Verinec Translation Module

Verified Network Configuration - Working Paper

David Buchmann

University of Fribourg, Switzerland

Abstract

The Verinec project aims to simplify network configuration. It is based upon an abstract definition of a network and the nodes in that network expressed in XML. Each node consists of its hardware (network interfaces) and a set of services such as DNS server, firewalls, and so on. The abstract configuration is translated automatically into configuration specific to the actual hard- and software used in the network. The simulator part of Verinec allows to check if the configuration will fullfill the desired behaviour prior to really configure the nodes. This paper describes the translation process.

Keywords: Generating network configuration, network simulation, automated verification

1 Introduction

Configuring a network of computers can be complicated, especially in heterogeneous environments. The aim of Verinec (Verified Network Configuration) is to facilitate this.

The Verinec system is designed to abstract from the concrete implementation for network services and focus on the functionality they provide. There exist many different packet-filtering firewalls both in software and as specialized hardware components. Some of them are configured using configuration files with varying syntax, others using Simple Network Management Protocol (SNMP). But they all do mostly the same: Testing on IP packets and based on source, destination and other properties of the packet, they decide whether to accept or reject it.

Using the Verinec application, you can establish the rules once and then generate them for all supported implementation. This allows to exchange one machine for an other and keep the configuration. Verinec can directly configure the target machines (currently only writing the configuration files with copy and scp are implemented, but SNMP and other protocols could be supported).

To help augmenting the confidence into the network configuration, Verinec incorporates a simulator. Using this simulator should reveal misconfigurations before the setup of the machines is really altered. It is envisioned to provide means to create requirement definitions for the desired network functionality that can be automatically tested. Finally it will even be possible to define the requirements for the network on a high level and let the Verinec system find a way to fulfill them and create an appropriate configuration.

For practical use, there will be a part that analyzes existing system setups and generates the abstract configuration. The vision is that an administrator can run Verinec and read in his existing configuration, create the abstract configuration and then deploy it on an other machine with some other operating system and this other machine will behave exactly the same way the first one did. This could be extended to allow users to move there entire system configuration from one system to an other. [1]

Verinec is based on the project NetSim [1].

[1] For example, Linux distributions often have different configuration files layout or different syntax, but the functionality is not different. It would be nice to be able to switch between distributions without having to reconfigure every service each time.

2 Architecture of Verinec

The base of Verinec is the network definition, consisting of “nodes” (Computers, Routers, Appliances, etc.) and network connections between the node’s network interfaces. This data is expressed in Extensible Markup Language (XML) and is abstracted from the actual hardware. A repository infrastructure can store and load named collections of nodes and networks.

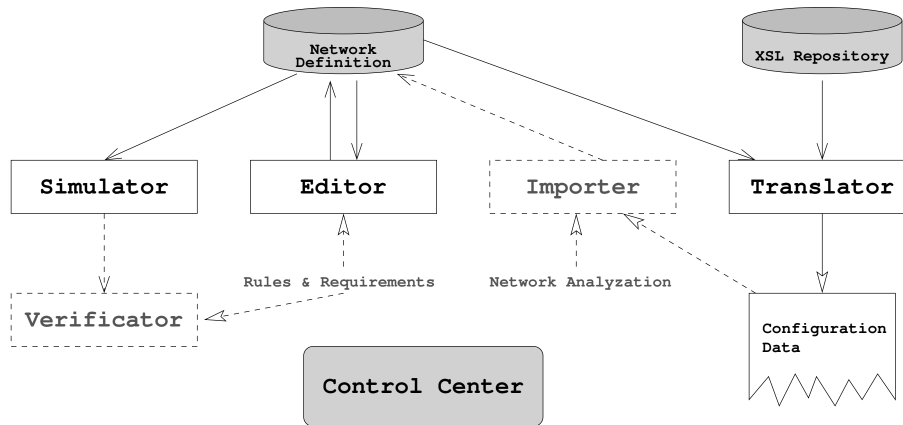

Fig. 1: The structure of the Verinec system.

Figure 1 shows the different modules of Verinec that operate on the data in the repository. The dotted lines denote parts of the system not yet implemented.

- The **editor** displays a network and provides the functionality to alter node configuration and network layout.
- The **simulator** takes the definition to simulate network behavior, the output of the simulation can be visualized in the editor.
- The **verifier** will test if a network fulfills specified requirements, using the simulator.
- The **translator** takes the definitions to create the actual configuration data.
- The **importer** analyzes networks and configuration files to create repository data.
- The **control center** is a graphical application that controls the different modules.

2.1 Network Definition

Networks are built by nodes, which might be workstation and server computers or specialized devices like switches or a hardware firewall. Each node contains a hardware section for its interface configuration and a services section, where all the services provided by this node are configured.

The base of Verinec is the abstracted network definition. Interfaces and services are described using an XML syntax. By studying the different implementations for a service, we tried to deduce a schema that covers most of the features that can be configured in one service implementation. Sometimes, features not existing in other implementations can be simulated using clever translations. If this is not possible, the translator module will create warnings for the user.

Every interface can be connected to networks. Ethernet interfaces may only be connected to one network (there is only one physical cable) while Wireless Local Area Network (WLAN) interfaces (several session-Id’s) and serial interfaces (different phone numbers) can be connected to several networks[2]. Each network is supposed to be connected by a bus architecture, for Ethernet this would typically be a hub. Switches however have to be implemented as nodes with many network interfaces and only layer 2 services.

So far, there are XML schemas and translators for Ethernet, WLAN and serial interfaces and for the services routing, DNS (i.e. bind) and packet-filters (i.e. iptables or ipchains).

[2] Of course, only interfaces of the same type may be connected together.

2.2 Simulation

Part of Verinec is the simulator. It allows to test a configuration before actually distributing it onto the systems.

...

2.3 Translation

The translation module generates instructions or configuration files from the abstract XML data. Using metadata, it selects the correct eXtensible Stylesheet Language Transformations (XSLT) for each service and its selected implementation.

The translation is done in two steps. First, the abstract configuration data is translated into an instruction file that typically contains orders to write files in the target systems native format. The second step is distributing the configuration to the target systems, for example copying config files, doing SNMP connections and pre- and post processing (i.e. stopping and starting services).

Because different implementations might not support everything that can be expressed using the abstract XML configuration, *restrictors* can produce warnings to the user. The transformation module extracts warnings for each feature present in the given configuration that can not be translated for the selected architecture.

3 Network Definition Schema

The core of Verinec lies in the abstracted network definition. All configuration details are stored in XML structures. How nodes and their services are represented is defined by the XML schema `node.xsd`[3]. Which interfaces are connected is stored according the schema `network.xsd`.

3.1 Nodes

A node is any network element, for example a switch, a desktop computer, a specific appliance or a server. Each node has a name, a hardware section for its network interfaces and a services section. An example node definition can be studied in Listing 1.

Listing 1: A sample node

```
<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://diuf.unifr.ch/tns/projects/verinec/node
      http://diuf.unifr.ch/tns/projects/verinec/node.xsd"
  xmlns="http://diuf.unifr.ch/tns/projects/verinec/node">

 <node hostname="pc01">
   <hardware>
     <ethernet>
       <ethernet-binding id='ebna3zt'>
         <nw id='i10' address='192.168.0.55' type='ip' />
       </ethernet-binding>
     </ethernet>
   </hardware>
   <services>
     <dns>
       <variable name="toonsdomain" value="toons.foo.net." />
       <Zone match="$toonsdomain$" type="master"
          primaryns="ns.$toonsdomain$"
          refresh="10800" retry="3600"
          ...
       </Zone>
     </dns>
   </services>
 </node>
</nodes>
```

The node definition consists of two parts. The first part is the hardware section. It defines all interfaces this node has. Currently, you can define ethernet, WLAN and serial interfaces.

A $< hint/ >$ section inside the interface allows to explicitely tell the translator, which hardware interface corresponds to this definition. Otherwise, they are just assigned in order they appear in the config file and in the computer system. Hints can be given for systems with pc architecture (numerated interfaces) or with junos and Cisco appliances (interfaces sorted per pic, port and slot).

By using different session keys, it is possible to bind a WLAN card to more than one network. A modem can also connect to different networks, depending on the number dialed. Ethernet cards however can physically only be connected to one network. To reflect this situations, an interface contains one or more bindings (ethernet interfaces only one, of course). We do not directly connect the interfaces, but bindings they contain. Therefore, it is the bindings which have the unique ID attribute.

With ethernet, it is possible to configure one interface to listen to more than one IP address. But it can still only be connected to one physical network. Virtual networks are on a higher level and can be configured using several nw tags inside of the binding.

[3] The full namespace is `http://diuf.unifr.ch/tns/projects/verinec/node`

The second part are services. Each service is enclosed in a tag with its service name. Please consult the schema for details about the definitions for a specific service.

Variables are allowed anywhere in the nodes document and can be referenced in any attribute. The variables are useful to isolate common information into one place, allowing changes to be more painless. For example, in a network with many clients all having the same dns server, you can define its IP as a variable. If for some reason you have to change that IP, you only have to update it in one place.

To reference a variable, the variable name is written in the attribute, enclosed in dollar signs, like `$variable$`. If you need a $ sign in the output, you have to escape it in the XML like '
$'. The variable name may consist of upper- or lowercase letters, digits and the underscore (a-zA-Z0-9_). Defining a variable is done using `<variable name="variable" value="value" />`. Variables must have been defined, otherwise an error during translation occurs. There exists also a special case of variables that are not defined within the XML document. Variables beginning with a percent sign % are resolved against the java environment. For example, '$%user.name$' will result in the username of the current user.

Something else are macros. A macro is specified in curly brackets and an exclamation point. Currently there exists only one macro, which is !hostname. It evaluates to the hostname attribute of the node within which the macro is. Important about macros is that they are resolved after variable expansion and target resolution took place. If you use a macro in a global variable or a target in tr:typedef, it will be resolved in the context of each place the variable / target is used.

3.2 Network

The network definition is quite simple. Networks are defined by specifying the bindings of interfaces belonging to the same segment. Under the root element 'networks', collections of binding references are grouped in 'network' tags. Of course, only interfaces of the same type may be grouped. The bindings are referenced by their unique ID. Because the binding belongs to one interfaces, which in turn is defined inside its node, it is defined which node is in which network.

A network consists only of interconnected interfaces (hub for ethernet, same session for wlan). Everything on a higher level, e.g. a switch, is considered a node with several interfaces, each belonging to one network.

Listing 2 shows a sample network. It consists of two groups of connected interfaces.

Listing 2: A small network

```
<networks>
  <network name="intranet">
    <connected binding="xyz" />
    <connected binding="uvw" />
    <connected binding="rst" />
  </network>
  <network name="extranet">
    <connected binding="abc" />
    <connected binding="def" />
  </network>
</networks>
```

To know how the network really looks, we have to know the nodes as well. Lets assume we have two clients with interfaces uvw resp. rst, a server with interfaces xyz and abc and a internet router with interface def. An intranet connects the clients to the server, which is in turn connected to the router. Figure 2 shows how this network would look in the network analyser.

Currently there is no notion of network latency (cable length, signal retardation). This would only be important if the simulation would care for the physical behavior of a network connection.

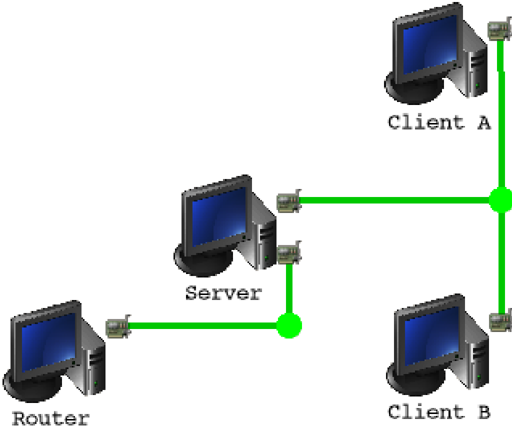

Fig. 2: How the small network could look in the editor.

4 Simulation

5 The translator in detail

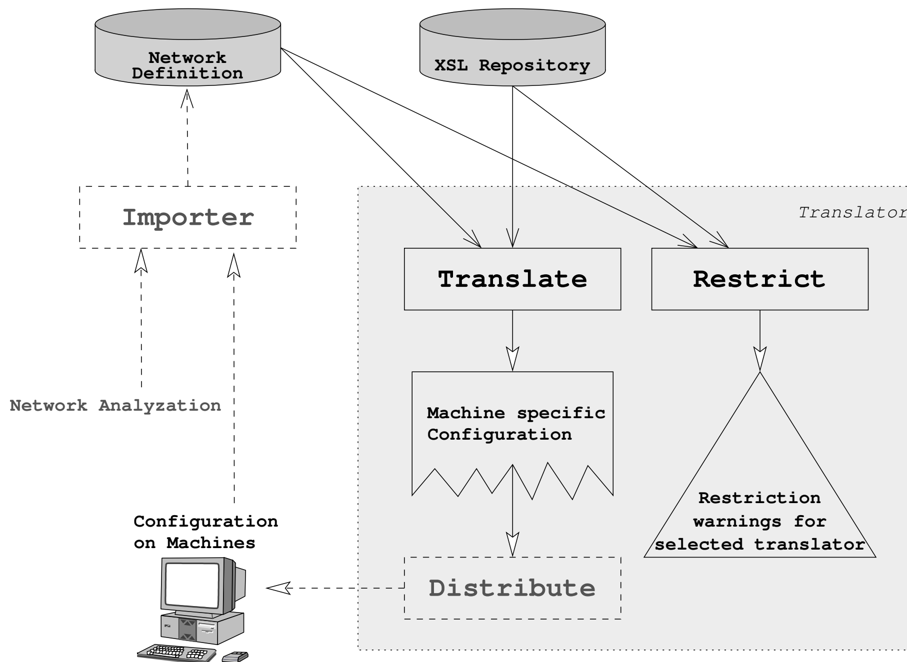

Fig. 3: The translation module.

Figure 3 shows the details of the translation process. Each machine will provide a number of services. There might be many implementations of the same service. For example, packet-filtering under Linux can be done using either iptables, ipchains, ipfilter or some other framework. Metadata tells the translation module which translator to use.

The XSL repository provides *translators* for each service implementation. The translation module applies the appropriate translators to the node configuration, generating system specific configuration orders in XML. Those orders define configuration files or scripts and tell the distribution part how to connect to the target system and what services have to be restarted and other things to be done before and after configuration is modified. To create warnings if some part of the abstract configuration is not supported by the selected translator, the restriction analyzer uses a service and translator specific XSL transformer to scan the configuration for the occurrence of unsupported constructs.

Note that in the translation context, the children of the 'hardware' section (Ethernet, WLAN, serial) are treated the same way services are.

5.1 Translation namespace

To help the translation process, the node must contain some hints. There is a schema for translation metadata, defining a *type* system for selecting the translators and the *targets* to define the method of connecting to the machine to configure.

The prefix for the translation data usually is `tr`, the namespace for this schema is
`http://diuf.unifr.ch/tns/projects/verinec/translation`

5.1.1 Node types and service translators

The translation metadata describes what translator to use for a certain service of a node by specifying the appropriate translator. A collection of service-translator relations is grouped in a type. Nodes can reference globally defined types and they can overwrite the translator for each service as necessary. This allows to change the architecture of each node at any time.

The translators are implemented in eXtensible Stylesheet Language Transformations (XSLT). They are responsible of translating the abstract configuration into a format compatible with the target system.

Listing 1: Type declaration in the translation Namespace

```
<nodes xmlns:tr='http://diuf.../verinec/translation'>
  <tr:typedef def-type-id='xy'>
    <tr:type name='linux-redhat' id='typ01'>
      <tr:service name='ethernet' translation='linux-redhat' />
      <tr:service name='serial' translation='wvdial' />
      <tr:service name='dns' translation='bind8' />
       ...
```

Listing 1 shows an example of a type declaration. Types are located in a typedef element directly under the root nodes tag. The typedef element also specifies a default type to be used for nodes with no type reference. Types should be referenced from nodes, as shown in listing 2. This listing also contains an example of how to overwrite some of the service translators.

Listing 2: A node using typ01 and overwriting some translators

```
<node hostname='machine666'
    xmlns:tr='http://diuf.unif.../verinec/translation'>
  <tr:nodetype type-id='typ01'>
    <tr:service name='ethernet' translation='linux-custom' />
    <tr:service name='wlan' translation='linux-redhat' />
  </tr:nodetype>
  ...
```

Before the actual translation, Verinec resolves the type information to determine which translator to use for a service of a node. The precedence is as follows:

1. Locally defined translator for that service.
2. Translator defined in the type referenced by the node.
3. Translator in the default type if the node does not specify its type.

The services of a node can not be translated into machine instructions without a translator. Each node must specify a translator existing in the repository for every service it provides; if there is no default type specified in the typedef element, each node must specify its type or define all service translators locally.

5.1.2 Targets for services

Similar to the type definition explained in the previous section, there is a system to define how the configuration data gets to the system to be configured. Target tags can contain different tags for means of accessing the system. They can specify a simple cp to copy files to the right directory (i.e. a directory mounted from the server) or an scp to copy configuration files over the network. Other options might be SNMP and similar remote management protocols.

Targets can be defined globally, in the nodetype tag or in a service translator declaration. Globally defined targets can be referenced from the nodetype or service tag.

Listing 3 shows some of the possibilities. Service *ethernet* will have the target 'copy to special' from 'sample type', while *serial* has its locally defined target. The service *wlan* has no specific target, the target of the default type referenced in nodetype is used.

Listing 3: Where to specify Targets

```
<nodes xmlns:tr='http://diuf.unif.../verinec/translation'>
  <tr:typedef def-target-id='tar01'>
    <tr:type name='sample type' id='typ02'>
      <tr:service name='ethernet' translation='linux-redhat'>
        <tr:target name='copy to special'>
          <tr:cp prefix='/special' />
        </tr:target>
      </tr:service>
      <tr:service name='serial' translation='wvdial' />
    </tr:type>

    <tr:target name=copy to temp' id='tar01'>
      <tr:cp prefix='/tmp' />
    </tr:target>
    <tr:target name=copy to pc02' id='tar02'>
      <tr:cp prefix='/mnt/pc02' />
    </tr:target>

    <node hostname='pc02'>
      <tr:nodetype type-id='typ02' def-target-id='tar02'>
        <tr:service name='ethernet' translation='linux-fedora' />
        <tr:service name='serial'>
          <tr:target name='serial'>
            <tr:cp prefix='/mnt/pc02/serial' />
          </tr:target>
        </tr:service>
        <tr:service name='wlan' translation='linux-redhat' />
      </tr:nodetype>
      ...
```

As can be seen from the previous example, there are several places where the target can be specified. The precedence rules make sure that local settings overwrite global ones. Targets inside a service tag always overwrite the ones in node tags and the global default.

1. Nodetype service target child element
2. Nodetype service target-id attribute
3. Global type service target child element
4. Global type service target-id attribute
5. Nodetype target child element
6. Nodetype target-id attribute
7. Global default target set in typedef
8. Write the file into local file system, relative to the local file system root '/'.[4]

There is only one point that might be surprising: tags inside a service tag in a global type have precedence over the tags in the nodetype element of a node. The reasoning is that global types will only specify targets specific to a service if that service needs a special treatment.

[4] This is the default target if there is no explicit target to be found.

5.2 The translation process

Before actually translating a node, Verinec must determine its type. Then it will produce warnings if some features are not supported by the currently selected translator. Afterwards, the translation process can begin. From the repository, the appropriate translators are chosen and applied to the node. The resulting configuration is distributed onto the systems using the targets. Figure 4 illustrates this process.

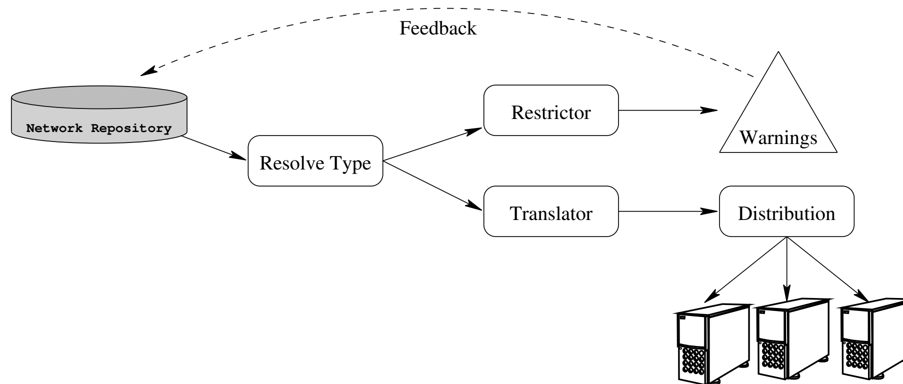

Fig. 4: Translation process details.

5.2.1 Resolve

Each node has to be completely resolved before it can be translated using the XSL transformer. All translator and target information is copied locally into the node's `tr:service` tags. This way, the rest of the translation module doesn't need to care about the type resolution and precedence rules for selecting restrictor and translator and the translators can copy the targets to the output documents.

5.2.2 XSL repository

As mentioned before, the translation is performed using XSLT. A repository provides instances of the JDOM `XSLTransformer` instances initialized to the specified translator. The repository also contains a restrictor for each translator. Both are identified by service name and their own name (name of the service implementation). Restrictors and translators have the same name, but are retrieved calling different methods.

A factory allows to use different implementations of repositories, i.e. for using an XML database. The default implementation reads the XSLT from files in the local file system.

5.2.3 Restrictions for selected type

The design of Verinec takes into account that sometimes, an implementation of a service can not support all features that the user is able to configure in Verinec. A restrictor is an XSLT transformer that extracts the constructs from the configuration data that are not usable with the translator that would currently be used for the service. The restrictors produce human readable explanations for each construct that is not translatable.

Restrictors are selected exactly the same way as translators, ensuring that the system produces the warnings specific to the currently targeted implementation. There must be a *restrictor* for every translator, even if it just produces an empty result document..

5.2.4 Translation

The translators transform services of a node into some implementation specific form. For example, they can produce configuration files in the custom format of a service implementation. Listing 4

shows the example of a configuration output for a network interface in fedora Linux.[5]

Listing 4: Where to specify Targets

```
<configuration xmlns="http://diuf.unif.../verinec/configuration"
            xmlns:tr="http://diuf.unif.../verinec/translation">
 <service name="hardware">
  <tr:target  name="pc07"><cp prefix="/mnt/pc07" /></tr:target>
  <result-file filename="/etc/sysconfig/network-skripts/ifcfg-eth0">
# Ethernet card configured by verinec: ethernet 0 [eth0]
DEVICE=eth0
IPADDR=192.168.0.55
BOOTPROTO=none
TYPE=Ethernet
ONBOOT=yes
</result-file>
 </service>
</configuration>
```

The resulting document is conforming to the configuration schema with the namespace `http://diuf.unifr.ch/tns/projects/verinec/configuration`. A `configuration` is a collection of `service` tags, one for each service. The service may contain a `tr:target` copied from the node data, pre- and postprocessing commands and one or several occurences of `result-file`. This last tag tells the distribution process to write a file to the specified location in the file system. The path is specified as absolute path, but targets can add a prefix to it for more complex scenarios.

5.2.5 Distribution of the configuration

The generated configuration is distributed onto the target machines using the informations in `tr:target`. Currently, there exist only targets for copying files, either somewhere on the machine Verinec is running on or to other machines using scp.

The translators provide a default path which is implementation dependent (e.g. different Linux distributions have been very creative as where in /etc files should be placed). Both cp and scp targets allow to specify a path prefix. This allows for example mounting the target machine file systems into the system of the local machine and just copy the generated files to the appropriate location, relative to the mount point.

For programs that are configured by user editable configuration files, this is about what you will need. Other systems might require SNMP configuration or something else (e.g. for windows). The idea is to generate an XML format to script a SNMP session or to create batch files.

There will also be need for specifying what services the distribution system has to stop and restart when exchanging configuration files. The pre and post tags are declared but need further detailing.

[5] Actually, the stylesheets will add some unused namespace declarations, these have been removed to make the example more readable.

6 Future Development

Verinec is far from complete. In the translation module, we have a rudimentary working translation framework. It will need more polishing and an exhaustive documentation for creating own translators. The final distribution part is yet to be implemented.

6.1 Configuration schemes

Currently, only configuration files are handled by Verinec. It is planned to extend the support to SNMP connections for configuring SNMP-enabled device and to a protocol suitable to configure Windows machines. This will require to add something like `snmp-commandos` and a similar tag for Windows besides `result-file` to the configuratio schema. The distribution part will need libraries to handle SNMP and Windows.

As an additional method to distribute configuration files, a ftp target could be implemented.

6.2 More supported services

Verinec should support many networking services and the translators for the important implementations of those services should be written. For example, we should have:

- user authentication
- DHCP server
- Routing
- SMTP server
- POP3 server
- IMAP server

Others could be done, but are not all very suitable. For example, HTTP servers are probably too complex for Verinec, the differences between the implementations too big to allow for anything efficient.

The network interfaces could also be more detailed. Currently there is simply 'ethernet', but we should separate the different classes (10MBit, Fast Ethernet, 1000MBit) and other topologies as for example token ring could be considered.

The base Verinec framework will need means to integrate own services. Currently, you have to edit the node schema to add a service. It should be possible to add schemas, translators and code for new services and targets.

A Acronyms

XML Extensible Markup Language [2]

XSL eXtensible Stylesheet Language: Used to translate XML documents. There are two different parts: Transformations for changing the format (XSLT) and Formatting Objects (XSL-FO) to layout XML data. Verinec does only use Transformations.

XSLT eXtensible Stylesheet Language Transformations: Transforms XML documents into other XML or plain-text documents [3].

SNMP Simple Network Management Protocol: Industry standard protocol to configure network devices remotely.

WLAN Wireless Local Area Network: Network protocol based on IEEE 802.11

References

[1] XML Network Manager, Simon Chudley, http://www.slyware.com/projects_xmlnetman.shtml

[2] Definition of XML, http://www.w3.org/XML/

[3] Definition of XSLT, http://www.w3.org/TR/xslt