Configuration Reasoning and Ontology For Web

Dana Glasner*
Columbia University
dglasner@cs.columbia.edu

Vugranam C. Sreedhar
IBM TJ Watson Research Center
vugranam@us.ibm.com

Abstract

Configuration plays a central role in the deployment and management of Web infrastructures and applications. A configuration often consists of assigning "values" to a pre-defined set of parameters defined in one or more files. Although the task of assigning values to (configuration) parameters looks simple, configuring infrastructures and applications is a very complex process. In this paper we present a framework for defining and analyzing configuration of an Apache server. We define the notion of "configuration space" of an Apache server as a set of possible values that can be assigned to configuration parameters. We then define the notion of an "obstacle" and "forbidden region" in the configuration space that should be avoided. We model configuration space using a logical framework based on OWL (Web Ontology Language). The obstacles and forbidden regions in the configuration space are modeled as constraints in the logical framework. These obstacles and forbidden regions are essentially "anti-patterns" that a typical installation should avoid. Given an instance of a configuration (that is, a "point" in the configuration space) we then check if the instance is "obstacle free" using logical reasoning.

1. Introduction

Configuration plays a central role in the deployment and management of Web applications and infrastructures. Web applications and infrastructures are often susceptible to malicious attacks. A naive and default configuration almost always leads to security and performance problems. A 2003 Gartner information security report concluded that 65% of attacks are due to poorly configured or mis-configured systems.[1] Configuring infrastructures and applications is a very complex process. Configuring a Web application involves many steps, and they include setting many different configuration parameters. Understanding the consistency of different configuration parameters is overwhelming. Also, often a system administrator has to deal with configuring many different and interacting Web applications and runtime environments. For instance, the configuration of Apache Web server can interact with the configuration of WebSphere and DB2 backend server. Such configuration interaction is even more pronounced in high-volume data centers. Also, in data centers configurations of different data center sub-systems are done by different people over a period of time. So it is extremely important to address the consistency problem of configurations.

In this paper we present a framework, called CROW (Configuration Reasoning and Ontology for Web), for defining and analyzing the configuration of an Apache Web server. Although our framework is general and can be applied to other system configurations, we focused on Apache server for several reasons. First, Apache server is the most popular Web server installation. Second, each application or system have their own way of setting configuration parameters, and we did not want to fall into the "tar pit" of understanding various systems configuration and deviating from our framework.[2] Apache server provides a simple way to set configuration parameter. Third, we wanted to focus on a specific application domain to implement our framework and use our framework to help our product team.

Our approach to configuration analysis is inspired from path planning in Robotics: given a robot, a workspace, a set of obstacles, and robot path, the goal is to determine whether the robot can navigate the path without colliding with any obstacles. The notion of "configuration space" is a fundamental concept in Robotics that simplifies path planning to motion planning of a "point" in "free space". In order to understand Apache configuration, we define the notion of *configuration space* as a set of possible values that can be assigned to configuration parameters. A *configuration* then is essentially a *point* in the configuration space. A configuration point is made of a tuple of *coordinates* that uniquely identifies the point in the configuration space. Each element of a tuple corresponds to a dimension in the coordination space. An *obstacle* is a configuration that should be avoided, and *forbidden region* is a set of obstacles. In our framework a forbidden region is a

*The work described in this paper was done when the author was doing her summer internship at IBM TJ Watson Research Center.

[1] Taxonomy of Software Vulnerabilities, John Pescatore, Gartner, Inc. 11 September 2003.

[2] Although we are currently investigating configuration of a large data center that has such tar pits (see Section 6).

2007 IEEE International Conference on Services Computing (SCC 2007)
0-7695-2925-9/07 $25.00 © 2007 IEEE

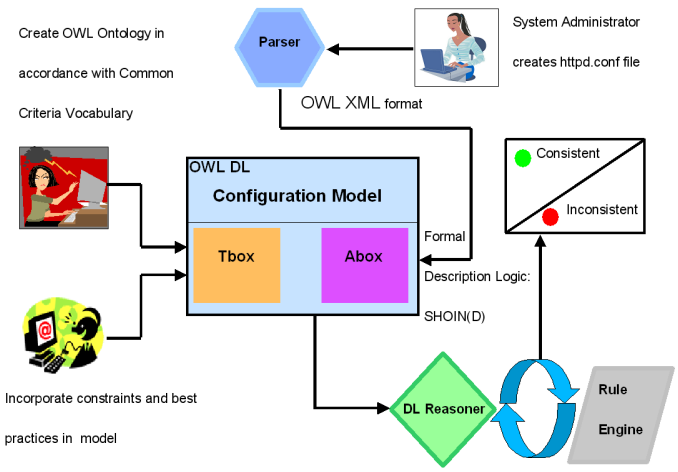

Figure 1. The CROW framework.

sub-set of configuration space that satisfies certain well known anti-patterns [2]. We reduce the problem of Aapche configuration analysis to collision free path planning in Robotics: given an Apache configuration space, the set of obstacles (anti-patterns), and a `httpd.conf` file, the goal is to determine whether the configuration parameter settings in `httpd.conf` will "collide" with any of the anti-patterns.

Example: The main Apache configuration file `httpd.conf` is a plain text file and the file simply contains a list of *directives*. A directive is a "command" or an "instruction" to the Apache runtime to respond or behave in a certain (directed) way. Consider the following snippets of a typical `http.conf` file

```
# This directive configures what you return as the Server
# HTTP response Header. The default is 'Full' which sends
# information about the OS-Type and compiled in modules.
# Set to one of: Full|OS|Minor|Minimal|Major|Prod, where
# Full conveys the most information and Prod the least.
#
ServerTokens Prod

# Optionally add a line containing the server version
# and virtual host name to server-generated pages
# (error documents, FTP directory listings, mod_status
# and mod_info output etc., but not CGI generated
# documents). Set to "EMail" to also include a mailto:link
# to the ServerAdmin. Set to one of: On|Off|EMail
ServerSignature Off
```

Each directive, such as `ServerToken`, contributes to a dimension of the configuration space. Notice that a directive can take on one of the many values, for example, `ServerSignature` can be one of `On`, `Off`, or `Email`. An obstacle or a forbidden region is a value of a directive that is considered to affect the Apache server in negative ways, and so should be avoided if possible. For instance, if we set `ServerTokens` to `Full`, outside users can learn which modules are running on the system and which version number is installed. This enables them to exploit vulnerabilities that are present in specific versions, and it is recommended to set `ServerTokens` to `Prod`.

There are several approaches to model configuration space and define forbidden region. and in our framework we will use OWL (Web Ontology Language). OWL is a language for describing ontologies, and an ontology is a (formal) description of concepts and their relations.[3] We chose OWL-DL as a starting point for modeling configuration space for several reasons: (1) OWL-DL is decidable and so any statement in the underlying logic is either *true* or *false*. This allows us to reduce the number of false errors. (2) OWL-DL is a simple modeling language and it is easy to express constraints within the language. (3) There are a number of off-the-shelf tools available to reason about the underlying logic, and we did not want to spend time to write our own reasoner or constraint solver. (4) OWL-DL is extensible and it allows us to go beyond DL to express certain anti-patterns that include logical implication. A model in OWL consists of a Terminological Box (T-box) and an Assertion Box (A-box). A T-box contains classes and the relationships between classes, including restrictions on classes, such as two classes that are defined to be disjoint, and the relations between those concepts. An A-box contains assertions about specific instances that can relate an instance to a class or relate two instances with each other.

Our approach to modeling the configuration space and forbidden region is quite simple. We represent each dimension as a class and forbidden regions as constraints on classes. Configuration points are then defined as class instances. Now given a configuration file `httpd.conf` file (path plan), we check whether the configuration will collide with anti-patterns using an off-the-shelf OWL reasoner. The configuration space, even for Apache server, is intractable, if not infinite, and it is impractical or even impossible to synthesis automatically the set of all obstacle-free configurations. Therefore in our framework we focus on checking whether a given configuration is in a forbidden region or not. The overall structure of our framework is illustrated in Figure 1. We wrote a simple Perl parser that essentially translates the `httpd.conf` file into instances of A-box.

During our modeling process we observed that there are a few cases of anti-patterns that cannot (naturally) be expressed using pure OWL-DL. To express such anti-patterns we then used OWL-DL-Safe rules [5]. OWL-DL-Safe rules combines OWL-DL and function-free Horn rules (clauses) by ensuring that every variable in a rule occurs in a non-DL atom (see Section 3 for more details). OWL-DL-Safe rule is once again decidable, and is more expressive than both OWL-DL and function free Horn rules. Currently our framework consists of 60 classes in the T-box, about 15 constraints on classes, and about 55 properties with constraints. The parser translates `httpd.conf` to OWL format generates about 500 A-box elements. Currently we have 3 DL-Safe rules, and we hope to

[3] http://www.w3.org/TR/2004/REC-owl-features-20040210/#domain

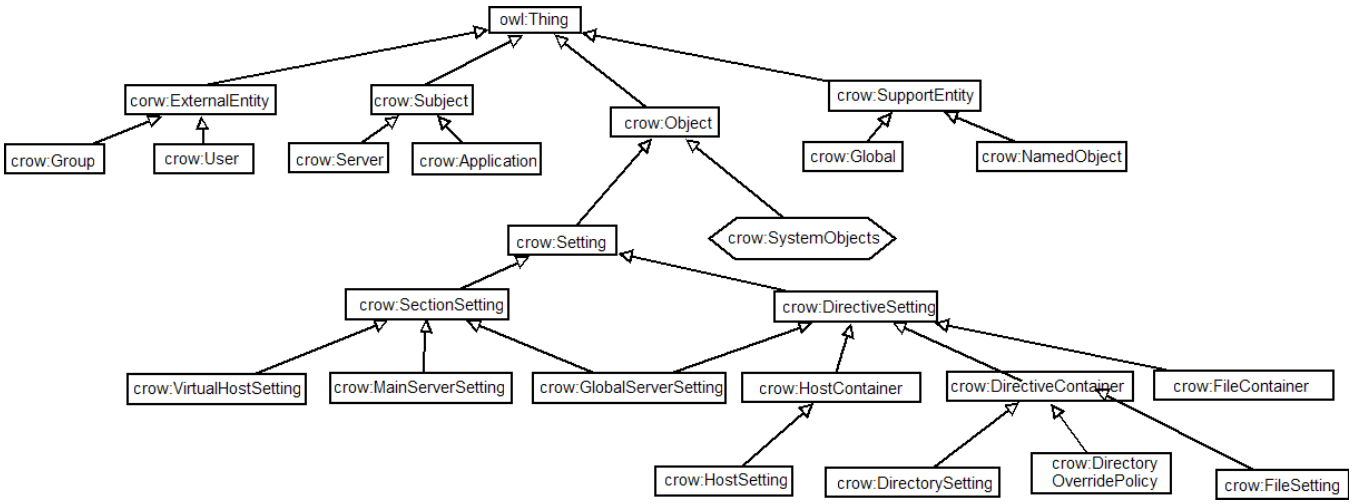

Figure 2. The T-box representation of Apache configuration model.

include more DL-safe rules in future. We use the OWL reasoner, Pellet, for reasoning about the A-box and T-box and the Jess Rule Engine for OWL-safe rules.

The rest of the paper is organized as follows: In Section 2 we introduce configuration space model and reasoning capabilities using OWL framework. In Section 3 we discuss OWL-DL safe rules. In Section 4 we present our experience on analyzing some Apache configuration files. In Section 5 we discuss related work and conclude in Section 6.

2. Configuration Space

Let C be the configuration space of Apache server, and $q \in C$ a point or configuration in C. Let $o \in C$ be an obstacle point and let $O = \{q \in C | q = o\}$ be the set of obstacle points or forbidden region. An obstacle point is essentially a configuration that can affect an Apache installation in a negative way, such as make the server vulnerable to attacks or can lead to poor performance. The set $G = C - O$ is the set of *free* or *well-behaved configuration space*. In Robotics a free configuration space essentially allows free navigation of a robot. In our framework a free configuration space will ensure that an Apache server is well-behaved with respect to the set of anti-patterns. Unfortunately computing the well-behaved configuration space for Apache is difficult. Recall that our focus in this paper is to determine whether a given configuration instance q_h, as defined in `httpd.conf`, will "collide" with any obstacles, that is, $q_h \in O$.

In Robotics computing configuration space consists of enumerating "contact surfaces" for every pair of features (such as vertex, edge, texture, etc.) from a robot and obstacles. In our framework an Apache configuration is a *specification* of values of all directives defined by Apache server. In our framework the specification of values are defined in terms of "classes" and predicates or relations among classes. In essence we map configuration space to a description logic framework.

2.1. OWL and Description Logic

To simplify the presentation we will use Protégé OWL notation throughout this paper.[4] An OWL ontology is made of *classes*, *instances* (also called as *individuals*), *properties*.

- We define configuration space in terms of OWL classes. Each class correspond to a dimension in the configuration space. Each dimension can have a set of values that "spans" dimension. For example, the class `File` is a set of file instances.
- Instances are the objects in the domains of discourse that we are modeling. In our case, instances are points in the configuration space.
- Properties are binary relations on instances, and in effect link two instances. For example, a particular file `isIn` in a particular path. A property can be an inverse of another property. For example, `contains` can be defined as being inverse the of `isIn` property: a file `httpd.conf` `isIn` the path `\usr\local\apache\` and the path `\usr\local\apache\` `contains` the file `httpd.conf`. A property can be defined to be *functional*, which are single valued properties. For instance, let us define the `isDirectlyIn` property as being functional and then define `httpd.conf isDirectlyIn \usr\local\apache\`. Now if `httpd.conf` is a also directly in another path, say, `\usr\local\apache\conf\` then we conclude that the two paths are the same, unless the two paths

[4] http://protege.stanford.edu/

are explicitly stated (i.e., anti-pattern or obstacle) to be different instances, in which case we trigger an inconsistency.

2.2. Defining Configuration Classes

Recall that classes in OWL can be related with one another in a hierarchical relation. A sub-class specializes a super-class. For example, `ConfigurationFile` is a sub-class of `File`. By default, classes in OWL overlap — an instance can be a member of more than one class. One can define two classes to be *disjoint*, in which case an instance cannot be a member of both classes.

To reduce the complexity of navigating through collision-free space, we induce some structure to the configuration so that we can "carve out" sub-spaces of interest. The only high level structure that is explicitly commented in the default `httpd.conf` are the three different sections of directives: (1) *Global Environment:* This section contains directives that affect the overall operation of the Apache server. (2) *Main Server Configuration:* This section contains directives that sets up the main server to respond to requests that are not handled by a virtual host. (3) *Virtual Host Configuration:* This section contains directives that sets up virtual hosts which allow Web requests to be sent to different IP addresses or hostnames and have them handled by the same Apache server process. We explicitly model these structures in our ontology.

To induce some more structure into the understanding of the Apache configuration we also use the following concepts in our ontology: (1) A *subject* is an active entity that performs operations or actions. (2) An *object* is a passive entity and a subject typically performs some action on one or more objects. (3) A *user* is an external user of the Apache server.

We first identify the directives that influence the subject, the object, and the user aspect of the Apache server. At the root of the hierarchy is the `owl:Thing` which is the base class for all OWL classes. We then define four main classes of the CROW T-box: (1) `crow:Subject`, (2) `crow:Object`, (3) `crow:ExternalEntity`, and (4) `crow:SupportEntity`.[5] The first three classes `crow:Subject`, `crow:Object` and `crow:ExternalEntity` correspond to the directives that influence the subject, the object, and the external entities (users) aspect of the Apache server. Figure 2 illustrates the main class hierarchy of the T-box for the Apache configurations. The directives that influence the subject aspect of Apache server are specialized under `crow:Subject`. For example, `crow:Server` and `crow:Applications` are specializations of `crow:Subject`. We similarly identify directives that influences the object aspect and specialize them as sub-classes of the `crow:Object`. For instance, directives that influence files, ports, sockets, etc are modeled as specialization of the `crow:Object` class. Similarly directives that influence external entities such as the users and groups are modeled as specialization of the `crow:ExternalEntity`.

For the three sections defined in the default `httpd.conf` file (see above) we create a `SectionSettings` class that specializes the `Settings` class. We then create three specializations of `SectionSettings` class. As mentioned above these are: `MainServerSettings`, `VirtualHostSettings`, and `GlobalServerSettings`. It is important not only to model the physical structure of the file, it is also important to model and categorize the `httpd.conf` file based on directive types. We create a dual view of the `httpd.conf` file since either view may be advantageous depending on the application. In this structure, we create another subclass of `Settings` called `DirectiveSettings` which in turn is classified into `HostContainer`, `DirectoryContainer`, `FileContainer`, and `GlobalServerSettings`, as before. In effect, this structure categorizes the elements of the `httpd.conf` file based on the `<Virtual Host>`, `<Directory>`, and `<File>` directives. Conceptually, `GlobalServerSettings` will contain global directives that pertain to the server itself. These include `serverRoot`, `serverTokens`, and `listen` directives. The `HostContainer` maps to the `<Virtual Host>` directive and also contains the settings of the default host even though within the actual `httpd.conf` file these are specified outside of a `<Virtual Host>` container, since structure-wise they are equivalent. The `DirectoryContainer` class corresponds to both the `<Directory>` and `<Location>` directives. Finally, the `FileContainer` class corresponds to the `<File>` directive.

2.3. Configuration Instances and Forbidden Space

A-box contains instances of T-box classes and assertions about specific instances that can relate an instance to a class or relate two instances with each other. We wrote a simple Perl tool to parse an existing `httpd.conf` file and then generate an OWL XML file that represents the A-box for CROW. We import the resulting XML file into the Protégé tool (see Figure 1). We then check the consistency of the configuration instance that is imported. In other words, we take a bottom-up approach for checking consistency of existing `httpd.conf` file of an installed Apache server.

Defining forbidden space consists of imposing restrictions on the elements of T-box and the properties in A-box. We follow a few modeling principles to simplify reasoning in CROW. OWL does not use the unique names assumption. To implicitly construct unique names, we use an `id` property that is functional and is unique for each instances of non-disjoint classes. This `id` property will effectively model unique-names

[5] We use `crow:` to define the CROW namespace. All CROW classes, instances, properties, and rules are part of this namespace.

Consistency Check	T-box Class	OWL Assertion	Test Case
Error Log is not located inside Document Root or any aliased Directory.	HostSettings	Necessary Condition: • not (errorlog some (isIn some (hasAlias some Location)))	documentRoot /usr/apache errorLog /usr/apache/error-log
Only CGI Directories or directories within CGI Directories have the option ExecCGI	NotCGIDirectories	Necessary and Sufficient Condition: • Path • not CGIDirec • isDirectlyIn some (cgidirec has notCGIDirectory) Necessary Condition: • cgidirec has notCGIDirectory • pathAssociatedWith only (options only (not {ExecCGI}))	documentRoot /usr/local/apache <Directory /usr/local/apache> options ExecCGI <Directory>
Access Control for Document Root is specified in httpd.conf file	UnspecifiedPath	Disjoint With: • isDocumentRootOf some HostSettings	documentRoot /usr/local/apache #does not appear in a #<Directory> directive
Every Server has exactly 1 ServerRoot defined	ServerContainer	Necessary Condition: • serverRoot exactly 1	serverRoot /usr/local/apache serverRoot /usr/apache/local
All Ports listened to by Hosts are listened to by the Server.	PortsListenedToByServer, PortsListenedToByHost	(Server)Necessary and Sufficient Condition: • Port and isListenedToBy some ServerContainer • $\{enumeratedinstances\}$ (Host)Necessary and Sufficient Condition: • Port and isListenedToBy some HostContainer • $\{enumeratedinstances\}$ (Host)Necessary Condition: • PortsListenedToByServer	<VirtualHost host2:443> . . <VirtualHost> #no Listen directive #for this port
Besides for root directory, non-aliased directories are not specified in httpd.conf (if it is, probably an error in directory name).	NotAliasedDirecs	Necessary and Sufficient Condition: • Path • not AliasedDirec • isDirectlyIn some (aliasdirec has notAliasedDirec) Necessary Condition: • aliasdirec has notAliasDirec Disjoint With: • $\{enumeratedinstances\}$ (SpecifiedPaths minus root directory)	documentRoot /usr/local/apache <Directory /usr/apache/local> . . <Directory>
ServerSig is on Off, ServerTokens is on Prod	ServerContainer	Necessary Condition: • serverSig has Off • serverTokens has Prod	• serverSig On • serverTokens Full

Table 1. Consistency rules and checks

assumptions for such instances that we will use implicitly during the reasoning process. Intuitively, this essentially creates a "unique name" for each instances of the class associated with this property in the A-box since now any two instances cannot be merged. Table 1 presents a subset of the consistency checking rules that we implemented in CROW. The table is self-explanatory and we encourage the reader to go over it in detail.

OWL's open world assumption can sometimes complicate the modeling process. In the Apache server the set of ports listened to by a virtual host must be a sub-set of the set of ports listened to by the server. This can be expressed in the `httpd.conf` file. Unfortunately modeling this using the "open world assumption" can lead to some confusion. To explain this further, let us create a property called `isListenedToBy` that relates an instance of a `Port` class with instance of a `HostContainer` class or an instance of a `ServerContainer` class. We then want to restrict that the set of ports listened to by an instance h of `HostContainer` to be a subset of the set of ports listened to by an instance s of `ServerContained`. With open world assumption, when a new port p is created that is not listened to by s but is listened to by h, the reasoner will simply relate p as being listened to by s. Rather we want the reasoner to trigger an inconsistency error in this case. To enable such inconsistencies we create an "enumerated" sub-class of the `Port` class for those ports that are restricted to be listened to by some instance of the `ServerContainer`. Such enumerated classes behave like closed-world classes, using which we can track such inconsistencies.

Whenever the imported A-box instances do not meet the constraints and restrictions of the elements of T-box and properties in A-box the logic reasoner will trigger inconsistencies in the model. For example, two classes may be specified to be disjoint, and yet an instance in the A-box is defined to be a member of the two disjoint classes. This is clearly inconsistent with the model and the reasoner will return an inconsistency error. In practice, most inconsistencies stem from the fact that

a class and its complement must be disjoint and then deriving that an instance is a member of both such classes.

3. OWL DL Safe Rules

During the development of CROW we observed that there are certain anti-patterns that cannot naturally be expressed using the base OWL DL. We wanted to confine CROW reasoning to be decidable, and yet more expressive than the base OWL DL. We then explored the possibility of using OWL DL Safe Rules (OSR), introduced by Motik et al [5], which combines OWL DL and function free Horn clause in a certain decidable way. DL safe rules are Horn Rules, where each variable in the rule occurs in a non-DL-atom in the rule body. These rules allow the extra expressivity of non-tree-structured relationships between variables and yet even in combination with OWL DL still maintain the decidability property. A predicate O that is not part of the description logic is chosen. The predicate is applied to each individual in the A-box and for DL-Safe rules, each variable in a rule appears in an atom that consists of this predicate [5]. Intuitively, this creates a closed-world policy only for those individuals that are directly participating in the rules. However, the existential operator can still be used to infer existence of individuals within the model. These inferences can actually affect the way the rules are applied, although the inferred individuals do not appear explicitly in the rules. Therefore, adding DL-Safe Rules is not equivalent to just enforcing a closed-world policy on the total reasoning. The resulting hybrid of DL and DL-Safe Rules is decidable. The following is an example of OSR that we implemented in CROW.

$$UnSpecifiedPath(?x) \wedge isDirectlyIn(?x, ?y) \wedge associatedWithDirectory(?z, ?y) \wedge allowFrom(?z, ?a) \wedge associatedWithDirectory(?b, ?x) \rightarrow allowFrom(?b, ?a)$$

Currently there is no open-source reasoner available for the combined reasoning. We use SWRL rules and a SWRL Rule Engine, called Jess, to add rules to our CROW model [4]. Currently, the way OWL interfaces with the SWRL Rule engine is that the DL Reasoner and the Rule Engine run in tandem. When the Rule Engine is initiated, the relevant individuals, classes, and properties are exported to the Rule Engine. Then the Rule Engine runs and the new knowledge it outputs is imported into the A-box of DL model. Then the DL Reasoner is initiated and reasons on the combination of A-box and T-box, new inferences are made and then the Rule Engine can run again. This process is guaranteed to reach a fixed point, but it may not catch all inconsistencies that a combined reasoner does. We hope to use a more powerful reasoner, when available, to possibly find more inconsistencies in CROW.[6]

[6]It is important to keep in mind that SWRL, in its full generality, is undecidable. OWL-DL-Safe rule language is still a decidable language.

4. Evaluation

Our product partners approached us to help them with major issues in understanding problems related configuration of large data center. So we started with an incubation project to analyze configuration of Apache Web servers. When we started working on the CROW project we were deliberating which modeling language to adopt for checking consistencies of configurations. Our long term goal is to develop a CROW-like tool for checking configurations of integrated applications in the context of a large data center. We explored the possibility of using modeling languages such as CIM (Content Information Model) and UML (Unified Modeling Language). Unfortunately these modeling languages are either informal or semi-formal and one typically has to write procedural logic or custom solvers for reasoning about the models. Our experience in using OWL, OSR, and the off-the-shelf tools has been quite positive.

We currently encode about 15 security constraints and rules within the ontology. Because the structure of DL lends itself to this purpose, we are able to express this with only one or two lines of "code" for each constraint. A small apache configuration file will input about 500 assertions into the A-box. Therefore, our method gives a very lightweight way to check for many inconsistencies. If the consistency checks were encoded in a standard programming language, it would require need hundreds of lines of code to achieve this expressiveness. We tested a few production-level Apache `httpd.conf` files for consistencies. In one particular Wiki application we found the following inconsistencies.

- `ServerTokens` is set to `Full`.
- `ServerSignature` is `On`.

  When `ServerTokens` directive is set to `Full`, outside users can learn which modules are running on the system and which version number is installed. This enables them to exploit vulnerabilities that are present in specific versions. We recommend that `ServerTokens` should be set to `Prod` `ServerSignature` should be set to `Off`.
- `C:/xampp/xampp/htdocs` (the document root) has `ExecCGI` option.
- `C:/xampp/xampp/webalizer` has `ExecCGI` option.

  It is best to have one location for CGI scripts. In this way, it is easier to administer and the environment the CGI scripts run in can be more controlled. It is especially a dangerous practice to have CGI enabled in the Document Root directory. Any CGI scripts in these directories should be moved to the `cgi-bin` directory.

We also ran our reasoner on other `httpd.conf` sample files voluntarily submitted and found the following errors:

1. `/home/www/default/web/WWW` was designated as Document Root but never specified within a `<Directory>` directive.

2. `/var/www/cgi-bin` was specified in a `<Directory>` directive without being aliased.

We also wrote several unit test cases of Apache configurations to verify our modeling approach. In particular we also modeled the best practice rules given by Sinz et al [7]. Sinz et al. use Common Information Model (CIM) framework for modeling and verifying configuration properties of the Apache servers. Table 2 shows the CIM model and the CROW model for the set of best practices given by Sinz et. al. We can easily and intuitively express all of Sinz et al. rules in CROW. Additionally, we extend some of their rules in a way that is simple in our model but cannot be done in their framework. For example, we changed the rule that Error Log cannot be located inside Document Root, to Error Log cannot be located inside any aliased Directory. Since Sinz et al. does not model the Directories themselves, but only the names of directories as strings, they are unable to express this more general property.

One aspect of an Apache configuration that we have not yet addressed in CROW is the Apache override policy behavior done through `.htaccess` files. Apache configuration allows `.htaccess` files to be specified for directories whose permissions may be overridden. The `.htaccess` file can override the permissions specified in the main httpd.conf file on a per directory basis. On the surface, the properties of the `.htaccess` files seem to require non-monotonic reasoning. This is because original permissions specified in the httpd.conf file can be reversed by the `.htaccess` file. We can get around this problem by enforcing a closed-world policy on the .htaccess files: All `.htaccess` files that will be used must be known at the time the Apache model is created. However, this may not completely reflect the nature of the `.htaccess` files. We are currently looking at the .haccess file semantics and its implication to CROW modeling.

5. Related Work

CROW is closely related to the CIM model of Apache configuration proposed by Sin et al. [7]. CIM Schemas are represented by UML Diagrams.[7] Association classes are used to model relationships between objects. Although, CIM is a popular modeling language for data modeling, it is a semi-formal model, unlike OWL. Sinz et al. define a custom formal semantics by mapping their CIM to a logic that is inspired by description logic. Unfortunately, the resulting logic is not decidable and also Sinz et al. wrote a custom reasoner and constraint solver for analyzing the consistency properties of the Apache configuration. Our work uses off-the-shelf modeling tool Protégé with off-the-shelf reasoner Pellet and Jess. We also confine our logic to a decidable sub-set that includes OWL-DL and OSR. Although one can envision security best practices that cannot easily be expressed using decidable logic, we believe that in practice this may not be an issue. Finally, Sinz et al. do not follow the CC principle for classifying and modeling configuration. We believe our approach of using the CC classification principle can help standardize vocabulary and ontology for modeling configurations.

Eilam et al. [3] propose a Model-Driven Architecture (MDA) approach for modeling configuration that complements our work. They use a refinement engine that automatically searches the space of service model transformations to produce a set of possible service physical deployment topologies and configurations. Agrawal et al. [1] describe how to validate a storage area network configuration based on a list of policies that has been created. They define Storage Area Networks (SANs) as an alternative storage pradigm that allows storage to be shared among servers using fast interconnects [1]. Agrawal et al. define a policy-based SAN configuration validation system that can be used to specify, store and evaluate configuration policies for SANs. They model SAN components and major concepts such as host, switch, storage device, host bus adapter (HBA), port-controller, port, link, fabric, and zone. Additionally, to extend the expressiveness of logical operators and comparison operators, they introduce five new operations on collections of elements. Based on this, they create a policy evaluator that operates on the boolean expressions that represent the policy conditions. They also compile a list of 64 policies that their evaluator validates.

Raghavachari et al. [6] propose a method for finding the best configuration parameters for a specific application. They propose varying the configuration parameters over all values within a specific range and testing the resulting configuration files to see which parameters are best. They randomly choose the value of the configuration parameter and continue the process until all possible values have been checked. This method does not check for effects of interacting configuration parameters since each one is tested independently. It is can also potentially generate a huge number of configurations that must be analyzed.

6. Conclusions

In this paper we presented the framework of CROW that follows the principle of the CC for classifying the elements (directives) of the Apache configuration. We used an off-the-shelf tool and reasoner for modeling and analysis of configurations. We modeled several well known best practices for configurations and used them to analyze extant production-level Apache configurations. We found a few inconsistencies in the configuration files, including innocuous ones. We are currently extending CROW to modeling and analyzing configurations of interdependent systems, such as applications running a data

[7] http://www.omg.org/technology/documents/formal/uml.htm

Constraint	CIM Model	CROWModel
The ServerRoot property is defined exactly once (per server)	$\exists^{=1} ServerProperties.ServerRoot$	(ServerContainer)Necessary Condition: (an additional enumerated class to simulate closed-world was created for all objects that have at least one serverRoot defined) • `serverRoot` exactly 1 • `associatedWithServer` exactly 1
MinSpareServer is less than MaxSpareServer	$[ServerConfig](ServerProperties.MinSpareServer < ServerProperties.MaxSpareServer) \wedge [ServerConfiguraion](ServerProperties.MaxSpareServer > 1)$	SWRL Rule: • $ServerContainer(?x) \wedge minSpareServer(?x, ?y) \wedge maxSpareServer(?x, ?z) \wedge lessthanorequal(?z, ?x) \wedge Global(?a) \rightarrow badMinMax(?a, "true'')$ • $ServerContainer(?x) \wedge maxSpareServer(?x, ?y) \wedge lessthanorequal(?y, ?1) \wedge Global(?a) \rightarrow badMax(?a, "true'')$ (Global)Necessary Condition: (there will always be exactly one instance of global in the A-box) • `badMinMax` has "false" • `badMax` has "false"
Each virtual host has its own unique server name	$\lvert HostProperties.ServerName \rvert = \lvert HostConfig.Name \rvert$	serverName property: (an additional enumerated class to simulate closed-world was created for all objects that have at least one serverName defined) • domain: HostSettings • range: Name • functional • inverseFunctional
Error Log should not be stored in DocumentRoot or a subdirectory	$[HostProperties] \neg isPrefixOf(HostProperties.DocumentRoot, HostProperties.ErrorLog)$	Necessary Condition: • not (`errorlog` some (`isIn` some (`hasAlias` some `Location`)))
The address/port pair of each virtual host must be an address/port the Web-server is listening to	$[HostConfig]HostProperties. < HostAddress, HostPort > \subseteq ListenSetting. < ListenAddress, ListenPort >$	(Server)Necessary and Sufficient Condition: • `Port` and `isListenedToBy` some `ServerContainer` • $\{enumeratedinstances\}$ (Host)Necessary and Sufficient Condition: • `Port` and `isListenedToBy` some `HostContainer` • $\{enumeratedinstances\}$ (Host)Necessary Condition: • `PortsListenedToByServer`
A configuration name and PID file must be specified for the Web-server	$\exists ServerProperties.ConfigName \wedge \exists ServerProperties.PIDFile$	(ServerContainer)Necessary Condition: • `configName` min 1 (an additional enumerated class to simulate closed-world was created for all objects that have at least one configName defined) • `pidFile` min 1 (an additional enumerated class to simulate closed-world was created for all objects that have at least one PIDFile defined) • associatedWithServer exactly 1

Table 2. Comparing CROW and CIM model

center.

Acknowledgement: We thank Gabriela Cretu, Achille Fokoue, and Doug Schales for technical discussion in various stages of our work. We also want to thank Larry Koved, Josyula Rao, and Doug Schales for management support.

References

[1] Dakshi Agrawal, James Giles, Kang won Lee, and Kaladhar Voruganti. Policy based validation of san configuration. In *Policy 2004: IEEE International Workshop on Policies for Distributed Systems and Netoworks*, 2004.

[2] Ryan C. Barnett. *Preventing Web Attacks With Apache*. Addison-Wesley, Upper Saddle River, NJ, USA, 2006.

[3] Tamar Eilam, Michael H. Kalantar, Alexander V. Konstantinou, and Giovanni Pacifici. Reducing the complexity of application deployment in large data centers. In *IM 2005: Ninth IFIP/IEEE International Symposium on Integrated Network Management*, pages 221–234, 2005.

[4] I. Horrocks and P. F. Patel-Schneider. A proposal for an owl rules language. In *Proceedings of the 13th Int'l World Wide Web Conf.* ACM, 2004.

[5] Boris Motik, Ulrike Sattler, and Rudi Studer. Query answering for owl-dl with rules. *Journal of Web Semantics*, 3(1):41–60, 2005.

[6] Mukund Raghavachari, Darrell Reimer, and Robert D. Johnson. The deployer's problem: Configuring application servers for performance and reliability. In *ICSE '03: International Conference on Software Engineering*, pages 484–489, 2003.

[7] Carsten Sinz, Amir Khosravizadeh, Wolfgang Kuchlin, and Viktor Mihajlovski. Verifying cim models of apache web-server configuration. In *QSIC '03: Third International Conference on Quality Software*, pages 290–297. IEEE Computer Society Press, 2003.