University of Southampton
Faculty of Engineering and Applied Science
Department of Electronics and Computer Science

A project report submitted for the award of
MEng Software Engineering

Project supervisor: Dr. Ulrich Ultes-Nitsche
Second examiner: Dr. Les Carr

Network Management using Abstracted XML Specifications

by Simon Chudley

May 7th, 2002

Abstract

Establishing and managing network infrastructures between distributed nodes can be time consuming and risk prone. Such a statement is often re-enforced by the incorporation of new technologies, the merging of existing systems and the rotation of sub-systems. However, the functional requirements of these enterprises remain constant, even though implementation specifics may differ greatly.

The driving force behind this project is to investigate the addition of an extra level of abstraction between the operation and implementation of such nodes within distributed environments. A processing architecture is to be established to enable dynamic configurations of system elements, including support for function calls and external library references within service specifications.

This report details the chosen architecture and the features it provides, along with an overview of the system as a whole. Possible new directions in node configuration are addressed, supported by implemented examples.

In conclusion, this project confirms that such an abstracted approach to system configuration can indeed be achieved, including the ability to dynamically expand to cover new services and implementations. In addition, the integration of simulation modules directly into this project would provide a powerful network validation tool; such research is currently underway.

Contents

Abstract 2

1. Introduction 5
   1.1 Project Goals 5
   1.2 Detailed Specification 5

2. Background Research 6
   2.1 Researched Approaches 7

3. Design 8
   3.1 Overall Structure of System 8
   3.2 Process Flow Control 9
   3.3 Variable Mapping Hierarchy 12
   3.4 Object Structure 12
   3.5 Design Changes 12

4. Implementation of Design 14
   4.1 System Implementation Overview 14
   4.2 Node and Service Description 14
   4.3 Variable Mapping Assignments 16
   4.4 Service Constructs 17
   4.5 External Service Constructs 17
   4.6 Function Calls 19
      4.6.1 Input to Function Calls 19
      4.6.2 Example Use of Function Calls 21

5. Service Translation Process 22
   5.1 Specifying the Translation 22
   5.2 Translation Restriction Calculation 23
      5.2.1 Overview of Restrictions 23
      5.2.2 Restriction Rule Specification 23
      5.2.3 Pre-processor Output 27
      5.2.4 Complications During Filter Process 27
      5.2.5 Example Translation Restriction 28
   5.3 The Translation Process 29

6. Program Testing 32
   6.1 Introduction 32
   6.2 XML Comparison Tool 32
   6.3 Implemented Network Testing 32
      6.3.1 The Testing Process 33
      6.3.2 Results 34

7. Summary 35
   7.1 Conclusions 35
   7.2 Future Work 36

8. References 37

9. Appendix 1 – User Manual 39
9.1 XML Network Element Descriptions 39
9.1.1 Creating Descriptions 39
9.1.2 Variable Definitions 40
9.1.3 Defining External Service Constructs 40
9.1.4 Specifying Functions 41
9.2 Accessing the Package 42
9.2.1 Loading and Manipulating Package Elements 42
9.2.2 Generating XML Output 43
9.2.3 Performing Translations and Restriction Analysis 44
9.3 Defining Translations 44
9.3.1 Translations Specification 44
9.3.2 Restriction Analysis Rules 45
9.4 Compilation and Execution 46
9.4.1 Tests 46

10. Appendix 2 – Test Script and Sample Files 47
10.1 Test Scripts 47
10.2 Sample Node Configuration with Firewall and DNS Services 48

11. Appendix 3 – Published Material 54

1. Introduction

1.1 Project Goals

- Provide a method of allowing users to gather and display structural details about nodes and links on their local area network.

- Allow the specification of network services and configuration for each required node on the network. The actual behaviour will be generalised so that it reflects the various different implementations and versions of that service available on different platforms.

- Store the state of the network layout and the service descriptions of each node to a series of XML files. These give an abstracted description of the system as a whole, allowing it to be restored and analysed at a later date.

- The XML node descriptions can then be parsed and converted into sets of configuration files specific to the required operating system and service types.

The main advantage of having the XML abstracted representation is that it separates the low level information, such as operating system and service version, from the higher-level behaviour of nodes. This implies that nodes can be interchanged, and once the XML configuration of that node has been re-applied it will maintain the same behaviour.

1.2 Detailed Specification

From the goals outlined in section 1.1, there are many smaller milestones that must be achieved in order to develop a suitable solution.

The first task is to establish an abstracted description of a firewall, taking into account the overall desired behaviour to create XML [18] specifications. In theory there must be defined procedures to translate between the generated description and the desired implementation level service applications.

The package must be able to parse the abstracted descriptions and store them in a modifiable object structure. An API should be provided to allow manipulation of internal services, and generate XML descriptions maintaining system state.

Within the XML configuration, added support to enable dynamic specification of services would be a great asset. Such features include being able to import common functionality from a pre-defined library, and inclusion of variable assignments within XML elements

With the desired descriptions, the user should be able to translate or compile them into files that can be directly applied to their chosen service implementation. In addition, a method to report back to the user any restrictions or problems that occurred during the translation process is envisioned.

2. Background Research

Modern networks consist of a variety of nodes chosen to provide network and user level services to peers. There are various solutions to publish these required services, often based on different operating systems due to performance and security issues. The system level configuration of these services is usually complex, and requires specific knowledge of both the service specification and the implementation being used. Network management solutions typically consist of four layers [17]. This project aims to address the top two layers, enabling service descriptions to be created and translated to implementation level. It will not, however, address the actual distribution and application of these configurations across the network itself.

Stevenson [17] addresses the Open Systems Interconnect (OSI) model for Management Function Areas (MFAs). These MFAs represent areas in the management system that we are focusing on. The OSI model presents the "FCAPS” acronym, representing the key elements of an integrated management system.

- Fault Management
- Configuration Management
- Accounting
- Performance Management
- Security Management

This project views the configuration management stages as the abstracted specification of services, analysis of configurations and finally the translation to implementation level files. The overall aim is that additional features such as security and performance management will be integrated into the simulation process, but these are outside the scope of this project.

Taking the firewall service as an example, configuration differs between implementations, even though the required behaviour is the same. An example of which is IPFW [10] and IPFilter [3]. The following two configuration scripts both give the same functionality, to prevent packets from private networks coming in via their public network interface, `tun0`, as described in RFC 1918 [15].

Example IPFilter configuration:

```
block in quick on tun0 from 192.168.0.0/16 to any
block in quick on tun0 from 172.16.0.0/12 to any
block in quick on tun0 from 10.0.0.0/8 to any
pass in all
```

Example IPFW configuration:

```
add deny all from 192.168.0.0/16 to any in recv tun0
add deny all from 172.16.0.0/12 to any in recv tun0
add deny all from 10.0.0.0/8 to any in recv tun0
add pass all from any to any
```

These two configurations differ purely on format; hence they could be automatically generated from an abstracted description. However, this may not always be the case as rule descriptions in IPFW and IPFilter differ also in functionality, which will need to be

addressed within this project. Some example firewall configurations to emphasise this point are available for IPFW [13] and IPFilter [3]. In addition, comparisons of various packet filter firewall configurations can be seen on the Firewall Filter Language Compiler page [14]. This project wishes to investigate creating such a conversion method that would be applicable to a wide range of network services.

Initial research was aimed towards developing a complete conversion for the firewall service from abstracted description through to various end level configurations. An analysis was carried out on IPFW [8] and IPFilter, two common firewall implementations used within the FreeBSD [5] operating system. Reference was also made to the specifications of the Internet Protocol v4 [6] and Transport Control Protocol [7], so that the various options supplied by each firewall implementation could be understood. Using these, a generic XML description of a firewall was generated, acting as the storage element of this service. Another useful source of information was the firewall section in the FreeBSD handbook [12].

An architecture was built to enable the XML service descriptions to be loaded, modified and saved back to XML. To enforce structural standards, XML schemas were used [16]. Schemas were built to describe every major element of the system. The translation process was integrated into the system using XML style sheets [19], and hence allowed conversions to be defined external to the code base.

2.1 Researched Approaches

The Filter Compiler Language project [14] has successfully implemented a conversion process from a set description to various firewall configurations. The approach taken uses the C pre-processor to execute the conversions. This allows the abstracted description to be generated using `if` statements, and variable mappings to be made. An example of this is as follows.

```
if ( in ) then {
      set protocol tcp
      if ( from host BAR and opening ) then {
            block .
      }
      if ( from foo and to host bar ) then {
            log body and block .
      }
      if ( to port 2049 ) then {
            log and block .
      }
      pass .
}
```

However, on closer examination, this approach would not be applicable to a wide range of services, as it relies on the fact that the configurations are rule based. Also, as one aim of this project is to introduce the possibility of simulation, such a service description would not be rich enough.

Another product was identified called the Firewall Builder [9]. This uses a similar approach to that proposed by this project, where nodes and other network elements are described using XML descriptions, and a GUI editor is used to configure the firewall. Such an approach includes all processing functionality within the GUI editor, and the

XML files purely used as storage for the system. This project aims to introduce more advanced dynamic configuration operations into the XML definitions themselves.

In addition, this method is still specific to a single service, but will be useful as a comparison during the development of the firewall service. This project also aims to create an architecture using Java and XML, to ensure that it remains as system independent as possible.

One example of a DNS [11] configuration tool is the DNSTools [4] software. This product enables DNS servers to be remotely configured via a web interface. However, this still does not feature a centralised abstracted description of the service, and would not be scalable to cover other services.

3. Design

3.1 Overall Structure of System

One primary aim of this project was to provide an architecture that could generically be applied over a wide service base. It should be able to transparently support the addition of new translations from the abstracted description to implementation level configurations, and expand easily to cover new services.

The most versatile approach to use is a fully object oriented design. A major aspect of the entities' responsibilities is to parse and generate the XML to represent themselves, being able to expand variable mappings and function calls for example. The idea is to maintain maximum encapsulation within the service objects, so that new services can be added by the alteration of the minimal number of external objects.

The diagram on the following page, figure 1, shows the overall system structure and processes that occur. Initially, the user's task is to generate descriptions of their network (stage 1), the nodes within it and the services they intend to configure. At this stage all descriptions use an abstracted syntax, implying that their specification is not tied to a single service implementation or architecture.

At stage 2 within the diagram, the system will maintain an XML abstracted description of all elements. Such syntax is used to maintain state; hence these descriptions can be stored and recalled from disk. This stage forms the central repository of the entire system, with other stages referring to it when performing further processing. XML service descriptions will contain unresolved external references and function calls, as these are evaluated prior to translation/simulation.

With the desired behaviour of the nodes and services encapsulated within abstracted XML descriptions, simulation can be performed to validate specifications prior to implementation. Simulation is outside the scope of this project, but research is in place to develop such a tool. It is envisioned that feedback criteria such as performance and security improvements will be fed back to the system editor, allowing the user to incorporate these new rules into their overall specifications.

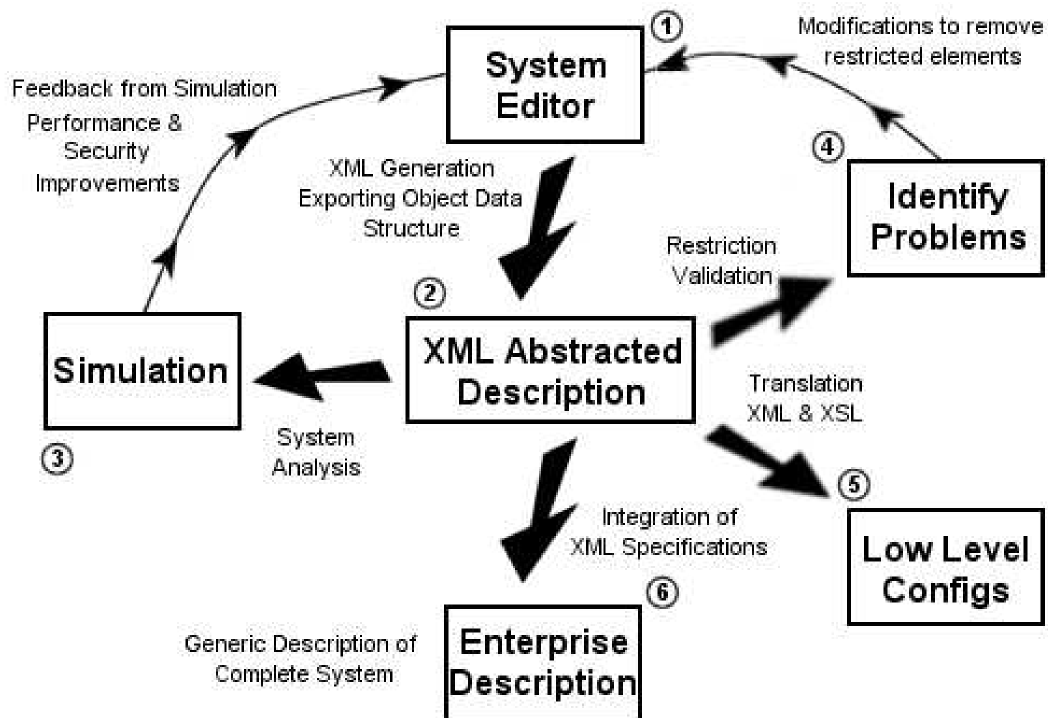

Figure 1 – Overview of System Structure

Stages 4 and 5 represent the process of creating implementation level configuration files from the descriptions now stored in stage 2, hence outputting files that can be directly loaded into service applications. Initially a level of restriction validation is performed. This takes a set of restriction filter rules, describing possible problems in the translation process, and reports to the user the location of such restrictions, and the XML elements they affect. Such a filter is required as different service solutions provide different functionality, and the user should be notified of any undesired changes in behaviour. As with simulation, the system editor will be used to make design changes in response to located restrictions.

Translation, in stage 5, aims to convert these descriptions to implementation level configuration files. At this stage, the user will be aware of the expected behaviour of the to-be implemented services, and have knowledge of any included restrictions. Finally, stage 6 represents the enterprise description, implying that the combination of all element specifications define the overall abstracted behaviour of the entire system.

3.2 Process Flow Control

Figure 2 shows the flow of processing and data throughout the system, from initial parsing through to restriction validation and service translation. The processing element *Generate Full XML* represents a key aspect of the system. This is performed prior to data being fed into the simulator, translator or restriction filter, hence represents the initial processing undertaken on all translations out of stage 2 in figure 1.

At stage 2 within figure 1, the node and service descriptions held will be in partial state. This implies that they will contain references to external constructs, function calls and

variable mappings. This is ideal for storage purposes; such descriptions should remain dynamic and evolve with the rest of the system with minimal user involvement. However they are not suitable for translation into configuration files, and also cannot be exported to components such as the simulator; without the functions and construct library they have limited meaning.

Three sub-processes spawn from the *Generate Full XML* process shown in figure 2. These pass sections of partial XML into the sub-processes with the intent of generating their full equivalent, which is then substituted back into the object tree. However, it is quite possible that recursive resolution can occur, causing further sub-processes to be evaluated. For example, a function call may take as input some external elements from a library, which will be transparently resolved by the package.

To implement such transparency of resolution from partial to full XML, the package architecture should provide operations to aid creators of new services for the system. Service constructs, described in section 4.4, are the building blocks of service descriptions. They are also the lowest level at which variable mappings can be declared, and are the elements that contain references to external constructs and function calls. Due to this, a vast amount of the functionality required to deal with such processing elements is hidden from the service developer; they can simply extend the behaviour of generic objects.

Every element within the package will be able to parse and generate its own XML definition; ensuring encapsulation of functionality is maintained. When elements are requested to generate XML, a flag is passed specifying whether full XML is required, allowing for finer control of processing.

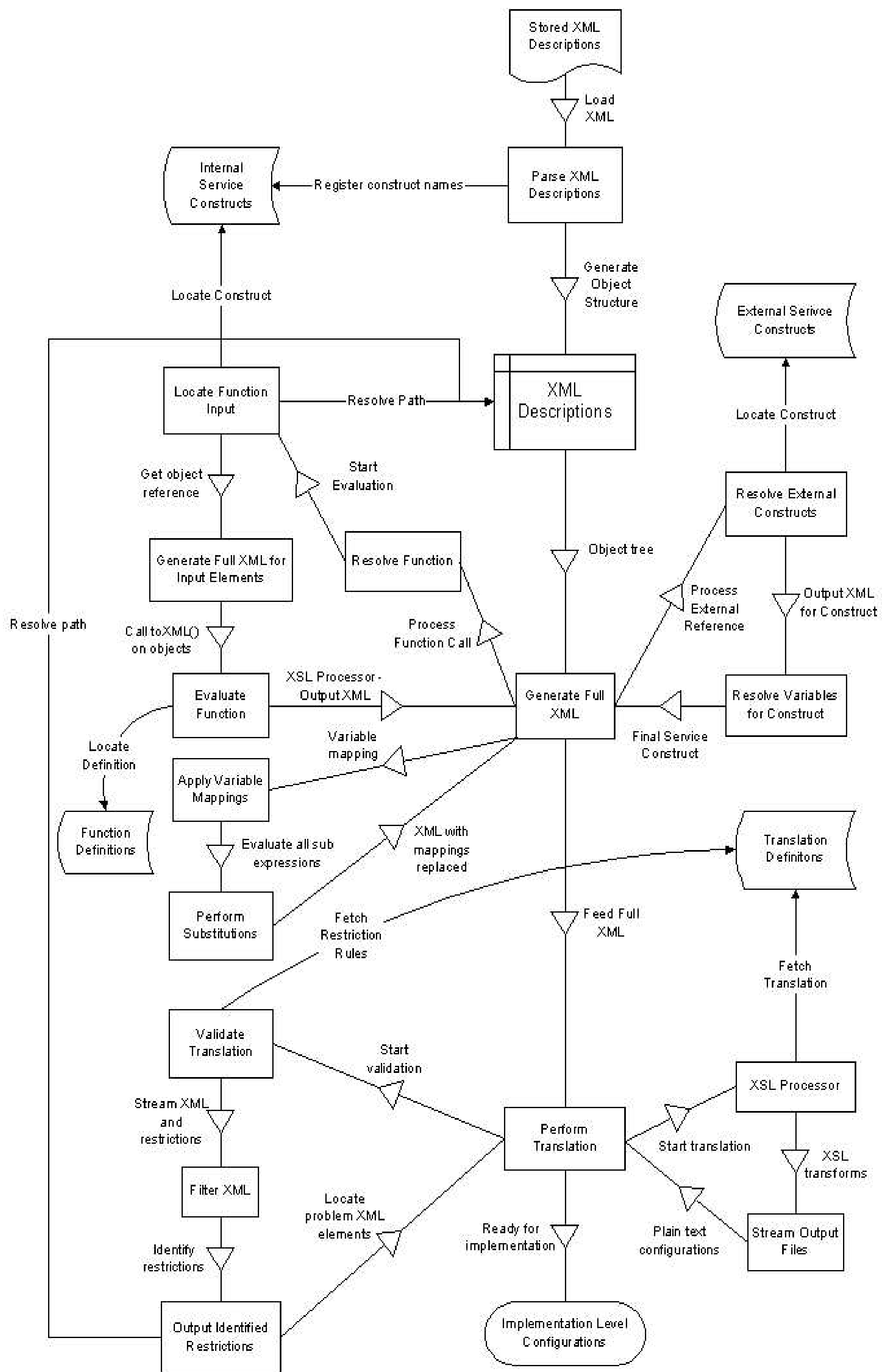

Figure 2 – System Processes

3.3 Variable Mapping Hierarchy

Variable mappings, explained in detail in section 4.3, are built in at a low level within the package. Each element can contain a level of variable mappings (down to service construct), with each layer overriding that of the previous (maintaining lexical scoping). When full XML is required, a hash table is passed to each `toXML()` call, representing the variable mappings defined up to that level in the hierarchy. Each element will integrate its own variable mappings prior to generating XML, allowing for complex arithmetic expressions and substitutions to be made.

3.4 Object Structure

Figure 3 on the following page is a class diagram showing the conceptual object structure within the main package. It can clearly be seen that the `NetworkElement` class acts as a central parent, providing basic parsing and XML generation functionality. It also defines how variable mappings should be resolved, recursing through the object structure incorporating variables at each level.

The `ServiceConstruct` class defines generic behaviour that is extended by `FirewallConstruct` and `DNSConstruct`. As mentioned in section 3.2, `ServiceConstruct` can also contain references to external constructs or functions, represented by `CallFunction`. All function processing is dealt with at this level; hence extending services need not be concerned with this.

3.5 Design Changes

Throughout the project there were many design changes, specifically to improve performance and add new features. Initially the aim was to use an extension of service objects to process the translation from abstracted to implementation level. However, this was soon identified as being inappropriate, as such functionality could be achieved using XSL and XSLT. In addition this enables users to write new service translations with no knowledge of the Java package running underneath.

Function calls were not initially planned to be included within the package, but were added when such a need was located. During the development of the DNS service, specifically writing configurations using forward and reverse start of authorities (mapping name to IP and IP to name respectively), a configuration dependency was identified. If a new host was added to the forward mappings section, a similar, yet slightly different format reverse entry was required. It was seen that such reverse mappings could be automatically generated, using the input forward mapping, hence leading to the introduction of function processing. See section 4.6 for further explanation.

During the development of the service translation from abstracted firewalls to IPFilter, mismatches were identified between elements within the two descriptions, leading to final configurations that contained undesired behaviour. To overcome this, a method of reporting restrictions to the user was introduced, and is explained further in section 5.2.

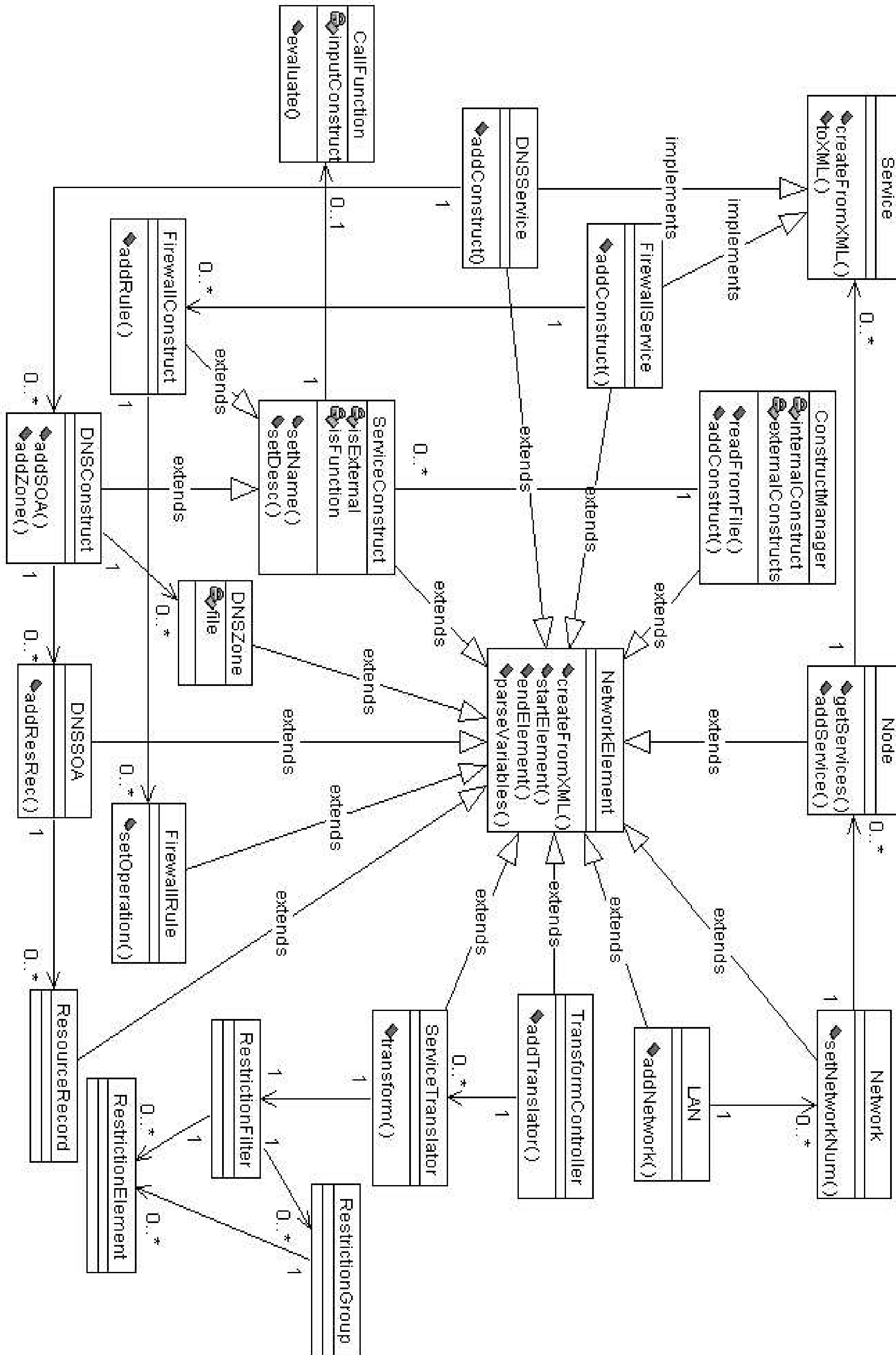

Figure 3 – Conceptual class diagram

4. Implementation of Design

4.1 System Implementation Overview

The primary deliverable of this project is a software package capable of allowing complex configurations of network nodes to be established. With the desired descriptions of each node in the network, the package will generate a set of XML files specifying the abstracted behaviour of each node. In addition to this, the system will then be able to perform a level of compilation or translation with the intent of generating implementation level configuration files.

Each node within the system has a single XML file describing the node settings and services running on it. In addition to the syntax for describing services, the package supports many additional configuration features such as variable mappings, service constructs and functions. These are described within the following sections, and allow for dynamic service descriptions to be created. The concept of networks and LANs are represented within the package as groupings of sub-elements. Variable mappings can be defined at both of these levels, such as specifying network number and subnet mask values.

When the desired node description has been generated, the next stage is translation. This aims to generate implementation level configuration files for the target service, which can then be directly loaded into the chosen application. This is a two-stage process, involving restriction calculations and then the actual translation process itself. During restriction calculation we aim to analyse the input descriptions, to determine if there will be any problems in the translation from abstracted to implementation level configuration files.

4.2 Node and Service Description

The central feature of this system is to be able to load, manipulate and save node and service descriptions. It was decided that each service within the system was to be validated using an XML Schema. Within such descriptions, all elements will be defined formally, stating the possible values that they could take and other relational constraints. When the XML file for a node is parsed, each service will be validated against their respective XML Schema to ensure that they can be processed correctly.

Packet filtering firewalls, such as IPFW, use sets of rules to determine how the traffic will be filtered. The first stage in development was to get the key elements that make up a single rule declaration. There are a finite amount of values for each element, hence the abstracted XML description of a firewall rule was straightforward to construct. Following is the definition of a firewall rule within IPFW:

```
index prob match-prob action log logamount number proto from src
                  to dst interface options
```

The words in bold indicate keywords in the rule, the others represent options that can be specified. Each element has a finite set of values, such as the source and destination descriptions. For these, they can be made up of an IP (Internet Protocol) address, an address mask, and a sequence of port numbers. There could also be other specific keywords in the address declarations that have special meanings, such as `me` and `any`.

Following is part of the XML Schema (XSD file) for a firewall, showing how the source of a packet should be defined

```
<xsd:element name="src">

   <xsd:annotation>
      <xsd:documentation>Source of the packet</xsd:documentation>
   </xsd:annotation>

   <xsd:complexType>
      <xsd:sequence>

         <!-- Definition of single port numbers to allow through -->
         <xsd:element ref="port-num" minOccurs="0"
                      maxOccurs="unbounded"/>

         <!-- Definition of a range of ports to match -->
         <xsd:element ref="inc-port-range" minOccurs="0"
                      maxOccurs="unbounded"/>
      </xsd:sequence>

      <!— Type of source definition -->
      <xsd:attribute name="type" use="required">
         <xsd:simpleType>
            <xsd:restriction base="xsd:NMTOKEN">
               <xsd:enumeration value="any"/>  <!-- Any source -->
               <xsd:enumeration value="ip"/>   <!-- Match the IP -->
            </xsd:restriction>
         </xsd:simpleType>
      </xsd:attribute>

      <!-- The address, in dotted form -->
      <xsd:attribute name="address" type="xsd:string"/>

      <!-- The mask, in dotted form -->
      <xsd:attribute name="mask" type="xsd:string"/>

      <!-- Negate this entry? -->
      <xsd:attribute name="negate">
         <xsd:simpleType>
            <xsd:restriction base="xsd:NMTOKEN">
               <xsd:enumeration value="true"/>
               <xsd:enumeration value="false"/>
            </xsd:restriction>
         </xsd:simpleType>
      </xsd:attribute>

      <!-- What type of port matches does this rule have -->
      <xsd:attribute name="ports">
         <xsd:simpleType>
            <xsd:restriction base="xsd:NMTOKEN">

               <!-- Inclusive range -->
               <xsd:enumeration value="inc-range"/>

               <!-- Set of single port numbers -->
               <xsd:enumeration value="set"/>
            </xsd:restriction>
         </xsd:simpleType>
      </xsd:attribute>
```

```
    </xsd:complexType>
</xsd:element>
```

Using the above schema, source addresses can be constructed with the following form. This will match on any source address not from the 192.168.2.0 network, hence outside the range 192.168.2.0 to 192.168.2.255.

```
<src type="ip" negate="true" address="192.168.2.0"
     mask="255.255.255.0"/>
```

A similar analysis was made of IPFilter, resulting in alteration to the abstracted firewall description schema. The package created by this project also supports the definition of a basic DNS service. For this, the syntax used with BIND8 was analysed, and similarly a schema was created, and support added to the package. Currently BIND8 is the only translation supported, but DNS systems typically use a standard format for start of authority definitions.

A complete description of a network node is given in section 10.2 of this document.

4.3 Variable Mapping Assignments

Variable mappings play an important role within the system, aiding the overall configuration task. Common data is typically available to services running on a network node, such as network settings and open ports. Such settings also apply at the network layer, as nodes within a single subnet will share data such as subnet mask and network number.

To prevent duplication of such data throughout the system, a hierarchy of system objects is used to maintain the scope of variables. At each level within the system, the user can specify variables to be mapped to some value. Local variable scope is maintained throughout the hierarchy, hence if a node defines `untrusted_if` to be mapped to `tun0`, it can be later accessed anywhere lower in the object model using the identifier `$untrusted_if`. However, this variable can be redefined to map to a different value at a lower level if required.

In addition, variables play an important role in situations where unique IDs are required for configuration rules. Within a firewall configuration for example, the user may wish to give each rule a unique ID number, but may not know in advanced the ranges required. This can be solved by defining a variable within a service construct, such as `$rule_id_base`, then specifying IDs as offsets from this value. The package supports arithmetic within variables, so IDs could be mapped simply as `$rule_id_base+1` for example. Along with this, recursive variable resolution occurs if part of one variable maps to another. Standard arithmetic operations are supported within variable mappings, using brackets to enforce precedence.

The lowest layer where variables can exist is that of the service construct, introduced in the next section. However, nested service constructs can be used, such as with function calls and external references, leading to a more complex mapping hierarchy. A set of global variables are also available to all elements within the system, such as `$this` returning the current node name, and others indicating the version of the package.

4.4 Service Constructs

Service constructs represent a logical grouping of service configuration, be it a sequence of rules that perform some overall task, or a collection of configuration statements. Each construct can define local variable mappings to aid configuration within that section of XML, and is essentially the building blocks of a service description.

A specific service construct is included for each service type, hence `FirewallConstruct` and `DNSConstruct`. These are defined at XML configuration level and allow functionality to be encapsulated into manageable sections. The definition of a service construct is purely a conceptual grouping; it has no meaning at the translation stages. Also, the package has no understanding of what a *logical group* may be. The following example shows a basic `FirewallConstruct` including local variable mapping definitions and contained `Rule`.

```
<fw:FirewallConstruct name="Loopback Handeling"
      description="Does stuff with the loopback device">

   <Mappings>
      <Variable name="block_rule_base" value="$rule_base+100"/>
   </Mappings>

   <fw:Rule Desc="Allow loopback" RuleID="$block_rule_base">
      <fw:action perform="pass"/>
      <fw:protocol type="all"/>
      <fw:src type="any"/>
      <fw:dst type="any"/>
      <fw:interface via="lo0"/>
   </fw:Rule>
</fw:FirewallConstruct>
```

To aid the configuration task, each service construct can have a name and description. The significance of naming is described in section 4.6, Function Calls, allowing constructs to be uniquely referenced and used as input to function calls.

4.5 External Service Constructs

In addition to service constructs, a library can be used to maintain a repository of commonly used constructs, which can then be imported into any other service description. The major advantage of this is that it reduces the maintenance overhead when modifying common sets of rules, and also provides an additional layer of abstraction to the end user.

With significant numbers of external constructs available to the user, they will be able to specify their whole service description using sections of previous tested and verified configurations. Such rule reuse gives confidence to the user in the overall behaviour of the system, hence producing more reliable service descriptions.

External constructs are implemented using a substitution method. When the user wishes to generate fully expanded XML configuration, such as prior to running through the XSL translator, the external construct references are resolved. However, variable scope also plays an important role. The reference to include an external construct can itself include a set of variable mappings; enabling data to be passed through to the external construct in a parameter style.

The following example demonstrates the reference to an external service construct providing basic IP address spoof prevention. In such a situation, we aim to prevent IP packets from non-routable source addresses arriving on our global interface. Local variable mappings have been used to specify the interface we intend to protect and the initial rule ID to use. These are passed onto the external construct when it is pulled in from the library.

```
<fw:FirewallConstruct name="Do Spoof Prevention" description="">

   <Mappings>
      <Variable name="block_rule_base" value="$rule_base+900"/>
      <Variable name="untrusted_if" value="tun0"/>
   </Mappings>

   <ExternalConstruct
      name="Firewall::ExternalConstruct:Spoof Prevention"/>

</fw:FirewallConstruct>
```

The following XML segment is part of the external service construct referenced from the above description, `Firewall::ExternalConstruct:Spoof`. The rules within this section are defined to use various variable mappings such as `$untrusted_if` and `$block_rule_base`, enabling the same description to be imported multiple times with different behaviour. Variable mappings can also be defined to be local to a whole library, and such variables override ones defined at the node and service levels.

```
<FirewallConstruct
   name="Firewall::ExternalConstruct:Spoof Prevention"
   description="Prevents spoofing over untrusted interface">

   <Rule RuleID="$block_rule_base">
      <action perform="deny"/>
      <log value="true" logamount="0"/>
      <protocol type="all"/>
      <src mask="255.255.0.0" address="192.168.0.0" type="ip"/>
      <dst type="any"/>
      <interface direction="in" via="$untrusted_if"/>
   </Rule>
      <Rule RuleID="$block_rule_base+1">
      <action perform="deny"/>
      <log value="true" logamount="0"/>
      <protocol type="all"/>
      <src mask="255.255.0.0" address="172.16.0.0" type="ip"/>
      <dst type="any"/>
      <interface direction="in" via="$untrusted_if"/>
   </Rule>
   <Rule RuleID="$block_rule_base+2">
      <action perform="deny"/>
      <log value="true" logamount="0"/>
      <protocol type="all"/>
      <src mask="255.0.0.0" address="10.0.0.0" type="ip"/>
      <dst type="any"/>
      <interface direction="in" via="$untrusted_if"/>
   </Rule>
</FirewallConstruct>
```

4.6 Function Calls

Function calls add a whole new level of scope to abstracted configuration descriptions. The purpose is to add computational functionality within the XML configuration stages, allowing descriptions to be processed prior to being passed through the translator. Within a service, an XML element can be used to describe a function call, taking as input some section of XML configuration from any other node along with a set of parameters.

Functions themselves are described as XSL files, taking as input the full XML generated from the specified input elements. The function then performs some evaluation of this input configuration with the intent of generating some XML configuration of its own. This output XML is then directly replaced over the original function call, hence completing the operation.

The reasoning behind introducing function calls is to allow automated configuration of service descriptions based on the specification of others. As there are no restrictions on the location of inputs to function calls, any service running on any node can be used. This allows for complex cross node configuration to be established.

An example of such situation is when there is a firewall protecting a network from the wider Internet. If another node on that internal network wishes to run a new service on a previously denied port, updates will need to be made to add new rules to the firewall. However, if the firewall had a function call, taking as input the service description from the internal node, with knowledge of the service it could automatically generate the required firewall rules, and add them to its configuration.

Section 4.6.2 demonstrates how function calls can be used internally to a node in the automatic configuration of part of a DNS service.

4.6.1 Input to Function Calls

A flexible method of specifying input to a function call has been implemented, allowing targeted sections of configuration to be used. There are various ways that input elements could be selected, either by using the global name of a service construct, or by using an explicit path to the required element.

Every service construct within the system has a globally addressable name (registered when parsed, as shown in figure 2). This is also true across multiple nodes, as the node name features as part of the address. The following is an example of such an address.

```
DNS::ServiceConstruct:bill/Forward
```

This specifies the DNS service construct called `Forward`, within the DNS service on the node name `bill`. There are no restrictions as to which service or node the construct exists on, as they are all available throughout the system.

The second way to specify input to a function is to use a wildcard within the reference, hence indicating that you wish all constructs of that service to be included.

```
DNS::ServiceConstruct:ben/*
```

The above reference will include every service construct within the DNS service running on the node `ben` as input to the function call. This can be useful when generating firewall rules for another service automatically.

The final way to specify input to a function is to use absolute paths. These can also work across nodes, and specify exactly the elements required as input, from the service level down to `Rule` level. Paths use indexes on elements to address positions within the XML configuration itself, which implies that they are more powerful but harder to maintain when extra configurations are added.

```
bill/Firewall(1)/FirewallConstruct(3)/Rule(2)
bill/Firewall(1)/FirewallConstruct(2)
```

The first path specifies the second `Rule` within the third `FirewallConstruct` of the first `Firewall` service definition on the node called `bill`. The second similarly indexes the second `FirewallConstruct` within the first `Firewall` service running on `bill`.

It is proposed that, as a future enhancement to this project, XML filter rules will be used to specify the exact elements passed to functions. This would use a similar syntax to the translation restriction filter described in section 5.2, enabling dynamic processing of `Rules` to select specific combinations of elements and attributes. Following is an example of the proposed syntax.

```
<CallFunction name="DNS::Functions:GenReverseSOA">

  <Group type="AND"
    matchOn="Firewall::Filter:bill/FirewallConstruct/ForwardSOA">

     <Element name="@match=foo.net"/>
     <Element name="/CNAME[!]"/>

     <Group type="OR"
        matchOn="Firewall::Filter:bill/FirewallConstruct/
           ForwardSOA">

        <Element name="A@match=bill.foo.net"/>
        <Element name="A@match=ben.foo.net"/>

     </Group>

  </Group>

</CallFunction>
```

The above calls the `GenReverseSOA` function that generates a `ReverseSOA` construct from an input `ForwardSOA` (see next section for details). In this example, the input to the function will be any `ForwardSOA` defined on the node `bill` matching the filter. The filter rules select whole `ForwardSOA` elements that are associated with the `foo.net` domain (i.e. authoritative to that domain), and which have an `A` sub-element (name to IP address mapping) for either host `bill` or `ben`. They must also have no `CNAME` sub-elements.

As described in section 5.2, the XML filter to be used for this proposed addition can accept complex combinations of filter groups and elements. It is envisioned that such a method could be used by a function to automatically create a completely new service description, taking into account the configuration of many others from various nodes on the network.

4.6.2 Example Use of Function Calls

An example where function calls are valuable is in the configuration of a DNS service. Within configurations there are typically two sets of data, forward start of authority and reverse start of authority. These map from name to IP address, and from IP address to name respectively. However, there is a great dependency between the two. The addition of a single node to the service would require two alterations to the configuration, both using a similar description.

The following XML output shows the configuration within the DNS service of a node. The first service construct is a definition of the forward name to address mappings, and the second contains a `CallFunction` element. This takes as input the forward definitions (`DNS::ServiceConstruct:$this/Forward`) and passes them through the function, to generate the required reverse rules. Note that `$this` is mapped to the name of the node this service is within.

```
<dns:DNSConstruct name="Forward"
                  description="Standard forward resolution">

   <Mappings>
      <Variable name="file" value="foo.net"/>
      <Variable name="NS" value="$this.$root_domain"/>
   </Mappings>

   <dns:ForwardSOA match="$root_domain"
                   primaryns="$NS"
                   adminmail="admin.$this.$root_domain"
                   file="$file"
                   serial="5"
                   refresh="10800"
                   retry="3600"
                   expire="604800"
                   min_ttl="86400">

      <dns:NS match="@" target="$NS"/>
      <dns:A match="localhost " target="127.0.0.1"/>
      <dns:A match="bill" target="$network.100"/>
      <dns:A match="ben" target="$network.101"/>

   </dns:ForwardSOA>
</dns:DNSConstruct>

<dns:DNSConstruct name="Reverse"
                  description="Standard reverse resolution">


   <CallFunction name="DNS::Functions:GenReverseSOA"
      onConstruct="DNS::ServiceConstruct:$this/Forward">

      <Parameter name="dbfile" value="db.192.168.2"/>
      <Parameter name="with_net" value="192.168.2"/>
```

```
      </CallFunction>

   </dns:DNSConstruct>
```

The function call used here is `DNS::Function:GenReverseSOA`, which in itself is an XSL style sheet.

```
   <dns:DNSConstruct name="Reverse SOA"
         description="Generated by DNS::Functions:ReverseSOAGen">

      <dns:ReverseSOA match="slybase.homeip.net."
         primaryns="bill.foo.net"
         adminmail="root.bill.slybase.homeip.net" serial="5"
         refresh="10800" retry="3600" expire="604800"
         min_ttl="86400" file="db.192.168.2">

         <dns:NS match="@" target="bill "/>
         <dns:PTR match="100"
                  target="bill.slybase.homeip.net"/>
         <dns:PTR match="101"
                  target="ben.slybase.homeip.net"/>
      </dns:ReverseSOA>

   </dns:DNSConstruct>
```

When evaluated, the construct referenced as input to the function, `DNS::ServiceConstruct:$this/Forward`, is called to generate its XML content. This XML is then passed as input to the function call, using the XSLT processor. The function itself generates a PTR (IP address to name mapping) resource record for each node within the original `ForwardSOA`, and formats it as a `ReverseSOA` element. This is directly replaced with the original function call, and an example of this output is included above.

5. Service Translation Process

5.1 Specifying the Translation

Translation is the process of converting the abstracted service description into rules and configuration elements that can be directly applied to the desired service implementation. The process used to achieve this is XSLT using XSL style sheets. For each implementation, such as IPFilter or IPFW for firewalls, an XSL sheet is created to perform the transformation to the end level rules. An XML file of the same name is used to act as a wrapper, providing extra functionality such as detailing limitations of the translation, described in the next section.

The aim is to create a rich enough translation between the abstracted description and the final configurations, so that the behaviour directly reflects that of the original XML specification. If this could be achieved, a node could in fact be swapped with one running a different implementation of a given service. Generating the configurations for this new node should give the same functionality as before, enabling services to be swapped for performance, security and testing reasons.

5.2 Translation Restriction Calculation

5.2.1 Overview of Restrictions

Due to the level of abstraction introduced in describing the behaviour of network services, there is commonly a mismatch between the two levels of configuration. Features supported by the XML description may translate directly into one implementation level service but not another, which implies that the overall behaviour of the system will not be as the user expected. However, this section describes a process that enables the user to test whether there will be any problems in the translation process. In addition it reports to the user information on the XML elements of their configuration that are restricted, along with textual reasoning.

Such restrictions can only be identified with knowledge of the specific target implementation language, all details of which are stored within the translation XSL file itself. Therefore, in addition to the XSL file, each translation also contains an XML file describing a set of restrictions of that process.

Restrictions are described as mapping a combination of XML configuration elements to some text reason explaining why they are not supported. These will be specified by the creator of the translation, and can contain multiple rules that an XML element must match to be declared as a restriction.

Restriction calculation is performed using a pre-processing style. The full XML output of the service is fed into the pre-processor, specifying which translation we desire to test for restrictions. The pre-processor itself is in fact a serial processing XML filter, using a set of predefined rules (specified as restrictions) to analyse the stream of XML as it passes. Elements that match the rules will be returned to the user, along with the text reason for that restriction matching.

The restriction filter provides a rich syntax for matching XML configuration elements, with the intent of being able to specify any restrictions that may occur. The basic element of a restriction maps from the status of an attribute or element to a text reason that is outputted on the match of such an attribute. However, combinations of these can be made and put into groups forming logical operations such as AND/OR. Further groups can be created consisting of other groups and sub-elements allowing complex node configurations to be matched.

5.2.2 Restriction Rule Specification

The following restriction segment shows matching on the existence of a sub-element of XML. This will therefore match any `Rules` that have one or more `protocol` sub-elements within them, hence the overall path to this element is `/Firewall/FirewallConstruct/Rule/protocol`. A specific filter syntax is used to address restrictions.

```
<RestrictionElement
   name="Firewall::Filter:FirewallConstruct/Rule/protocol"
   reason="Matching existence of sub-element"/>
```

The following restriction matches `src` elements of XML that have an attribute `address` defined, with or without a specific value. The lower rule matches `interface` specifications that don't have an attribute `via` defined.

```
<RestrictionElement
   name="Firewall::Filter:FirewallConstruct/Rule/src@address"
   reason="Existence of element attribute"/>

<RestrictionElement
   name="Firewall::Filter:FirewallConstruct/Rule/interface@!via"
   reason="Absence of element attribute"/>
```

The ability to match on attributes with or without a specific value is shown below. The first matches `Rules` that have a `RuleID` attribute set at the value of `1004`. The second will match all `interface` elements that have an attribute called `direction`, whose value does not equal `in`.

```
<RestrictionElement
   name="Firewall::Filter:FirewallConstruct/Rule@RuleID=1004"
   reason="Existence of an attribute with specific value"/>

<RestrictionElement
   name="Firewall::Filter:FirewallConstruct/Rule/
      interface@!direction=in"
   reason="Absence of an attribute with a specific value"/>
```

Restriction groups have a different syntax. They can represent some logical operation of other groups and element, be it AND, OR or ANY, specified in the `type` attribute. When the type ANY is specified it is logically equivalent to OR, but indicates that there is no connection between the rules. Groups match at a defined path within the XML structure, as specified by the `matchOn` attribute. During pre-processing, this group will become active when we enter this path, and the group will be checked to see if it matched when we leave that path, and hence the group becomes deactivated.

Sub-elements of a restriction group must process attributes within the path that the group is active for; hence the topmost path should be specified in the outer group. Again, a text description of the restriction is specified to be outputted when a match is found.

Elements within the group are defined as offsets from the `matchOn` path attribute. In the following example, it will match `Rules` that have a `RuleID` attribute of `1002`, `1004` or `1006`, as the overall `type` of the group is OR. A similar syntax to that described above is used for each restriction element, with the leading path removed.

```
<RestrictionGroup
      type="OR"
      matchOn="Firewall::Filter:FirewallConstruct/Rule"
      reason="Attribute with either one of a range of values">

   <RestrictionElement name="@RuleID=1002"/>
   <RestrictionElement name="@RuleID=1004"/>
   <RestrictionElement name="@RuleID=1006"/>
</RestrictionGroup>
```

Due to the nature of the serial pre-processor, to match the absence of an XML element the rule must be enclosed within a restriction group. This is because the processor tests elements and attributes as it meets them on the XML stream, but as the missing attribute would never occur, it needs a position to launch the test to see if it has been found. The following example will match all `Rules` that don't have an `options` sub-element or an `interface` sub-element. Note that the `type` of `ANY` has been used here, and each restriction element within the group has its own reason. This simply means that there is no connection between the two elements, and when a match is found for one its reason should be outputted immediately.

```
<RestrictionGroup
     type="ANY"
     matchOn="Firewall::Filter:FirewallConstruct/Rule">

   <RestrictionElement name="/options[!]" reason="No options"/>
   <RestrictionElement name="/interface[!]" reason="No interface"/>
</RestrictionGroup>
```

The following restriction group demonstrates a more complex combination of rules. It will match `Rules` that have an `RuleID` attribute, that have a `direction` of travel as either `in` or `out` within the `interface` sub-element and which also either `pass` or `deny` that overall `Rule`. Also, the `Rule` must be for a `protocol` other than `all`, be from some specific source IP address (hence not *any*) and destined for no specific port number.

```
<RestrictionGroup
      type="AND"
      matchOn="Firewall::Filter:FirewallConstruct/Rule"
      reason="Matching some random group of data">

   <RestrictionElement name="@RuleID"/>

   <RestrictionGroup
      type="OR"
      matchOn="Firewall::Filter:FirewallConstruct/Rule">

      <RestrictionElement name="/interface@direction=in"/>
      <RestrictionElement name="/interface@direction=out"/>
   </RestrictionGroup>

   <RestrictionGroup
      type="OR"
      matchOn="Firewall::Filter:FirewallConstruct/Rule/action">

      <RestrictionElement name="@perform=pass"/>
      <RestrictionElement name="@perform=deny"/>
   </RestrictionGroup>

   <RestrictionElement name="/protocol@!type=all"/>
   <RestrictionElement name="/src@type=ip"/>
   <RestrictionElement name="/dst/port-num[!]"/>

</RestrictionGroup>
```

The following `Rule` would indeed match the above restriction group.

```
<fw:Rule RuleID="1007" >
   <w:action perform="pass"/>
   <fw:protocol type="other" name="udp"/>
   <fw:src type="ip" mask="255.255.255.255"
      address="192.168.2.100"/>
   <fw:dst type="ip" mask="255.255.255.0" address="192.168.2.0"/>
   <fw:interface direction="out"/>
</fw:Rule>
```

In addition to the rule formats mentioned above, an option has been added to allow exact counts of sub-elements to be matched. These apply to all `RestrictionElements` within `RestrictionGroups`, hence can be used to count how many sub-elements match that specific rule construct.

```
<RestrictionGroup type="AND"
   matchOn="Firewall::Filter:FirewallConstruct/Rule"
   reason="Rules with more than 2 icmptype sub-elements">

   <RestrictionElement name="/options/icmptype[m2]"/>
</RestrictionGroup>
```

The method shown above allows multiples of elements to be matched. The square brackets at the end of the rule denote such a count, and can take various forms. By default, as when the square brackets are not included, `[+]` is assumed. This simply means to match that rule where there are one or more occurrences of the element within the enclosing group. As seen previously, `[!]` can be used to specify the absence of a rule match. An integer can be placed within the brackets, hence `[5]` would match when exactly five instances of that rule matched within the group. Negation can be used with this, therefore `[!5]` would match where there is anything but 5 exact element matches.

Finally, more than and less than can be represented using the `[mX]` and `[lX]` definitions respectively, where X denotes some non-inclusive boundary. With multiple rules such as this, complex situations can be represented allowing finer control of element matches.

The following group will match on whole `FirewallConstructs`. To match, the construct must have more than one `Rule`, and one `Rule` or more must have a destination address of `192.168.2.100`. Also, more than two `Rules` must define `options`, and one or more of these `options` contains `icmptype` sub-elements.

```
<RestrictionGroup type="AND"
   matchOn="Firewall::Filter:FirewallConstruct"
   reason="Combination of expressions">

   <RestrictionElement name="/Rule[m1]"/>
   <RestrictionElement name="/Rule/dst@address=192.168.2.100[+]"/>
   <RestrictionElement name="/Rule/options[m2]"/>
   <RestrictionElement name="/Rule/options/icmptype[+]"/>

</RestrictionGroup>
```

5.2.3 Pre-processor Output

The aim of the pre-processing stage is not just to identify the existence of possible restrictions, but also to direct the user towards the offending parts of their XML service description. To perform this the package can produce additional information when restrictions are found to help the user identify the location of the problems.

Following is an example of the matched output produced by the pre-processor.

```
<Restriction line="57" col="18"
      path="/Firewall(1)/FirewallConstruct(1)/Rule(7)"
      reason="Matching some random group of data">

   <fw:Rule RuleID="1007" >
      <fw:action perform="pass"/>
      <fw:protocol type="other" name="udp"/>
      <fw:src type="ip" mask="255.255.255.255"
         address="192.168.2.100"/>
      <fw:dst type="ip" mask="255.255.255.0"
         address="192.168.2.0"/>
      <fw:interface direction="out"/>
   </fw:Rule>

</Restriction>
```

Initially details on the restriction are outputted, including the line and column numbers that the problem XML starts on, and also the exact processing path to that element. This path is detailed by including the index of the various sub-elements that were encountered to get to the start of that element.

In the above example we can conclude that the matched XML element is the seventh `Rule`, within the first `FirewallConstruct` all within the first `Firewall`. As the input XML is just a service, there will be no additional node definitions on the path.

In addition to reporting the path, the package also fetches the actual source XML input, using the reported `path`, and outputs this within the final restriction report. In effect this can report straight back to the user the source of the restriction, allowing them to make the required changes. To aid interpreting systems using this package, such as GUIs, the restriction report is itself valid XML, hence enabling further processing to be performed.

If the restriction element or group specifies the `outputXML` attribute to equal `“no”`, then only the path and restriction reason details will be generated with no source XML.

5.2.4 Complications During Filter Process

The ability of the filter pre-processor to generate source XML on restriction match is itself not trivial. As the filter is processing in a serial manner, minimal state is maintained about seen XML, and there is no knowledge of future XML. This implies that the filter is unable to remember matched XML, hence needs to use the stored `path` and fetch the source XML directly from the object structure.

However, fetching the source object can cause complications. Initially the input to the pre-processor was simply a stream of fully expanded XML (implying variables, external constructs and functions have been resolved). It is quite possible that the restriction the

package is attempting to locate was defined within an external construct, or even the output of some function call.

The package will attempt to fetch the exact XML element that matched the restriction, which is fine with standard definitions and external constructs. Variable mappings at that exact level are also resolved to provide the user with as much information as possible. However, function calls cannot be inspected by the pre-processor, as they don't return objects (they in fact return another XML stream), so in this situation, the result of the whole function is returned to the user.

5.2.5 Example Translation Restriction

One of the restrictions of the IPFilter translation is that it does not support the `gid` and `uid` attributes of the `options` element, representing group and user ID respectively. Therefore, the following restriction group will highlight to the user the translation problem if they have these options defined within their XML service specification.

```
<RestrictionGroup
      type="OR"
      matchOn="Firewall::Filter:FirewallConstruct/Rule/options"
      reason="UID and GUI are not supported within IPFilter">

   <RestrictionElement name="@uid"/>
   <RestrictionElement name="@gid"/>
</RestrictionGroup>
```

Another restriction that has been identified within IPFilter is with `Rule` direction. IPFW, along with the abstracted XML firewall description, allows `Rules` to be specified without an explicit direction of travel. Within the firewall implementation these will match rules going in either direction. However, IPFilter requires an explicit direction statement, hence there may be a mismatch between the two descriptions. The IPFilter translation process tries to resolve this (see section 5.3) by duplicating rules, but in some situations it may result in undesired behaviour. Therefore, a restriction rule was added to the IPFilter translation description, to indicate to the user that such a problem might exist.

The following restriction group represents the match required for identifying XML firewall `Rules` without explicit direction definitions. The first restriction element will match `Rules` that don't have an `interface` definition at all, and the second matches `Rules` that have an `interface` definition, but no `direction` attribute. Finally, the restriction sub-group will match `Rules` with a `direction` attribute not equal to `in` or `out`. If any of these restriction filter elements match, the user will be notified as explained in section 5.2.3.

```
<RestrictionGroup
   type="OR"
   matchOn="Firewall::Filter:FirewallConstruct/Rule"
   reason="Rules without set directions will be duplicated">

   <RestrictionElement name="/interface[!]"/>
   <RestrictionElement name="/interface@!direction"/>

   <RestrictionGroup
```

```
        type="AND"
        matchOn="Firewall::Filter:FirewallConstruct/Rule/interface">

         <RestrictionElement name="@!direction=in"/>
         <RestrictionElement name="@!direction=out"/>

      </RestrictionGroup>

   </RestrictionGroup>
```

5.3 The Translation Process

The next stage is to generate implementation level configuration files. Prior to translation, the fully expanded XML representation of the chosen service is generated, involving the resolution of external constructs and function calls. This is then fed into an XSL processor, along with the XSL style sheet describing the translation for the desired implementation level service. The output from this process will be a series of configuration files that can be applied directly to the intended service. Once implemented, these should have the same behaviour as previous specified in the abstracted XML description.

A translation is described using XSL, mapping combinations of elements within the abstracted description to their lower level syntax. Such a process should be able to convert all elements within the input to their equivalent at this lower level, unless specified otherwise within the restrictions stage.

Following is a section from the XSL service translation sheet for IPFW, a firewall implementation typically used with FreeBSD. The extract is from the operation that processes the `options` element of XML, dealing with IP options and TCP flags.

```
<!-- The IP and TCP options -->
<xsl:template match="fw:options">

   <xsl:if test="@frag = 'true'">
      <xsl:text>frag </xsl:text>
   </xsl:if>
   <xsl:if test="@established = 'true'">
      <xsl:text>established </xsl:text>
   </xsl:if>
   <xsl:if test="@setup = 'true'">
      <xsl:text>setup </xsl:text>
   </xsl:if>

   <xsl:if test="self::node()[@keep-state]">
      <xsl:text>keep-state </xsl:text>
      <xsl:if test="@keep-state != ''">
         <xsl:value-of select="@keep-state"/>
      </xsl:if>
   </xsl:if>

   <!-- Process all the IP options elements -->
   <xsl:if test="count(fw:ipoption) > 0">
      <xsl:text>ipoptions </xsl:text>
      <xsl:for-each select="./fw:ipoption">

         <xsl:if test="@absent='yes'">
            <xsl:text>!</xsl:text>
```

```
            </xsl:if>

            <xsl:choose>
               <xsl:when test="@spec='ssrr'"> <xsl:text>ssrr</xsl:text>
               </xsl:when>
               <xsl:when test="@spec='lsrr'"> <xsl:text>lsrr</xsl:text>
               </xsl:when>
               <xsl:when test="@spec='rr'"> <xsl:text>rr</xsl:text>
               </xsl:when>
               <xsl:when test="@spec='ts'"> <xsl:text>ts</xsl:text>
               </xsl:when>
            </xsl:choose>

            <xsl:if test="not(position()=last())">
               <xsl:text>,</xsl:text>
            </xsl:if>
         </xsl:for-each>

         <xsl:text> </xsl:text>
      </xsl:if>

      <xsl:if test="self::node()[@uid]">
         <xsl:text>uid </xsl:text>
         <xsl:value-of select="@uid"/> <xsl:text> </xsl:text>
      </xsl:if>

      <xsl:if test="self::node()[@gid]">
         <xsl:text>gid </xsl:text> <xsl:value-of select="@gid"/>
         <xsl:text> </xsl:text>
      </xsl:if>

   </xsl:template>
```

The extract below is from the XML `options` processing translation within IPFilter, another firewall implementation that this project supports. The `setup` and `established` flags do not have a straight translation to IPFilter as they did with IPFW, hence the flags of the TCP packet have to be specified explicitly (in this case identifying connection set up packets with just the `SYN` flag set). It can also be seen that IPFilter doesn’t support the `gid` and `uid` elements, for specifying group and user IDs respectively. In such a situation, a restriction will be identified during the restriction filter pre-processing stage (see section 5.2.5).

```
   <!-- The IP and TCP options -->
   <xsl:template match="fw:options">

      <!— If we need some extra flags -->
      <xsl:if test="@established = 'true' or @setup = 'true'">

         <xsl:text>flags </xsl:text>
         <xsl:if test="@established = 'true'">
            <xsl:text>S/SA</xsl:text>
         </xsl:if>
         <xsl:if test="@setup = 'true'">
            <xsl:text>S/AUPRFS</xsl:text>
         </xsl:if>

         <xsl:text> </xsl:text>
      </xsl:if>
```

```
      <!-- Process all the IP options elements -->
      <xsl:if test="count(fw:ipoption) > 0">
         <xsl:text>with </xsl:text>

         <xsl:for-each select="./fw:ipoption">
            <xsl:if test="@absent='yes'">
               <xsl:text>not </xsl:text>
            </xsl:if>

            <xsl:choose>
               <xsl:when test="@spec='ssrr'">
                  <xsl:text>opt ssrr</xsl:text>
               </xsl:when>
               <xsl:when test="@spec='lsrr'">
                  <xsl:text>opt lsrr</xsl:text>
               </xsl:when>
               <xsl:when test="@spec='rr'">
                  <xsl:text>opt rr</xsl:text>
               </xsl:when>
               <xsl:when test="@spec='ts'">
                  <xsl:text>opt ts</xsl:text>
               </xsl:when>
            </xsl:choose>

            <xsl:if test="not(position()=last())">
               <xsl:text>,</xsl:text>
            </xsl:if>

         </xsl:for-each>

         <xsl:text> </xsl:text>
      </xsl:if>
   </xsl:template>
```

The following XML segment is a simple rule with various extra `options` definitions. It matches a TCP packet, travelling from anywhere to anywhere, that is a connection set up request (TCP SYN flag set). In addition, the packet must not be set as using source routing (either strict of loose), used for specifying a fixed route of travel for that packet.

```
<fw:Rule Desc="Test Rule" RuleID="100">
   <fw:action perform="pass"/>
   <fw:protocol type="other" name="tcp"/>
   <fw:src type="any"/>
   <fw:dst type="any"/>
   <fw:options setup="true" established="no" gid="200">
      <fw:ipoption spec="lsrr" absent="true"/>
      <fw:ipoption spec="ssrr" absent="true"/>
   </fw:options>
</fw:Rule>
```

When using the input XML rule above, processing with the IPFW translation, the following rule is produced.

```
add 100 pass tcp from any to any setup ipoptions lsrr,ssrr gid 200
```

The IPFilter translation produces the following two rules. Note that the `gid` option is not supported by IPFilter, and hence is removed (refer to section 5.2.5). Another

restriction of IPFilter is that rules must have an explicit direction of travel, whereas the abstracted firewall description does not enforce this. The translation process can resolve these conflicts automatically by generating duplicate rules, creating two rules with different IDs and directions of travel.

```
@100 pass in quick proto tcp all flags S/AUPRFS with opt lsrr,opt ssrr
@101 pass out quick proto tcp all flags S/AUPRFS with opt lsrr,opt ssrr
```

6. Program Testing

6.1 Introduction

Due to the nature of this project, the main package is concerned with reading in XML descriptions, processing, manipulating and producing new configurations. Therefore, the majority of testing has been focused on ensuring that the package correctly processes specifications. However, during the translation process, this package produces configuration files for services such as firewalls. These cannot easily be tested, and in addition testing that two different firewall implementation are conceptually the same (as their original abstracted description states) is beyond the scope of this project.

The testing phase described within the test scripts in section 10.1 used the black box method of testing. The XML processing package is fed a series of XML input files, specifying various operations, and then the output of the package is compared to the expected definitions. All tests within section 10.1 are automatic, using the comparison tool described in section 6.2 to look for differences.

A different testing strategy is described in section 6.3, aiming at testing the capabilities of the package to perform valid translations to implementation level configuration files. For this, a real test network was established, containing many nodes, and three different subnets. Routers were used to join these networks together, and a DNS service and two firewall services were specified in XML. Automatic configuration generation was used to create the require files, which were then directly applied to the application services running on the nodes. The behaviour of the system was then analysed to ensure that the service translation had indeed given us a network that operates as expected.

6.2 XML Comparison Tool

To aid the testing of the XML processing element of this project, a tool was written to compare XML documents. This tool can compare whilst taking into account the various subtle differences that can be present in syntactically identical XML files. If differences are located between the two input files, the tool outputs the path to both elements that differ. For the majority of processing tests, an expected output file is used, which contains the correct result that we intend to obtain. During regression testing these can all be quickly re-run.

6.3 Implemented Network Testing

The sole purpose of the following series of tests was to establish if the service translation aspect of the project behaves as expected. Initially, a network was created and implemented featuring many nodes, and three subnets. Routers were then placed to join these subnets, featuring both firewalls and a DNS service for the domain. Rules were created to ensure that the test coverage was sufficient.

Figure 4 shows the layout of the network. It centres on having a single trusted subnet, containing various nodes that should have full access to each other. A firewall router then bridges this subnet with another, serving wireless users. The security model implemented here is that no users within the wireless subnet should be able to contact any machines within the trusted network, but be able to access the Internet as a whole. Another router firewall bridges the trusted subnet with the main parent network. This has a router within it (not part of the specification of the test network) that is responsible for routing traffic onto the rest of the Internet. For this, we assume that the node performs no filtering; hence the primary gateway into the trusted network should perform all required filtering.

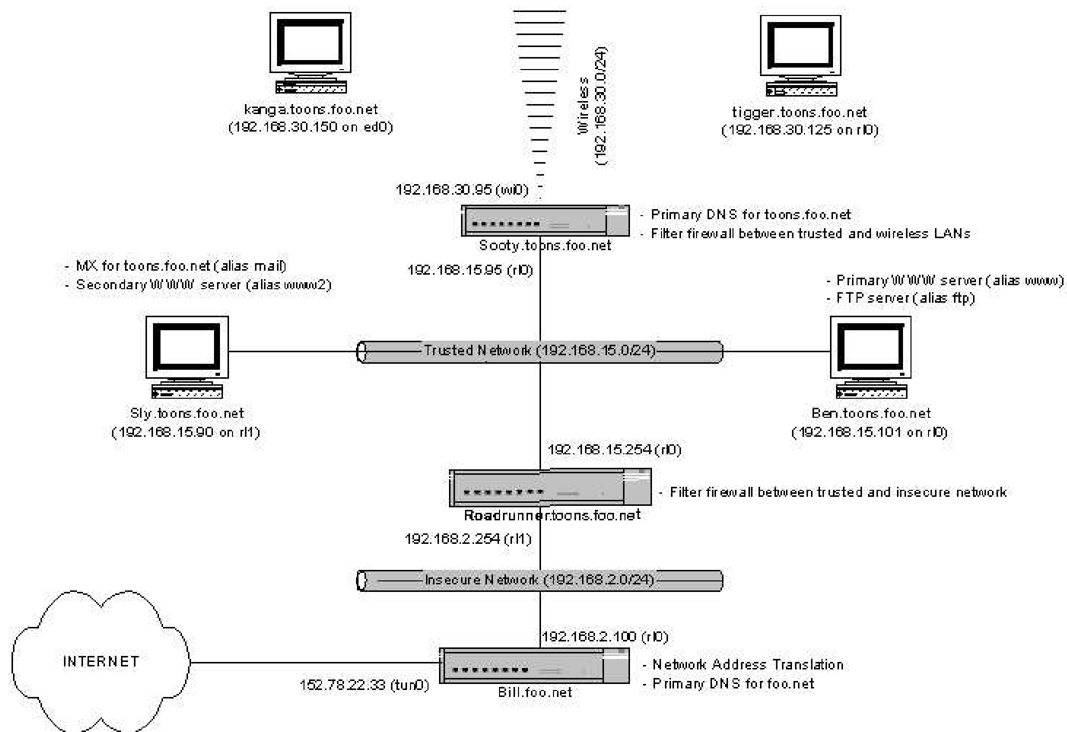

Figure 4: Layout of test network

Users within the trusted network can ping hosts within the wireless network, and also establish TCP communications with them. Hosts on both the trusted and wireless networks can query the DNS server running on the bridge between those networks, acting as the primary authority on the `toons.foo.net` domain.

Section 10.2 lists the complete XML description of the node *sooty* in the above network.

6.3.1 The Testing Process

XML configurations for the two firewall descriptions and the single DNS service were created for the nodes `Sooty` and `Roadrunner`. Stub XML descriptions were generated for the other nodes to identify their interfaces and IP addresses. Each node was then established, and the configuration files generated using the primary package, employing function analysis and service translation.

Once the desired network configuration was operating, the system was analysed to ensure that it did indeed behave as originally specified in the abstracted XML descriptions. This enabled us to determine that the translation process was correct, though may not be complete. The DNS service was queried to ensure that it maintained the required behaviour, and that nodes that required access to that service indeed did have. A query was made for all forward names to address mappings (A records), including any canonical names (CNAME records). In addition, reverse lookups were performed to validate the address to name resolution (PTR records).

Opening standard connections between different parts of the system, and looking for undesired behaviour verified the implementation of the firewalls. Initially ICMP messages were sent, such as ping requests. Our original description states that hosts within the wireless network should not be capable of pinging hosts within the trusted network, but should be able to ping hosts within the wider Internet. Secondly, this was performed in reverse; hence nodes within the trusted network pinging hosts within the wireless network.

Finally TCP connections were verified. An ftp server was executed on both the host `Roadrunner` and the host `Bill`. From the wireless network, hosts should be able to establish an ftp connection to `Bill`, but not `Roadrunner`. Hosts within the trusted network should be able to establish connections to all other hosts. In addition, this test was performed from outside the two networks, hence within and beyond the network 192.168.2.0/24. For the DNS tests to pass, the querying host must have had access to send UDP queries on port 53, therefore we know this functionality is included.

6.3.2 Results

When this test was executed, the expected behaviour was achieved using the service translations for IPFW and BIND8. This therefore proves that for the subset of functionality employed within the firewall and DNS services, the descriptions indeed have a correct translation through to implementation level. However, as not all the aspects of a firewall could be included, we cannot conclude that the translation process is yet complete.

In addition, to strengthen such a test, varying firewall implementations could be used. If a strict testing process was to be constructed, with a set of pre-defined connection requests, such a test could be re-run on the different firewalls to establish if they performed the same. It is envisioned that with a rich abstract firewall syntax, precise service translations and a detailed simulation model, users could verify the expected behaviour prior to actually implementing nodes.

7. Summary

7.1 Conclusions

During the development of this project the implementation of an abstracted service processor and architecture have been researched. This project has essentially produced a package for the centralised management of network services. There are indeed various stages to the configuration process, with this project concentrating on service management [17].

Initially the abstracted definitions of services were created to act as the core of the system. These were not simply static XML representations of the implementation level files, but included various elements of dynamic functionality. The addition of variable mappings, external constructs and function calls allows complex configurations to be created, based on the definitions of other nodes and services within the surrounding network.

Descriptions in an abstracted state can be modified using the package interface, allowing the addition of new nodes and services. When desired, these enterprise descriptions can be analysed by the package in order to highlight any possible restrictions that may occur in the translation process. The desired result is to provide feedback to the configuration stages, allowing restricted elements to be resolved

In addition, such descriptions enable network simulations to be performed, analysing the network against common security threats. Again, the feedback from simulation aims to provide security and performance improvements to the original descriptions prior to any service being implemented. Simulation is beyond the scope of this project, but future work on this topic is discussed in the following section.

The final process introduced by this project is translation. Abstracted XML descriptions of services are fed into the translator processor, with the intent of generating implementation level configurations files. Such files can then be directly applied to the service applications themselves, and we can be sure that they will behave as the original description specified.

This project has primarily concentrated on the firewall and DNS services. Due to this, translations have been created for IPFW and IPFilter for firewalls, and BIND8 for DNS. With the central architecture in place, it is proposed that the addition of new services and translations can be made easily, making the overall tool more applicable within larger network environments.

During this project, a paper [1] was written for the Business Information Systems 2002 conference in Poznan, Poland. The conference featured papers associated with Mobile E-business, and I presented this paper during the *Web Languages and Web Personalization* session. A copy of the paper is included in section 11, and a copy of the presented slides can be obtained from the main project web site [2].

7.2 Future Work

This project was aimed at designing and implementing the require architecture to enable abstracted and dynamic configuration of nodes within a network environment. Using such descriptions, and the functionality provided by the package, there are many areas of further interest that would integrate directly into the package created so far.

The major area of interest is simulation, and another project has already started to investigate the addition of such functionality. With the abstracted description of the networks, nodes and services, as presented by this project, the ability to simulate the whole life of a TCP connection as it travels across the network is desired. This will in effect create a virtual service, reading from the abstracted description and responding to requests in a similar way to the implemented service. For example, a virtual firewall service has been constructed which, on receipt of a packet, will process the firewall rules (currently in a stateless manner). Upon a match, it will return the resulting rule, which can then be processed by the higher level GUI.

Such processing will be expanded to allow whole network simulations, such as tracking a DNS request from one host in a single network, through all routing nodes to the final destination. Obviously the various firewalls running on nodes along this connection route will be evaluated.

The desired result is to indicate to the user, via a GUI interface, the exact trace of the packet across the system, hence details on where the packet gets dropped, if appropriate. The simulator will return a path to such simulated element, so that it can be parsed and the Java object itself returned.

It is intended that the simulator will use XML based specifications to describe the operations to be performed during the simulation process. This will enable common network threats to be described and re-run, testing the abstracted configurations. With a complete description of the entire network, users could analyse the security of their system, and establish whether they are protected against these common threats.

To establish how this project will scale to cover new services and translations, it is planned that new, non-rule based services will be added to the system. This will cover configurations such as the Apache web server, using blocks rather than explicit rules. Also new translations to various implementation level configurations will be added for the currently supported services. It is then intended that two differing firewalls services will be tested in parallel, to establish if the abstracted description does indeed behave the same on both systems.

One other future area of research is the addition of support for stateful firewalls. It is envisioned that TCP sessions could be described rather than rules as in the current firewall implementation. This would effectively allow users to create their configurations based on higher-level connections, such as allowing traffic to communicate with a web server. As stateful firewalls aim to treat traffic as flows rather than individual packets, it provides a great canvas for developing user-friendlier abstracted descriptions.

8. References

[1] CHUDLEY, S.R. AND ULTES-NITSCHE, U. Simulation and Implementation of an E-Commerce Communications Infrastructure using XML Specifications. Proceedings of Business Information Systems 2002 Conference, Poznan, Poland, 2002. http://www.ecs.soton.ac.uk/~src299/xmlnetman/bispaper.pdf

[2] CHUDLEY, S.R. XML abstracted network management, 2002. http://www.slyware.com/projects_xmlnetman.shtml

[3] CONOBOY, B, AND FICHTNER, E. IP Filter Based Firewalls HOWTO, 2002 http://www.obfuscation.org/ipf/ipf-howto.html

[4] DNSTools software. http://www.dnstools.com/index.html

[5] FreeBSD Operating System. http://www.freebsd.org.

[6] Information Sciences Institute, University of Southern California Internet Protocol v4 (RFC 791), 1981. http://www.faqs.org/rfcs/rfc791.html

[7] Information Sciences Institute, University of Southern California Transport Control Protocol (RFC 793), 1981. http://www.faqs.org/rfcs/rfc793.html

[8] IPFW firewall manual pages, 2000. http://www.gsp.com/cgi-bin/man.cgi?section=8&topic=ipfw

[9] KURLAND, V AND ZALIVA, V. Firewall Builder, 2001. http://www.fwbuilder.org/

[10] LAVIGNE, D. IPFW firewall configuration details. O'Reilly & Associates http://www.onlamp.com/pub/a/bsd/2001/06/01/FreeBSD_Basics.html

[11] MOCKAPETRIS, P. DNS Domain Names – Concepts and Facilities. RFC1034, 1987. http://www.faqs.org/rfcs/rfc1034.html.

[12] PALMER, G AND NASH, ALEX. FreeBSD Handbook – Firewall Section http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/firewalls.html

[13] POSKANZER, J. A. IPFW firewall example configuration files. http://www.acme.com/firewall.html

[14] REED, D. Filter language compiler specification http://coombs.anu.edu.au/~avalon/flc.html

[15] REKHTER, Y, MOSKOWITZ, B, KARRENBERG, D, DE GROOT, G, AND LEAR, E. Address allocation for private networks (RFC 1918), 1996 http://www.faqs.org/rfcs/rfc1918.html

[16] SPERBERG-MCQUEEN, C. M. AND THOMPSON, H. World Wide Web Consortium. XML Schemas, 2000. http://www.w3.org/XML/Schema

[17] STEVENSON, D. Network Management, 1995.
http://netman.cit.buffalo.edu/Doc/DStevenson/

[18] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/TR/2000/REC-xml-20001006

[19] World Wide Web Consortium. XML Style Sheets (XSL)
http://www.w3.org/Style/XSL/

9. Appendix 1 – User Manual

9.1 XML Network Element Descriptions

9.1.1 Creating Descriptions

Major network elements within the system are described using an abstracted XML format. The top-level element is a LAN, and the complete definition of which is stored within a single directory. One primary XML file describes the LAN itself, featuring information on the contained networks and references to the nodes within it. The following shows the format of the LAN file.

```
<?xml version="1.0" encoding="UTF-8"?>

<LAN xmlns="http://www.ecs.soton.ac.uk/~src299/xmlnetman"
     xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
     xsi:schemaLocation="http://www.ecs.soton.ac.uk/~src299/xmlnetman
     Node.xsd">

   <Mappings>
      <Variable name="trusted_net" value="192.168.15.0"/>
      <Variable name="wlan_net" value="192.168.30.0"/>
      <Variable name="untrusted_net" value="192.168.2.0"/>
   </Mappings>

   <Network>

      <Mappings>
         <Variable name="net_num" value="192.168.15.0"/>
         <Variable name="net_mask" value="255.255.255.0"/>
      </Mappings>

      <Node location="sly.xml"/>
      <Node location="ben.xml"/>
      <Node location="roadrunner.xml"/>
      <Node location="sooty.xml"/>
   </Network>

   <Network>

      <Mappings>
         <Variable name="net_num" value="192.168.30.0"/>
         <Variable name="net_mask" value="255.255.255.0"/>
      </Mappings>

      <Node location="tigger.xml"/>
      <Node location="kanga.xml"/>
      <Node reference="sooty.xml"/>
   </Network>
</LAN>
```

Using the syntax introduced in section 4.2, an individual file for each node describes the abstracted services. Refer to the test network, described in section 6.3 for further details. The configuration files for this network are included within the package `testnet` directory. Section 10.2 lists a complete XML description of a network node.

9.1.2 Variable Definitions

Variables can be defined at all major element boundaries within the abstracted descriptions. Standard lexical scoping is adopted; hence local variables can be used to override previous definitions. Such variables can be defined at LAN, network, node, service and service construct levels. However, more complex variable mappings can be obtained when using functions calls and external constructs, which can produce deeper mappings.

Variables are defined within `Mappings` elements at the head of the major network element boundaries, as shown below.

```
<fw:Firewall>

  <Mappings>
    <Variable name="rule_base" value="($ruleid+1000)/2"/>
    <Variable name="upstream_dns" value="192.168.2.100"/>
  </Mappings>

</fw:Firewall>
```

Variables map a given name to some value. However, the value can be another variable, or an expression containing arithmetic on static numbers and other nested variables. See section 4.3 of the project report for more details on variable arithmetic.

Service constructs represent the lowest element within the variable mapping tree. However, if a service construct is defined as referencing some external construct, the variable mappings defined within that will override ones of the same names within the standard variable tree.

To access the value of a variable, the character `$` should be pre-pended to the variable name. For variable arithmetic, brackets should be used to enforce precedence.

9.1.3 Defining External Service Constructs

External service constructs, discussed within section 4.5 of the project report, enable common service functionality to be grouped into logical chunks. They allow standard service constructs to be defined within an external file, and referenced to from any node within the system.

By default, external service constructs are defined within the `constructs` directory. All XML files appearing within this directory will be parsed and loaded into the system. Within such files, the top most elements should be the `ConstructLibrary`, containing additional `name` and `description` attributes. These are purely for ease of management and have no conceptual meaning within the system.

Within this top most element, a sequence of service constructs can follow, and need not be all for the same service. For example, a sequence of `FirewallConstructs` can be followed by a sequence of `DNSConstruct` elements. These should be defined as with standard constructs, but using a different naming convention for the `name` attribute. The form `SERVICE::ExternalConstruct:NAME` should be used, hence `DNS::ExternalConstruct:Loopback` would be valid.

In addition to defining variables at the standard service construct level; variables can also be defined at the `ConstructLibrary` level. These are directly above the contained constructs in the variable tree, hence will override any re-declared variables within the importing service or node.

To import the functionality within an external construct in a service description, the following notation should be used:

```
<!-- Just some standard protection. Externally defined. -->
<fw:FirewallConstruct name="Standard Protection"
                      description="External">

      <Mappings>
            <Variable name="block_rule_base"
                  value="$rule_base+1000"/>
            <Variable name="untrusted_if" value="$wlan_if"/>
      </Mappings>

      <ExternalConstruct
            name="Firewall::ExternalConstruct:Protection"/>

</fw:FirewallConstruct>
```

When imported, the whole calling construct will be replaced by the corresponding external declaration. The following segment of code demonstrates how to create a `ConstructManager`, the owner of external service constructs, and reading in all constructs defined within that directory. It also shows how to load function definitions into the system.

```
ConstructManager manager = new ConstructManager("../constructs/",
                                                "../functions/");

manager.readConstructs();
manager.parseFunctions("funclist.xml");
```

9.1.4 Specifying Functions

Function definitions, described within section 4.6 of the project report, are typically defined within the `functions` directory. Function definitions themselves are XSL style-sheets, expecting some valid XML service input, and generating the required output. As they are initiated from service constructs, as with external references, they should output valid service constructs in response.

To define the mapping from function name to XSL file, a single index file within the functions directory is used. As shown in the example at the end of the previous section, the file `funclist.xml` is used. The syntax of this file should be as follows, mapping a function name to the XSL file it represents.

```
<FunctionList>
  <Function name="DNS::Functions:GenReverseSOA"
            file="dns_revsoa.xsl"/>
  <Function name="Global::Functions:Duplication"
            file="duplication.xsl"/>
</FunctionList>
```

As with external constructs, a naming convention should be followed, and they should be defined in the form `SERVICE::Functions:NAME`.

Functions accept XML input that could be any part of a service or node defined within the system. These can either be service construct segments, or explicit elements addressed using paths. Refer to section 4.6.1 of the project report for details on function inputs. The following segment demonstrates how to call a function using a service construct as input.

```
<dns:DNSConstruct name="Reverse"
                  description="Standard reverse resolution">

  <CallFunction name="DNS::Functions:GenReverseSOA"
                onConstruct="DNS::ServiceConstruct:$this/forward">

    <Parameter name="dbfile" value="db.$network"/>
    <Parameter name="with_net" value="$network"/>
  </CallFunction>

</dns:DNSConstruct>
```

Within the above function call, the input is defined as being the service construct named `forward`, on the current node (`$this` is a pointer to the current node). This example also demonstrates the passing of parameters, which are defined in a similar manner to variable mappings.

When a function is evaluated, the selected source XML is passed to the XSL function processor. It is encapsulated within a top level XML element named `FunctionApplication`, containing name space information. In addition, this contains the parameters that may be passed into the function, which can be accessed within the XSL sheet using the following declaration:

```
<xsl:value-of select="/*/@with_net"/>
```

The above will select the `with_net` parameter from the example function call (parameters are simply turned into attributes of the top most element). Within the function, any analysis can be performed on the input XML, as long as the result is valid service XML, contained within a service construct. To ensure clean XML output is obtained, the following declaration should be placed at the head of the XSL style-sheet:

```
<xsl:output method="xml" indent="yes" omit-xml-declaration="yes"/>
```

9.2 Accessing the Package

9.2.1 Loading and Manipulating Package Elements

The package created by this project exports an interface to allow elements to be fetched, processed, modified and then saved back to abstracted XML format. The top most element within the system, the `LAN`, contains all child networks, which in turn contain all child `Nodes`. A `LAN` is contained within a single directory, and one XML file is used to describe the various networks and nodes that are on them. Additional nodes are defined within separate XML files in the same directory.

The following section of code demonstrates how to create a new `LAN` object, and start the parsing process. This will not only parse the networks, but also will iterate through all nodes and load them into the system.

```
 // As defined within section 9.1.3
ConstructManager manager = ...

// Where the example LAN is located
LAN lan = new LAN(manager, "../example/");

// Load up the LAN, with all nodes as well
lan.createFromXML("mylan.xml");
```

Once a `LAN` and all corresponding nodes have been located, either the networks and nodes can be enumerated over directly or a *find* method can be used. The following shows how to obtain the reference to the node called *bill*.

```
// Get bill back out of the network
Node bill = lan.findNodeByName("bill");
```

From this reference, all details of that node and its services can be manipulated. Refer to the JavaDoc package interface documentation for details on how this can be achieved.

9.2.2 Generating XML Output

With any network element of the system, full or partial XML output can be generated. All major objects of the package extend `NetworkElement` and hence import this functionality. The following example demonstrates how to output the partial XML description for a given node reference to standard out.

```
// Output partial XML for this node to standard out
bill.toXML(System.out, 0, false, null);
```

The basic `toXML()` call defined within the `NetworkElement` class takes four parameters. The first specifies an `OutputStream`, hence the following could be used to save the output XML to a given file.

```
PrintStream output = new PrintStream(new FileOutputStream(
               new File("test.xml")));

      // Generate the XML output
      node.toXML(output, 0, false, null);
```

The second parameter indicates the level of indentation to start aligning the output XML to. This is an indicator of units, hence a few spaces. The third parameter specifies whether full or partial XML should be produced. The exact differences between these are described within section 3.1 of the project report, but essentially full XML generation will expand all external construct references and resolve variables and function calls.

The final parameter specifies the variable mappings to use as the top of the variable mapping tree. This is a `HashMap` mapping from variable name to value, and will be

used when full XML generation is required. You may wish to redefine these to customise the behaviour of the generated XML. However, typically the value `null` is passed. This indicates to the element to import all variable mappings defined by its parent into its tree before resolving, and is the normal behaviour. On the example above, it will import all variables from the parent network and LAN.

9.2.3 Performing Translations and Restriction Analysis

Translations and restriction analysis, described in section 5 of the project report, are the processes involved in converting abstracted service configurations to implementation level files.

Initially, the required translations have to be loaded from their source XML description files. These contain both restriction rules and the reference to the external translation XSL style-sheet. The following code demonstrates how to create a new `TransformController`, read in all transforms and then obtain an object reference to a particular translation.

```
// Create a tranform controller and read in all external transforms
TransformController trans =
   new TransformController("../transforms/");

// Read in all transforms
trans.readTransforms();

// Get the translators for the conversions
ServiceTranslator ipfw =
   trans.getTranslatorByName("Firewall::Translator:IPFW");
```

Once a reference to a service translator is obtained, a restriction analysis can be performed using the following call, assuming we wish to use the first service definition within the node *bill*.

```
// Perform the translation
ipfw.validateTranslation((Service)bill.getServices().elementAt(0),
                         System.out);
```

To perform the translation itself, the following call can be used, again using the first service definition within the node *bill*. In addition, this will save the generated configurations to the file *output*.

```
// Perform the translation
ipfw.transform((Service)bill.getServices().elementAt(0),
               new PrintStream(
                   new FileOutputStream(
                       new File("output"))));
```

9.3 Defining Translations

9.3.1 Translations Specification

Translations are specified using two files, a description file containing the restriction rules, and an XSL style-sheet to perform the translation itself. By default, all XML files with the `translations` directory will be loaded. The format of the primary XML file is given below.

```
<?xml version="1.0" encoding="UTF-8"?>

<ServiceTranslation xmlns:xmlnetman="XMLNetMan"
                    xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-
                    instance"
                    xsi:schemaLocation="XMLNetMan
                    ServiceTranslation.xsd">

   <Name>SERVICE::Translator:NAME</Name>
   <Description>TEXT DESCRIPTION</Description>
   <XSLTrans>filename.xsl</XSLTrans>

   <RestrictionGroup>
   </RestrictionGroup>

</ServiceTranslation>
```

The name should be in the `SERVICE::Translator:NAME` format, hence `Firewall::Translator:IPFW` for example. The `XSLTrans` element should point to the actual style-sheet file, located within the same directory. Finally, between the `RestrictionGroup` elements should be a list of restrictions, as described in the following section.

The translation style-sheet should expect to receive some valid XML describing a service description, in abstracted format. It is then the purpose of the translator to convert this into low-level configurations for the desired service. When the service XML description is fed out, it will be between a single top-level `ServiceDefinition` tag, including details on the required namespaces. The translation process should ideally return output that can be directly applied to the intended service implementation.

See section 5 for further details on the translation process.

9.3.2 Restriction Analysis Rules

Restriction analysis is the process used to identify any problems that may occur in the translation from abstracted to implementation level configurations. It is essentially a pre-processor stage, aiming to match sequences of rules against some service description.

These rules are specified within the translation XML file, between the two `RestrictionGroup` elements. Complex sequences of rules can be used to pinpoint problem elements and return information to the user. See section 5.2 for further details on the process, along with detailed guidelines on the rule syntax.

The following example shows how a firewall translation could locate rules that don't have any explicit direction information, indicating to the user that this may be a problem.

```
<!-- Restrictions within the translation to IPFilter -->
<RestrictionGroup>

   <!-- Pick out rules that don't have explicit directions within
        them, as IPFilter likes this. The translation is able to
        duplicate rules, but might not get the correct desired
        result -->
   <RestrictionGroup type="OR"
```

```
            matchOn="Firewall::Filter:FirewallConstruct/Rule"
            reason="Rules without set directions will be duplicated"
            outputXML="no">

        <RestrictionElement name="/!interface"/>
        <RestrictionElement name="/interface@!direction"/>

        <RestrictionGroup type="AND"
          matchOn="Firewall::Filter:FirewallConstruct/Rule/interface">

            <RestrictionElement name="@!direction=in"/>
            <RestrictionElement name="@!direction=out"/>

        </RestrictionGroup>

      </RestrictionGroup>

   </RestrictionGroup>
```

9.4 Compilation and Execution

Compilation of the `xmlnetman.*` package can be initiated using the `make all` command within the `src` directory. More details on the operations supported can be viewed within the `Makefile`. The package comes with a default test network, as described within section 6.3, featuring a collection of nodes and services. Operations can be performed on this test network using the in built demonstrations.

The various demonstrations are shown below, and can be started by executing `make demoX`.

- **Demo 1:** This parses the description of node *sooty* and then generates the partial XML description and outputs to standard out.
- **Demo 2:** This parses the description of node *sooty* and then generates the full XML description and outputs to standard out.
- **Demo 3:** This parses the description of node *sooty* and then uses its firewall description to generate IPFW rules.
- **Demo 4:** This parses the description of node *sooty* and then uses its firewall description to generate IPFilter rules.
- **Demo 5:** This parses the description of node *sooty* and then uses its firewall description to perform a restriction analysis with IPFilter.
- **Demo 6:** This parses the description of node *sooty* and then uses its DNS description to generate BIND8 DNS rules.

To view the code that performs these demonstrations edit the file `src/xmlnetman/test/TestNet.java`.

9.4.1 Tests

This package includes built in automatic regression testing scripts. The actual scripts are detailed in section 10.1 of this document, and can be evaluated using the `make regression` command. Results will be both outputted to screen and saved to file. View the `test/testX.out` and `test/testX.res` for details, where `X` is the number of the test performed.

10. Appendix 2 – Test Script and Sample Files

10.1 Test Scripts

The following section details the various test scripts used for regression testing within the main processing package. All of these tests are automatic, and upon failure details will be reported to the results file. They can be evaluated using `make regression`.

Test	Description	Input/Output	Pass Criteria
1	*Parsing and storage of partial XML specification.* Reads in partial XML description of a node, parses and create object tree. Then generates partial XML descriptions and saves to file.	**Input:** `node1.xml` **Output:** `test1.out` **Results:** `test1.res`	The two files `node1.xml` and `test1.out` should be syntactically identical.
2	*Parsing and storage of full XML specification.* Reads in fully expanded XML description of a node, and creates object tree. Then generates full XML description and saves to file.	**Input:** `node1_full.xml` **Output:** `test2.out` **Results:** `test2.res`	The two files `node1_full.xml` and `test2.out` should be syntactically identical.
3	*Partial XML to generate full XML specification.* Reads in partial XML description of a node, parses and creates object tree. Then generates full XML descriptions and saves to file.	**Input:** `node1.xml` **Output:** `test3.out` **Results:** `test3.res`	The two files `node1_full.xml` and `test3.out` should be syntactically identical.
4	*Variable mappings and expressions.* Tests the functionality of variable mappings and expression resolution.	**Input:** `node2.xml` **Output:** `test4.out` **Results:** `test4.res`	The two files `node2_full.xml` and `test4.out` should be syntactically identical.
5	*Internal package creating of node description.* Creates internal representation of a basic DNS and firewall service, then generate the full XML description of the node.	**Input:** Internal objects **Output:** `test5.out` **Results:** `test5.res`	The two files `newnode.xml` and `test5.out` should be syntactically identical.
6	*Skeleton node description creation and storage.* Creates a skeleton description of firewall and DNS services, and	**Input:** Internal objects **Output:** `test6.out` **Results:**	The two files `skele.xml` and `test6.out` should be syntactically identical.

	then generate the full XML.	`test6.res`	

7	*Standard translation restrictions analysis*
	
	Tests the basic functionality of restriction analysis on a service description.
		Input:
		`node3.xml`
		Output:
		`test7.out`
		Results:
		`test7.res`
			The two files `restres.xml` and `test7.out` should be syntactically identical.

8	*Extended translation restrictions analysis.*
	
	Tests ability of the restriction analyser to fetch restriction source service XML elements.
		Input:
		`node3.xml`
		Output:
		`test8.out`
		Results:
		`test8.res`
			The two files `rest_full.xml` and `test8.out` should be syntactically identical.

9	*Path resolution with translation restriction analysis.*
	
	The source restriction analysis is required to evaluate functions and external references prior to returning the source service XML elements.
		Input:
		`node3.xml`
		Output:
		`test9.out`
		Results:
		`test9.res`
			The two files `rest_dns.xml` and `test9.out` should be syntactically identical.

10	*Function call input and path resolution.*
	
	Tests whether the package can correctly deal with path references using nested function calls.
		Input:
		`node4.xml`
		Output:
		`test10.out`
		Results:
		`test10.res`
			The two files `func_ana.xml` and `test10.out` should be syntactically identical.

11	*Advanced translation restriction analysis.*
	
	Tests advanced features of restriction rules such as more/less than, and count matches of sub-elements.
		Input:
		`node5.xml`
		Output:
		`test11.out`
		Results:
		`test11.res`
			The two files `restcoun.xml` and `test11.out` should be syntactically identical.

10.2 Sample Node Configuration with Firewall and DNS Services

The following XML represents a complete description of an abstracted node. It includes both a DNS and a firewall service.

```
<?xml version="1.0" encoding="UTF-8"?>

<ntman:Node xmlns="http://www.ecs.soton.ac.uk/~src299/xmlnetman/firewall"
            xmlns:ntman="http://www.ecs.soton.ac.uk/~src299/xmlnetman"
             xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
             xsi:schemaLocation="http://www.ecs.soton.ac.uk/~src299/xmlnetman Node.xsd">

        <ntman:Mappings>
                <ntman:Variable name="node_name" value="Sooty"/>
                <ntman:Variable name="interface_1_ip" value="192.168.15.95"/>
                <ntman:Variable name="interface_1" value="rl0"/>
                <ntman:Variable name="interface_2_ip" value="192.168.30.95"/>
                <ntman:Variable name="interface_2" value="wi0"/>
```

```
        <ntman:Variable name="wlan_if" value="$interface_2"/>
        <ntman:Variable name="trusted_if" value="$interface_1"/>

        <ntman:Variable name="network" value="192.168.15"/>
        <ntman:Variable name="network_1" value="192.168.30"/>
</ntman:Mappings>

<!-- Firewall to protect trusted internet against wireless LAN traffic -->
<fw:Firewall>

        <Mappings>
                <Variable name="rule_base" value="1000"/>
                <Variable name="upstream_dns" value="192.168.2.100"/>
        </Mappings>

        <!-- Standard loopback processing -->
        <fw:FirewallConstruct name="Loopback Handeling"
                              description="Protect loopback device">

                <Mappings>
                        <Variable name="block_rule_base" value="$rule_base"/>
                </Mappings>

                <fw:Rule Desc="Allow loopback" RuleID="$block_rule_base">
                        <fw:action perform="pass"/>
                        <fw:protocol type="all"/>
                        <fw:src type="any"/>
                        <fw:dst type="any"/>
                        <fw:interface via="lo0"/>
                </fw:Rule>
                <fw:Rule Desc="Prevent spoofing of loopback"
                         RuleID="$block_rule_base+1">
                        <fw:action perform="deny"/>
                        <fw:log logamount="0" value="true"/>
                        <fw:protocol type="all"/>
                        <fw:src type="any"/>
                        <fw:dst mask="255.0.0.0" address="127.0.0.1" type="ip"/>
                        <fw:interface/>
                </fw:Rule>

        </fw:FirewallConstruct>

        <!-- Just some standard protection. Externally defined. -->
        <fw:FirewallConstruct name="Standard Protection"
                     description="External">

                <Mappings>
                        <Variable name="block_rule_base" value="$rule_base+1000"/>
                        <Variable name="untrusted_if" value="$wlan_if"/>
                </Mappings>

                <ExternalConstruct name="Firewall::ExternalConstruct:Protection"/>
        </fw:FirewallConstruct>

        <!-- Just some standard protection. Externally defined. -->
        <fw:FirewallConstruct name="Standard Protection"
                              description="External">

                <Mappings>
                        <Variable name="block_rule_base" value="$rule_base+2000"/>
                        <Variable name="untrusted_if" value="$trusted_if"/>
                </Mappings>

                <ExternalConstruct name="Firewall::ExternalConstruct:Protection"/>
        </fw:FirewallConstruct>

        <!-- Automatically create the firewall rules required to allow DNS queries
             in on the two networks we serve -->
       <fw:FirewallConstruct name="Dynamic DNS Service"
                     description="Automatically create firewall Rules to allow DNS
                       service operation">

                <CallFunction name="Firewall::Functions:DNSAccess"
                              onPath="$this/DNS(1)/Bindings(1)">
                        <Parameter name="networks" value="$network,$network_1"/>
                        <Parameter name="ips"
```

```
                        value="$interface_1_ip,$interface_2_ip"/>
                <Parameter name="interfaces"
                        value="$interface_1,$interface_2"/>
                <Parameter name="ruleid" value="$rule_base+3000"/>
                <Parameter name="port" value="53"/>
        </CallFunction>

</fw:FirewallConstruct>

<!-- Allow me to query my upstream DNS server -->
<fw:FirewallConstruct name="Forwarding of DNS queries"
             description="Allow me to query my upstream DNS server">

        <Mappings>
                <Variable name="block_rule_base" value="$rule_base+3500"/>
        </Mappings>

        <fw:Rule Desc="Allow DNS queries out to upstream DNS server"
                  RuleID="$block_rule_base">
                <fw:action perform="pass"/>
                <fw:protocol type="other" name="udp"/>
                <fw:src type="ip" address="$interface_1_ip"
                         mask="255.255.255.255" ports="set">
                        <fw:port-num port="53"/>
                </fw:src>
                <fw:dst type="ip" address="$upstream_dns"
                         mask="255.255.255.255" ports="set">
                        <fw:port-num port="53"/>
                </fw:dst>
                <fw:interface direction="out" xmit="$trusted_if"/>
        </fw:Rule>

        <fw:Rule Desc="Allow DNS queries back from upstream DNS server"
                  RuleID="$block_rule_base+1">
                <fw:action perform="pass"/>
                <fw:protocol type="other" name="udp"/>
                <fw:src type="ip" address="$upstream_dns"
                         mask="255.255.255.255" ports="set">
                        <fw:port-num port="53"/>
                </fw:src>
                <fw:dst type="ip" address="$interface_1_ip"
                         mask="255.255.255.255" ports="set">
                        <fw:port-num port="53"/>
                </fw:dst>
                <fw:interface direction="in" recv="$trusted_if"/>
        </fw:Rule>

</fw:FirewallConstruct>

<!-- Deal with other incomming ICMP connections -->
<fw:FirewallConstruct name="Incomming ICMP Connections"
             description="Handle ICMP requests">

        <Mappings>
                <Variable name="block_rule_base" value="$rule_base+4000"/>
        </Mappings>

        <fw:Rule Desc="Allow ICMP queries from our trusted network to us"
                  RuleID="$block_rule_base">
                <fw:action perform="pass"/>
                <fw:protocol name="icmp" type="other"/>
                <fw:src type="ip" address="$trusted_net" mask="$net_mask"/>
                <fw:dst type="ip" address="$interface_1_ip"
                         mask="255.255.255.255"/>
                <fw:options>
                        <fw:icmptype type="echo-request"/>
                </fw:options>
        </fw:Rule>

        <fw:Rule Desc="Allow ICMP queries from our wireless network to us"
                  RuleID="$block_rule_base+1">
                <fw:action perform="pass"/>
                <fw:protocol name="icmp" type="other"/>
                <fw:src type="ip" address="$wlan_net" mask="$net_mask"/>
                <fw:dst type="ip" address="$interface_2_ip"
                         mask="255.255.255.255"/>
```

```
                <fw:options>
                        <fw:icmptype type="echo-request"/>
                </fw:options>
        </fw:Rule>

        <fw:Rule Desc="Allow ICMP replies from wireless LAN to trusted
                network" RuleID="$block_rule_base+2">
                <fw:action perform="pass"/>
                <fw:protocol name="icmp" type="other"/>
                <fw:src type="ip" address="$wlan_net" mask="$net_mask"/>
                <fw:dst type="ip" address="$trusted_net" mask="$net_mask"/>
                <fw:options>
                        <fw:icmptype type="echo-reply"/>
                        <fw:icmptype type="ts-reply"/>
                        <fw:icmptype type="info-reply"/>
                        <fw:icmptype type="add-mask-reply"/>
                        <fw:icmptype type="dst-unreach"/>
                        <fw:icmptype type="source-quench"/>
                        <fw:icmptype type="ttl-exceed"/>
                        <fw:icmptype type="header-bad"/>
                </fw:options>
        </fw:Rule>

</fw:FirewallConstruct>

<!-- Deal with other incomming connections -->
<fw:FirewallConstruct name="Incomming Connections"
                      description="What to do with incomming connections">

        <Mappings>
                <Variable name="block_rule_base" value="$rule_base+5000"/>
        </Mappings>

        <fw:Rule Desc="Let established TCP connections flow through"
                  RuleID="$block_rule_base">
                <fw:action perform="pass"/>
                <fw:protocol type="other" name="tcp"/>
                <fw:src type="any"/>
                <fw:dst type="any"/>
                <fw:options established="true"/>
        </fw:Rule>

        <fw:Rule Desc="Deny incomming connections directly to me from
                WLan" RuleID="$block_rule_base+1">
                <fw:action perform="deny"/>
                <fw:protocol type="all"/>
                <fw:src type="any"/>
                <fw:dst type="ip" address="$interface_2_ip"
                        mask="255.255.255.255"/>
                <fw:interface direction="in" recv="$wlan_if"/>
        </fw:Rule>

        <fw:Rule Desc="Deny all other incomming connections from trusted
                lan" RuleID="$block_rule_base+2">
                <fw:action perform="deny"/>
                <fw:protocol type="all"/>
                <fw:src type="any"/>
                <fw:dst type="ip" address="$interface_1_ip"
                        mask="255.255.255.255"/>
                <fw:interface direction="in" recv="$trusted_if"/>
        </fw:Rule>

</fw:FirewallConstruct>

<!-- Connection requests travelling through us -->
<fw:FirewallConstruct name="Throughput"
                      description="What we allow to travel across us">

        <Mappings>
                <Variable name="block_rule_base" value="$rule_base+6000"/>
        </Mappings>

        <fw:Rule Desc="Allow packets to be routed through over trusted
                LAN" RuleID="$block_rule_base">
                <fw:action perform="pass"/>
                <fw:protocol type="all"/>
```

```
                <fw:src type="ip" address="$wlan_net"
                        mask="255.255.255.0"/>
                <fw:dst type="ip" address="$trusted_net"
                        mask="255.255.255.0" negate="true"/>
                <fw:interface recv="$wlan_if"/>
        </fw:Rule>

        <fw:Rule Desc="Allow all packets back out to wireless LAN"
                  RuleID="$block_rule_base+1">
                <fw:action perform="pass"/>
                <fw:protocol type="all"/>
                <fw:src type="any"/>
                <fw:dst type="ip" address="$wlan_net"
                        mask="255.255.255.0"/>
                <fw:interface recv="$trusted_if"/>
        </fw:Rule>

</fw:FirewallConstruct>

<!-- Traffic originating from this node -->
<fw:FirewallConstruct name="Local traffic"
                      description="Allow outgoing local traffic">

        <Mappings>
                <Variable name="block_rule_base" value="$rule_base+7000"/>
        </Mappings>

        <fw:Rule Desc="Outgoing from trusted interface"
                  RuleID="$block_rule_base">
                <fw:action perform="pass"/>
                <fw:protocol type="all"/>
                <fw:src type="ip" address="$interface_1_ip"
                        mask="255.255.255.255"/>
                <fw:dst type="any"/>
                <fw:interface direction="out"/>
        </fw:Rule>

        <fw:Rule Desc="Outgoing from wireless interface"
                  RuleID="$block_rule_base+1">
                <fw:action perform="pass"/>
                <fw:protocol type="all"/>
                <fw:src type="ip" address="$interface_2_ip"
                        mask="255.255.255.255"/>
                <fw:dst type="any"/>
                <fw:interface direction="out"/>
        </fw:Rule>

</fw:FirewallConstruct>

</fw:Firewall>


<!-- DNS service configuration follows -->
<dns:DNS>

        <!-- DNS wide global variable mappings -->
        <Mappings>
                <Variable name="root_domain" value="toons.foo.net"/>
                <Variable name="NS" value="sooty.$root_domain"/>
        </Mappings>

        <!-- Options - i.e. named.conf in BIND8 : Defines all our zones -->
        <dns:Bindings>
                <dns:Zone name="$root_domain" type="master"
                          file="db.toons.foo.net"/>
                <dns:Zone name="15.168.192.in-addr.arpa" type="master"
                          file="db.$network"/>
                <dns:Zone name="30.168.192.in-addr.arpa" type="master"
                          file="db.$network_1"/>
                <dns:Zone name="0.0.127.in-addr.arpa" type="master"
                          file="db.127.0.0"/>
        </dns:Bindings>

        <!-- All the forward resolution files -->
        <dns:DNSConstruct name="Forward" description="Standard forward
                resolution">
```

```
        <Mappings>
                <Variable name="file" value="db.slybase.homeip.net"/>
        </Mappings>

        <!-- Standard forward lookup -->
        <dns:ForwardSOA match="$root_domain"
                        primaryns="$NS"
                        adminmail="root.$this.$root_domain"
                        file="db.$root_domain"
                        serial="5"
                        refresh="10800"
                        retry="3600"
                        expire="604800"
                        min_ttl="86400">

                <dns:NS match="@" target="$this.$root_domain"/>
                <dns:A match="localhost" target="127.0.0.1"/>

                <!-- Primary network -->
                <dns:A match="sooty" target="$network.95"/>
                <dns:A match="sly" target="$network.90"/>
                <dns:A match="ben" target="$network.101"/>
                <dns:A match="roadrunner" target="$network.254"/>

                <!-- Secondary network (WLan) -->
                <dns:A match="sooty" target="$network_1.95"/>
                <dns:A match="tigger" target="$network_1.125"/>
                <dns:A match="kanga" target="$network_1.150"/>

                <!-- Some CNAME for out LAN -->
                <dns:CNAME match="www" target="ben"/>
                <dns:CNAME match="ftp" target="ben"/>
                <dns:CNAME match="www2" target="sly"/>
                <dns:CNAME match="mail" target="sly"/>

                <!-- Mail Exchanger -->
                <dns:MX match="@" target="mail.$root_domain"
                        priority="10"/>

                <dns:A match="@" target="$network.95"/>
        </dns:ForwardSOA>

</dns:DNSConstruct>

<!-- Use a function to generate the reverse resolution file (SOA) -->
<dns:DNSConstruct name="Reverse" description="Standard reverse
                resolution">

        <!-- Execute the function call on the SOA defined above -->
        <CallFunction name="DNS::Functions:GenReverseSOA"
                      onConstruct="DNS::ServiceConstruct:$this/forward">
                <Parameter name="dbfile" value="db.$network"/>
                <Parameter name="with_net" value="$network"/>
        </CallFunction>

</dns:DNSConstruct>

<!-- Use function to generate the reverse resolution (SOA) for the WLAN-->
<dns:DNSConstruct name="Reverse" description="Standard reverse
        resolution">

        <!-- Execute function call on the forward SOA defined above -->
        <CallFunction name="DNS::Functions:GenReverseSOA"
                      onConstruct="DNS::ServiceConstruct:$this/forward">
                <Parameter name="dbfile" value="db.$network_1"/>
                <Parameter name="with_net" value="$network_1"/>
        </CallFunction>

</dns:DNSConstruct>

<!-- Use externally defined loopback feature -->
<dns:DNSConstruct name="Loopback" description="Pull in external
        definitions">

        <Mappings>
```

```
                                <Variable name="primaryns" value="$NS"/>
                                <Variable name="adminmail"
                                          value="root.$this.$root_domain"/>
                                <Variable name="domain" value="$root_domain"/>
                        </Mappings>

                        <!-- Pull in loopback definition. -->
                        <ExternalConstruct name="DNS::ExternalConstruct:Loopback"/>

                </dns:DNSConstruct>
        </dns:DNS>

</ntman:Node>
```

11. Appendix 3 – Published Material

The following four pages consist of the paper [1] that was presented at the Business Information Systems 2002 conference in Poznan, Poland.