Semantic Web and Firewall Alignment

Simon N. Foley #, William M. Fitzgerald #,*

#*Department of Computer Science,*
University College Cork,
Ireland.
s.foley@cs.ucc.ie

**Telecommunications Software & Systems Group,*
Waterford Institute of Technology,
Ireland.
wfitzgerald@tssg.org

Abstract— **Secure Semantic Web applications, particularly those involving access control, are typically focused at the application-domain only, rather than taking a more holistic approach to also include the underlying infrastructure (for example, firewalls). As a result, infrastructure configurations may unintentionally hinder and prohibit the normal operation of the Semantic Web. This paper, discusses an approach involving Description Logic and the Semantic Web Rule Language to provide synergy and alignment between firewall configurations and semantic-aware application configurations.**

I. Introduction

The Semantic Web [1] builds on application domain ontologies in order to provide a framework for web-resource reasoning and interoperation. Semantic Web applications are typically modeled at the application-domain (knowledge) level [2], [3] and tend not to consider the underlying infrastructure: it is assumed that the underlying infrastructure is suitably configured to support the application and its web-resources. However, there are situations where the infrastructure configuration may work against the normal operation of the Semantic Web and it becomes necessary to consider some knowledge about the infrastructure and how it relates the application knowledge.

A practical reality of any system—regardless of the application it supports—is that a firewall policy is applied to incoming and outgoing traffic. A Semantic Web application may span many systems and, as a consequence, its proper operation is dependent on the firewall configuration at each system. An overly-restrictive configuration may prevent normal interaction of web-resources resulting in application failure. An overly-permissive configuration, while permitting normal operation of the application, may leave the system vulnerable to attack, for example, across open ports.

While the Semantic Web may provide applications with security services that are domain-knowledge aware [4], [5], it is argued that firewalls still have a role to play in securing the low-level infrastructure. Not only do firewalls protect services that do not provide built-in application-level security, it is considered best practice to rely on multiple layers of security, providing 'belt and braces'. In practice, deploying a firewall for a web-server or web-client is not simply about opening port 80 on the server for all traffic; one may wish to deny certain nodes (IP addresses, etc.), only accept HTTP traffic from some nodes, require other nodes to use HTTPS and also deal with HTTP traffic that is tunneled through proxies available on other ports. Furthermore, web-services do not necessarily communicate on port 80. In addition, firewall content sanitation (application layer) provides fine-grained access control that may cut across the host-based access controls; for example, certain content may be permitted (or denied) only to/from particular nodes.

Firewall configurations can be complex, run to many thousands of rules and are typically maintained on an ad-hoc basis [6], [7], [8]. New rules are added with little regard to existing rules and may result an overly-restrictive and/or overly-permissive configuration. The ideal firewall configuration is one that is *aligned* with the application supported by the system, that is, it permits valid application traffic, and, preferably, no more and no less.

The contribution of this paper is the alignment of firewall configurations with Semantic Web applications. An ontology is proposed for Linux Netfilter which is used to represent and reason about instances of firewall configurations. By bridging this ontology with the application ontology, it becomes possible to reason about how knowledge within the application may affect a firewall configuration, and vice-versa. For example, an airline service provider agent (web-service) may be willing to offer special discount services to preferred travel broker agents connecting through port 22 (`ssh`). Semantic agents can use this reasoning about the 'big picture' to infer new knowledge and better align firewall configurations to the application in an autonomic way.

The paper is organized as follows. Description Logic (*DL*) is used to represent the ontology; Section II provides an overview of *DL*; a more detailed discussion can be found in [9]. A *DL* representation of the Netfilter firewall ontology is given in Section III. Section V describes how the firewall ontology is bridged with the Semantic Web application E-Tourism case-study outlined in Section IV. How the resulting ontology can be reasoned about is discussed in Section VI.

II. Description Logic & Ontologies

An ontology is an explicit specification of a conceptualisation using an agreed vocabulary and provides a rich set of constructs to build a more meaningful level of knowledge. An important characteristic of the Semantic Web (as constrained by *DL*) is that the information contained in it is specified using a formal language in order to enable automated reasoning and the derivation of new knowledge from existing knowledge.

Description Logic (*DL*) is a family of logic-based formalisms that forms part of the W3C recommendation for the Semantic Web [9]. *DL* uses classes (concepts) to represent sets of individuals (instances) and properties (roles) to represent binary relations applied to individuals. For example, the *DL* assertion:

$$\begin{aligned} ServerNode \sqsubseteq & Node \sqcap \\ & \exists hasSemanticService.Service \sqcap \\ & \exists hasFirewall.Firewall \end{aligned}$$

specifies that a server node (class) hosts services (class) and has (property) a firewall (class) protecting them. Note that properties are conventionally prefixed by "*has*"; for instance, $hasSemanticService$, is the property over the individuals of the class $ServerNode$ (domain) that host a Semantic Web service (range).

The Semantic Web Rule Language, (*SWRL*), complements *DL* providing the ability to infer additional information in *DL* ontologies, but at the expense of decidability. *SWRL* rules are Horn-clause like rules written in terms of *DL* concepts, properties and individuals. A *SWRL* rule is composed of an antecedent part and a consequent part, both of which consist of positive conjunctions of atoms [10]. For example, the requirement: *servers hosting ssh based semantic services protected by a firewall require that firewall to open port 22* is expressed in SWRL as:

$$\begin{aligned} & ServerNode(?n) \wedge hasSemanticService(?n, s) \wedge \\ & hasPort(?s, ssh) \wedge hasFirewall(?n, ?f) \\ & \quad \rightarrow hasPortOpen(?f, ssh) \end{aligned}$$

III. Netfilter Ontology

Netfilter is a framework that enables packet filtering, network address translation (NAT) and packet mangling. As a firewall, it is both a stateful and stateless packet filter that is characterised by a sequence of firewall decisions against which all packets traversing the firewall are filtered. Each firewall decision takes the form of a series of conditions representing packet attributes that must be met in order for that decision to be applicable, with a consequent action for the matching packet (accept, drop, log and so forth).

Netfilter requires the specification of a *table* (`filter`, `NAT` or `mangle`), a *chain*, the accompanying decision *condition* details and an associated *target* outcome. A table is a set of chains and it defines the global context, while chains define the local context within a table. Our research focuses on the firewalling aspects of Netfilter and hence our current model only incorporates the `filter` table attributes. A chain is a set of firewall decisions and those decisions in a chain are applied to the context defined both by the chain itself and the particular table. By default there is no need to specify the `filter` table (using `-t` option) when defining firewall decisions. A Netfilter decision has the following components.

$$[Table][ChainType][FilterConditions][TargetDecision]$$

For example, a decision that states that HTTP traffic along the `FORWARD` chain outward bound from the trusted internal interface `eth0` is permitted is:

```
iptables -t filter -A FORWARD -o eth0
         -p tcp --dport 80 -j ACCEPT
```

A. Firewall Decision Components

The Netfilter `filter` table or overall firewall configuration policy is composed of a number of sub-governing local policies controlled by each of its in-built chains ($Chain$ class) to which various protective packet condition filters ($ConditionFilter$ class) and their respective verdict permissions ($Target$ class) are applied. Netfilter provides a mechanism of three separate firewall or filtering chains to police various kinds of network traffic (Figure 1). These chains filter traffic being routed to, from and beyond the firewall device itself [11]. Classes in *DL* represent concepts within

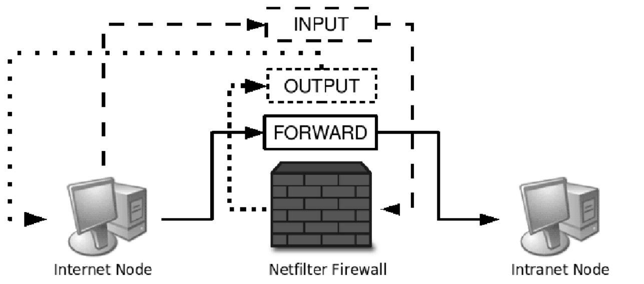

Fig. 1. Linux Netfilter Packet Traversal

the domain of interest and in our formal ontology model, various domain specific classes provide knowledge of the key features of Netfilter's filtering capabilities. Individual objects that belong to a class are referred to as instances of that class. We have developed a *DL*-based ontology for Netfilter. Figure 2 depicts a fragment of the class taxonomy for this model. The taxonomy provides the classes, subclasses and individuals that are inferred from the *DL* specification of the ontology. In the following subsections, we briefly illustrate some abstract examples to provide the reader with enough information as to how we codified the salient features of Netfilter in *DL*. Both disjoint and covering axioms are used extensively throughout our formal model. However, for reasons of space, we have opted to omit these axioms when presenting *DL* definitions in this paper. The reader should assume that all sibling classes defined in the paper are disjoint unless stated otherwise.

Chain Types & Chain Decision Policy. Each Netfilter firewall chain must be assigned a default decision policy of 'accept' or 'drop', such that, if packets that have been correlated

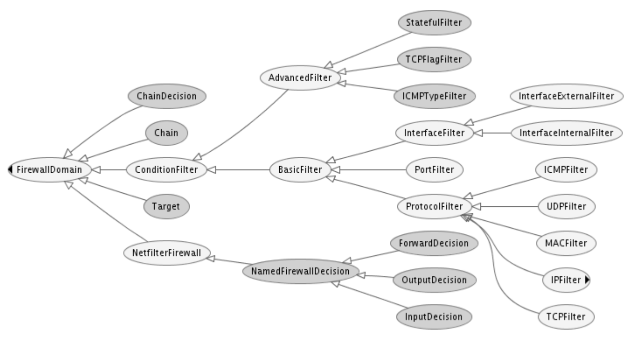

Fig. 2. Netfilter Class Taxonomy

against the complete set of firewall decisions within that chain have not met any of the filter conditions then the default decision policy is executed to decide the fate of those packets. The following is a detailed description of the $Chain$ class definition; it is also used to explain to the reader some of the constructors provided by *DL*.

The $Chain$ class is defined as a *complete* class ($\equiv$), subsummed by the $FirewallDomainConcept$ class, and as a domain concept with restrictions applied to the $hasChainDecision$ property that binds individuals of the $Chain$ class to individuals of another class within the firewall domain called $ChainDecision$. The $hasChainDecision$ property is a binary relation that is constrained with an existential ($\exists_{=1}$) restriction across that property with a cardinality value of 1. It states that there exists a single relationship between individuals of the $Chain$ class and individuals of the $ChainDecision$ class. There is also a closure axiom applied to the $hasChainDecision$ property via the universal ($\forall$) restriction. It states that individuals of the $Chain$ class can only ever have $hasChainDecision$ relationships to individuals of the $ChainDecision$ class using that property.

$$\begin{aligned} Chain \equiv\ & FirewallDomainConcept\ \sqcap \\ & \exists_{=1} hasChainDecision.ChainDecision\ \sqcap \\ & \forall hasChainDecision.ChainDecision\ \sqcap \\ & \{inputChain, outputChain, forwardChain\} \end{aligned}$$

The class $ChainDecision$ defines the vocabulary for a default policy decision across a firewall chain.

$$\begin{aligned} ChainDecision \equiv\ & FirewallDomainConcept\ \sqcap \\ & \{decisionDeny, decisionAccept\} \end{aligned}$$

Within Netfilter there are two approaches: the first is a *deny everything by default*, thereafter explicitly permitting selected packets; the second approach, is to *accept everything by default*, thereafter explicitly denying selected packets. The first approach is usually implemented as security best practice [12]. The $ChainDecision$ class is composed of two distinct individuals: $decisionDeny$ and $decisionAccept$ to represent the default policy decisions that can be applied to a chain. Chains have a default policy decision in order to govern how it processes unmatched traffic. Netfilter provides a mechanism for traffic to be inspected and analyzed, depending on various pre-firewall routing decisions (Figure 1). Thus, the $Chain$ class provides a finite/enumeration set of three individuals namely, $inputChain$, $outputChain$ and $forwardChain$. For example, the $inputChain$ individual (instance of class $Chain$) is a chain to which packet condition filters and corresponding target decisions can be applied to, in order to analyze traffic incoming to local services hosted on the actual firewall itself.

Filter Conditions. Netfilter firewall chains `INPUT`, `OUTPUT` and `FORWARD` govern routed traffic and can (and normally do) contain a set of packet filter conditions. Hence, chains act as containers for firewall decisions. The class $ConditionFilter$ represents the kinds of filters defined in our model:

$$ConditionFilter \sqsubseteq FirewallDomainConcept$$

Filtering criteria that can inspect individual packets can be further subdivided into two more specialised categories: basic filtering techniques (Class $BasicFilter$) and advanced filtering techniques (Class $AdvancedFilter$). Basic filtering has the ability to inspect each packet. It can be applied in various levels of granularity: to a particular interface or set of interfaces, to various protocols (MAC address, IP address or range of IP addresses, TCP, UDP and ICMP), or to a particular port or set of ports. Netfilter's advanced filtering techniques involve deep packet header inspection (TCP flags and ICMP types) and it can filter based on stateful connection state (previous packet streaming context).

$$BasicFilter, AdvancedFilter \sqsubseteq\ ConditionFilter$$

In the following two examples, we demonstrate of how condition filters can be modelled in more depth for port and state filtering.

Basic Filter by Port Type. Port condition filtering occurs at Layer 4 of the OSI stack. Ports (for example, individuals like $portSSH$) in our model are defined as a class of individuals that should be constrained to protocols TCP or UDP. Hence, we define a closure axiom that states: should a port be associated to a protocol along the $hasProtocol$ property relationship then it must only be to $TCPFilter$ or $UDPFilter$.

$$\begin{aligned} PortFilter \sqsubseteq\ & BasicFilter\ \sqcap \\ & \forall hasProtocol(TCPFilter \sqcup UDPFilter) \end{aligned}$$

Advanced Filter by Stateful Operands. The model defines three advanced filtering techniques (TCP flags, ICMP Types and Statefulness) that the Netfilter framework is capable of providing. The Netfilter firewall framework has stateful capabilities ($StateFilter$) that can filter at Layer 4 (TCP, UDP) and Layer 3 (ICMP) of the OSI model to filter packets based on the context of the traffic's current stream. Packets can be filtered based on the following attributes: `NEW` (equivalent to the TCP SYN request or initial UDP packet) , `ESTABLISHED` (equivalent to ongoing TCP ACK traffic after connection has been established), `RELATED` (ICMP error messages or FTP secondary connections etc) and `INVALID` operations. The stateful capabilities augment the stateless, static packet filter protection. State information is recorded when a TCP connection or UDP exchange is initiated and subsequent packets are examined not only based on stateless tuple decisions but also on the context of the ongoing connection [7]. Stateful filtering does not apply to MAC addresses or IP addresses so we define a complement along the universal restriction of $MACFilter$ or $IPFilter$.

$$\begin{aligned} StateFilter \equiv\ & AdvancedFilter\ \sqcap \\ & \neg(\forall hasProtocol.(MACFilter \sqcup IPFilter))\ \sqcap \\ & \{stateESTABLISHED, stateNEW, \\ & stateRELATED, stateINVALID\} \end{aligned}$$

Target Permissions. When a filter condition matches a packet traversing a particular chain, a firewall target option specifies the fate of that packet (for example, `DROP` or `ACCEPT` the packet). Netfilter provides a mechanism of packet authorisations (class individuals) represented by the $Target$ class in our model for this purpose. The $Target$ class is defined as a *complete* class that details the *necessary & sufficient* conditions of class membership. The reader is referred to [9] for an introduction to *DL's*.

$$\begin{aligned} Target \equiv\ & FirewallDomainConcept \sqcap \\ & \{returnTarget, rejectTarget, ulogTarget, \\ & logTarget, dropTarget, acceptTarget\} \end{aligned}$$

B. Firewall Decision Composition

In this section, we illustrate some examples of firewall decision constraints that are constructed in terms of the model vocabulary proposed above. Various kinds of firewall decisions can be defined as subclasses of the $NamedFirewallDecision$ class in our model. This class defines the *necessary & sufficient* conditions for the composition of a firewall decision. A firewall decision is composed of exactly one chain, one or more condition filters and a single permission target.

$$\begin{aligned} NamedFirewallDecision \equiv\ & NetfilterFirewall \sqcap \\ & \exists_{=1} hasChain.Chain \sqcap \\ & \exists_{\geq 1} hasCondition.Filter \sqcap \\ & \exists_{=1} hasTarget.Target \end{aligned}$$

To instantiate `FORWARD` decisions, for example, it is first necessary to define the membership constraints of class $ForwardDecision$ (a more specialised $NamedFirewallDecision$ class) whereby `FORWARD` decisions must have a relathionship to a specific individual of the Chain class called $forwardChain$:

$$\begin{aligned} ForwardDecision \equiv\ & NamedFirewallDecision \sqcap \\ & \in hasChain.forwardChain \end{aligned}$$

C. Firewall Configuration

The previous sections describe an ontology for firewall configuration. A firewall decision instance, defined in terms of specific ports, protocols, etc., is represented as an instance of the ontology defined in terms of class individuals. For example, the firewall decision

```
iptables -A FORWARD -i eth1
         -s 4.3.2.1 -d 1.2.3.4
         -p tcp --dport 22 -j ACCEPT
```

is represented by an individual fd within the ontology whereby $ForwardDecision(fd)$ holds; intuitively, we can think of fd as an individual of class $ForwardDecision$. This individual is inferred across the ontology whereby,

$$\begin{aligned} & hasChain(fd, \texttt{forwardChain}) \\ \sqcap\ & hasExternalInterface(fd, \texttt{eth1}) \\ \sqcap\ & hasSrcIP(fd, \texttt{ip4.3.2.1}) \\ \sqcap\ & hasDstIP(fd, \texttt{ip1.2.3.4}) \\ \sqcap\ & hasProtocol(fd, \texttt{tcp}) \\ \sqcap\ & hasDstPort(fd, \texttt{portSSH}) \\ \sqcap\ & hasTarget(fd, \texttt{acceptTarget}) \\ & \rightarrow ForwardDecision(fd) \end{aligned}$$

Atomic individuals are written in a `typewriter` font, while inferred individuals are given in an *italic* font. Note that the low-level facts of a firewall configuration are presented as individuals rather than classes on the basis that they are atomic and will not be further decomposed. Using instances (rather than subclasses) allows subsequent reasoning of collections of firewall decisions using *SWRL*, as outlined in [13].

IV. E-Tourism Application

A semantics-aware travel broker provider service (Figure 3) provides travel broker (intelligent) agent clients with an ability to query and purchase complete vacation/business packages based on user preferences. Typically, the travel broker client

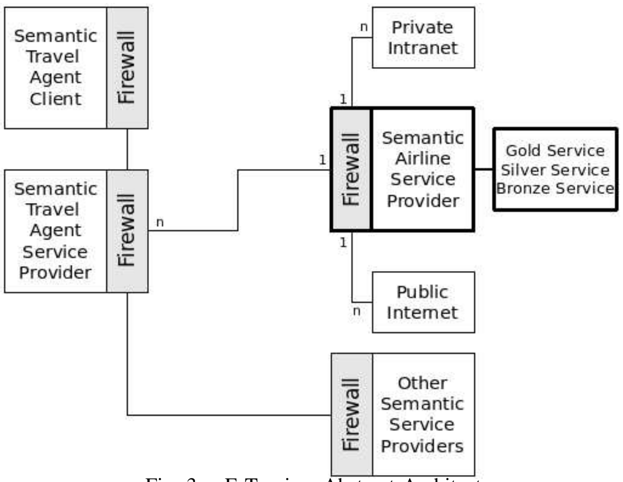
Semantic Travel Agent Client
Firewall
Semantic Travel Agent Service Provider
Firewall
n
Private Intranet
n
1
1
Firewall
Semantic Airline Service Provider
Gold Service
Silver Service
Bronze Service
1
n
Public Internet
Firewall
Other Semantic Service Providers

Fig. 3. E-Tourism Abstract Architecture

will interface with one or more travel broker service providers with which they have a subscription. The travel broker agent service provider interacts with various service providers (transport, accommodation, activities and so forth) on behalf of client requests. The $ETourismDomain$ ontology defines the knowledge within the E-Tourism application. For example, $ETourismDomain$ is sub-classed to describe the kinds of classes that are transport related:

$$TransportService \sqsubseteq ETourismDomain$$

A simple taxonomy of the overall E-Tourism ontology of this application is provided in Figure 4. In this example, we focus on the operation of the airline service that provides three different levels of service (gold, silver and bronze) based on client contract. These service types are modeled as individuals of the FlightService class:

$$\begin{aligned} FlightService \sqsubseteq\ & TransportService \sqcap \\ & \{\texttt{gold}, \texttt{silver}, \texttt{bronze}\} \end{aligned}$$

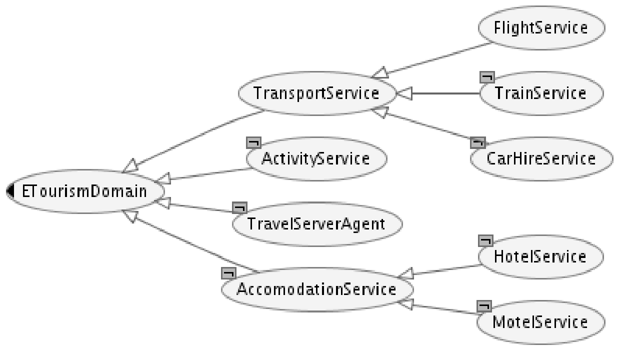

Fig. 4. E-Tourism Taxonomy

The gold service provides unlimited access to the airline booking system which is offered to the airline's sales branch itself along with premium business partners (for example, travel brokers). The silver grade service provides a more constrained service offering but at a cheaper rate, while the bronze grade is offered to all public users.

V. Bridging Application & Firewall

A. Mapping Services to Infrastructure

The E-Tourism application requires, among other things, network resources and associated business partners.

$$\begin{aligned} ETourismDomain \sqsubseteq & \exists hasNetRequirement.NetDomain \sqcap \\ & \forall hasPartner.ETourismDomain \end{aligned}$$

The kinds of network resources (class $NetDomain$) required by Semantic Web applications include (Figure 5): hosting nodes modeled as IP addresses (class $IPAddress$), communication protocols (class $Protocol$) and listening ports (class $Port$). Service individuals and their corresponding

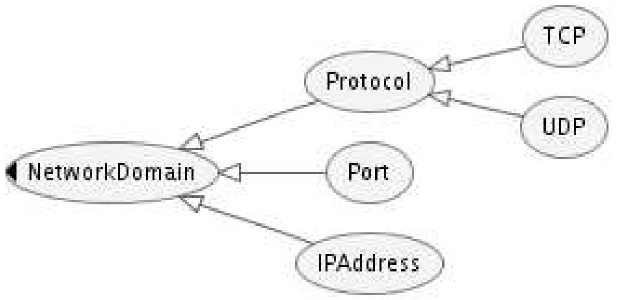

Fig. 5. Service Network Taxonomy

relationships to network resources can be instantiated from the ontology model. For example, a `gold` service individual may have a requirement to be hosted on a particular node (`ip1.2.3.4`) opening a `tcp` communication channel listening on port 22, (`ssh`), to provide secure access for premium business partners (for example, at IP address `4.3.2.1`).

$$\begin{aligned} & hasNetRequirement(\texttt{gold}, \texttt{ip1.2.3.4}) \\ \sqcap \; & hasNetRequirement(\texttt{gold}, \texttt{ssh}) \\ \sqcap \; & hasNetRequirement(\texttt{gold}, \texttt{tcp}) \\ \sqcap \; & hasPartner(\texttt{gold}, \texttt{ip4.3.2.1}) \\ & \rightarrow FlightService(\texttt{gold}) \end{aligned}$$

B. Relating Services to Firewall Decisions

A business domain ontology is used to relate service access requirements and firewall decisions. Figure 6 outlines the taxonomy of the $BusinessPolicyDomain$ ontology. Within

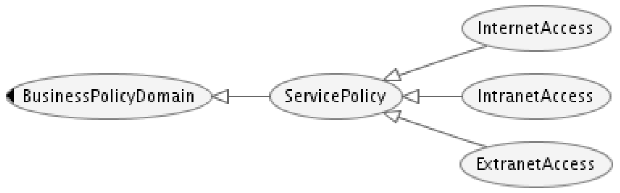

Fig. 6. Business Policy Taxonomy

this ontology, we define a class $ServicePolicy$ to represent how we map semantic services to firewall access control:

$$\begin{aligned} ServicePolicy \sqsubseteq \; & BusinessPolicyDomain \sqcap \\ & \exists hasManagedService.ETourismDomain \sqcap \\ & \exists hasFirewallDecision.NamedFirewallDomain \end{aligned}$$

This class is further subdivided into three disjoint classes: $IntranetAccess$, $ExtranetAccess$ and $InternetAccess$. An (inferred) individual gep (gold Extranet policy) of the $ExtranetAccess$ class that associates a `gold` service to an appropriate (forward) Netfilter firewall decision fd is described as follows:

$$\begin{aligned} & hasManagedService(gep, \texttt{gold}) \\ \sqcap \; & hasFirewallDecision(gep, fd) \\ & \rightarrow ExtranetAccess(gep) \end{aligned}$$

While the $NetDomain$ and $BusinessPolicyDomain$ ontologies bridge firewall knowledge and application knowledge, the concrete examples (bridge configuration of individuals fd, `gold` and gep) are manually constructed, presumably extracted from configuration data. The following section considers how a suitable firewall configuration can be automatically selected from an existing knowledge-base and based on an E-Tourism configuration.

VI. Synthesizing Firewall Configurations

Reasoning provides the ability to analyze a configuration in order to discover conflicts, for example, whether the firewall blocks a service or has conflicting decisions. It also allows the synthesis of a suitable firewall configuration, given application and bridging configuration. In this section, we consider firewall configuration synthesis. While DL-based reasoners support inferences about classes within an ontology, SWRL supports reasoning about relationships between individuals. In this paper, *SWRL* is used to provide synthesis of new knowledge about configuration and its integration back into the ontology.

Firewall synthesis relies on the existence of a knowledge base of candidate firewall decisions. These could, for example, represent considered best-practice for systems that host semantic web applications. The synthesis process selects the firewall decisions that are consistent with the configuration of the services (and bridge).

The following *SWRL* rule selects a firewall decision given a particular Extranet service access policy that currently manages an application service.

$$ExtranetAccess(?x) \wedge FlightService(?y)\wedge$$
$$ForwardDecision(?z) \wedge hasManagedService(?x, ?y)\wedge$$
$$hasNetRequirement(?y, ssh) \wedge hasDestIP(?z, ip1.2.3.4)\wedge$$
$$hasPort(?z, sshFilter) \wedge hasTarget(?z, acceptTarget)$$
$$\rightarrow hasFirewallDecision(?x, ?z)$$

The *SWRL* variable $?x$ represents a service policy that currently manages a service, ($?y$), but has no firewall decision applied that provides it with the required network access controls. Should the firewall decision be deemed appropriate, this new knowledge can be asserted back into the ontology by setting the $hasFirewallDecision$ property of the $ExtranentAccess$ individual $?x$ to reflect its relationship to the firewall decision $?z$.

SWRL also provides an SQL-like notation that can be used to query the *DL* knowledge base. For example, the query fragment:

$$\ldots$$
$$FlightService(?y) \wedge ForwardDecision(?z)\wedge$$
$$hasNetRequirement(?y, ssh) \wedge hasPort(?z, sshFilter)$$
$$\rightarrow query : select(?y, ?z)$$

returns a list of tuples of the form $y \mapsto z$, where y is a application service and z the firewall decision that provisions network access control. For example, it would be expected to return the tuple gold $\mapsto fd$. In practice, the above rule should contain additional filtering information to consider the protocol, interface, etc.

SWRL can be used to synthesize a variety of configuration scenarios. For example, *SWRL* rules can be provided, which given a specific firewall configuration can determine which services of an application can be reliably supported. This is useful when a network topology requires specific controls for systems in specific locations.

VII. Discussion & Conclusion

This paper, outlined a novel approach to using a *DL* constrained ontology to represent: Netfilter, semantic-aware E-Tourism and business policy configurations. The Netfilter ontology reflects the semantic knowledge that a firewall administrator should 'keep in their head' when writing and/or updating firewall decisions based on the semantic application it offers protection to. We provided an E-Tourism case study to illustrate how Semantic Web applications cannot operate in a standalone manner when incorporating security features because there are situations where the infrastructure configuration (firewalls, Web proxies and so forth) may work against the normal operation of the semantic application. The business policy ontology, through the use of *SWRL* inferencing, facilitates access control alignment between a list of appropriate Netfilter firewall decisions and various services grades within the E-Tourism application.

The provision of reasoning, in particular within the context of the open world assumption, provides our ontology with flexibility and extendability of incorporating new knowledge. For example, our firewall ontology not only provides infrastructure knowledge that supports alignment with semantic-aware applications but it also facilitates subsequent reasoning of firewall configuration conflict anomalies as outlined in [13].

The ontologies have been implemented in OWL-DL, a language subset of OWL, a W3C standard that includes *DL* reasoning semantics [14]. Protégé [15], with an incorporated OWL-DL plug-in, provide a frame-like GUI structure to create the ontologies. Pellet [16], an open world *DL* compliant reasoner, provides consistency checking and automatic polyhierarchy structuring accross the ontology. The *SWRL* [10], [17] Protégé plug-in, *SWRLTab*, interfaces with the Jess closed world inference engine to reason over relationship properties (for example, $hasFirewallDecision$) between individuals of our ontology.

We are currently developing a prototype autonomic architecture that is based on this ontology and reasoning framework. Firewall semantic agents are responsible for managing the configuration of a firewalls. These agents negotiate firewall settings that are constrained by the current knowledge base, which is in turn controlled by the firewall agents and other application agents, managing for example, the business rules. The knowledge base is controlled by adding or deleting facts based on new knowledge and inferences by the agents. For example, a business agent informs a firewall agent of a new customer whereby the firewall agent must reconfigure (new facts) to enable access.

References

[1] H. P. Alesso and C. F. Smith, *Thinking on the Web: Berners-Lee, Gdel and Turing*. Wiley-Interscience, September 2006.

[2] J. Cardoso, *Semantic Web Services: Theory, Tools and Applications*. IGI Global, March 2007.

[3] J. R. S. A. F. Salam, *Semantic Web Technologies and E-Business: Toward the Integrated Virtual Organization and Business Process Automation*. IGI Global, January 2007.

[4] S. K. Nair, B. Crispo, and A. S. Tanenbaum, "Zodac-Towards a Secure Policy Enforcement Architecture," *Fourteenth International Workshop on Security Protocols, March 27-29*, 2006.

[5] N. Kodali, C. Farkas, and D. Wijesekera, "Enforcing semantics-aware security in multimedia surveillance," *Journal of Data Semantics*, 2004.

[6] L. Gheorghe, *Designing and Implementing Linux Firewalls with QoS using netfilter, iproute2, NAT and l7-filter*. PACKT Publishing, October 2006.

[7] S. Suehring and R. L. Ziegler, *Linux Firewalls: Third Edition*. Novell Publishing, 2006.

[8] E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan, "Conflict Classification and Analysis of Distributed Firewall Policies," *In IEEE Journal on Selected Areas in Communications, Volume 1-1*, 2005.

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

[10] M. J. O'Connor, H. Knublauch, S. W. Tu, B. Grossof, M. Dean, W. E. Grosso, and M. A. Musen., "Supporting Rule System Interoperability on the Semantic Web with SWRL," *Fourth International Semantic Web Conference (ISWC2005)*.

[11] R. Russell, "Linux 2.4 Packet Filtering HOWTO," *www.netfilter.org*, January 2002.

[12] J. Wack, K. Cutler, and J. Pole, "Guidelines on Firewalls and Firewall Policy: Recommendations of the National Institute of Standards and Technology," *NIST, Special Publication 800-41*, 2002.

[13] W. M. Fitzgerald, S. N. Foley, and M. O'Foghlu, "Confident Firewall Policy Configuration Management using Description Logic," *Twelfth Nordic Workshop on Secure IT Systems, short presentations (unpublished), Reykjavik, Iceland, October 11-12*, 2007.

[14] M. K. Smith, C. Welty, and D. L. McGuinness, "OWL Web Ontology Language Guide," *W3C Recommendation, Technical Report.*

[15] J. Gennari, M. A. Musen, R. W. Fergerson, W. E. Grosso, M. Crubezy, H. Eriksson, N. F. Noy, and S. W. Tu., "The Evolution of Protege: An Environment for Knowledge-Based Systems Development," in *Proceedings of International Journal of Human-Computer Studies, Volume 58 , Issue 1*, 2003.

[16] B. Parsia and E. Sirin, "Pellet: An OWL DL Reasoner," *3rd International Semantic Web Conference, ISWC*, 2004.

[17] E. J. Friedman-Hil, "Jess the Rule Engine for the Java Platform," *Version 7.0p1*, 2006.

APPENDIX

APPENDIX: DESCRIPTION LOGIC

DL belongs to a family of logic that represents a decidable portion of first-order logic. The logic is characterised by a set of constructors (Table I) that allows the construction of complex concepts and roles from atomic concepts or roles. Classes (concepts) represent sets of individuals and properties (roles) represent binary relations applied to individuals. Tables I and II illustrate parts of *DL* that are used in this paper. The reader is referred to [9] for further information.

TABLE I
DL CONSTRUCTORS

Constructor	DL Syntax	Example
Intersection Of	$C_1 \sqcap \ldots \sqcap C_n$	$Protocol \sqcap Port$
Union Of	$C_1 \sqcup \ldots \sqcup C_n$	$PrivilegedPort \sqcup UnprivilegedPort$
Complement Of	$\neg C$	$\neg PrivilegedPort$
Universal Quantifier	$\forall P.C$	$\forall hasPort.PrivilegedPort$
Existential Quantifier	$\exists P.C$	$\exists hasPort.PrivilegedPort$
Max Cardinality	$\leq_n P$	$\leq_{16} hasPort$
Min Cardinality	$\geq_n P$	$\geq_1 hasPort$
Exact Cardinality	$=_n P$	$=_1 hasPort$

TABLE II
DL AXIOMS

Axiom	DL Syntax	Example
SubClass Of	$C_1 \sqsubseteq C_2$	$TCPParam \sqsubseteq TCPFlag \sqcap Port$
Equivalent Class	$C_1 \equiv C_2$	$Port \equiv PrivilegedPort \sqcup UnPrivilegedPort$
Disjoint With	$C_1 \sqsubseteq \neg C_2$	$PrivilegedPort \sqsubseteq \neg UnPrivilegedPort$
Sub Property OF	$P_1 \sqsubseteq P_2$	$hasSrcPort \sqsubseteq hasPort$

a) Classes & Properties: Classes are interpreted as sets of individuals and can be organised into a superclass-subclass hierarchy. For example, $Protocol$ is a class that represents the set of all individual protocols and its subclasses include TCP and UDP classes. Subsumption represents the superclass-subclass hierarchy, for example, $TCP \sqsubseteq Protocol$ indicates that TCP is a subclass of $Protocol$.

Properties are used to construct binary relationships between classes. They are used when making statements about classes. For example, the following defines the concept of "ports are either privileged or unprivileged but not both":

$$\begin{aligned} Port &\equiv PrivilegedPort \sqcup UnPrivilegedPort \\ PrivilegedPort &\sqsubseteq \neg UnPrivilegedPort \end{aligned}$$

Like classes, subproperties specialise their superproperties. For example, the property $hasSrcPort$ specialises the property $hasPort$. This states that if two classes are related by the $hasSrcPort$ property then an attributed source port is a more specific relationship than the general case of having a port relationship. Properties in our model are prefixed with the word "*has*". *DL* has the following properties: functional, inverse functional, transitive and symmetric.

b) Property Restrictions: Object property restrictions are used to create constraints on individuals that belong to a particular class. Restrictions fall into three categories: *Quantifier, Cardinality* and *hasValue* restrictions. An existential ($\exists$) restriction requires *at least one* relationship for a given property to an individual that is a member of a specific class. A universal ($\forall$) restriction mandates that the *only* relationships for the given property that can exist must be to individuals that are members of the specified class. A property restriction effectively describes an anonymous or unnamed class that contains all the individuals that satisfy the restriction. When restrictions are used to describe classes they specify anonymous superclasses of the class being described.

c) Partial & Complete Classes: A *partial* class definition is specified with *necessary conditions* and is of the form $Class \sqsubseteq SuperClass \sqcap PropertyConditions$. This states that if an individual is a member of the defined class it must satisfy the conditions that it characterises. However, it cannot be said that any (random) individual that satisfies these conditions must be a member of this class. When a class is defined with *necessary and sufficient conditions* ($\equiv$), like partial classes,then if an individual is a member of the class then it must then satisfy those conditions. However, with the *sufficient condition* included, then any (random) individual that satisfies these conditions must be a member of this class. Classes that have at least one set of *necessary and sufficient conditions* are known as *complete* classes.

d) Open World Assumption: The Closed World Assumption (*CWA*) and the Open World Assumption (*OWA*) represent two different approaches of how to evaluate implicit knowledge in a knowledge base [9]. The CWA approach makes the presumption that what is not currently known to be true in a knowledge base is false, hence the interpretation of negation as failure. However, OWA assumes that its knowledge of the world is incomplete. If something cannot be proved to be true in the known knowledge base, then it does not automatically become false. A simple example of OWA would be to assume that we know that a firewall decision has been applied to a range of privileged ports and from this information using the OWA approach one can not conclude that a firewall decision also has or has not some unprivileged ports assigned. Hence, if a class is to be confined to certain constraints, it must be explicitly stated that other unwanted constraints do not exist. In *DL*, OWA characteristics can be contained by stating exactly the components of a class using both *disjointness* and *covering axioms*.