far as automated management and verification purposes are concerned: The relevant data is distributed over several files at different locations, and it is hard for a management or verification system to get an integrated overview of the whole system. A special-purpose verification program working directly on all the configuration files would have to deal with access issues, format- and syntax-checks and much more besides its real task, the semantically motivated checks. In addition, such a system could hardly be used for verifying other configuration data (in another format) or different constraints without major rewritings. In contrast, using a CIM-based configuration model as an abstraction layer between the real configuration and the verifying program leads to a great degree of flexibility and abstraction on the verifier's side. CIM shares these advantages with other high-level data description languages like, e.g., the ontology language OWL [14]. However, CIM gradually evolves into a quasi-standard for system management.

### 3.2. Apache Configuration with CIM

In our CIM model, access to the configuration data of the Apache Web-server is made available through the Apache HTTP Service class. This class is the entry point to all other CIM classes related to Apache server configuration. For increased manageability and structuring, configuration directives are in a first step grouped according to the Web-server's entities they belong to. So there is a coarse distinction between directives pertinent to the whole server, to only a virtual host or merely to a directory. These groups of configuration options are further split up, e.g. into directives valid only for a certain Apache module. That way, the whole unstructured set of configuration options is hierarchically organized into smaller managed CIM elements, as is shown in Figure 1.

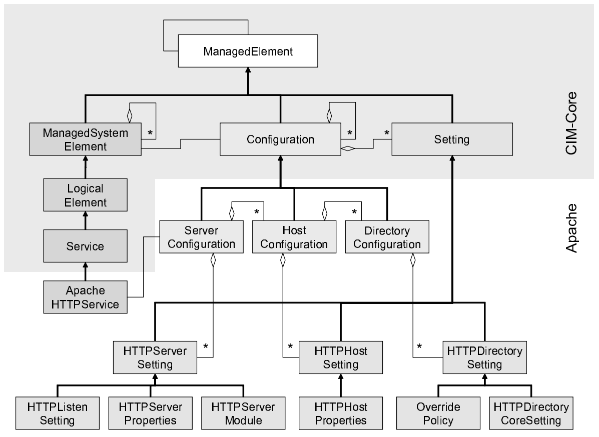

**Figure 1. CIM Classes for Apache Configuration.**

The primary classes to hold configuration information for managed elements of the Apache HTTP Service are descendants of CIM's *Setting* class. So there are, e.g., classes containing properties of a virtual host (*HTTP Host Properties*), or for overriding access policies for directories (*Override Policy*). To group settings into larger managed elements, *Configurations* are employed. Several Apache specific configuration classes (*Server Configuration*, *Host Configuration* and *Directory Configuration*) are derived from CIM's core configuration class. These configuration classes do not directly contain any Apache directives, but are containers for instances of the appropriate setting classes. Configurations themselves may also be part of larger configurations, reflecting the structure of the elements managed by Apache. So we end up with a hierarchical, tree-like structure, where *Directory Configurations* are components of *Host Configurations* that are unified by *Server Configurations*.

Looking at the real configuration devices (mainly files) of an Apache Web-server, we can say that the server configuration approximately maps to the *httpd.conf* file, host configurations to `<VirtualHost>` directives and directory configurations to `<Directory>` directives.

Excerpts from two Apache CIM classes in their MOF representation may illustrate how the directives of a Web-server configuration are structured and distributed over CIM classes:

```
class Apache_HttpHostConfiguration :
      CIM_Configuration
{
      [Key] String Name;
};

class Apache_HttpServerProperties :
      Apache_HttpServerSetting
{
      [Key] String ConfigName;
      String BindAddress;
      String CoreDumpDirectory;
      uint16 MaxClients;
      uint32 MaxRequestsPerChild;
      int16 MaxSpareServers;
      int16 MinSpareServers;
      ...
};
```

Note, that aggregations are not stored within classes, but are modeled using additional association classes. So, in our example, we do not see any properties reflecting associated settings in the HTTP Host Configuration class.

Instantiation of the Apache specific classes of Figure 1 with concrete instances of a server configuration leads to a hierarchical configuration tree. This tree is generated by the aggregation relation of the relevant classes of the CIM model. It consists of two kinds of nodes: setting nodes and configuration nodes. In this tree the inner nodes represent configurations whereas the leaves stand for settings. A schematic example of such a tree is shown in Figure 2.

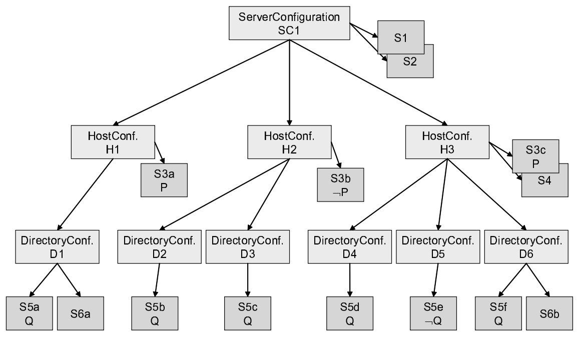

**Figure 2. Example Hierarchy of Configuration and Setting Instances.**

In our example, Web-server SC1 is configured to possess three virtual hosts (H1-H3). These hosts are in turn configured by directives contained in the setting classes S3a, S3b, S3c and S4. For each host further directory configurations are present. So, host H1 has an additional directory configuration consisting of the setting classes S5a and S6a.

The tree structure captures an additional semantics of the Web-server configuration, concerned with the correlation between instances of setting classes and managed elements: A setting S associated with a managed element E is supposed to be relevant for all descendant nodes of element E, too. Thus, in our example, the directives of setting S3a are also valid for directory configuration D1, whereas they are irrelevant for host configurations H2 and H3, or directory configurations D2 through D6. The semantical structure can be reproduced by traversing the part/whole-relations (*ConfigurationComponent* and *SettingContext* association classes in CIM) between these nodes. Each configuration node can be considered generating a causally closed semantical context for constraint evaluation. In specifying and verifying constraints describing interdependencies between different settings, it is crucial to consider the relevant settings only.

Some constraints also require some kind of "horizontal navigation" in this tree, allowing selection of all present instances of a particular class. This can be seen as a form of quantification over configuration instances. Using the WBEM API, quantification over objects can be accomplished using the class name, whereas addressing a special object requires additional knowledge of its key values.

## 4. Apache Configuration Constraints

There is a vast range of possible syntactical and structural errors in a configuration file that can be recognized and remedied in the early stage of transformation of the configuration file into CIM classes. However, there are other, mainly semantical errors that cannot be treated by lexical analysis alone. Some examples may illustrate the kind of conditions that 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 want to make sure that *MaxSpareServers* $>$ *MinSpareServers* and that *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* or a subdirectory thereof.
- 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 [1], others may have to be collected and specified by Web-server administrators or other personnel.

Using the CIM model as a starting point for modeling and checking such constraints offers the advantage of having a semi-formal basis on which the constraint modeling language can be built up. That way, a separation between constraint modeling and low-level configuration processing is achieved.

## 5. Constraint Checking the CIM Model

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 systems seems particularly promising.

The 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.

### 5.1. Syntax

We will now present our CIM constraint language, $\mathcal{CCL}$, which is partly influenced by Description Logic [BMNP03] and partly resembles variable-free predicate logic. The language of $\mathcal{CCL}$ 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.

## 5.2. Semantics

All expressions of our language are interpreted with respect to a set $I$ of instances of CIM classes, where each instance has properties according to its class definition, and each property has a value matching the property's type (always including the value NULL, denoting an undefined value; see also [3], Sec. 4.11.6).

By $C.P$ we (intuitively) want to denote the set of values occurring under property $P$ of any instance of class $C$; $C.\langle P_1, \ldots, P_k\rangle$ selects all tuples $(v_1, \ldots, v_k)$ that occur under properties $P_1, \ldots, P_k$ of an instance of class $C$; $v$ denotes an arbitrary value constant; $f(s_1, \ldots, s_k)$ denotes application of function $f$ to the sets of values $s_1, \ldots, s_k$; the expression $R(s_1, \ldots, s_k)$ is true, if the values $s_1, \ldots, s_k$ are in relation $R$; $\exists^{\geq n} C$ is true, if there are at least $n$ different instances of class $C$; $\exists^{\geq n} C.P$ is true, if there are at least $n$ instances of class $C$, for which property $P$ is defined; and $[C]F$ is true, if formula $F$ is true in evaluation context $C$ (determined by class name $C$, for details see below). We will also use the abbreviations $\exists C$ resp. $\exists C.P$ instead of the a-expressions $\exists^{\geq 1} C$ and $\exists^{\geq 1} C.P$, and make use of the notation $\exists^{=n} C$ as short form of the formula $\exists^{\geq n} C \wedge \neg\exists^{\geq (n+1)} C$ (and similar for $\exists^{=n} C.P$). Among the relations, we always assume the equality predicate $=$ to be present, and use the relation symbol $\neq$ as abbreviation for the negation of the equality predicate. Moreover, we will use the notation $C.P \in \{v_1, \ldots, v_k\}$ as a short equivalent for $C.P = v_1 \vee \cdots \vee C.P = v_k$.

Before giving examples on the use of our language $\mathcal{CCL}$, we will give a precise definition of its semantics. We will define the truth of a- and f-expressions model-theoretically. To refer to a class instance's properties we will use some (meta-language) auxiliary predicates. For an instance $i$, we define:

- $\text{class}(i)$: the CIM class that $i$ is an instance of,
- $\text{props}(i)$: the set of properties defined for instance $i$,
- $\text{prop}(i, p)$: the value assigned to property $p$ of instance $i$.

For the context operator $[C]F$ we will also need to have access to the aggregational (tree-)structure of the whole set of CIM instances. We therefore use the following predicates:

- $\text{parent}(i)$: delivers the set of parent nodes of instance $i$ (i.e. a singleton set, if $i$ is not the root node, and the empty set otherwise),
- $\text{children}(i)$: delivers the direct descendants of instance $i$ in the aggregation tree.

So, for the aggregation tree structure of Figure 2, we get (in extracts, assuming that host configuration H1 is named "Host 1"):

$$\begin{aligned}
\text{class}(\text{H1}) &= \text{HostConfiguration} \\
\text{props}(\text{H1}) &= \{\text{Name}\} \\
\text{prop}(\text{H1}, \text{Name}) &= \text{“Host1''} \\
\text{children}(\text{H1}) &= \{\text{D1}, \text{S3a}\}
\end{aligned}$$

We can now define the precise semantics of expressions in $\mathcal{CCL}$ ($\sigma_v$ for v-expressions, $\sigma_a$ for a-expressions, and $\sigma_f$ for f-expressions):

$$\begin{aligned}
\sigma_v(C.P, I) &= \{v \mid (\exists i \in I)(\text{class}(i) = C \wedge \\
&\qquad\qquad \text{prop}(i, P) = v)\} \\
\sigma_v(C.\langle P_1, ..., P_k\rangle, I) &= \{(v_1, ..., v_k) \mid (\exists i \in I) \\
&\qquad (\text{class}(i) = C \wedge \text{prop}(i, P_1) = v_1 \\
&\qquad\qquad \wedge ... \wedge \text{prop}(i, P_k) = v_k)\} \\
\sigma_v(v, I) &= \{\mathbf{v}\} \\
\sigma_v(f(s_1, ..., s_k), I) &= \mathbf{f}(\sigma_v(s_1, I), ..., \sigma_v(s_k, I))
\end{aligned}$$

$$\begin{aligned}
\sigma_a(R(s_1, ..., s_k), I) &= \mathbf{R}(\sigma_v(s_1, I), ..., \sigma_v(s_k, I)) \\
\sigma_a(\exists^{\geq n} C, I) &= (\exists^{\geq n} i \in I)(\text{class}(i) = C) \\
\sigma_a(\exists^{\geq n} C.P, I) &= (\exists^{\geq n} i \in I)(\text{class}(i) = C \wedge \\
&\qquad \text{prop}(i, P) \neq \text{NONE})
\end{aligned}$$

$$\begin{aligned}
\sigma_f(F, I) &= \sigma_a(F, I) \quad \text{if } F \text{ is atomic} \\
\sigma_f(F \wedge G) &= \sigma_f(F, I) \wedge \sigma_f(G, I) \\
&\vdots \\
\sigma_f(F \Rightarrow G) &= \sigma_f(F, I) \Rightarrow \sigma_f(G, I) \\
\sigma_f([C]F) &= \bigwedge_{\substack{i \in I \\ \text{class}(i) = C}} \sigma_f(F, I'(i))
\end{aligned}$$

In these definitions we denote by boldface letters the constants, functions resp. relations of the concrete domain. The counting quantifiers ($\exists^{\geq n}$), taken from an extension of predicate logic, are true, if there are at least $n$ different values fulfilling the condition following the operator. So a formula $(\exists^{\geq n} x)F$ can be translated to $(\exists x_1, \ldots, x_n)(F(x_1) \wedge$

$\cdots \wedge F(x_n) \wedge x_i \neq x_j$ f.a. $i \neq j$). The modification of the set of considered instances in the definition of the context operator $[\cdot]$, $I'(i)$, is defined over the aggregation tree structure using auxiliary functions $A(i)$ and $N(C, i)$:

$$I'(i) = A(i) \cup \bigcup_{j \in \text{parent}(i)} N(\text{class}(i), j)$$

$$A(i) = \{i\} \cup \bigcup_{j \in \text{children}(i)} A(j)$$

$$N(C,i) = \begin{cases} \emptyset & \text{if class}(i) = C \\ \{i\} \cup \bigcup_{j \in \text{children}(i)} N(C,j) & \text{otherwise} \end{cases}$$

Considering node H1 of our example of Figure 2, we obtain

$A(\text{H1}) = \{\text{H1}, \text{D1}, \text{S5a}, \text{S6a}, \text{S3a}\}$ and
$N(\text{HostConfiguration}, \text{SC1}) = \{\text{SC1}, \text{S1}, \text{S2}\}$, and thus
$I'(\text{H1}) = \{\text{H1}, \text{D1}, \text{S5a}, \text{S6a}, \text{S3a}, \text{SC1}, \text{S1}, \text{S2}\}$.

Therefore, $I'(H1)$ is a set containing (among others) all instances of setting classes relevant for node H1, which justifies our definition of the context operator via function $I'$. Now, as usual, a formula $F$ is said to be satisfied by a set of instances $I$, if $\sigma_f(F, I) = \text{true}$.

Note, that all non-logical functions and predicates exclusively take set-valued arguments. This enables a wide variety of functions being uniformly and naturally definable. A comparison operator $<$, ranging over sets of natural numbers, may, e.g., be defined as

$$\sigma_a(s_1 < s_2) = \bigwedge_{x \in s_1, y \in s_2} x < y \quad .$$

Set-containment, as another example, can be defined directly. But there are further possibilities, e.g. in defining vary-adic operators. That way, sum- or minimum/maximum-operators can be defined, and statements such as $C_1.P < \text{sum}(C_2.Q)$ become expressible. This allows formulation of complex, but common dependencies.

## 5.3. Examples

Turning back to Apache configuration, we now want to give some examples on how to use the constraint specification language $\mathcal{CCL}$. In Figure 3 we give formal variants of part of the specification stated in natural language in Section 4 above, as well as some examples taken from the Apache documentation [1]. 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 in Section 4. Here we also used the comparison operator $<$ and its converse $>$ as defined above.
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 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 [1]).
6. A configuration name and PID file must be specified for the Web-server.

## 5.4. Constraint Evaluation

Constraint sets as those of the last section have to be evaluated in order to check whether or not they hold for a concrete configuration. We therefore implemented *CIMVerifier*, a prototypical constraint checker. This verifier uses a variant of our language $\mathcal{CCL}$, called *ConQuery*, in which all logical operators have a textual representation. CIMVerifier is a Java client that employs the CIM and WBEM infrastructure to access the Apache configuration data. Constraints are checked sequentially by evaluating the constraints in an innermost fashion, using Java *Reflection* to evaluate user-defined predicates and functions.

In case of a violated constraint a simple yes/no answer is often not enough to locate an error, and assistance of the constraint verification system to guide the user can be of great help. So we implemented a mechanism to find possible configuration errors by identifying properties that are set to a wrong value with highest probability. Therefore, we assume that the values of all parameters of a violated constraint are responsible for the failure with equal probability. Then we can proceed as follows: Let $C = \{c_1, \ldots, c_n\}$ be the set of violated constraints, and $\#\text{occ}(p, ci)$ the number of occurrences of parameter $p$ in constraint $c_i$. We then compute for each parameter $p$ occurring in $C$ its weight,

$$w(p) = \sum_{i=1}^{n} \frac{\#\text{occ}(p, c_i)}{|c_i|},$$

where $|c_i| = \sum_{p \in P} \#\text{occ}(p, c_i)$ denotes the total number of properties occurring in constraint $c_i$. The parameter with highest weight $w(p)$ is then assumed to be most likely responsible for the constraint violation.

## 6. Conclusions, Related and Future Work

We presented a verification approach for Apache Web-server configurations. The verification is based on an

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

**Figure 3. Formalization of some Consistency Properties in $\mathcal{CCL}$.**

object-oriented semi-formal CIM model of the configuration data and a specialized constraint specification language. Our extensible specification language reflects typical constructs found in CIM and currently allows formulation of constraints containing predicates and functions over numbers, sets and strings. We also implemented a prototypical constraint evaluator based on Java Reflection and the WBEM infrastructure. This implementation facilitates error recovery by computing weights for probably faulty configuration settings.

Representative for other work on formal verification of (semi-formal) UML-diagrams we want to mention Dupuy-Chessa and du Bousquet's validation of UML models [6] and Meyer and Souquières' formalization based on the specification language B [12]. In contrast to there 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 [2], 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 [5]. Work on validation and integrity checking of XML data can also be found in the literature [13].

Future work may include application of automatic theorem proving methods to CIM verification using $\mathcal{CCL}$. This would offer additional possibilities, e.g. in checking the consistency constraints by themselves, in automatic completion of partially specified configurations, and in automatic error correction. Moreover, a complexity theoretic analysis of our specification language $\mathcal{CCL}$ and a comparison with current description logics could be of interest.

## References

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

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

[3] Distributed Management Task Force, Inc. *Common Information Model Specification*, 1999. `http://www.dmtf.org/standards/documents/CIM/DSP0004.pdf`.

[4] Distributed Management Task Force, Inc. *CIM Operations over HTTP*, 2002. `http://www.dmtf.org/standards/documents/WBEM/DSP200.html`.

[5] J. 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.

[6] 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.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[8] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[9] IBM Linux Technology Center. *Standards Based Linux Instrumentation for Manageability*, 2000. `http://oss.software.ibm.com/sblim`.

[10] W. Küchlin and C. Sinz. Proving consistency assertions for automotive product data management. *J. Automated Reasoning*, 24(1–2):145–163, Feb. 2000.

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

[12] E. Meyer and J. Souqui"'eres. 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.

[13] 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.

[14] *Web Ontology Language (OWL) Reference Version 1.0*, 2002. W3C Working Draft 12 November 2002. Latest version available at `http://www.w3.org/TR/owl-ref`.

[15] C. Sinz, T. Lumpp, and W. Küchlin. Towards a verification of the rule-based expert system of the IBM SA for OS/390 automation manager. In *Proceedings of the 2nd Asia-Pacific Conference on Quality Software (APAQS 2001)*, pages 367–374, Hong Kong, Dec. 2001. IEEE Computer Society.

[16] C. Sinz, T. Lumpp, J. Schneider, and W. Küchlin. Detection of dynamic execution errors in IBM System Automation's rule-based expert system. *Information and Software Technology*, 44(14):857–873, Nov. 2002.

[17] R. Waldinger and M. Stickel. Proving properties of rule-based systems. *Intl. J. Software Engineering and Knowledge Engineering*, 2(1):121–144, 1992.