Meta–CLI Configuration Model for Network Device Management

Rudy Deca

A Major Report

in

the Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science

Concordia University
Montreal, Quebec, Canada

January 2003

Your file Votre référence
ISBN: 978-0-494-20787-1
Our file Notre référence
ISBN: 978-0-494-20787-1

Abstract

Meta–CLI Configuration Model for Network Device Management

Rudy Deca

The astounding Internet revolution brings more and more new and sophisticated technologies and services, like : MPLS, VPN, QoS, RSVP, DiffServ, VLANs, bandwidth-on-demand, VoIP, etc. Moreover, the sheer number of elements in a network is skyrocketing. For instance, an ISP may have to deal with hundreds of routers and thousands of interfaces. The diversity and heterogeneity of the network elements, domains, hierarchies, routing technologies, services and management policies gives yet another dimension to the problem.

This manifold complexity poses new challenges to the network engineers and specialists. The error-prone and slow manual device configuration process involves risks like bringing the elements or the systems into undefined states or rendering them unreachable from the rest of the network and is ineffective when faced with the network's fast-growing size and heterogeneity. In this context, an integrated fabric of high- and low-level, complementary approaches is demanded, involving global- and domain-level, business policies, automated configuration, combined with outsourced policies, filtering techniques, fine-grained instance- or device-specific configuration approaches and policy error and conflict avoidance and resolution mechanisms.

The report presents a configuration model which translates the manual command line information into meta-CLI constructs and allows the manipulation and composition of configurations, features, services and parameters, in order to facilitate service activation, support, invoking and monitoring, policy integration at different abstraction levels, allow better control, validation and verification, optimisation, operational efficiency and a more reliable, scalable, flexible and cost-effective configuration of the network resources and traffic.

Acknowledgements

First of all, I would like to thank my mentor and co-supervisor from the Université de Québec à Montréal, Dr. Omar Cherkaoui, for his generous patience, invaluable step-by-step guidance and support especially in the domains of telecommunications and internetworking.

I wish to give all my gratitude and credit to my mentor and co-supervisor from the Concordia University, Dr. Gregory Butler, for his exigent perspective, inspired advices and valuable guidance especially in the fields of software development and methodology.

Last but not least, I would like to mention Mr. Elmi Hassan for his support when I was using the network devices at UQAM's *Networking Laboratory.* In addition, I wish say that I had the privilege of using the equipment and documentation of the *Laboratoire de Téléinformatique.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The exponential growth of the Internet has revolutionised society and technology and is one of the most interesting and exciting phenomena in networking. Two decades ago, the Internet, which is the biggest network of networks, or internetwork, was a research project that involved a few dozen sites and few people had access to a network. Today, the Internet has grown into a production communication system that reaches millions of people in all the countries in the world [8] and computer communication has become an essential part of our infrastructure and culture [21].

Networking is used in every aspect of business, including advertising, production, shipping, planning, billing, and accounting. Consequently, most corporations have multiple networks. Schools, at all grade levels from elementary through post-graduate, are using computer networks to provide students and teachers with instantaneous access to information in online libraries around the world. Federal, state and local government offices use networks, as do military organizations. In short, computer networks are everywhere [8].

According to Network Wizards [47], as of year end 1996, the Internet linked over 60,000 networks, 9.5 million computers [1] and 35 million users in 150 countries [21]. However, the size is not as surprising as the rate of growth. The Internet has experienced exponential growth over two decades, i.e. has been doubling in size every nine to twelve months.

[1] There were 162 million, in July, 2002 [47].

New and sophisticated technologies and network services, like : MPLS, VPN, QoS, RSVP, DiffServ, VLANs, bandwidth-on-demand, VoIP, etc. are available on the Internet and other networks.

In order to deal with the ever-increasing network management exigencies and to perform configurations in a constantly changing system, some new models, mechanisms and tools are required, to simplify, structure and abstract the configuration process.

1.1 Network Configuration Problems

Some preliminary definitions are necessary to set up some aspects of the terminology used in this chapter.

An *internetwork* is a collection of individual networks, connected by intermediate networking devices, that functions as a single large network. *Internetworking* refers to the industry, products and procedures that meet the challenge of creating and administering internetworks [12]. The *Internet* is a network of networks, linking computers to computers sharing the TCP/IP protocols [12].

The problems posed by the network configuration may be expressed in terms of network complexity and from the point of view of the means to manage this complexity.

1.1.1 Network Management

In this paragraph we shall expose some main issues related to the network management.

When speaking of network management, we have some more terms and notions to define from the beginning. A *configuration* is a set of parameters in network elements and other systems that determine their function and operation.

A network *service* is the behavior or functionality provided by a network, network element or host [34]. A service involves: functions to be performed, information required to perform these functions and the information made available by the element

to other elements of the system. Examples of policy actions might include relegating the packet to best effort service, dropping packets, reshaping the traffic, or marking non-conforming traffic in some fashion [36].

A network *policy* is a set of rules to administer, manage, and control access to network resources [34]. Policy can be used to configure a "service" in a network or on a network element/host, invoke its functionality, and/or coordinate services in an inter-domain or end-to-end environment.

The *policy-based management* is the practice of applying management operations globally on all managed elements that share certain attributes [24].

The complexity of the network has several aspects:

- number of elements : routers, switches, gateways, firewalls, etc. For instance, the ISPs manage networks with hundreds of routers and thousands of interfaces;
- fast growing number and quality of the services demanded and deployed on the networks;
- heterogeneity of the composing elements, domains and hierarchies, policies and decision mechanisms; and
- the immense number of configuration commands and parameters.

In this context, configuration management has to tackle multiple challenges: conflicting services, domain policies and overlapping domain hierarchies, opposite effects of business and domain policies and fine-grained, instance-specific configurations, multiple domain or element policies, multiple intra-domain sources of policy creation, antagonist effects of outsourced and provisioned policies combination, possibility of incompatible feature compositions, capable of bringing network systems or elements into weird, indefinable states and possibly rendering them unreachable, the need for error detection, avoidance and resolution mechanisms, for validation, verification and automation of the configuration process.

Due to the network size and heterogeneity, the traditional ways of configuration using the command-oriented configuration approach and, in particular, by sequentially entering the commands, can create a bottleneck effect.

1.1.2 Command–Line Interface Limitations

The *Command-Line Interface* (*CLI*) is the basic means to configure the network devices and consists of sequences of commands. These commands are text sequences containing parameters and values, keywords, options and command switches that are input by the user at the command line to the operating system of the network device. Based on this commands, the state of the device changes according to the service or policy intended by the user.

Some of the major network device vendors like *Juniper*, *Nortel* and *Cisco* companies provide the best-known CLIs used in the networking industry [37] [38].

There are several drawbacks encountered when using the CLI:

- sequential operation of the CLI commands;
- context, argument and result dependencies of the commands, which may create problems when a context, a parameter or another command are not entered correctly during or before the command execution;
- default configuration information is not stored in the configuration files;
- default values are not known by the user;
- default values vary for the same command according to some options or switch parameters;
- there may be no default for a command or parameter;
- context-dependent meaning of some commands, options and parameters; and
- the huge number of commands, options and parameters

1.2 Aim

The aim of this paper is to provide a solution to the problems posed by the translation of the configuration data into higher-level user interfaces that give the user leverage over the manifold complexity of the network resources and traffic configuration.

1.3 Motivation

The need for a business-oriented management requires high-level approaches and modelling techniques, expressed in more functional and service- and policy-based user interfaces. Since one of the main means to perform the configurations is the CLI, the problems posed by its utilisation need to be solved in order to facilitate the global task.

The *configuration file* is a text representation and the result of the device configuration. It is a simple representation of the state of the device, which does not reflect the entire information of the device. If we try to change the parameters or commands directly in these files, we risk messing up everything, because of the CLI complexity, which cannot be summarised by a simple text file. The file text does not have the means to interact in depth with the device, it is just a snapshot.

A configuration process involves both service- or policy-specific information and instance- or device-specific information. The business-oriented network-specific configuration has a coarse granularity, involving sets of commands, whereas the instance- and device-specific information has a fine granularity, involving parameters and commands. Thus, for the latter case, we would need to access and process individual parameters in a text file. Thus, changes in the text of the file should be verified by means of complex algorithms and the resulting overhead would limit the efficiency of this approach.

This shows that we need to construct a *high-level model.* A *data model* is, basically, the rendering of an *information model* according to a specific set of mechanisms for representing, organizing, storing and handling data. It has three parts [34]:

- a collection of data structures such as lists, tables, relations, etc.;
- a collection of operations that can be applied to the structures such as retrieval, update, summation, etc.; and
- a collection of integrity rules that define the legal states (set of values) or changes of state (operations on values).

An *information model* is an abstraction and representation of the entities in a managed environment, their properties, attributes and operations, and the way that they relate to each other [34].

Such a model would reflect all of the inherent complexity, architecture, commands and parameters of the device and will be directly accessible to the user and will work with the configuration files, just like the CLI.

At the same time, a higher-level approach would provide sequences of commands that belong to the same context, configure or activate a service or a feature, or represent a provisioned or outsourced policy. A function of the high-level model is to eliminate some of the drawbacks encountered when using the CLI and the configuration files. For instance, such an interface would spare the user the struggles with the order and values of numerous options and parameters, the changing syntax of the commands, confusions over the meanings of some commands, options and parameters and focus instead on tasks, services, features and policies.

A higher-level approach would also offer more structural coherence and a better combination of complementary techniques and methods, like the combination between the provisioned and outsourced policies, the policy-specific and the instance-specific configuration information, etc. As a consequence of the structuring and abstraction effort captured by a higher-level configuration model, the management policies would become more efficient, reliable, optimised, and the networks would be easier to upgrade, extend and support newer technologies and services.

1.4 Contribution of the Report

The report proposes a business-driven, goal-oriented configuration management solution for the configuration of the network resources and traffic. The major focus falls on some main network elements like the *routers* and *LAN switches*. Based on their CLI, a configuration model is defined and its properties, attributes and operations are explained.

The meta-CLI model is conceived to eliminate some of the drawbacks of using CLI commands and to facilitate high-level, goal-oriented manipulations of the configuration data. The model abstracts the CLI commands, modes, architecture, and contexts and creates the framework for high-level constructs that allow the creation, application, combination and validation of services and configuration features at network and domain level, and provide a means for business-driven network management.

At the same time, the model retains all the flexibility and diversity of the CLI commands, allowing fine-grained access and manipulation of the configuration data like, for example, the instance- and device-specific configuration information. A graphical user interface translates the CLI commands into meta-CLI structures that are easier to manipulate, conserve the CLI architecture, contexts and modes and give easy access to their options and parameters for manipulations and compositions required during the configuration process.

The model takes into account the hierarchical architecture of the CLI modes and contexts, the sequential character and the dependences of the CLI commands, the way in which the default values of the commands and parameters are dealt with by the startup and running configuration files, etc. The model proposes the following approach :

- the CLI commands are translated into meta-CLI *structures* and the network device configuration files are organised into tree *constructs*, which are easy to manipulate;

- the hierarchic structure, the modes and contexts of the CLI are modelled such that the dependencies and the CLI complexity are hidden by the model, while

the configuration process functionality and goal-oriented capability are facilitated and enhanced by the model's user interface;

- the services and configuration features are translated into tree structures called capsules;
- the business-driven management is performed at a high level, through compositions between the device configuration constructs and configuration capsules;
- this composition enables a fast, reliable, simple configuration process, since it eliminated the CLI commands dependencies, the low-level struggle with the syntax and quasi-infinite set of commands, parameters and values of the CLI; and
- the granularity of the CLI commands is preserved; thus, the options and parameters are directly accessible through instance- or device-specific operations, which enable the high-level balanced coordination of the network-specific and instance- and device-specific configuration information.

The *trees* of the Meta-CLI Model provide structures that contain all of the information complexity of a network device configuration and allow both the device- and instance-specific information manipulation through *vertical, intra-configuration* (*intra-entity*) operations and the policy-based management through *horizontal, inter-configuration* operations.

1.5 Layout of the Report

Chapter 2 presents the main features and problems of the command-line interfaces, configuration contexts and files.

Chapter 3 presents a couple of *solutions* to the problems mentioned in the previous chapter.

Chapter 4 presents the *functionality* of the Meta-CLI-based interface.

Chapter 5 establishes the Meta-CLI Model, presents the *architecture, structure, rules and operations* that apply to the configuration of the network devices.

Chapter 6 analyses the characteristics of the CLI and discusses the issues of the *translation* of the CLI commands into meta-CLI constructs.

Chapter 7 illustrates the use of the Meta-CLI Model in a concrete *example*. The services for $MPLS$ and VPN are configured in a network using and composition features provided by the Meta-CLI Model.

Chapter 8 concludes this report with an outline of its *contributions.*

Chapter 2

The Problem

Some of the most relevant aspects of the CLI limitations will be exposed in the following paragraphs, since they are the basis of our analysis and design for a meta-CLI model.

2.1 Size of the CLI

Some of the major network devices vendors like *Cisco* [2] and *Juniper* [37], have created an extremely diverse line of routers and other network products. The CLI is the unifying thread that runs through their product line : virtually all of their products run it. This is both a great advantage and disadvantage. On one hand, when you are familiar with one of the vendor's devices, you are reasonably familiar with them all. Someone using a small $ISDN$ router in a home office could look at a configuration file for a high-end router at an ISP and not be lost. He might not understand how to configure the more esoteric routing protocols or High-end network interfaces, but he would be looking at a language that was recognisably the same.

On the other hand, this uniformity means that just about everything has been crammed into CLI at one time or another. CLI is massive – there is no other way to say it and it has evolved over many years. The CLI is an extremely powerful and complex operating system with an equally complex configuration language. There

are many commands, with many options and if you get something wrong, you can easily take your company offline [19].

For instance, the command reference for *Cisco IOS 11.3* is in excess of 1,000 pages. *Cisco Systems Inc.* publishes a software command summary for each of the IOS versions and these books are as thick as the New York City telephone directory [20]. The "Cisco IOS Software Command Summary" (referred to humourously as the "pocket guide") is $8.5'' \times 11''$ and approximately 900 pages long and it is just a "summary", not a fully detailed manual [18]. Moreover, there are probably over 100,000 pages of documentation of the Cisco IOS. The number of configurable options is downright daunting [18]. To put it mildly, finding what you need to know is a challenge [19].

2.2 Hierarchy of Modes and Contexts

One of the confusing things about working with a router is the notion of a *command context.* Most commands are legal only in limited situations [19]. The configuration modes and sub-modes provide the *context* in which certain commands are legal and others disallowed. It is one way that the CLI tries to prevent you from making mistakes when configuring a router [19]. For instance, the *Cisco routers* offer, by default, two levels of command access : *non-privileged mode* (in Cisco terminology *user exec mode* or simply (*user mode*) and *privileged mode* (in Cisco terminology *privileged exec mode*).

The *user mode* is where you get when you first connect to the router and it does not allow you to edit or view configurations. The `show` commands are limited to a few basic levels. From the *user mode* you can get to the *privileged* with the command `enable`. In the *privileged* mode you can edit configurations. From the *privileged mode*, you can access the global or configuration mode with the command `configure terminal`. In the *global mode* the commands enter directly into the router's configuration and do not require any specific context.

From the *global* mode you can enter several modes, like: the *interface* mode, the *sub-interface* mode, the *line* mode, the *router* mode, the *named access list* mode, etc. (You return from these modes to the configuration and global modes with the commands `exit` and `end` or Ctrl+Z, respectively.) Each one of these modes has its own prompt on the command-line but it does not have it in the configuration files [19]. Thus, the configuration files do not provide the context information.

Even more, the command-line prompt gives sometimes only limited and generic information. For example, the prompt for the interface `(conf-if)#` is one and the same for any interface, and we know that a router can have tens of interfaces.

Another context (called *command context* by [19]) is for interactive use. The user does not need to be in the configuration mode in order to give commands for this context. The commands in this context are not entered into the router's configuration and cannot be included into a configuration file that you upload.

2.3 Dependencies of the CLI Commands

There are a couple of dependencies that add to the complexity of the CLI architecture [27]:

- context dependency;
- argument dependency; and
- result dependency.

2.3.1 Context Dependency

In this dependency relationship, the execution of a command is conditioned by the context in which it is executed. This context is reached by means of executing a hierarchically "superior" command and expressed by a specific prompt string.

For example, consider the following *Cisco IOS* command sequence that sets the *IP address* of a router's serial interface *0* :

```
Router (config) # interface Serial 0
Router (config-if) # ip address 131.119.251.201 255.255.255.255.0
```

Here, the command:

```
Router (config) # interface Serial 0
```

will get the user into interface *Serial 0* where he/she can then enter interface-specific commands like :

```
Router (config-if) # ip address 131.119.251.201 255.255.255.255.0
```

2.3.2 Argument Dependency

In this kind of dependency, a command's success depends on the correct number and sequential order of its options and parameters. For example, in the following command, (which generates a default route into an OSPF routing domain):

```
default-information originate always metric 20 metric-type 1 level-1
```

the argument `level-1` is mandatory and, in consequence, its absence will fail the command [1].

2.3.3 Result Dependency

The execution result of some command may be one of the following :

- *error report* string and *prompt* string, which means that the command failed for some reason;
- *request for additional input*, which shows that the system is pending awaiting for some additional input; and
- *prompt* string, which shows that the execution was successful.

[1] On the contrary, arguments like: `metric` XX, `metric-type` Y are optional – which shows that the command's syntax may vary.

The success or failure of a command execution determines which schedules of subsequent commands should be executed. The decision requires sometimes additional command-line inputs for which the configuration workflow will be pending.

For example, when performing an `exit` command on a *Juniper* router, we encounter the following command dependence :

```
admin@host # exit
The configuration has been changed but not committed
Exit with uncommitted changes ? [yes, no] (yes)
```

which shows that an additional input is required because there are uncommitted configurations.

2.4 CLI Commands Ambiguity

The CLI is not user-friendly; it has a cryptic command-line interface with thousands of commands, some of which mean different things in different situations [19]. The command-line interface is not graceful and is sometimes confusing or ambiguous: many commands do not do what you think they should, and the same command verbs can mean completely different things in different contexts. (This inconsistency is probably a natural result of evolution at an extremely large company with an extremely large number of developers, but it does not make life any easier.)

For example, in the *Cisco IOS*, the `default-information originate` command, which is available in the *router* submode, has different meanings, according to the value taken by the *protocol* switch that follows the `router` command [22].

- when used for the BGP:
  - syntax:
    - `default-information originate`
    - `no default-information originate`
  - meaning:

    * allow (or not) the redistribution of network 0.0.0.0 into BGP;

- when used for the EGP:
  - syntax:
    * `default-information originate`
    * `no default-information originate`
  - meaning:
    * explicitly configure (or not) EGP to generate a default route;
- when used for the IS-IS:
  - syntax:
    * `default-information originate [route-map` *map-name*`]`
    * `no default-information originate [route-map` *map-name*`]`
  - where:
    * `route-map` *map-name* : (Optional) Routing process will generate the default route if the route map is satisfied;
  - meaning:
    * generate (or not) a default route into an IS-IS routing domain;
- when used for the OSPF
  - syntax:
    * `default-information originate [always] [metric` *metric-value*`]` `[metric-type` *type-value*`]` `level-1 | level-1-2 | level-2` `[route-map` *map-name*`]`
    * `no default-information originate [always] [metric` *metric-value*`]` `[metric-type` *type-value*`]` `level-1 | level-1-2 | level-2` `[route-map` *map-name*`]`
  - where:

* `originate`: Causes the Cisco IOS software to generate a default external route into an OSPF domain if the software already has a default route and you want to propagate to other routers.
* `always`: (Optional) Always advertises the default route regardless of whether the software has a default route.
* `metric` *metric-value*: (Optional) Metric used for generating the default route. If you omit a value and do not specify a value using the default-metric router configuration command, the default metric value is 10. The value used is specific to the protocol.
* `metric-type` *type-value*: (Optional) External link type associated with the default route advertised into the OSPF routing domain. It can be one of the following values:
  - 1 – Type 1 external route
  - 2 – Type 2 external route

  The default is Type 2 external route.
* `level-1`: Level 1 routes are redistributed into other IP routing protocols independently. It specifies if IS-IS advertises network 0.0.0.0 into the Level 1 area.
* `level-1-2`: Both Level 1 and Level 2 routes are redistributed into other IP routing protocols. It specifies if IS-IS advertises network 0.0.0.0 into both levels in a single command.
* `level-2`: Level 2 routes are redistributed into other IP routing protocols independently. It specifies if IS-IS advertises network 0.0.0.0 into the Level 2 subdomain.
* `route-map` *map-name*: (Optional) Routing process will generate the default route if the route map is satisfied.

– meaning:

* generate (or not) a default route into an OSPF routing domain.

2.5 Default Values

There are cases when the default value of a command differs according to the subcontext: The command `default-metric`, which is used to configure the default metric has different default values in different contexts: *router rip, router bgp, router igrp, router eigrp, router ospf*[2].

2.6 Configuration Files

An important part of working with a network device is the ability to retrieve, review and save configuration files. It is important to know how a device is configured in order to manage it. A network device maintains two different configuration files : the `startup-config` and the `running-config`, which store (in the NVRAM and RAM) the configurations loaded at startup and currently being used, respectively. The changes made to the router take effect immediately and are made into the running-config file. These changes are also made in the startup-config file only when they are manually saved [18].

The configuration files only show the "deviations" from the base configuration. If a change of a value shows up in the configuration file, the value has deviated from the base setup [18]. Thus, a lot of configuration information is not explicitly presented in the configuration files.

For example, a *Cisco router* has *Finger* services listening on the router even though *Finger* does not show up in the configuration file. The reason there is no entry in the configuration files is that *IOS* enables *Finger* services by default. You would not get an entry in the configuration file unless you disabled *Finger* services, because having the service disabled is a deviation from the default setup.

[2]Besides, as in the previous example, it has also different parameters, when applied to various IP routing protocols (BGP, RIP, IGRP/EIGRP, OSPF).

2.7 Negating Commands

The CLI uses a simple syntax for deleting commands, which consists in adding a `no` in front of the syntax that was used during the command creation, as the example in paragraph 2.4 shows. This command can have sometimes unexpected results, in the sense that it does not do what you think it should do. For instance, this command can have a greater effect than predicted. For example, we might create a filter preventing anyone using *Telnet* to connect to the host located at *200.200.200.1*:

```
Router(config)# access-list 101 deny TCP any 200.200.200.1 0.0.0.0 eq 23
```

The command :

```
Router(config)# no access-list 101 deny TCP any 200.200.200.1 0.0.0.0 eq 23
```

would negate not only the previous one, but *every filter* associated with access list 101. This infringes upon the basic rules of a user interface regarding the usability of the *undo* type of operation, because you cannot just undo *only* what you did. The logic behind this is that you cannot delete a specific entry but you can delete just the whole group. (In fact, the latter command is equivalent to:

```
Router (config) # no access-list 101
```

because it ignores the subsequent parameters in the previous command.)

Chapter 3

State of the Art

Several approaches have been proposed for the policy-based configuration management of the network devices. One way involves the network management protocols and their information databases, like *SNMP/MIB* and *COPS/PIB* [29] [30] [31] [32]. Another way is based on the CLI and the transmission of the CLI commands with protocols like *Telnet*, *SSH*, etc.

These approaches have both their strengths and weaknesses. Thus, the capabilities of SNMP version 1 are over-passed by the Internet growth, whereas its subsequent versions did not have an impact on the Internet community. However, SNMP MIB-based solutions have been provided that use scripts to materialise management policies. The methods that control network devices using the CLI have a maintenance problem, since the changing CLI commands syntax requires modifications in the implementation.

The problem with most of these approaches is that they rely on pre-defined *configuration templates* or *models* they lack the versatility and power of the CLI and restrict the variety and freedom of the user's configuration strategy.

3.1 COPS/PIB

An alternative to SNMP, the *COPS* protocol suggested by the *IETF*, although has valuable features like:

- *transaction management*;
- *security*; and
- *object-oriented message format.*

has also drawbacks, like [27] [33]:

- its *Policy Information Base* (PIB) has been standardised more slowly;
- its *SPPI* (Structure of Policy Provisioning Information) language has been incompletely defined;
- it is unclear whether more complex *PRIDs* (Policy Rule Instance Identifiers), which identify all instances, are legal;
- there are no *context* mechanisms for the instances, as in SNMP; Thus, each instance of a rule class can exist once.
- the PIBs do not support evolution, because the install/remove/notify operations are bound to a MIB class (which corresponds to a MIB table) and the COPS-PR protocol only transfers sequences of values. Thus, changes like addition of an element to a class (table) requires to completely redefine the whole table;
- there is no algorithm that can automatically translate between PIBs and MIBs;
- the advantages of COPS/PIB over SNMP/MIB do not outweigh the importance of a single, consistent and coordinated management approach, based on the existing SNMP; and
- it has not been widely accepted by network device vendors.

3.2 The *Cabletron* Solutions

Method and Apparatus for Defining and Enforcing Policies for Configuration Management in Communication Networks

The method and apparatus proposed by Malik et al. [25] [26] use the *templates* for generating configuration records of network devices of a selected model type. A *database* of models is provided, each model representing an associated network device and including attribute values for the parameters of the associated network device.

3.2.1 Configuration Templates

Templates are used to screen a model in order to retrieve values for each of the attributes and create a configuration record.

The configuration records may be stored in the configuration manager or other storage device, and/or transferred to the preexisting model database for use by a network management system in reconfiguring the associated network devices.

3.2.2 Database Models

The *models* of the database represents an associated network device and includes attribute values for the parameters of the device. A configuration manager accesses a set of model types, each *model type* having an associated set of attributes. The configuration manager creates a template by selecting a model type and one or mode attributes from the associated set of attributes and then screens the selected model with the template to retrieve the values for each of the attributes in the template from the attribute values in the database, to create a configuration record for the model. The configuration model can then be stored, modified, transferred to a model and/or displayed to a user on a user interface.

3.2.3 Model Types

A "model type" is analogous to a "class" and the term "model" to an "object" in object-oriented terminology. Thus, the models are implemented as software "objects" containing both "data" (attributes) relating to the corresponding network entity and one or more "inference handlers" (functions) for processing the data.

3.2.4 Configuration Records

More specifically, the configuration manager enables a user to create configurations with a template. A template is a list of attributes for a device of a certain model type. When creating a template, the configuration manager provides the user with a list of all readable/writable and non-shared attributes for a model type (which includes the specific device). The user then selects the attributes needed for the template which, depending on the purpose of the template, might include a single attribute (port status, for example) or dozens of attributes.

The configuration manager then captures the values of the attributes listed in the template, by retrieving the values from the network management system model. The template functions like a filter, blocking out unwanted attributes (IP address, for example) and capturing the values of those attributes found in the template.

The resulting configuration created with the template contains the attributes from the template and the values collected from the model. The configuration may be stored in the configuration manager or in another storage device.

3.2.5 Operations on Models

The operations on models are :

- capture, that stores all attribute/value pairs, obtained by interrogating the selected models through a template. That is, the value of only those attributes that can be found within the template are obtained by interrogating the model;

- load, which places the values of the attributes listed in the selected configuration into selected models; and
- verify, which compares the model's actual attributes/values with the attribute/value pairs of a configuration.

The data in the database may be used for generating topological displays of the network, showing hierarchical relationships between network devices, isolating a network fault and reviewing statistical information.

3.2.6 Implementations : *Spectrum* and *Spectrograph*

An implementation of a network management system is the "Spectrum" product of the *Cabletron* company. A user interface associated to this network management system is the "Spectrograph", which provides a highly graphical multi-perspective view into the network model. This user interface enables the navigation through a landscape in which cables, networks and even rooms show up as icons that indicate the health and performance characteristics of those elements. These icons can be further queried for additional information.

3.3 Policy-Based Management MIB

The approach [24] defines a portion of the *Management Information Base* (MIB) for use with network management protocols in TCP/IP-based internets. In particular, this MIB defines objects that enable policy-based configuration management of SNMP infrastructures.

The SNMP Management Framework presently consists of five major components:

- An overall architecture;
- Mechanisms for describing and naming objects and events for the purpose of management;
- Message protocols for transferring management information;

- Protocol operations for accessing management information; and
- A set of fundamental applications and the view-based access control mechanism.

3.3.1 Architecture

The main concepts of this component are :

- the policies;
- the management stations; and
- the elements.

3.3.1.1 The Concepts of the Architecture Component

3.3.1.1.1 Policies. The *policies* are intended to express a notion of:

```
if
   (an element has certain characteristics)
then
   (apply operation to that element)
```

The policies take the following normal form:

```
if
    (policyCondition)
then
    (policyAction)
```

A `policyCondition` is a script that results in a Boolean to determine whether or not an element is a member of a set of elements upon which an action is to be performed. A `policyAction` is an operation performed on an element or a set of elements.

3.3.1.1.2 Management Stations. A *management station* is responsible for distributing an organization's policies to all of the managed devices in the infrastructure. The `pmPolicyTable` provides managed objects for representing a policy on a managed device.

3.3.1.1.3 Elements. An *element* is an instance of a physical or logical entity and is embodied by a group of related MIB variables such as all the variables for *interface #7*. This enables policies to be expressed more efficiently and concisely. Elements can also model circuits, CPUs, queues, processes, systems, etc.

```
for each element for which policyCondition returns true
    execute policyAction on that element
```

Each unique combination of policy and element is called an *execution context.* Within a particular execution context, the phrase "this element" is often used to refer to the associated element, as most policy operations will be applied to "this element". The address of "this element" contains the object identifier of any attribute of the element, the SNMP context the element was discovered in, and the address of the system on which the element was discovered.

3.3.2 Policy–Based Management MIB–Defined Objects

Whereas many device characteristics are already defined in MIBs and are easy to include in `policyCondition` expressions (`ifType == ethernet`, `frCircuitCommittedBurst < 128K`, etc.), there are three missing areas: *roles*, *capabilities* and *time.* In order to meet today's needs, the Policy-Based Management MIB defines MIB objects for this information.

3.3.2.1 Roles

A *role* is an administratively specified characteristic of a managed element. It is a selector for policies, to determine the applicability of the policy to a particular managed element. Roles in this model are human defined strings that can be referenced by policy code.

Policy scripts may inspect roles assignments to make decisions based on whether or not an element has a particular role assigned to it. The `pmRoleTable` allows a management station to learn what roles exist on a managed system.

3.3.2.2 Capabilities

The capabilities table allows a management station to learn what capabilities exist on a managed system.

3.3.2.3 Time

Managers may wish to define policies that are intended to apply for certain periods of time. This might mean that a policy is installed and is dormant for a period of time, becomes ready, and then later goes dormant. Sometimes these time periods will be regular (Monday–Friday 9–5) and sometimes ad-hoc. This MIB provides a schedule table that can schedule when a policy is ready and when it is dormant.

3.3.3 Policy Execution Environment

The execution environment has a *terminology* defined and a *procedure* executed in several *steps.*

3.3.3.1 Execution Environment Terminology

The memo [24] defines a terminology for the concepts:

- the active schedule.

  A schedule specifies certain times that it will be considered active. A schedule is active during those times.

- the valid and ready policy.

  A *valid policy* is a policy that is fully configured and enabled to run. A valid policy may run unless it is linked to a schedule entry that says the policy is not currently active.

  A *ready policy* is a valid policy that either has no schedule or is linked to a schedule that is currently active.

- the precedence group.

  Multiple policies can be assigned to a precedence group with the resulting be-

havior that for each element, of the ready policies that match the condition, only the one with the highest precedence value will be active.

- the active execution context.

  An *active execution context* is a pairing of a ready policy with an element that matches the element type filter and the policy condition.

- the run time exception.

  A *run-time exception (RTE)* is a fatal error caused in language or function processing.

3.3.3.2 Elements of Procedure for the Execution Environment

There are several steps performed in order to execute policies in this environment:

- Element Discovery;
- Element Filtering; and
- Policy Enforcement.

3.3.3.2.1 Element Discovery. Sometimes various attributes of an entity will be described through tables in several standard and proprietary MIBs – as long as the indexing is consistent between these tables, the entity can be modeled as one element. For example, the `ifTable` and the `dot3Stats` table both contain attributes of interfaces and share the same index (`ifIndex`), therefore they can be modeled as one element type.

The *element type registration table* allows the manager to learn what element types are being managed by the system and to register new types if necessary. An element type is registered by providing the OID of an SNMP object (i.e., without the instance).

Agents may configure elements for whom discovery is optimized in one or both of the following ways:

- The agent may discover elements by scanning internal data structures; and

- The agent may receive asynchronous notification of new elements.

When an element is first discovered all `policyConditions` are run immediately and `policyConditions` that match will have the associated `policyAction` run immediately.

3.3.3.2.2 Element Filtering. The first step in executing a policy is to see if the policy is ready to run based on its schedule. If the `pmPolicySchedule` object is equal to zero, there is no schedule defined and the policy is always ready. If the `pmPolicySchedule` object is non- zero, then the policy is ready only if the referenced schedule group contains at least one valid schedule entry that is active at the current time.

If the policy is ready, the next step in executing a policy is to see which elements match the policy condition.

3.3.3.2.3 Policy Enforcement. For each element that has returned non-zero from the policy condition, the corresponding policy action is called.

3.3.4 The *PolicyScript* Language

Policy conditions and policy actions are expressed with the *PolicyScript* language.

PolicyScript is intended to be familiar to programmers that know one of several common languages, including *Perl* and *C*. *PolicyScript* is nominally a subset of the *C* language.

If the returned value of a `policyCondition` is zero, the associated `policyAction` script is not executed.

3.3.5 Index information for "this element"

The *PolicyScript* code provides two mechanisms for getting the components of the index for "this element" in a convenient way, so that they can perform SNMP operations on it or on related elements.

The two mechanisms are :

- the $\$n$ token; and
- the *ec* and *ev* functions.

3.3.5.1 The $\$n$ Token

For all OID input parameters to all SNMP Library Functions (but not OID utility functions), the token $\$n$ ("$" followed by an integer between 0 and 128) can be used in place of any decimal sub-identifier. This token is expanded by the agent at execution time to contain the n'th subid of the index for the current element

3.3.5.2 The *ec*() and *ev*() Functions

The *ec*() and *ev*() functions allow access to the components of the index for "this element".

3.3.6 The Library

A standard base library of functions, registered with the name `pmBaseFunctionLibrary`, is available to all systems that implement the specification.

The library contains four types of functions:

- SNMP library functions;
- Policy library functions;
- Utility functions; and
- Library Functions.

3.3.7 The Schedule Table

The *policy schedule table* allows control over when a valid policy will be ready, based on the date and time.

A policy's `pmPolicySchedule` variable refers to a group of one or more schedules in the schedule table. At any given point in time, if any of these schedules are active, the policy will be ready and its conditions and actions will be executed as appropriate.

3.4 X–CLI and Wise <TE>

Another approach used by the network administrators for the network management uses software systems implemented on CLI commands of the network devices.

Those systems have a software component written in a script language which translates a policy into a sequence of CLI commands which are sent to the network devices using some transmission protocol, like *Telnet.*

3.4.1 Overview

The *X-CLI* [27] is a CLI-based solution which eliminates the dependency of the policy translation on the CLI command syntax, thus separating the software from the CLI syntax. X-CLI was proposed as a module of the *Wise<TE>* policy management server for traffic engineering using MPLS, VPN and QoS, developed at the *Electronics and Telecommunications Research Institute* in Daejon, Korea [28].

Administrators apply policy rules (by means of a GUI), which are delivered, through a network management transmission protocol and some proxy, to the network devices, where they are translated and *materialised* into CLI commands. X-CLI is an XML wrapper API for CLI, which defines the syntax of the XML template to represent a group of CLI commands and provides the means to load the template, pass arguments and send generate CLI commands to the network devices.

X-CLI takes into account three dependency types encountered in the set of CLI commands, namely the *hierarchical, argument* and *result dependencies.*

3.4.2 XML Representation of CLI Commands

The concept of *XML template*, which corresponds to the *function* in general programming languages like *C++*, represents the hierarchical, argument and result dependencies between the software and the CLI command syntax and separates the software implementation from the policy-to-CLI translation.

3.4.3 XML Template

An XML template is loaded into the memory by the X-CLI API and *materialised* into a sequence of CLI commands by passing the required arguments. The XML template is an hierarchy of *<cli> </cli>* tags which have attributes that express the CLI commands dependencies and are described in the *DTD* (*Document Type Definition*) of the *<cli>* tag: *tag*, *command*, *prompt1*, *prompt2*, *error*, *always*, *ainput*, *ainresponse*. The *containment* relationship between the *<cli>* tags expresses the *hierarchical dependency*.

3.4.4 Attributes of the *<cli>* Tag

The purposes of the above-mentioned attributes are succinctly described below :

- *tag* : uniquely identifies the *<cli>* tag;
- *prompt1*: a string sent by the server to the client before the latter starts sending the message (to the server);
- *prompt2*: a string sent by the server to the client to stop the latter's sending the message;
- *command*: a stringified CLI command, containing "required" or "optional" keywords and "$"–prefixed formal arguments;
- *error*: error notification string;
- *always*: flag indicating that a CLI command can be executed in spite of the execution failure of the previous CLI command;

- *ainput*: request for the additional input; and
- *ainputresponse* : response for the "ainput".

3.4.5 Monitor String

The *monitor string* is a regular expression-like string that may appear between *<cli> </cli>* and analyses the response sent by a network device to a CLI command execution. The results are "pattern-matched" with the monitor string by converting the latter into an *NFA* (*Non-deterministic Finite Automaton*) and are accessed through output parameters.

3.4.6 X–CLI API

In order to facilitate the implementation of the connection pooling mechanism and relieve the overhead of including the connection procedure in every XML template, two template types are provided:

- *DCPF* (*Device Connection Procedure File*)
- *CLIF* (CLI File)

The *DCPF* is a login procedure-dedicated template, whereas the *CLIF* is a normal CLI command template. The X-CLI API loads the template and parses it into a tree topology of *CLI* C++ objects, then sends the *materialised* XML template to the network device and processes the response with a monitor string. The *XCLI* class has a `set_argument ()` and a `send ()` in order to materialise and send the template.

The results of the materialisation is a sequence of instances of the `CLI_transformed` C++ class which basically includes all the `CLI` class attributes and has an additional attribute *BTEF* (*Branch Target for Execution Failure*) to represent the result dependency. Every materialised CLI command has two possible branch targets, which are chosen according to the following scenario:

- if it is a success : the branch target is always the next materialised CLI command; and

- otherwise : the branch target is the next sibling CLI command.

3.4.7 Wise<TE>

Wise<TE> is an integrated server system that incorporates with a MPLS-based ISP backbone, and provides management, verification, analysis and optimization functions for MPLS TE with network operators. Since *Wise<TE>* not only manages *LSPs* (*Labelled Switched Paths*) and traffic trunks, TED (*Traffic Engineering DB*), per-LSP traffic statistics, network topologies, routing protocol information and path computation algorithms (SPF & CSPF) but also performs link/node failure simulation and global usage optimization, the system can play critical role for real-world deployment of MPLS TE.

Wise<TE> works on typical *Unix* systems, and currently can manage *Cisco* and *Juniper* routers. For the sake of inter-module communication, CORBA technology is utilized. A RDBMS and a LDAP server are required to efficiently manage various network information. A Java-based GUI provides an integrated management environment.

3.5 The *Intelliden* Products

Intelliden Corporation provides solutions and tools for business-driven device management, which eliminate the error-prone manual configuration and automate the configuration process [41]. This approach enables the implementation, monitoring and enforcement of business policies in the network and network-based applications.

3.5.1 *R-Series* Software Suite

Intelliden's *R-Series* software suite [35] features a patent-pending configuration workflow solution, which:

- automates the device configuration (thereby dramatically reducing the number of manual configuration errors);

- can control different devices from different vendors (using the same Web-based GUI or XML over Java API, which significantly reduces the complexity in managing heterogeneous devices);
- presents a single common interface rather than multiple device-dependent interfaces;
- guarantees the integrity if the configuration process: if the configuration rules are violated, the R-Series software suite discovers those changes made outside of its control;
- creates *audit trails* and *reports* which help discover bad logic and misconfigurations before damage is done;
- simplifies the *version control* by using multiple versions of a configuration of a device that are stored in a directory. A former configuration is recorded as a previous version;
- has a common repository, specifically, a *master directory* which contains policies that determine the end-user rules.

3.6 The *Directory Enabled Networks* (*DEN*) and the *Directory Enabled Networks – Next Generation* (*DENng*)

The *Directory Enabled Networks* (*DEN*) has been proposed by the Distributed Management Task Force Inc. [48].

DEN (as well as *DENng*) is described [42] [43] [44] [45] [46] as a network and service management approach and a holistic method to manage services according to business policies. It consists of an object-oriented information model, a mapping to several data models and is based on systems and networking standards – including path and bandwidth issues.

3.5.2.1 Object–Oriented Information Modelling

The *DEN* uses object-oriented techniques. Thus, it:

- abstracts the networks, subnetworks, devices, ports, users, applications, locations and services as *objects* that are modelled into classes;
- uses a combination of standard-based directory and object-oriented data models;
- uses the object-oriented information modelling to represent management entities in an environment and their inter-relationships;
- contains relationships that model the dependencies between the managed objects of the system;
- equates different functionalities from different devices to each other and to appropriate classes of services; and
- abstracts different vendors' policing and shaping mechanisms with different command syntaxes and side-effects.

3.5.2.2 Intelligent Services and Intelligent Networks

DEN views the network as a provider of intelligent services rather than a collection of individual interfaces, it abstracts the network into a *set of intelligent services*, changing it into an *intelligent network*. Thus, *DEN*:

- offers an alternative to current management approaches like SNMP and CLI, which mostly view a large-scale network as a collection of individual interfaces to the thousands of disparate routers, switches and other network devices, and which offer no *service* concept, but rather lets the administrator work on individual interfaces;
- transforms the network – from a set of "knobs" affecting the traffic – into a "service-oriented network" (i.e., a set of services available to, and used by, the application);

- enables network interoperability: applications with different GUI, running on different platforms and using different APIs, can share and re-use data;
- allows the end users to: (a) use a same set of services no matter when, where and how they log on to the network; (b) have a personalised view of the network, according to their job, rank and role in the company.
- models the network (as a whole), as opposed to the "silo approach" to provisioning, which concentrates on activating a particular service (without regard to other services already deployed over the network) and: (a) can send you into an endless loop of configuring one service only to harm another; (b) can mean device-configuration conflicts (a single device can have multiple, competing configurations).
- constructs an intermediate layer of network services that can be mixed and matched in building-block fashion;
- provides two things for that purpose: (a) an abstraction layer that enables the application to request network services (rather than using commands on a device-by-device basis); (b) a policy framework that allocates to each application the appropriate network resource and adjusts to the changing business environment;
- equates different commands and operations present in different devices into a common set of commands that can be assigned to different classes of service;

3.5.2.3 Layered Policy Models

DEN enables the policy-based network management. On this account, *DEN*:

- defines also a set of data models which describe how to implement the information models into repositories like directories and relational databases;
- is built as a set of layered models, where each layer has its level of abstraction;
- provides:

  - an extensible *information model*, which: (1) translates the semantics and behaviour of the objects into "middleware"; and (2) defines a "framework of classes" that can represent: the current state of the (managed) object, the settings that can change the object's state and the policies that control and apply the settings to the objects; and
  - a set of *data models.*

- assigns policies to respond to business rules to automatically control device configuration and prioritise network services;
- defines a single repository that contains archived device configuration files;
- has a directory that logs new devices to the network and performs network moved, adds, changes and deletes;
- maps business rules and service level agreements (SLAs) to a common *set of policies* (which automate the process of end-to-end network management);
- has as its top-most layer the *Policy Core Information Model* (*PCIM*), (described by RFC 3060), which defines:
  - a common structure and representation of policy, that is independent of technology;
  - the semantics of policy through a set of classes and relationships, that represent a *policy* in the form of a *condition* clause and an *action* clause; and
  - a set of extension models, each focused on a specific technology (e.g. QoS and IPsec).
- the policies are contained in a *continuum of policies*, which is a set of inter-related policies (rather than just one), corresponding to different policy abstraction layers, tailored to specific domains of users (e.g. business and systems administrators who talk in business terms, as opposed to network programming

developers, who talk in terms of "WFQ queuing parameters" and "WRED thresholds").

3.5.2.4 Policy Mapping and Language

There are six *constituencies*, or levels, that define *DEN*'s *policy mapping and language*:

- the *business view*, which is concerned with the implementation of the processes, guidelines and goals and often takes the form of SLAs;
- the *system view*, where the administrator translates the business requirements to system operation;
- the *administrator view, technology- and device-independent*, which translates the overall requirements into a networking-specific form (avoiding to choose a specific technology, like the *Differentiated Services* – *DiffServ*, etc.);
- the *administrator view, technology-specific and device-independent*, which translates the third level into technology-specific implementation (e.g.: include devices that can handle *DiffServ* and devices that cannot);
- the *device-specific view*, which translates the chosen implementation into a device-type-specific form; and
- the *instance-specific view*, which takes into account the *software version* and *memory configuration* and develops a network device-specific configuration.

3.5.2.5 Finite State Machine

DEN has a combination of layered policy models that intersects with the user, resource and service models, to construct *finite state machines* that represent the full life-cycle of the device management and control network configurations. In this context, policies are used to control when, where, who and how the configurations are changed, by modelling these changes as *state transitions* in the finite state machine.

3.5.2.6 Architecture

DEN's conceptual architecture features several constituents, among which the following may be mentioned:

- the intelligence is distributed among various components of the managed system, using *Policy Decision Points* (PDPs) and *Policy Enforcement Points* (PEPs);
- multiple PDPs are necessary for different applications from different vendors. (For instance, a *logon policy*: logging into a network requires several functions like: IP address allocations, user- and group-specific QoS and security treatment allocation.) A PDP can provide control over one or multiple functions;
- each PEP can have different capabilities (e.g.: a firewall, a router and a content switch);
- *middleware* is used to implement the semantics and behaviour. The static portion of the business rules are stored in the directory, whereas the dynamic portion (involving relationships between entities, resource allocation, etc.) is stored and implemented in the middleware;
- a *publish/subscribe bus* enforces the intelligence distribution among the system components;
- *policy proxies* interface the legacy components (that use SNMP or CLI, for instance, and that the PEPs understand), to the policy systems;
- there are a *policy language* and *policy protocols* of the system;
- the *domain knowledge* consists of a *logical repository*, constructed from many specialised repositories like the directory and the relational database systems;
- policies are stored in *repositories*; and
- a *policy console* provides the graphical and programmatic interface for defining, editing and managing policies;
- a *broker* is used for policy translation.

3.5.2.7 Objectives and Benefits

Several objectives and benefits of the *DEN* approach can be mentioned:

- simplify IP device configuration – which leads to increased operational efficiency and lower management costs;
- ensure that consistent policies are applied to all network elements – which brings optimisation benefits, like:
  - automatically obtain bandwidth without disrupting other services;
  - *router optimisation*: establishment of SLAs with customers that give priority to certain types of delay-sensitive traffic;
  - *content optimisation*: disk mirroring is used to make copies of information in order to save customer data during system outages. Disk mirroring admits small delays and is thus provided as a *service* and supplied with appropriate *policies* that *prioritise* it.
- reduce the gap between the network devices' functionality and the services required by the applications – which leads to faster service creation.

Chapter 4

Meta–CLI Interface

4.1 The Text Challenge

The network device configuration is done by means of CLI, through commands, which are string sequences, given as input to the system. The constraints and checks that enforce the syntactic rules and other CLI information come as a response to the input text. This "reactive" or "interactive" approach leaves it to the user to search in a big pool of commands, parameters and values and find the seemingly right ones, in view of a configuration task. In the case of configuration files, the problem is more complex. The files, rather than store all of the commands executed on an equipment, store some of the latest executed commands and parameter values, more exactly, those, which are a "deviation" from the standard default state of the system.

Since the configuration file is a text file, it could be edited manually, offline (like in figure 4.1) but, in this case, the only way for it to be checked for correctness is to be loaded into the device as the running configuration. This trial-and-error approach is risky and error-prone, since the CLI interactive checking is not performed.

As a consequence of the textual nature of the commands and files, the parameters in the configuration files cannot be uniquely and securely identified, accessed and manipulated. The interaction with the text in a configuration file would not be much safer as, but it would certainly be more dangerous than, performing cut-

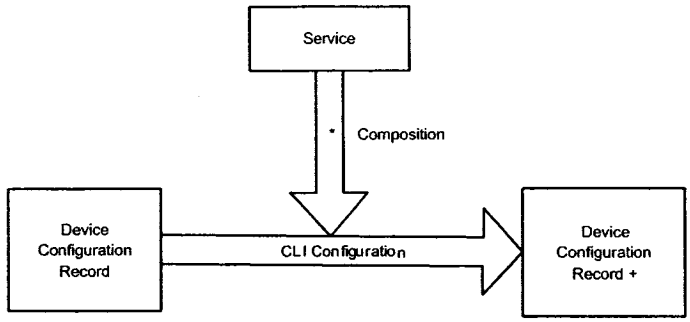

Figure 4.1: CLI-Based Configuration Process

and-paste in a Latin text without knowing the grammar. Using the CLI to access parameters is not a best solution either, considering its context-dependent nature.

4.2 The Idea of the Meta–CLI Interface

The idea of the configuration model is simple. Instead of interacting with the device's information by text-only commands and files, we can *translate* the commands and the files or other records into abstract structures, do the configuration, add or activate services according to our goals and using the rules of composition provided by the abstract structures and *reconvert* the configuration result into commands and configuration files or registers (figure 4.2).

4.3 Goals and Features of the Meta–CLI Interface

In this way, we achieve several goals :

- the abstract structures are specially designed to do some checking for us "actively" rather than "reactively", i.e. it could "suggest" us a list, menu, etc. of current choices, instead of responding negatively to wrong inputs;
- using the configuration files for configuration compositions becomes more feasible because less risky and error-prone, more transparent, and direct;

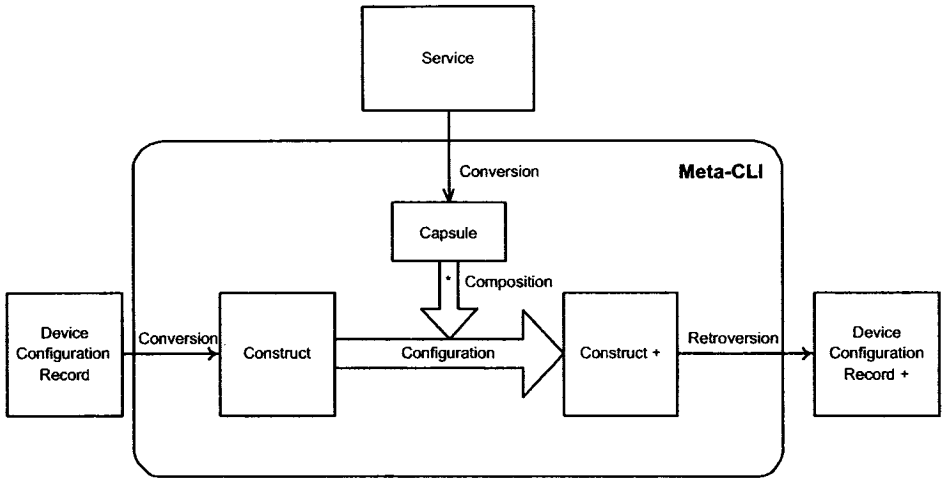

Figure 4.2: Meta-CLI-Based Configuration Process

- the abstract interface provides us with CLI-specific information like :
  - commands' or sub-command's context, mode or sub-mode;
  - list of context-specific commands;
  - command for accessing contexts;
  - commands' syntax, parameters, arguments, options, keywords; and
  - parameters' default values, legal values, etc.
- the abstract structures provide a means to uniquely identify, access and securely manipulate the parameters; and
- the abstract structures allow entity manipulations, in which the elements of the entities match automatically and are composed individually.

4.4 Meta–CLI Topology

The proposed model uses the tree topology, which is a common, universal topology. In this way, the model may be used in conjunction with other tools and models and has a wide potential are of applicability. Many topologies, including the networks,

may be "transformed" or "converted" into tree topologies and manipulated by means of this model.

4.5 Meta–CLI Interface Concepts

The abstract structures are tree-like, model-specific, structures and have several hierarchical levels. The command tree puts the parameter information in the inner nodes and leaves and thus ensures unique identification, access, matching and composition of the command information. The context tree maps the context or mode hierarchy of the CLI. In this way, the abstract structures have a "twofold" tree-like aspect. The trees have several hierarchical levels :

- element;
- branch; and
- entity.

We can extend this hierarchy by adding the *data* level, which is a sub-element level that deals with the *type*, *value*, etc., ignoring the element's *linkage* (i.e. parent and children), and the *network* level, which is a super-entity level that maps the network into an tree, where the domains are descendants and the backbones are ancestors.

The *element* is a *node* in a tree that translates a command name, parameter, argument, option or keyword into node's data and the relationship with the rest of the command components into *descendant elements* ("children" nodes). The element can have zero or more descendants under control and can a parent that controls it. The elements are the smallest tree-like unit in a tree and have a recurrent nature, since the descendants may have their own descendants and the ancestor may have its own ancestor.

The *ancestor* and *descendant* properties are transitive and thus, by way of recurrence, we can deal with *branches*. The *top* element of a branch controls its elements. The *topmost* element in an entity with many branches is its *root*. The *root* has the

Meta-CLI Tree	Level	Tree				
			Branch		Entity	
	Type	Element	Command Tree	Context Tree	Capsule	Construct
CLI Concept		Command Name/ Parameter, etc.	Command	Context/ Mode	Service	Device Config File

Table 4.1: CLI/Meta-CLI Modelling Concepts.

entire *entity* under control. Thus, the element and the entity are the extreme levels of the branchings. The entity can be a device entity, called *construct* or a service entity, called *capsule.*

The *construct* translates the collection of configuration information of a network device, available in some record as, for instance, a configuration file. The *capsule* translates the collection of configuration information of a Internet service, policy or feature. We have thus the following translation relationship between the CLI concepts and the Meta-CLI Model concepts, as shown in table 4.1.

4.6 Stages of the Configuration Process

The Meta-CLI-based configuration process consists of the following stages:

- the device configuration information, stored in some record, for instance, in a config file and containing command instances, is read in;
- the Meta-CLI tool has some lists of command definitions and context dependencies, default values, etc. (please refer to figure 4.3)
- the Meta-CLI tool thus identifies the commands and their contexts from the file or record and accesses the parameters' values;

- the Meta-CLI tool has a collection of generic, predefined command trees that translate the command definitions and another collection of generic predefined context trees that translate the names of the commands specific to each context;
- the model *converts* the actual command instances from the device configuration record into actual tree instances, obtaining thus the construct, based on the generic command and context trees and taking into account the parameters' values from the command instances of the device configuration record; .
- the service is a predefined module or group of commands, that is provided by the tool interface and selected and customised by the tool user;
- as in the case of the device configuration record, the Meta-CLI tool uses its command definitions and context-dependencies lists to identify and put in place the commands and their parameters' values;
- with the aid of the generic command and context trees, the information from the previous step is converted into effective trees and the capsule is thus obtained. Not all of the capsule's information might be in place, depending on whether more specific information must be added individually, later on, for each instance or device;
- the composition and manipulation process is performed in the Meta-CLI with the construct and capsule, resulting in some value-added construct;
- the value-added construct is retroverted into a device configuration record through a process that basically inverts the direction and sequential order of the steps and operations performed during the conversion process (i.e. the first five steps described above);
- based on the collections of generic command and context trees, the model recognises the tree instances of the construct and identifies the parameters, command names and contexts in the tree; and

- with the aid of the command definitions and context-dependencies lists, the effective information from the tree instances is retroverted into CLI command instances and stored in a device configuration record, such as a configuration file.

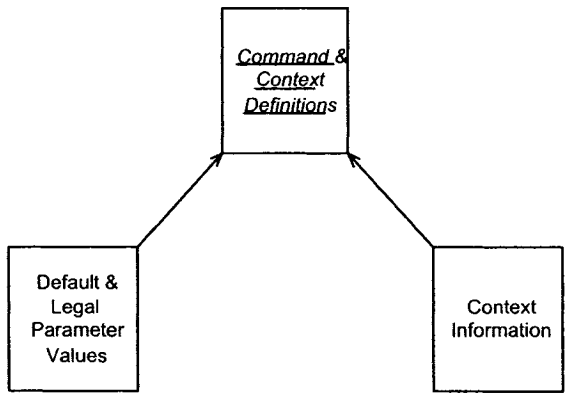

Figure 4.3: Command and Construct Definitions

Schemata of the translation process involving the construct and the capsule are depicted in figures 4.4 and 4.5, respectively.

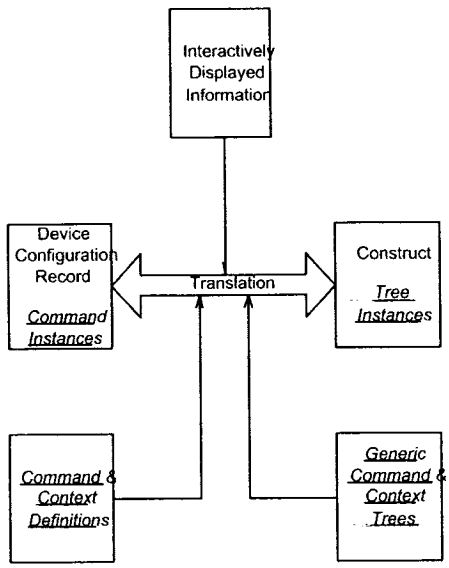

Figure 4.4: Construct Translation

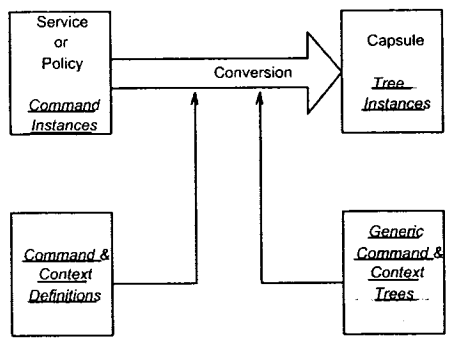

Figure 4.5: Capsule Translation

Chapter 5

Meta–CLI Model Concepts

5.1 Overview

The configuration model proposed by the Meta-CLI Model places the information into tree structure models and defines operations to retrieve, modify, add and remove data. The Meta-CLI models have an architecture that contains several hierarchical *levels*: the data (which resides within the node), the element (consisting of a node and its links towards its children), the branch and the entity. Yet another perspective has three structural *layers*: value, linkage and element layers.

The access and manipulation of the data is based on the *identification* and *matching* mechanisms. The information is accessed by query operations and modified either *directly* within the configuration (i.e. *intra-entity*) or through *inter-entity* compositions (which involve two or more configurations).

A *composition algebra* or *language* is further defined, that describes the composition operations that can be applied to different levels and layers of the proposed configuration model. These operations are the "vectors" or "immediate means" of the configuration that must be outlined on a higher level of the configuration model based on the business rules, policies, services and functions that concur to the configuration process, and using these formal "means".

The compositions are described for the node elements – since they are the basic constituents of the configuration model's tree structures – but they are recurrently

applicable to branches and entire trees. The composition operations are explained for the three layers and illustrated with examples.

5.2 Rationale

The problems mentioned in chapter 2 can be solved by means of a structured configuration model that allows the composition of the features and services the with the device configuration information. As mentioned in section 2.2, the configuration files do not provide the context information that the command-line offers. There is no prompt and no syntax checking of the commands in these files. The default values are not stored in the files. Thus, we must know the syntax and the default values of the commands before editing them in the files and this requires a structured model and a set of rules for the structured, policy-based configuration.

In order to manipulate and control the configuration information, we need to store and organise the data according to a logical, structured model. The CLI commands and structure must be abstracted and modelled in order to obtain a specific data organisation that allows the composition of the configurations. An important feature of the model is that it can store information about the *context*, the *syntax*, the *effective* and *default* values of the parameters.

5.3 The Tree Data Structure and the Meta–CLI Model's Configuration Tree

The Meta–CLI Configuration Model trees (in short, *trees*) are based on, but are different from, the tree data structures. The tree data structures are generic data structures, whereas the Meta-CLI trees are modelling structures and are specific to the Meta-CLI Model. Since there are differences between them, we need to stress out the difference. The trees are particular types of trees, or potions of these trees. Some features of the trees have been used and refined in order to accommodate the

translations of the CLI commands, context-dependencies, default and legal values and all of the rest of the relevant CLI configuration information.

Therefore, there are certain similarities and differences between those structures, and we shall highlight some of them. The *Meta-CLI Model tree* refers generically to an *element*, a *branch* or an it entity. There is a functional difference between the tree structure notion of *node* and that of *element*, in the Meta-CLI Model tree: essentially, the latter has more features and functionality than the former. Some of the differences are enumerated below.

Thus, the element (of an tree) has a recurrent potential and controls its descendants, whereas the node does not have these features and is just an element of the tree. If the element's recurrence is used, the descendants contained in a branch are involved (accessed, modified, manipulated, etc.). The data, element, and branch levels are implemented at the node level.

The tree data structure has nodes and each node contains an object. Finding a node is based on identifying the object. When changing the node contents, i.e. the object, we also change the means to recognise it. More specifically, the node's object is included in the path of the children. Even if we keep an id for the node unchanged, changing the value of another field of the object, changes the path of the children.

Besides, the tree functionalities (at node and root levels) are insufficient for our purposes. The Meta-CLI Model adds recurrence to its logic, enabling in-depth, or multi-level, control and manipulation of the data. Speaking in terms of trees, the Meta-CLI Model is based on branch manipulations, i.e. the information is dealt with from an element, through its descendant elements, up to all its *leaf elements.*

The Meta-CLI Model takes the tree structure and adds an immutable *type* to the node. The changes are only made to the object's *value.* The Meta-CLI Model may allow cycles, e.g. when a command like `router` brings the user from the *config* into the *router* context, whereas the command `exit` sends him/her back from the *router* context to the *config* context.

The possible existence of multiple contexts for an element brings in the spotlight various virtual functionalities. For instance, the same data may be referenced rather

than being copied in different contexts. This feature may be used for integrity verification and consistency checking.

5.4 Architecture of the Meta–CLI Model

The Meta-CLI Configuration Model has a *multiple level hierarchy* and a *layered structure.*

5.4.1 Hierarchical Levels

The hierarchy levels are the *data*, the *element* the *branch* and the *entity* levels (according to the number of elements involved and whether the links are ignored or taken into account):

- the *data level* – deals with the data inside an element, i.e. the *type* and *effective value* and, in subsidiary, with the *default value* and the value *range* or *enumeration*;
- the *element level* – on top of the element's *data*, deals with its descendant *linkage*, i.e. the *elements* that are its *direct descendants*;
- the *branching level* – deals with hierarchies or branches of elements; and
- the *entity level* – deals with global configuration entities that have an identity and represent translations of *device configuration files* or *services, policies* and *configuration features.*

5.4.1.1 Data Level

The *data-level* includes the following :

- *type*, which is a string and is immutable;
- *effective value* (in short, the *value*) which may be a string, a numeric value or other data type and can be set or *reset* during the configuration process;

- *default value*, (in short, the *default* which is an immutable value, to which the *effective* value is, occasionally, *reset*; and
- *value range or enumeration*, specify the *legal* values that can be assigned to the *effective* value (and which include the *default* value).

The *type* and *value* are the basic data information, whereas the rest are auxiliary. Compositions involve the basic data, whereas the auxiliary data help in the configuration process.

5.4.1.2 Element Level

There is a subtle distinction between *node* and *element*: the *element* includes the *node* and its *links* to other elements. A *link* is a hierarchical relation between two elements, and involves an *ancestor* ("parent") and a *descendant* ("child") element. The ancestor element controls its direct descendants, i.e. it can change them (but not vice-versa).

According to the existence or non-existence of the ascending or descending links of an element, we can classify the elements into :

- *edge* elements, which can be further divided into:
  - *roots*, which lack ancestors;
  - *leaves*, which have no descendants; and
- *core* elements, which have both ascending and descending links.

The element is a tree *en miniature*. It is inherently or potentially recurrent. Since it has control only over the descendants, it can be considered the basic functional component of the branchings and trees from the two levels presented in the following paragraphs. If the operations applied on the element are effectively recurrent, the control of the element extends to all of its direct and indirect descendants. In terms of visual representation, it represents an *upside-down* tree, since the *top* element is the *ancestor*. In written representation, it stretches horizontally, with the *top* element on the *left-hand* side and its *descendants* indented on the *right-hand* side.

5.4.1.3 Branch Level

The *branch* level can be obtained from the previous level, the element level through element *recurrence.*

The *branching* accounts for the element and its direct and indirect descendants. To be more accurate, we may distinguish it form the more restrictive *branch*, which is a *branching* that reaches the *leaves.* The indirect descendants are related to an element through *chains* or sequences of descendant relationships.

The *lineage* of an element represents a chain of ancestors, starting from the *root.* A *path* is a chain of data, starting from the *root* and tracing down the links of an element. If the descendants of an element have different types, their link in the path may omit the value. If the data do not uniquely identify some elements, the linkages may be used to differentiate between the paths of the descendants. This issue is further discussed in the *Identification* section. The path can be *absolute*, when it starts from the *root*, and *immediate*, when it contains only data from the element's *parent.* The immediate paths that compose an absolute path are separated by slashes (/), e.g. :

```
Router1:/configure/interface/FastEthernet:'2/1'
```

The absolute path and the branch of an element are somewhat complementary : the former reaches through the element's *lineage* towards the *root* element, whereas the latter reaches through the *descendants* towards a *leaf* element.

5.4.1.4 Entity Level

The tree level can be obtained from the previous one, by starting from the *root* and thus *encompassing* the entire tree.

We deal at this level with *hierarchic*, *tree* structures that are identifiable entities containing all the related elements descending from a root. There are two kinds of entities:

- the *device configuration construct*, (or, in short, *construct*), which *translates* a device configuration *file* and other configuration information (*default* values, "showed" i.e. displayed information, etc.); and

- the *encapsulated service, policy or feature structure*, (or, in short, *capsule*) which translates a service, policy or configuration feature.

There is no structural distinction between the construct and the capsule. Thus, the capsule may be considered a special case of construct, more specifically, an incomplete configuration (in the sense that it could not be translated back into a valid configuration file). The capsule lacks the basic features, commands and contexts contained in the construct and its role is to add or modify the features on the construct. Therefore, the distinction is solely functional. Thus, we may use a construct instead of a capsule and vice-versa.

The interaction between the construct and the capsule is asymmetrical, namely the capsule will usually modify the features inside the construct rather than vice-versa. Since we are finally interested in the resulted construct, the terminology in centered on the construct and its metamorphosis.

5.4.2 Layered Structure

There are two *structural layers* within the higher levels (the elements, branches, entities), according to whether the values are changed or the descendant links are added or removed. Since these two layers may be overlapped, we get actually *three* structural occurrences:

- a *value layer*, in which the values of the elements are modified;

- a *linkage layer*, in which child elements are deleted from, or added to, the parent; and

- an *element layer*, in which the *value layer* and the *linkage layer* manipulations are simultaneously performed, i.e. the layers are overlapped.

The structural layers correspond to the hierarchy levels. For instance, a value-layer operation does data-level manipulations (on data, elements, branches and entities), whereas a linkage-layer operation does branch-level manipulations (on elements, branchings and trees).

5.5 Syntax of the Meta–CLI Model

We may express the Meta-CLI Model using the concepts of formal languages.

5.5.1 Data Level

On the data level, we have :

```
<data> ::= <type> <value> <default> [<range>| <domain>]
```

where :

- <type> is a string;
- <value> is a string;
- <default> is a string;
- <range> is an (ordered) pair of numerals; and
- <domain> is a string or numeral enumeration

The node has several mandatory fields like the *value* and the *default*, and other optional fields, like the *range* or the *domain* :

5.5.2 Element Level

On the element level, we have :

```
<element> ::= <data> [; <element>*]
```

where :

- <data> is the element's data;
- the square brackets ([...]) show that the sequence <element> * is *optional*;
- the semicolon (;) represents the (descendant) *link*; and
- the asterisk (*) shows the *multiplicity*.

We do not need to represent the ascending element, since this is represented in the parent's linkage, as a descending link.

5.5.3 Branch Level

On the branch level, we have :

```
<branching> ::= <data> ; [<element> | <branching>]*}
```

where :

- <data> is the *top* element's data; and
- <element> is a *descendant* element.

5.5.4 Entity Level

On the entity level, we have :

```
<entity> ::= <construct> | <capsule>
<construct> ::= <name> <branching>
<capsule> ::= <name> <branching>
```

Here, the branching starts at the root.

5.6 Representation of the Meta–CLI Model Concepts

The trees may be represented in a graphical way, based on their architecture.

5.6.1 Element Representation

An element can be represented in *compressed*, *condensed* or in *expanded* form. For better visualization, the three representations may be preceded by different icons, like: "+", "×" and "–", respectively.

5.6.1.1 Compressed Representation

The *compressed representation* of an element will display, say – when clicked, on one line, the *type* and *value* of the element:

```
+ <type>:<value>
```

5.6.1.2 Condensed Representation

The *condensed representation* of a form will display, on a line, the *type and value* of the element and of its descendants :

```
× < type >< value > [< type >< value >]*
```

This representation is important because the user condenses, say – with a double click, the tree, in order to see all the parameters on one line. Thus he/she sees the *command tree* just like a *CLI command* on the *command line.*

5.6.1.3 Expanded Representation

The *expanded representation* gives full freedom to access individual parameters. It displays on a line the element's *type and value* and, on subsequent lines, *tabulated*, the *descendant* elements :

```
- <type>:<value>
    - <type>:<value>
    - <type>:<value>
      ...
```

The expansion of an element may be performed in different degrees, depending on whether its descendants are represented *compressed, condensed or expanded.* We may wish to expand only the direct links, a few or all of the indirect links, along the branch stemming from an element.

5.6.2 Branch Representation

The branch representation does not critically differ from the element one, except for the descendants aspect: the branching condenses or expands the *indirect* descendants too, rather than just the *direct* descendants, as the element.

5.6.3 Entity Representation

The entity's representation uses its `<name>` and its (compressed, condensed or expanded) root element:

```
<name> [:/ <element>]
```

5.7 Identification and Matching

5.7.1 Element Identification

An *identifier* of an element is a sequence of strings that uniquely identifies the element. The *types*, *values*, *paths* and *indices* are used during the identification process, according to the concrete situation and the kind of operation to be performed.

5.7.1.1 Identification Modes

On one hand, according to whether the path is taken into account during the identification, we can have two identification modes:

- *path-free*, when the absolute path is not part of the identifier; and
- *path-based*, when the absolute path is part of the identifier.

For instance, the identifier `ip address:'1.1.1.1'` may occur in *several paths* of a construct: *router*, *address-family*, etc. On the other hand, several data may be required by the identifier, in order to ensure more accuracy.

5.7.1.2 Identification Levels

According to the means utilised for the query (element's type and value, linkage, absolute paths) we can have several possible *identification levels* :

- *type-based*, which takes into account the type of the element;
- *data-based*, which is based on type and the value of the element; and
- *element-based*, which checks the type, the value of the element and involves supplementary means of recognition, such as (the types and values of) specific descendant elements and/or indices [1].

The *path-free* identification is used for intra-entity operations. The *type-based* identification is used for *leaf elements*. In this case, we compose the values of these elements. The *data-based* identification is used for *parent* elements because we usually want to change their *descendants* and keep their *values* unchanged.

The *element-identification* is used for the same purposes as the type identification, when the *absolute path*, *type* and *value* do not provide a *unique* identification to the elements. In this case, the type and value of additional *descendants* provide the identification uniqueness. The immediate path of an element is provided by its ancestor's identifier.

5.7.2 Element Matching

The elements' identifiers are compared in order to find out whether they *match* or not and to apply the appropriate operations. Two elements *match* when their respective identifiers are equal. Two identifiers are equal when the corresponding strings that compose those identifiers are equal.

[1] Indices are meant for recognition rather than matching.

According to the identification types involved, we have several modes:

- path-free; and
- path-based.

and levels of matching:

- type-based;
- data-based; and
- element-based.

5.8 Classification of the Configuration Operations

We may speak of a *configuration process* that involves and modifies *configuration entities* by means of *configuration operations.*

The operations can be classified in several categories:

- *query operations*, where match-based element collections are selected;
- *direct intervention operations*, where trees are directly modified (elements are added, removed, values are set, etc.); and
- *composition operations*, where trees are combined.

These categories interact and superpose throughout the configuration process. For instance, we can match trees and obtain element and construct pools that we modify by means of compositions (i.e. set values and add or remove elements).

5.8.1 Query Operations

The query operations are based on element matching, involve an identification condition, require selected trees and result in element selections or existence confirmation. According to the type of matching, the result of a query may be a selection

of a single element or a collection of elements. The modifications performed by the other types of operations (direct manipulations and compositions) may take place during the search operation.

5.8.2 Direct Operations

In this category we may have direct, interactive operations like : the creation, deletion, permutation (reordering), etc. of descendant elements, value setting, type change, object-contents display, etc. There is a close relationship between the matching and the type change and whether the elements are leaves or core elements, more specifically, the factors involved in this relationship should be compatible within the configuration operation.

Thus, we usually match the types and values of core elements without changing them, when we recurrently search through the elements of a tree and match the types and modify the values of the leaf elements. Conversely, if we change the core elements' values, we should be aware that the identifications used by the queries might be affected and need to be adjusted accordingly.

5.8.3 Composition Operations

In this category, the query and direct operations are performed sequentially on elements found along the branches of the composing elements or constructs. The composition operations are covered in the next section.

5.9 Types of Composition Operations

Operation can be classified according to different criteria:

- *hierarchical* levels;
- *structural* layers; and
- *number of entities*,

involved.

5.9.1 Hierarchical Level Classification of the Operations

From the point of view of the number of levels on which the changes are performed, there are four categories of operations, which correspond to the hierarchical levels :

- *data-level operations*, which are performed on the *value* of an element;
- *element-level operations*, which are performed on the *value* and *linkage* of an element, non-recurrently;
- *branch-level operations*, which are performed at element-level, recurrently; and
- *entity-level operations*, which are performed on the elements of an *entity*.

5.9.2 Structural Layer Classification of the Operations

The configuration operations can be classified according to the structural layers involved:

- the *value-layer operations*, which are performed on the value layer;
- the *linkage-layer operations*, which are performed on the linkage layer; and
- the *element-layer operations*, which are performed both on the *value* and the *linkage* layers.

There is a strong connection between the *value* and the *linkage* operations. Thus, a *value* operation can *reset* a value of an element, i.e. can set the effective value to the default value and a *linkage* operation can remove the element because its value is set to default. It should be mentioned that, in CLI, the configuration files contain the commands that represent "deviations" of the system from its default configuration.

5.9.3 Classification of the Operations According to the Number of Operands

According to the number of operands (entities) involved in the composition operations, there are two kinds of operations : *inter-* and *intra-entity* configurations. The *inter-entity configurations* refer to the compositions that take place between two or more entities, while the *intra-entity configurations* refer to the data manipulations that take place within a configuration or capsule entity.

The inter-entity configuration compositions may be considered *horizontal* because they involve similar elements from different constructs and capsules, whereas the intra-entity operations may be considered *vertical*, since they involve different elements from the same entities. Furthermore, usually the inter-entity configuration operations implement *policy-* or *service-based* configurations whereas the intra-entity configuration operations perform *device-* and *instance-specific* configuration operations.

5.9.3.1 Inter–Entity Operations

These operations act at all the hierarchical levels and may take place between two or three entities. In the former case, an entity, usually a construct, plays both roles of an operand and of the result of the operation, whereas in the latter case, the result of the composition performed with two operands is placed in a third configuration entity.

When dealing with two operands, we can also use a shortcut notation. Thus, since an entity is both *operand* and *result*, and we do just one manipulation, we can condense the assignment symbol (=) and the other operation symbol, for example the generic *value composition* symbol (*) like in *C*, *C++* and *Java*:

$$\boxed{(A = A * B) \Longleftrightarrow (A *= B)}$$

From the two-operand compositions, between a capsule and a construct, there is just a step to the three-operand compositions. The conceptual focus falls on the

two-operand compositions since from their combinations can be obtained the three-operand ones. Suppose, for instance, that we perform some operation $(*)$ on three operands; then :

$$\boxed{(A = B * C) \Longleftrightarrow (A = B; A *= C)}$$

5.9.3.2 The Intra–Entity Operations

The *intra-entity operations* are performed within an entity and basically do the following:

- change the *value* of one or more descendants (by means of a composition operation);
- *remove* an element and its branching; and
- *create* and *add* a new descendant to an element.

Unlike direct operations, these composition operations require recurrent search and matching, and therefore, they need as input an element and a value.

For instance, we can edit a new element and compose its value with the elements of another entity or branch .

5.10 Composition Algebra

The *element-level operations* will be tackled in this section since the element is the basic, central concept in the Meta-CLI Model. The other level operations are deductible form this one and do not need further attention. Thus, the *data level* is illustrated by the *value-layer* operations, the *element level* corresponds to the *non-recurrent element-layer* operations, the *branch level* corresponds to *recurrent core element-layer* operations and the *entity level* corresponds to the *root element-layer* operations.

The names of the operations have been chosen such that they express the composition interaction from the construct's perspective (rather than the capsule's).

We shall further discuss the three layers of the element-level composition operations: *value, linkage* and *element* layers.

5.10.1 Value–Layer Operations

As already mentioned, the *value-layer* operations concern the *data* level.

The *value operations* compose the values of the *leaf* elements, whereas the *core* elements' values serve just for *identification* and *matching*. Value operations are generally symbolised by an *asterisk* $(*)$:

$$\boxed{* \Longleftrightarrow [\times | + | \cdot | < | > | / | \stackrel{*}{=}]}$$

5.10.1.1 Overview

In the tables 5.1 and 5.2, is presented a summary of the definitions of several value operations $(*)$. The table is not exhaustive. The sampled operations are not always independent, i.e. an operation might be obtained from a combination of others. In the choice of these operations the usefulness and implementation feasibility have played a major role.

The symbols have the following meanings :

- A and B represent two (matching) elements whose descendants are composed;
- Φ represents the *default value* of an element (which, in its turn, maps the *parameter* or *command default value*; we make the assumption that both elements are supposed to have the same defaults; if the default values are different or unknown, we can use a hypothetical default value thereby preserving the validity of the operation definitions); and
- α, β and γ represent *non-default, distinct* values of the elements.

We assume that the forms A and B have similar *types* and we compose their *values*. For the case when they have *different* values, the symbols used for these values are distinct (α, β), whereas a common symbol was used (γ) if they have *equal*

values. The case when both *effective values* are equal to the *default value* (Φ) is simple (because the result is a *default* value, for any operation) and has not been presented in the tables.

Table 5.1 presents the *exclusive addition* ($\times$), the *symmetric addition* ($+$) and the *identical selection* ($\cdot$), and table 5.2 presents the *subtraction* ($/$), the *conservative addition* ($>$) and the *substitutive addition* ($<$) and *assignment* ($\stackrel{*}{=}$) value operations.

	Operand		Operation		
Element	A	B	$A \times B$	$A + B$	$A \cdot B$
Value	α	β	Φ	Φ	Φ
	γ	γ	Φ	γ	γ
	α	Φ	α	α	Φ
	Φ	β	β	β	Φ

Table 5.1: Definitions of the exclusive addition ($\times$) symmetric addition ($+$) and identical selection ($\cdot$) value operations.

	Operand		Operation			
Element	A	B	A/B	$A > B$	$A < B$	$A \stackrel{*}{=} B$
Value	α	β	α	α	β	β
	γ	γ	Φ	γ	γ	γ
	α	Φ	α	α	α	Φ
	Φ	β	Φ	β	β	β

Table 5.2: Definition of the subtraction ($/$), conservative addition ($>$), substitutive addition ($<$) and assignment ($\stackrel{*}{=}$) value operations.

5.10.1.2 Precedence Symbol

A value composition operation can be locally preempted by a value of an element, using a *precedence sign* ($), according to the table 5.3 (A and B are the operand ele-

ments and a and b are their values, respectively). The functionality of the *precedence*

	Operand		Operation
Element	A	B	$A * B$
Value	$\$a$	b	a
	a	$\$b$	b
	$\$a$	$\$b$	$a * b$
	a	b	$a * b$

Table 5.3: Definition of the precedence operator ($).

operator may be extended from the use within leaf elements to other levels, by means of recurrence.

For instance, it can be applied to *core* elements. Thus, the *precedence* sign used with a core element's value, will indicate the *precedence* operation for all its leaf descendants. The core element's value is not composed and does not need a *precedence* status.

When used at the *root*, the *precedence* nears the behaviour of the *value assignment* ($\stackrel{*}{=}$). The *precedence* preempts the *value* operations, because it's more "local" and "manual", whereas the *value* operations are more "general" and "automatic".

5.10.2 Linkage–Layer Operations

Linkage-layer operations are complementary to the value operations within the element. Linkage operations are generally symbolised by a small circle (o):

$$\boxed{\circ \Longleftrightarrow [\oplus | \otimes | \odot | \oslash | \stackrel{\circ}{=}]}$$

Some of them are not "stand-alone", i.e. they need to be composed with some *value* operation in order to be applied to the configurations.

5.10.2.1 Overview

In the table 5.4 is presented a summary of the definitions of several linkage operations.

The symbols in the figure have the following meanings :

- A and B : two (matching) elements whose descendants ("links") are composed;
- $\exists$: the (operand) element's link has a match among the other element's descendants; and
- $\nexists$: the element's link does not have a match among the other element's descendants.

Table 5.4 presents the *symmetric addition* ($\oplus$), the *exclusive addition* ($\otimes$), *identical selection* ($\odot$), *subtraction* ($\oslash$) and *assignment* ($\overset{\circ}{=}$) linkage operations.

	Operand		Operation				
Element	A	B	$A \oplus B$	$A \otimes B$	$A \odot B$	$A \oslash B$	$A \overset{\circ}{=} B$
Link	$\exists$	$\exists$	$\exists$	$\nexists$	$\exists$	$\nexists$	$\exists$
	$\exists$	$\nexists$	$\exists$	$\exists$	$\nexists$	$\exists$	$\nexists$
	$\nexists$	$\exists$	$\exists$	$\exists$	$\nexists$	$\nexists$	$\exists$

Table 5.4: Definition of the symmetric addition ($\oplus$), exclusive addition ($\otimes$), identical selection ($\odot$), subtraction ($\oslash$) and assignment ($\overset{\circ}{=}$) linkage operations.

5.10.3 Element–Layer Operations

Element-layer operations are, in fact, compositions of *value* and *linkage* operations. These compositions are commutative because the two kinds of operations are independent and decoupled.

$$\boxed{(*, \circ) \Longleftrightarrow (\circ, *)}$$

In this case, both the *value* and the *linkage* of the changing element are affected. *Element* operations are generally symbolised by the pair of symbols that represent the

value and *linkage* operations: a small circle and an asterisk $(*, \circ)$, or by a superposed symbol $(\circledast)$.

$$\boxed{(*, \circ) \Longleftrightarrow (\circledast)}$$

Specific value and linkage operations can be composed and applied to the operands. In this case, the generic symbols for *value* and *linkage* operations from the previous notation will be replaced by the specific *value* and *linkage* operation symbols. For example, the combination between the *value symmetric addition* and the *linkage symmetric addition* is written: $(+, \oplus)$. In the case of the *element assignment*, we may suppress the $\circledast$ from the $\overset{\circledast}{=}$ symbol without major risks of confusion, in order to simplify the notation, since this symbol is used very often.

$$\boxed{(\overset{\circledast}{=}) \Longleftrightarrow (=)}$$

We can write a three-operand composition between B and C and assigned to A, thus :

$$\boxed{A = B\ (+, \oplus)\ C}$$

We can write [2] a *two-operand* composition between A and B, thus :

$$\boxed{A\ +\oplus = B}$$

A *three-operand* operation can be decomposed into two two-operand operations. For instance:

$$\boxed{(A = B\ *\circ\ C) \Longleftrightarrow (A = B;\ A\ *\circ = C)}$$

5.11 Composition Examples

5.11.1 Examples Background

In the subsequent sections we shall exemplify the operations of the *Composition Algebra.*

[2] In order to avoid the confusion with the notations used for argument functions (e.g. as in: $f(x, y)$), we will omit the braces "(...)" and the coma (,) from now on (consequently, the previous statement looks like this: $A = B\ +\oplus\ C$).

We shall use the `neighbor (OSPF)` *Cisco IOS* command, which is used in the *router* context to configure OSPF routers interconnecting to non-broadcast networks [23] (the *no* form should be used to disable this feature). The command syntax is:

```
neighbor ip-address [priority number] [poll-interval seconds]
                    [cost number]
no neighbor ip-address [priority number] [poll-interval seconds]
                       [cost number]
```

where :

- *ip-address* is the interface IP address of the neighbor;
- `priority` *number* (optional) indicates the router priority value of the non-broadcast neighbor associated with the IP address specified. The default is **0**;
- `poll-interval` *seconds* (optional) indicates the poll interval. The default is **120**; and
- `cost` *number* (optional) assigns a cost to a neighbor The default is the cost of the interface, based on the `ip ospf cost` command.

We map it into the element:

```
neighbor:<ip-address>
- priority:<priority_number>
- poll-interval:<seconds>
- cost:<cost_number>
```

We see that, the *neighbor* element has three descendants. We make the assumption that a construct, A, and a capsule, B, have such descendant elements. Since we may have several neighbors and `neighbor` commands, the identifier includes the *value* of the *neighbor* element.

5.11.2 Value–Layer Operations Examples

The *value* operations are "stand-alone", i.e. they can compose elements with or without combination with linkage operations.

5.11.2.1 Assignment

The *assignment* symbol is: "$\stackrel{*}{=}$". An expression like "$X = Y$" means that the (direct and, if recurrent, indirect descendant) leaves of the left-hand element (X) are assigned the values of the corresponding leaves of the right-hand element (Y).

Suppose that the construct A and the capsule B have the following shapes [3]:

```
A: - neighbor:'131.108.3.4'
     - priority:XXX
```

```
B: - neighbor:'131.108.3.4'
     - priority:YYY
```

Suppose we performed the following assignment:

$$(A: / \textit{neighbor}: \text{'131.108.3.4'}) \stackrel{*}{=} (B: / \textit{neighbor}: \text{'131.108.3.4'}) \qquad (5.1)$$

The `neighbor:'131.108.3.4'` is an element identifier and will not change, whereas the `XXX` and `YYY` are leaf element values and should be composed. No matter whether the values of `XXX` and/or `YYY` are 0 (i.e. the default) or not, the latter replaces the former and we obtain the configuration for A depicted in figure 5.1:

```
A: - neighbor:'131.108.3.4'
     - priority:YYY
```

Figure 5.1: Construct A.

5.11.2.2 Additions

There are several ways to add information to a configuration. Thus, if the capsule's (non-default) data preempts the router's configuration (non-default) data, the

[3]The symbols XXX and YYY are just the place-holders for any (0 or non-zero) values.

addition is called *substitutive* (or "overriding"), if – on the contrary – the router's configuration data is preserved despite the capsule, the addition is called *conservative* and if none of the non-default data is retained and the default value is set instead, the addition is called *symmetric.*

The *exclusive* addition is mentioned mostly for its potential use in combined compositions. It acts based on the consensus principle and sets to *default* an element's effective value if the (non-default) effective values of the two operands differ.

5.11.2.3 Symmetric Addition

Let us tackle the *non-default* value cases, first. Suppose we have the construct A and capsule B, depicted in figure 5.2.

```
A: -  neighbor:'131.108.3.4'
      -  priority:1
      -  poll-interval:180
```

```
B: -  neighbor:'131.108.3.4'
      -  priority:2
      -  poll-interval:180
```

Figure 5.2: Entities A and B.

If we perform the composition :

$$(A: / \textit{neighbor};'131.108.3.4') \mathrel{+}= (B: / \textit{neighbor};'131.108.3.4') \tag{5.2}$$

we obtain the following configuration for A:

```
A: -  neighbor:'131.108.3.4'
      -  priority:0
      -  poll-interval:180
```

We see that none of A's and B's values is retained when *non-default unequal* values are composed. Now, let us tackle the *default* value cases. Suppose we have a construct A and a capsule B showed in figure 5.3: If we perform again the composition expressed by 5.2, we obtain the configuration for A showed in figure 5.4.

```
A: - neighbor:'131.108.3.4'
   - priority:1
   - poll-interval:120
```

```
B: - neighbor:'131.108.3.4'
   - priority:0
   - poll-interval:180
```

Figure 5.3: Entities A and B.

```
A: - neighbor:'131.108.3.4'
   - priority:1
   - poll-interval:180
```

Figure 5.4: Entity A.

If we consider just the addition (+) alone, i.e. without assignment (=) to either of the operands, we see that it is *symmetric* indeed (or *commutative*) [4]:

$$\boxed{A + B \Longleftrightarrow B + A}$$

5.11.2.4 Conservative Addition

We use the same cases as in the previous paragraph. For the *non-default* value cases we use the initial A and B depicted in figure 5.2, section 5.11.2.3 (page 73).

If we perform the *conservative addition* between A and B :

$$\boxed{(A: / \mathit{neighbor}: \text{'131.108.3.4'}) >= (B: / \mathit{neighbor}: \text{'131.108.3.4'})} \quad (5.3)$$

we obtain the following configuration for A:

```
A: - neighbor:'131.108.3.4'
   - priority:1
   - poll-interval:180
```

We see that A conserves its values when *non-default unequal* values are composed.

For the *default* value cases, expressed in figure 5.3, section 5.11.2.3, if we perform the *conservative addition* (equation 5.3), we obtain the same results as for the *symmetric addition*, i.e. A has the configuration expressed by 5.4.

[4]This quality is evident in table 5.1.

If we apply the conventions stipulated for symmetry in the previous section (5.11.2.3), we can see that the *conservative addition* is an *asymmetric* operation. This feature is evident in table 5.2 [5].

5.11.2.5 Substitutive Addition

This operation mirrors the *conservative* addition. Thus, we shall see that A's value is "overridden" by B's value when *non-default unequal* values are composed. With the A's and B's non-default values depicted in figure 5.2 in section 5.11.2.3, if we perform the *substitutive addition* between A and B:

$$(A: / \mathit{neighbor}: '131.108.3.4') <= (B: / \mathit{neighbor}: '131.108.3.4') \tag{5.4}$$

we obtain the following configuration for A:

```
A: - neighbor:'131.108.3.4'
   - priority:2
   - poll-interval:180
```

For the *default* value cases, expressed by (A.2) and (B.2) in section 5.11.2.3, if we perform the composition 5.4, we obtain the same results as for the *symmetric* and *conservative additions*, i.e. A has the configuration expressed by 5.4.

If we apply the conventions stipulated for symmetry in section 5.11.2.3, we can see that the substitutive addition is an *asymmetric* operation.

5.11.2.6 Exclusive Addition

With the A's and B's *non-default* values showed in figure 5.2, section 5.11.2.3, if we perform the *exclusive addition* between A and B:

$$(A: / \mathit{neighbor}: '131.108.3.4') \times = (B: / \mathit{neighbor}: '131.108.3.4') \tag{5.5}$$

[5]The reason why the order of the operations in tables 5.1 and 5.2 and the order of the examples differ is that the tables group the symmetric and asymmetric operations, respectively. In plus, for a better visualisation, with the exception of the assignment operations, the *symmetric* symbols correspond to *symmetric* operations, whereas the *asymmetric* symbols correspond to the *asymmetric* operations.

we obtain the following configuration for A:

```
A: - neighbor:'131.108.3.4'
   - priority:1
   - poll-interval:120
```

We see that A's *effective value* is set to the *default value* when *non-default* (equal or unequal) effective values are composed.

For the *default* value cases, expressed in figure 5.3, section 5.11.2.3, if we perform the composition 5.5, we obtain the same results as for the *symmetric* and *conservative additions*, i.e. A has the configuration showed in figure 5.4.

According to the symmetry conventions stipulated in section (5.11.2.3), we can see that the *substitutive addition* is a *symmetric* operation.

5.11.2.7 Identical Selection

With the A's and B's *non-default* values displayed in figure 5.2, section 5.11.2.3, if we perform the *identical selection* between A and B:

$$(A: / \mathit{neighbor}: \text{'131.108.3.4'}) \cdot = (B: / \mathit{neighbor}: \text{'131.108.3.4'}) \tag{5.6}$$

we obtain the following configuration for A:

```
A: - neighbor:'131.108.3.4'
   - priority:0
   - poll-interval:180
```

For the *default* value cases, depicted in figure 5.3, section 5.11.2.3, if we perform the composition 5.6, we obtain the following configuration for A:

```
A: - neighbor:'131.108.3.4'
   - priority:0
   - poll-interval:120
```

We see that A's *effective value* is preserved when it's identical with B's corresponding value. According to the symmetry conventions stipulated in section 5.11.2.3, the *identical selection* is a *symmetric* operation.

5.11.2.8 Subtraction

Subtraction is somewhat complementary to the *identical selection*, because the former sets to default the (non-default) value in case of equality and preserves the rest, whereas the latter does just the opposite (i.e. preserves the value in case of equality and sets the rest to default).

With the A's and B's *non-default* values displayed in figure 5.2, section 5.11.2.3, if we subtract B from A :

$$\boxed{(A\text{: / } \textit{neighbor}\text{: '131.108.3.4'}) \mathbin{/=} (B\text{: / } \textit{neighbor}\text{: '131.108.3.4'})} \qquad (5.7)$$

we obtain the following configuration for A:

```
A: -  neighbor:'131.108.3.4'
   -  priority:1
   -  poll-interval:120
```

For the *default* value cases, expressed by 5.3, section 5.11.2.3, if we perform the composition 5.7, we obtain the following configuration for A:

```
A: -  neighbor:'131.108.3.4'
   -  priority:1
   -  poll-interval:120
```

We see that A's *effective value* is reset when it is identical with B's corresponding value. As expected, this operation is *asymmetric*.

5.11.2.9 The Precedence

Let's take a random value operation, for instance the *symmetric addition* (+). We shall set A's `priority` to have *precedence*, by appending the *dollar* sign prefix ($) to that particular value [6].

Suppose we have a construct A and a capsule B depicted in figure 5.5. If we

[6]The `poll-interval` parameter has no precedence and serves just to emphasize, by contrast, the case illustrated by the *priority* parameter.

```
A: - neighbor:'131.108.3.4'
   - priority:$1
   - poll-interval:180
```

```
B: - neighbor:'131.108.3.4'
   - priority:2
   - poll-interval:240
```

Figure 5.5: Entities A and B.

perform the *symmetric addition* (as in the expression 5.2):

$$\boxed{(A:/\ neighbor:\ '131.108.3.4') \mathrel{+}= (B:/\ neighbor:\ '131.108.3.4')} \tag{5.8}$$

we obtain the following configuration for A [7]:

```
A: - neighbor:'131.108.3.4'
   - priority:1
   - poll-interval:120
```

We see that none of A's priority prevailed over every other possible value (B's value or the *default* value normally resulting from the *symmetric addition*).

5.11.3 Linkage Operations Examples

Since the some of the linkage-layer operations are not "stand-alone", but must be separately explained in this section, we make the following convention: we compose a *generic value operation* ($*$) with the *specific linkage operation* that is presented and represent only the latter one, based on the following formal artifice:

$$\boxed{(*, \circ) \Longleftrightarrow (\circ)}$$

where the generic linkage operation symbol ($\circ$), is a "wildcard" for any linkage operation ($\oplus$, $\otimes$, $\odot$, $\oslash$, $\overset{\circ}{=}$).

5.11.3.1 Assignment

The symbol for the *linkage assignment* is "$\overset{\circ}{=}$". An expression like $X \overset{\circ}{=} Y$ means that the *stray* (direct and indirect descendant) elements of the left-hand element (X) are *removed* and descendants of the right-hand element (Y) *copied instead* to X.

[7] The `poll-interval` default value is 120.

Suppose that the construct A and the capsule B are such as depicted in figure 5.6 [8]. We perform the following assignment:

```
A: - neighbor:'131.108.3.4'
   - priority:XXX
   - poll-interval:VVV
```

```
B: - neighbor:'131.108.3.4'
   - priority:YYY
   - cost:UUU
```

Figure 5.6: Entities A and B.

$$(A: / \textit{neighbor}: '131.108.3.4') \stackrel{\circ}{=} (B: / \textit{neighbor}: '131.108.3.4') \tag{5.9}$$

and obtain the configuration for A showed in figure 5.7.

```
A: - neighbor:'131.108.3.4'
   - priority:ZZZ
   - cost:UUU
```

Figure 5.7: Entity A

We see that, as in the previous example, the *descendant leaf* element `priority` remains in A with yet undecided value, whereas, unlike the previous example, the `poll-interval` element is copied from B to A.

The linkage assignment is not a "stand-alone" operation, it relies on a complementary value operation to compose the values on the *value layer*. Thus, the value of the "place-holder" `ZZZ` is either `XXX` or `YYY`, depending on the value operation that is composed with the type assignment, i.e. $ZZZ = XXX * YYY$ [9].

5.11.3.2 Symmetric Addition

We shall use the same cases as in the previous section (please refer to figure 5.6 in section 5.11.3.1). In order to illustrate the cases expressed in the first two rows of

[8] The symbols *UUU*, *VVV*, *XXX* and *YYY* are just the "place-holders" for (default or non-default) effective values.

[9] Written more formally: $(A: / \textit{neighbor} / \textit{priority}) * = (B: / \textit{neighbor} / \textit{priority})$. For an example of such a composition, refer to section 5.11.4.

the table 5.4 we perform the following *symmetric addition*:

$$\boxed{(A:\ /\ neighbor:\ '131.108.3.4') \oplus= (B:\ /\ neighbor:\ '131.108.3.4')} \tag{5.10}$$

This operation has a different behaviour depending whether it is used in combination with a value operation or as a stand-alone operation.

5.11.3.2.1 When used in combination with a value operation When performing the operation 5.10 [10] we obtain the following configuration for A:

```
A: -  neighbor:'131.108.3.4'
      -  priority:ZZZ
      -  poll-interval:VVV
      -  cost:UUU
```

where $VVV = XXX * YYY$.

We see that descendants from both elements are retained and the common descendants are composed by means of the (*generic*) *value* operation. The result is predictable, if we speak just in terms of the linkage operation, because it is a *symmetric* operation. Regarding the value, the matter is decided by the *effective* value operation that hides behind the *generic symbol* $(*)$.

5.11.3.2.2 When used as a stand alone operation When performing the operation 5.10 [11] we obtain the following configuration for A:

```
A: -  neighbor:'131.108.3.4'
      -  priority:XXX
      -  priority:YYY
      -  poll-interval:VVV
      -  cost:UUU
```

In our specific example, this command tree does not represent a valid **neighbor** command. However, the behaviour of the stand-alone symmetric operation might be

[10] In this case, the generic value composition $(*)$ is implied.

[11] In this case, there is no generic value composition $(*)$ involved.

useful in intermediate operations. For instance, one of the `priority` siblings might be pruned later, according to some algorithm or schedule using a *linkage subtraction* operation.

Besides, there might exist other contexts, commands or sub-commands, that admit sibling elements.

For instance, the command `network`, which configures the network which the routing process is responsible for or, when used in the *no* form, removes an entry from the list of networks. The command has the following syntax, for *IGRP*, *EIGRP* and *RIP* protocols [23]:

```
Router (router) # network network-number
Router (router) # no network network-number
```

where *network-number* represents the IP address of the directly connected networks [12].

Suppose that the construct A and the capsule B are such as depicted in figure 5.8 and we perform the *stand-alone symmetric addition* expressed by (5.11).

```
A: -  eigrp:'109'
      -  network:'131.108.0.0'
```

```
B: -  eigrp:'109'
      -  network:'192.31.7.0'
```

Figure 5.8: Entities A and B.

$$(A: / \mathit{eigrp}: \text{'109'}) \oplus= (B: / \mathit{eigrp}: \text{'109'}) \tag{5.11}$$

We obtain the following configuration for A:

```
A: -  eigrp:'109'
      -  network:'131.108.0.0'
      -  network:'192.31.7.0'
```

Since it cannot choose between the two values *network-number*, in the absence of the *value* operation that would do that, the linkage symmetric addition keeps both of the `network` elements, which suits us, in this example, because we wanted to create a list of network addresses for the *EIGRP* protocol.

[12] for other protocols, like *BGP* or *OSPF*, the command has a slightly different form.

5.11.3.3 Exclusive Addition

Using the same cases as in the previous section (figure 5.6 in 5.11.3.1), we perform the following *exclusive addition* operation:

$$(A: / \textit{neighbor}: \text{'131.108.3.4'}) \otimes= (B: / \textit{neighbor}: \text{'131.108.3.4'})$$

and obtain the following configuration for A:

```
A: -  neighbor:'131.108.3.4'
      -  poll-interval:VVV
      -  cost:UUU
```

The *exclusive addition* is a *symmetric* operation. The distinct elements are retained by this operation whereas the common features are discarded and consequently, no value operation is further needed. In other words, this operation is "stand-alone".

5.11.3.4 Identical Selection

Using the same cases as in the previous section (figure 5.6 in 5.11.3.1), we perform the following *identical selection* operation:

$$(A: / \textit{neighbor}: \text{'131.108.3.4'}) \odot= (B: / \textit{neighbor}: \text{'131.108.3.4'})$$

and obtain the following configuration for A:

```
A: -  neighbor:'131.108.3.4'
      -  priority:ZZZ
```

The *exclusive addition* is *symmetric*. This operation and the previous one are complementary, since now, the distinct elements are discarded by this operation whereas the common features are retained and consequently, a value operation is further needed.

The *symmetric addition* is obtained from the combined effect of the *identical selection* and the *exclusive addition*.

5.11.3.5 Subtraction

In order to illustrate this operation, we shall use the same cases as in the previous section (figure 5.6 in 5.11.3.1). We perform the following *subtraction* :

$$(A: / \textit{neighbor}: \text{'131.108.3.4'}) \oslash= (B: / \textit{neighbor}: \text{'131.108.3.4'})$$

and obtain the following configuration for A:

```
A: -  neighbor:'131.108.3.4'
      -  poll-interval:VVV
```

Obviously, the *subtraction* is *asymmetric*. The distinct elements of the left-hand side operand are retained by this operation whereas the others are discarded and consequently, no value operation is further needed. In other words, this operation is stand-alone.

It is interesting to mention that the *exclusive addition* can be obtained by applying twice the *subtraction*, once with the operands in order and once with them in inverted order.

5.11.4 Element–Layer Operations Examples

5.11.4.1 Assignment

The *element-layer assignment* is composed of the *value assignment* and the *linkage assignment*.

Suppose that the construct A and the capsule B look like in figure 5.9. We

```
A: -  neighbor:'131.108.3.4'
      -  priority:XXX
      -  poll-interval:VVV
```

```
B: -  neighbor:'131.108.3.4'
      -  priority:YYY
      -  cost:UUU
```

Figure 5.9: Entities A and B

perform the following assignment :

$$(A: / \textit{neighbor}: \text{'131.108.3.4'}) = (B: / \textit{neighbor}: \text{'131.108.3.4'}) \tag{5.12}$$

and obtain the following configuration for A:

```
A: -  neighbor:'131.108.3.4'
   -  priority:YYY
   -  cost:UUU
```

Thus, the *linkage assignment* operation gives a similar result as operation 5.9 (please refer to figure 5.7) and the *value assignment* decides the value of the *priority* element, as the *value assignment* operation (5.1) in section 5.11.2.1 (please refer to figure 5.1).

5.11.4.2 The Value Conservative Addition and Linkage Symmetric Addition Compound

Obviously, there are many possible combinations between the *value* and *linkage* operations that have been presented. We shall take just an example to illustrate the operation composition.

Suppose that the construct A and the capsule B are such as depicted in figure 5.10 and we perform the following compound operation:

```
A: -  neighbor:'131.108.3.4'
   -  priority:1
   -  poll-interval:VVV
```

```
B: -  neighbor:'131.108.3.4'
   -  priority:2
   -  cost:UUU
```

Figure 5.10: Entities A and B.

$$(A: / \textit{neighbor}: \text{'131.108.3.4'}) > \oplus = (B: / \textit{neighbor}: \text{'131.108.3.4'}) \quad (5.13)$$

and obtain the configuration for A depicted in figure 5.11.

The *value* of the `priority` is obtained according to the *value conservative addition* that was illustrated in section 5.11.2.4 at page 74 (refer to operation 5.3). Suppose that the construct A and the capsule B are such as depicted in figure 5.12.

If we perform the operation expressed by 5.13, we obtain the following configuration for A:

```
A: -  neighbor:'131.108.3.4'
      -  priority:1
      -  poll-interval:VVV
      -  cost:UUU
```

Figure 5.11: Entity A

```
A: -  neighbor:'131.108.3.4'
      -  priority:1
      -  poll-interval:VVV
```

```
B: -  neighbor:'131.108.3.4'
      -  priority:1
      -  cost:UUU
```

Figure 5.12: Entities A and B.

```
A: -  neighbor:'131.108.3.4'
      -  priority:1
      -  poll-interval:VVV
      -  cost:UUU
```

which is similar to 5.11, obtained in the previous case.

Suppose now that A has the following values :

```
A: -  neighbor:'131.108.3.4'
      -  priority:0
      -  poll-interval:VVV
```

and we perform again the operation expressed by 5.13, we get the following values for A:

```
A: -  neighbor:'131.108.3.4'
      -  priority:1
      -  poll-interval:VVV
      -  cost:UUU
```

which is again similar to 5.11.

If we use the values for the construct A and capsule B depicted in figure 5.13, the resulting A has the shape :

```
A: - neighbor:'131.108.3.4'
   - priority:1
   - poll-interval:VVV
```

```
B: - neighbor:'131.108.3.4'
   - priority:0
   - cost:UUU
```

Figure 5.13: Entities A and B.

```
A: - neighbor:'131.108.3.4'
   - priority:1
   - poll-interval:VVV
   - cost:UUU
```

in which we find again 5.11.

Obviously, if both values involved in the value composition are equal to the *default value*, (which, in the case of the `priority` parameter, is *0*) we get the effective value equal to 0 as the result.

For instance, if the construct A and the capsule B are those depicted in figure 5.14. the resulting A has the shape depicted in figure 5.15:

```
A: - neighbor:'131.108.3.4'
   - priority:0
   - poll-interval:VVV
```

```
B: - neighbor:'131.108.3.4'
   - priority:0
   - cost:UUU
```

Figure 5.14: Entities A and B.

```
A: - neighbor:'131.108.3.4'
   - priority:0
   - poll-interval:VVV
   - cost:UUU
```

Figure 5.15: Entity A.

5.11.4.3 Other Compound Operations

Several considerations regarding the composition of value and linkage operations into element operations may be relevant:

- there are many operations and consequently many more potential combinations among them;
- the configuration composition is typically *asymmetric*, i.e., usually an entity (the capsule) changes another entity (the construct);
- some operations are *dependent*, i.e. they can be obtained from the others;
- while independent operations are useful from axiomatic perspective, some dependent operations may be convenient for the user;
- usually, it is less risky to *add* than to *remove* information. For instance, an additional information may be ignored, while a missing one may cause problems; That's why, we defined more additions than subtractions, etc.;
- we can pre-configure some sets of services, policies or configurations and switch between them, mask parameters, etc.;
- there are some functional similarities and correspondences between the some *value* and *linkage* operations, as their respective names may suggest, which might be successfully combined into a convenient and effective operation system; and
- a balance must be set between the *symmetric* and the *asymmetric* operations.

There are specific combinations of *value* and *linkage* operations that seem to be better suited for the configuration process. For instance, the *value* and *linkage additions* can be combined into useful *element* operations. Nevertheless, we can use other combinations, according to our purposes.

Chapter 6

CLI/Meta–CLI Modelling

There are several CLI features that must be taken unto account in the CLI/Meta-CLI modelling. The modelling process is not cast on fixed rules, it involves a degree of creativity and is specific to each configuration command, feature or service. The modelisation is based on the syntax, semantics and behaviour of the CLI commands.

The process of modelling the CLI information depends on, and must be adapted to, the concrete type of CLI. In the following paragraphs we shall illustrate some aspects of this procedure with examples from the *Cisco IOS*.

There are several *levels* of the modelling process from CLI to meta-CLI :

- the *command* level, which deals with the command and its options, switches, parameters and their values; access to this information requires fine-grained modelling structures and is necessary especially for and corresponds to the instance-specific configuration component;
- the *context* level, which deals with the commands opening a context, mode or submode and the group of commands belonging to that context; and
- the *service and configuration record* level, which deals with the commands and information that belong to a service or a device configuration record and corresponds to the policy- and service-specific information.

6.1 Command–Level Translation

The *CLI commands* are translated into *Meta-CLI tree structures.* The purpose of translating the parameters and arguments into a tree is to enable fine-grained access to the information in order to implement services and policies and to perform the instance-specific configuration operations. If the command information does not need to be addressed individually, the command can be taken as a whole, translated into an element and dealt with directly at context-level.

At this level, there are several specific issues that involve both creativity and the existence of some translation patterns or rules. The following sections will highlight some of these issues.

6.1.1 Command Definitions and Instances and Generic and Concrete Structures

Since we can have several occurrences of a command, there should exist a corresponding *generic structure*, which is a like a class structure or a template and corresponds to the *command definition* and *concrete structures*, which are like instance structures and correspond to the *command instances.* The *name of the command* is mapped into the *top element* and the *parameters, switches* or *options* are added as descendant elements of the first element or of its descendants.

The *keywords* of the parameters will be the *types* of the descendant elements and the *values* will be element *effective values.* The shape of the tree must take into the account the dependencies among the command's parameters, with the elements that control the others as ancestors and the controlled elements as descendants.

We may take into consideration, as an example, the `network area` command, which defines the interfaces on which OSPF runs and the area ID for those interfaces [19]. (To disable ISPF routing for interfaces defined with the *address wildcard-mask* pair, the *no* form should be used.). The context of the command is *router.*

The syntax of this command is:

```
network address wildcard-mask area area-id
no network address wildcard-mask area area-id
```

where :

- *address* is an IP address;
- *wildcard-mask* is an IP-address-type mask; and
- *area* is the area that must be associated with the OSPF address range;

A (concrete) *instance* of this command would look like this:

```
Router (router) # network 131.119.0.0 0.0.0.255 area 0
```

which defines the interface with the address *131.119.0.0 0.0.0.255* on which *OSPF* runs and associates area *0* to it.

The translation of the `network area` *command definition* into a *generic structure* will be as follows: the element from parameter `area` will be added (as a descendant) to the `network` element, whose value is represented by the *address wildcard-mask* [1].

```
- network:<address> <wildcard-mask>
   - area:<area-id>
```

while the *instance* of the `network area` command would take the following shape:

```
- network:'131.119.0.0. 0.0.0.255'
   - area:0
```

6.1.2 Command Negation

Usually, the syntax provides a negation of a command, starting with the word *no* and continuing with the command's name and the (mandatory) parameters (e.g. `no network` *address wildcard-mask* `area` *area-id*). In this case, this *name*, using the *tilde wildcard* (~) is assigned to the effective value of the top element. For instance, if we negate the previous command, we get the following tree :

[1] The angular brackets (< ... >) designate tokens (placeholders) that must be replaced with actual values, when the command actually occurs during the configuration process.

```
- network:no ~ '131.119.0.0. 0.0.0.255'
  - area:0
```

The *tilde* represents the "shorthand" notation for the *type* of the element.

6.1.3 Independent Parameters

If the parameters are *independent*, they should be translated into *sibling* elements. For example, the `neighbor (OSPF)` command translation, presented in section 5.11.1:

`neighbor` *ip-address* `[priority` *number*`] [poll-interval` *seconds*`] [cost` *number*`]`

Here, the `priority`, `poll-interval` and `cost` are mutually *independent* parameters and we can translate them into *siblings*; thus the command definition maps to the following generic structure:

```
- neighbor:<ip-address>
  - priority:<priority_number>
  - poll-interval:<seconds>
  - cost:<cost_number>
```

6.1.4 Dependent Parameters

If there is a *dependency* between options or parameters, they should be linked to each *en cascade*, so that the link showed the *hierarchical* relationship between them.

Let us consider the *Cisco IOS* command `offset-list`, which is used in the *router* mode, to add an offset to incoming and outgoing metrics to routes learned via *RIP*. (To remove an offset list, the *no* form of this command must be used.) The command has the following syntax:

`offset-list` {*access-list-number* | *name*} {`in` | `out`} *offset* [*type number*]
`no offset-list` {*access-list-number* | *name*} {`in` | `out`} *offset* [*type number*]

where :

- *access-list-number — name*, is the standard access list number or name to be applied;
- `in` or `out`, applies to access list to incoming or outgoing metrics, respectively;
- *offset*, is the offset to be applied to metrics for networks matching the access list;
- *type* (optional), is the interface type to which the offset list is applied; and
- *number* (optional), is the interface number to which the offset list is applied.

In the following example of an `offset-list` command, the router applies an offset of 10 to the router's delay component only to *access list 21* [23]:

```
offset-list 21 out 10
```

In the next example, the router applies an offset of 10 to the routes learned from Ethernet interface 0 :

```
offset-list 21 in 10 ethernet 0
```

In this command, each of the parameters "refines" or "restricts" the information given by the previous parameters. For example, an offset list with interface type and number (which are optional), will take precedence over a list without interface and type number. Thus, we have *cascading* dependencies of the parameters:

- the *interface type* and *number*, if any specified, applies to the offset;
- the *offset* applies to a direction; and
- the direction (*in* or *out*), applies to a specific access list.

The translation into an tree should therefore organise this information into a hierarchical way; for instance the `offset-list` command would be translated thus:

```
- offset-list:21
   - direction:in
      - offset:10
         - interface:'Ethernet 0'
```

6.1.5 Parameter Translation

A parameter may have various representations within the command. The command translation should provide a means to deal with all these situations:

- there exist both a *keyword* and its *value*;

  For example, in the `neighbor (OSPF)` command translation, presented in section 5.11.1 (pages 70 and subsequent) the parameter `cost` has both a keyword and a value. In this case, we translate the parameter into an element whose *type* is the keyword and *effective value* is the parameter value;

- there is no *keyword* but there exists a *value*;

  For example, in the `offset-list` command instances exemplified in section 6.1.4, at page 92, the *offset* has no keyword but the value 10 is indicated; in this case, the corresponding element will have its effective value set to the value specified by the command and the type of the element will be set to some string, which may be suggested by the command definition; for instance, in this command's translation at page 92, the type *offset* was created to accompany the value : 10; and

- there is a *keyword* but no *value*;

  If the keyword is an alternative, (like *in* and *out*) in the `offset-list` command at page 91, it may be treated as a *value* in the previous case; in the tree shown at page 92 it was coupled with the string *direction.*

6.1.6 Command Name Translation

The command may start with *several keywords* in a row; in this case they may be placed in the same element (as its *type*). For instance, the command `ip address`, which is valid in the *interface* mode and sets the IP address of an interface (or removes it when used with the *no* form) and has the following syntax:

```
ip address ip-address mask [secondary]
no ip address ip-address mask [secondary]
```

where :

- *ip-address* represents the IP address;
- *mask* represents the mask of the associated IP subnet; and
- `secondary` (optional) specified that the configured address is a secondary address.

starts with two strings: `ip address`. These will be translated into the *type* and the two strings *ip-address mask* can be translated into *the value* of an element.

6.1.7 Parameter Permutation

The translation should handle various situations encapsulating the commands into the tree. This encapsulation should allow some degree of freedom of expression but keep the equivalence of the commands and tree intact, so that conversion and retroversion would happen without information loss or distortion.

Thus, for instance, the order of the arguments and parameters may be permuted within the tree, if necessary. We may take as an example the command `ip access-group`, which is used in the *interface* mode and assigns an access list to an interface in a specific direction – `in` or `out` (to disable an access-group command, the *no* form should be used). The syntax of the command is :

```
ip access-group access-list [in | out]
no ip access-group access-list [in | out]
```

where :

- *access-list* is the number of an access list; and
- `in` or `out` is the direction. Each interface can support only *one access list* in either direction.

We can set the command's element to the value of the second parameter and keep the first value as the value of its descendant. We can call the former "access-list" (therefore, the *type* is *access-list*):

Assuming a concrete command:

```
Router (config-if) # ip access-group 10 in
```

we can map it to an element in the following way:

```
- ip access-group:in
   - access-list:10
```

In this way, we manipulate the element according to a type (`in` or `out`) rather than a number (the *access-list* value).

As mentioned in chapter 4, the *generic structures* representing the CLI commands must be pre-defined in the program. When a configuration file is read in, the effective values must be set to the corresponding values found in the configuration and thus the structures are instantiated. The (immutable) default and range or enumerated values must pre-exist in the generic structure.

The program must parse and recognise the commands and extract the options, parameter, and argument information from the commands and place it into the corresponding tree elements.

6.2 Context–Level Translation

We deal at this level with groups of commands in which the first one changes the context or mode in which the configuration process is performed (the `interface` (e.g. `interface Ethernet 0`), `router` (e.g. `router ospf 109`), `line`, etc. commands) whereas the rest of the commands are specific to that context or mode (for example, `ip address` command within the *interface* mode). The context-specific commands should be translated into trees that are descendants of the context-changing command, more exactly, of its *top* element.

Thus, we have two kinds of trees descending from the same top element of the context-changing command tree :

- command's own tree containing the parameter information; and
- trees representing the context-specific commands.

As mentioned in chapter 4, the context-level Meta-CLI tree must pre-exist and translate into branches all of the generic context-specific commands. When such a tree is instantiated, the branches that translate existing commands will be instantiated (i.e. kept and filled with corresponding information), whereas those who represent non-existing commands will be deleted.

6.3 Lineage Aggregation

If some of the instances of the same command have one or more similar elements and paths, these can be unified into the same ancestor and lineage. We may consider as an example the command `access-list` (standard), which defines a standard IP access list and has the following syntax (to remove standard access-lists, the *no* form is used):

```
access-list access-list-number deny | permit source [source-wildcard]
no access-list access-list-number
```

where:

- *access-list-number*, is the number of the access-list, from 1 to 99;
- `deny` (default) or `permit`, denies or permits, respectively, access if conditions are matched;
- *source*, is the number of the network or host sending the packet, in dotted-decimal format or keyword `any` as an abbreviation of the source and wildcard *0.0.0.0 255.255.255.255*; and
- *source-wildcard*, are the wildcard bits to be applied to the source, in dotted-decimal format or keyword `any` as an abbreviation of the source and wildcard *0.0.0.0 255.255.255.255*.

For instance, the following access-list:

```
Router (config) # access-list 110 permit 10.0.1.0 0.0.0.255
Router (config) # access-list 110 permit 10.0.2.0 0.0.0.255
```

allows access for only those hosts on the networks *10.0.1.0/24* and *10.0.2.0/24* and implicitly denies all other accesses.

According to the above-mentioned considerations, this access list can be translated into the following tree :

```
- access-list:110
    - permit:'10.0.1.0 0.0.0.255'
    - permit:'10.0.2.0 0.0.0.255'
```

The contexts have an tree hierarchy. However, the context-changing commands that allow the transition of the user's configuration control from one context to the other do not have an tree hierarchy because they can have opposite effect, i.e. they can invert the context hierarchy. More specifically, commands like *control+Z*, *end* or *exit* do not have parameters and are, in general, implied by the configuration files. Therefore, they should not cause problems to the translation of context-level groups of commands.

6.4 Device Configuration Record–Level Translation

At this level, the context-level trees are combined into trees that represent device configuration files and services that should be composed with these configurations. If some elements on the trees are descendants of more than an ancestor, these elements may be accessed from different roots and along different paths. This can be the case with instance-specific information.

A construct or a capsule can get alternative trees, which may be used for configuration *validation* and *verification.*

Chapter 7

Case Study: VPN Service Configuration

Configuration of the Services for Virtual Private Networks (VPNs) Using the Meta–CLI Model

7.1 MPLS and VPN Services Outline

In this chapter, we shall illustrate the use of our model to activate a MPLS/VPN service, which provides us with a good example that allows us to make our case. A VPN service [1] consists of multiple sub-services (e.g., the configuration of MPLS, an IGP for connectivity, and a BGP for route advertisement), and, since there is no single CLI command available for its activation, it requires multiple CLI commands [46].

7.1.1 MPLS Description

The *Multiprotocol label switching* (*MPLS*) is a versatile solution to address the problems faced by present-day networks-speed, scalability, quality-of-service (QoS)

[1] Sometimes it is considered a superservice.

management, and traffic engineering. MPLS has emerged as an elegant solution to meet the bandwidth-management and service requirements for next-generation

For Internet protocol (IP)-based backbone networks, MPLS addresses issues related to scalability and routing (based on QoS and service quality metrics) and can exist over existing asynchronous transfer mode (ATM) and frame-relay networks [9].

The premise of multiprotocol label switching (MPLS) is to speed up packet forwarding and provide for traffic engineering in Internet protocol (IP) networks. To accomplish this, the connectionless operation of IP networks becomes more like a connection-oriented network where the path between the source and the destination is pre-calculated based on user specifics.

To speed up the forwarding scheme, an MPLS device uses labels rather than address matching to determine the next hop for a received packet. To provide traffic engineering, tables are used that represent the levels of quality of service (QoS) that the network can support. The tables and the labels are used together to establish an end-to-end path called a label switched path (LSP) [10].

7.1.2 VPN Description

Private networking involves securely transmitting corporate data across multiple sites throughout an entire enterprise. Creating a truly private corporate network generally requires an intranet. A *virtual private network* (VPN) is one means of accomplishing such an implementation using the public Internet [11] [39].

7.2 Configuration of an MPLS/VPN Application

We will illustrate the use of the Meta-CLI with the following example. We suppose that a service provider needs to provide virtual private networking services to two local Intranets via MPLS.

One Intranet, known as the *RED VPN*, will have a 10MB Ethernet connection to the provider. The subscriber owns a router and plans to use the provider as the default gateway to their headquarters facility. The other Intranet, known as the

BLUE VPN, will have a 100 MB connection to the provider. The subscriber and the provider are using overlapping private address space.

This is allowed because once an interface is configured as part of a VPN, it is removed from the global routing table. Each VPN will have its own virtual route-forwarding instance, resulting in secure transport across the MPLS cloud. Multiple OSPF routing processes are required in order to exchange topology information between the subscriber and the provider. The provider runs an internal OSPF routing process to exchange reachability information among other MPLS peers. The details of this example are depicted in Figure 7.1.

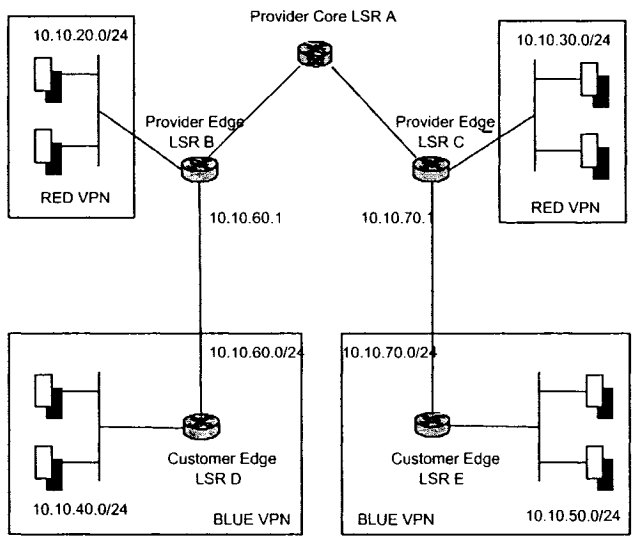

Figure 7.1: MPLS VPN Configuration Example.

Since the MPLS may be also regarded as a service *per se*, and pre-exist, we shall consider has already been set up [2].

[2]by enabling `ip cef` and configuring `tag-switching ip` on the interfaces

7.3 The Addition of VPN Services

We start with the configuration files described in the figures 7.2, 7.3 and 7.4).

```
!
version 12.1
!
hostname A
!
ip subnet-zero
ip cef
!
interface Loopback0
   ip address 10.10.10.1 255.255.255.255
   no ip directed-broadcast
!
interface Ethernet 1/0
   no ip address
   no ip directed-broadcast
   shutdown
   no cdp enable
!
interface Ethernet 1/1
   no ip address
   no ip directed-broadcast
   shutdown
   no cdp enable
!
interface FastEthernet 2/0
   ip unnumbered loopback0
   tag-switching ip
!
interface FastEthernet 2/1
   ip unnumbered loopback0
   tag-switching ip
!
router ospf 10
   network 10.0.0.0 0.255.255.255 area 0
!
ip classless
no ip http server
!
no cdp run
!
line con 0
   exec-timeout 0 0
   transport input none
line aux 0
line vty 0 4
   password cisco
   no login
!
end
```

Figure 7.2: LSR A config file.

The router configurations that need to change are B and C.

7.3.1 B's Configuration

We shall first map the B configuration file into the configuration construct depicted in figure 7.5. The services to be added are depicted in figure 7.6. We construct the capsule VPN containing the features of the *Red* and *Blue* VPN services to be added, as represented in figure 7.7. (Instead of $b.b.b.b$, which indicates an IP address data type, we shall leave an *empty space.*)

```
!
version 12.1
!
hostname B
!
ip subnet-zero
ip cef
!
interface Loopback0
   ip address 10.10.10.2 255.255.255.255
   no ip directed-broadcast
!
interface Ethernet 1/0
   no ip address
   no ip directed-broadcast
   shutdown
   no cdp enable
!
interface Ethernet 1/1
   no ip address
   no ip directed-broadcast
   shutdown
   no cdp enable
!
interface FastEthernet 2/0
   ip unnumbered loopback0
   tag-switching ip
!
interface FastEthernet 2/1
   ip unnumbered loopback0
   tag-switching ip
!
router ospf 10
   network 10.0.0.0 0.255.255.255 area 0
!
ip classless
no ip http server
!
no cdp run
!
line con 0
   exec-timeout 0 0
   transport input none
line aux 0
line vty 0 4
   password cisco
   no login
!
end
```

```
!
version 12.1
!
hostname C
!
ip subnet-zero
ip cef
!
interface Loopback0
   ip address 10.10.10.3 255.255.255.255
   no ip directed-broadcast
!
interface Ethernet 1/0
   no ip address
   no ip directed-broadcast
   shutdown
   no cdp enable
!
interface Ethernet 1/1
   no ip address
   no ip directed-broadcast
   shutdown
   no cdp enable
!
interface FastEthernet 2/0
   ip unnumbered loopback0
   tag-switching ip
!
interface FastEthernet 2/1
   ip unnumbered loopback0
   tag-switching ip
!
router ospf 10
   network 10.0.0.0 0.255.255.255 area 0
!
ip classless
no ip http server
!
no cdp run
!
line con 0
   exec-timeout 0 0
   transport input none
line aux 0
line vty 0 4
   password cisco
   no login
!
end
```

Figure 7.3: LSRs B and C config files.

We perform now several *policy-based*, inter-configuration operations, followed by several *instance-* or *device-specific*, intra-configuration operations.

Policy–Specific Operations

In the first two steps, we *copy* the trees responsible for the creation of new VPN routing tables and address families, from the capsule to the construct, since they do not exist in the latter one.

7.3.1.1 B's First Operation

We perform the following composition operation:

$$\boxed{(vpn{:}\ /\ configure\ /\ ipvrf)\ copy\ to\ (b{:}\ /\ ipcef)}$$

```
!
version 12.1
!
hostname D
!
ip subnet-zero
ip cef
!
interface Ethernet 1/0
   ip address 10.10.40.1 255.255.255.0
!
interface Ethernet 1/1
   no ip address
   no ip directed-broadcast
   shutdown
   no cdp enable
!
interface FastEthernet 2/0
   ip address 10.10.60.2 255.255.255.0
!
interface FastEthernet 2/1
   no ip address
   no ip directed-broadcast
   shutdown
   no cdp enable
!
router ospf 20
   network 10.0.0.0 0.255.255.255 area 0
!
ip classless
no ip http server
!
no cdp run
!
line con 0
   exec-timeout 0 0
   transport input none
line aux 0
line vty 0 4
   password cisco
   no login
!
end
```

```
!
version 12.1
!
hostname E
!
ip subnet-zero
ip cef
!
interface Ethernet 1/0
   ip address 10.10.50.1 255.255.255.0
!
interface Ethernet 1/1
   no ip address
   no ip directed-broadcast
   shutdown
   no cdp enable
!
interface FastEthernet 2/0
   ip address 10.10.70.2 255.255.255.0
!
interface FastEthernet 2/1
   no ip address
   no ip directed-broadcast
   shutdown
   no cdp enable
!
router ospf 20
   network 10.0.0.0 0.255.255.255 area 0
!
ip classless
no ip http server
!
no cdp run
!
line con 0
   exec-timeout 0 0
   transport input none
line aux 0
line vty 0 4
   password cisco
   no login
!
end
```

Figure 7.4: LSRs D and E config files.

corresponding to the CLI commands *ip vrf Red* and *ip vrf Blue* which create two new VPN routing tables called *Red* and *Blue*, respectively. The above-mentioned operation *copies* the tree *ip vrf* from the capsule VPN as a sibling of the tree *ip cef* of the construct B. Thus, the B construct will take the shape presented in figure 7.8 (for the sake of the clarity, we will *compress* (symbol: +) the elements that do not change or are simply copied from an operand to the other and *expand* (symbol: −) the elements whose descendants have changed in value or number).

7.3.1.2 B's Second Operation

Next, we perform the following composition operation:

$$(vpn\text{: } /\ configure\ /\ address\ family)\ copy\ to\ (b\text{: } /\ configure\ /\ router)$$

```
B :
  - configure : terminal
    - version : '12.1'
    - hostname : b
    - ip subnet-zero :
    - ip cef :
    - interface
      - Loopback : 0
        - ip address : '10.10.10.2 255.255.255.0'
        - ip directed-broadcast :
      - Ethernet : '1/0'
        - ip address : '10.10.20.1 255.255.255.0'
        - ip directed-broadcast :
        - shutdown : no
        - cdp enable :
      - Ethernet : '1/1'
        - ip address : no
        - ip directed-broadcast : no
        - shutdown :
        - cdp enable : no
      - FastEthernet : '2/0'
        - ip unnumbered :
          - type : loopback
          - number : 0
        - tag-switching ip :
      - FastEthernet : '2/1'
        - ip address : '10.10.60.1 255.255.255.0'
        - ip directed-broadcast :
        - shutdown : no
        - cdp enable :
    - router
      - ospf : 10
        - network
          - address & wildcard mask : '10.0.0.0 0.255.255.255'
          - area : 0
    - ip classless :
    - ip http server : no
    - cdp run : no
    - line
      - type : aux
        - line number : 0
      - type : vty
        - line number : 0
        - ending line number : 4
        - password : cisco
        - login : no
      - type : console
        - exec timeout
          - minutes : 0
          - seconds : 0
        - transport input : none
```

Figure 7.5: B construct.

This operation adds the *address family* element from the capsule VPN to the construct B as a sibling of the element *address family* and which adds the following features to B:

- configure the address family for VRF *Red* and *Blue* (`address-family ipv4 vrf Red`, `address-family ipv4 vrf Blue`, respectively);
- redistribute the routes from OSPF to BGP Red VPN routing table (`redistribute ospf 17`);
- disable summarisation (`no summary`);
- redistribute the static routes (`redistribute static`);

```
ip vrf Red
   rd 65050:1
   route-target export 65050:1
   route-target import 65050:1
interface Ethernet 1/0
   ip vrf forwarding Red
address-family ipv4 vrf Red
   redistribute static
   redistribute static connected
   exit address-family
address-family vpnv4
   neighbor b.b.b.b activate
   neighbor b.b.b.b send-community extended
```

```
ip vrf Blue
   rd 65051:1
   route-target export 65051:1
   route-target import 65051:1
interface FastEthernet 2/1
   ip vrf forwarding Blue
router ospf 20 vrf Blue
   network 10.0.0.0 0.255.255.255 area 0
   redistribute bgp 65500 metric-type 1 subnets
router bgp 65500
   no synchronization
   no bgp default ipv4-unicast
   neighbor b.b.b.b remote-as 65500
   neighbor b.b.b.b update-source loopback 0
address-family ipv4 vrf Blue
   redistribute ospf 20
   no autosummary
   exit address-family
```

Figure 7.6: Red and Blue VPNs commands.

- redistribute the connected routes (`redistribute static connected`);
- configure the address family using VPN IPv4 prefixes (`address-family vpnv4`);
- activate IBGP neighbor (`neighbor a.a.a.a activate`); and
- forward VPN extended attributes (`neighbor a.a.a.a send-community extended`).

which brings the B construct into the state showed in figure 7.9.

7.3.1.3 B's Third Operation

In the following three operations, we will compose the *symmetric addition* (+) for the values with the *(linkage) addition* ($\oplus$) for the connectivity of the elements.
First, we associate the *Ethernet:'1/0'* interface with the *Red* VPN with the following composition operation :

$$(b{:}\ /\ interface\ /\ Ethernet{:}\ '1/0') \ +\oplus= \ (vpn{:}\ /\ interface\ /\ Ethernet{:}\ '1/0')$$

This composition performs a *symmetric addition* (+) of the values of the similar descendants and a *(linkage) addition* ($\oplus$) of missing descendants of the element *Ethernet:'1/0'* from the capsule VPN to the element *Ethernet:'1/0'* of the construct B, which results in the B construct depicted in figure 7.10.

7.3.1.4 B's Fourth Operation

Similarly, we associate *FastEthernet:'2/1'* interface with the *Blue* VPN, with the following composition operation :

$$\boxed{(b\text{: / } interface \text{ / } FastEthernet\text{: } '2/1') \ +\oplus= \ (vpn\text{: / } interface \text{ / } Ethernet\text{: } '1/1')}$$

This composition performs a *symmetric addition* (+) of the values of the similar descendants and a *(linkage) addition* ($\oplus$) of missing descendants of the element *Ethernet:'2/1'* from the capsule VPN to the element *FastEthernet:'1/1'* of the construct B. It results in the B construct represented in figure 7.11. The two previous compositions add the feature from the CLI commands *ip vrf forwarding Red* and *ip vrf forwarding Blue*, respectively.

7.3.1.5 B's Fifth Operation

Next, we perform the following combination :

$$\boxed{(b\text{: / } configure \text{ / } router) \ +\oplus= \ (vpn\text{: / } configure \text{ / } router)}$$

This composition performs again a *(value) symmetric addition* (+) of the values of the matching elements and a *(linkage) addition* ($\oplus$) of missing elements from the capsule VPN's element *router* to the construct B's element *router*. This composition corresponds to the following features and CLI commands :

- enable routing process 20 for the *Red* VPN (`router ospf 20 vrf Red`);
- specify the network directly connected to the router and the OSPF area membership (`network 10.0.0.0 0.255.255.255 area 0`);
- redistribute BGP routes and inject BGP routes into OSPF as *type*1 routes (`redistribute bgp 65500 metric-type 1 subnets`);
- enable BGP routing for autonomous system 65500 (`router bgp 65500`);
- disable synchronization : (`no synchronization`);

- specify the IBGP neighbor and autonomous system number (`no bgp default ipv4-unicast`);
- add an entry to the BGP neighbor table (`neighbor z.z.z.z remote-as 65500`); and
- force the router to use the IP address assigned to *Loopback*0 as the source address for BGP packets (`neighbor z.z.z.z update source loopback 0`).

The shape of the resulting B is presented in figure 7.12.

Device–Specific Operations

We have finished the policy-based operations with regard to B and proceed now with the interface and device-specific operations.

7.3.1.6 B's Sixth Operation

We set the address of the neighbor to the appropriate value wherever it occurs in the *router* and in the *family address* contexts. For the *router* context, we perform the following operation :

$$\boxed{(b\text{: } / \textit{configure} / \textit{router} /.../ \textit{ip address}) = \text{'10.10.10.3'}} \tag{7.1}$$

In operation 7.1 the notation /.../ represents a *wildcard* for any *relative path* between the element *router* and the element *ip address*. The element *router* is uniquely determined, since it has an absolute path indicated and is the only element of this type within its context, whereas the element *ip address* is not uniquely specified, since it has a *generic path*. In this way, *all the descendants* of type *ip address* of the element *router* match the identification of the operation 7.1 and will be assigned the value *'10.10.10.3'*.

We have here an occurrence of a *device-specific operation*, where a search and multiple matchings followed by assignments take place within the same construct. In fact, the assignment is a value assignment and if we wanted to be more precise

and restrictive, we could use the *value assignment* ($\stackrel{*}{=}$) symbol instead of the *element assignment* (=). The shape of the B construct is displayed in figure 7.13.

7.3.1.7 B's Seventh Operation

We repeat the previous operation for the *address family* context:

$$\boxed{(b\text{: / } configure \text{ / } address\ family \text{ /.../ } ip\ address) = \text{'10.10.10.3'}}$$

and obtain the B construct showed in figure 7.14.

7.3.2 C's Configuration

Now, we have finished the feature composition for *LSR B* and start the it for the *LSR C*. To speed up the procedure for *LSR C*, we can use *LSR B* rather than the VPN capsule and then change some IP addresses where necessary.

We map the C configuration file into the configuration construct presented in figure 7.15.

Policy–Specific Operations

7.3.2.1 C's First Operation

We have already added the services to the B construct and we can use it as a template to add these services to C too. However, we have to take into account that, whereas the policy-specific information is correct, the device- and instance-specific information for C must be configured separately, where appropriate. This specific information will be set straight shortly.

We perform the following composition between B and C:

$$\boxed{(c\text{: / } configure) >\oplus= (b\text{: / } configure)}$$

This composition performs a *conservative addition* (>) of the values of the matching elements and a (linkage) addition ($\oplus$) of missing elements from the capsule VPN's

element *configure* to the construct B's element *configure*. (The *conservative addition* (>) preserves always the *non-default effective values* of the B construct.) We get the C construct depicted in figure 7.16. We see now the advantages of using the Meta-CLI Model composition approach. The constructs are interacting in an intelligent way, in which each parameter is carefully considered and the relevant old values are not simply overridden by the new values. In this way, the policy configuration does not interfere with the device-specific information, when this is not required and there is already some pertaining information in place.

Device–Specific Operations

We have finished the *policy-* and *service-oriented* operations with regard to C and proceed now with the *instance-* and *device-specific* operations.

7.3.2.2 C's Second Operation

We set the addresses of the neighbor to the appropriate value wherever it occurs in the *router* context.

$$\boxed{(c\text{: } /\ configure\ /\ router\ /.../\ ip\ address\ /)\ =\ '10.10.10.2'}$$

We get the C construct into the state described in figure 7.17.

7.3.2.3 C's Third Operation

As in the previous paragraph, we set the address of the `neighbor` in the *address-family* context:

$$\boxed{(c\text{: } /\ configure\ /\ address\ family\ /.../\ ip\ address\ /)\ =\ '10.10.10.2'}$$

As in the case of the B construct, the above-mentioned operations *find* all the occurrences of descendants of the elements *router* and *address family*, respectively, of the B construct that *match* the *type*: *"ip address"* without regard of their *paths* and assign them all the same *value*: *10.10.10.2*. We obtain the C construct depicted in figure 7.18. The configuration constructs for the B and C will look like in figures

7.19 and 7.20 [3]. We can see that, during the configuration process of *LSR C*, we needed just *one* service- or policy-based, inter-entity operation, rather than *five*, as in the case of the configuration of *LSR B*, whereas the device-specific, intra-entity operations which were setting the neighbor's IP address, were similar for the two cases.

The device-specific operations were necessary since there was no information in the constructs for the neighbors' IP addresses. All the other specific information, like the IP addresses of the interfaces, was preserved in *LSR C* after the infusion of information from *LSR B*, due to the correct selection and usage of the value operation, which was, in this case, the *conservative addition* (>).

In the *FastEthernet:'2/1'* interfaces, there are a couple of default value elements that will not show up as commands in the configuration file : *no shutdown*, *cdp enable* and *ip directed broadcast*. Finally, the configuration files for the LSRs *B* and *C* will look like in figure 7.21. The network has now the *Red* and *Blue* VPNs configured.

[3]To compact the figures, long lines, like those containing the IP addresses, etc., have been broken in two.

```
VPN :
   - configure : terminal
      - hostname : vpn
      - ip vrf
         - name : Red
            - rd : '65050:1'
            - route target : ~
               - direction : import
               - route target ext community : '65050:1'
            - route target : ~
               - direction : export
               - route target ext community : '65050:1'
         - name : Blue
            - rd : '65051:1'
            - route target : ~
               - direction : import
               - route target ext community : '65051:1'
            - route target : ~
               - direction : export
               - route target ext community : '65051:1'
      - interface
         - Ethernet : '1/0'
            - ip vrf forwarding : Red
      - interface
         - Ethernet : '1/1'
            - ip vrf forwarding : Blue
      - router
         - ospf : 20
            - vrf : Blue
               - network
                  - address & wildcard mask : '10.0.0.0 0.255.255.255'
                  - area : 0
               - redistribute
                  - protocol : bgp
                  - process id : 65500
                  - metric type : 1
                  - subnets : ~
         - bgp : 65500
            - synchronization : no ~
            - bgp default ipv4-unicast : no ~
            - neighbor
               - ip address :
                  - remote as : 65500
            - neighbor
               - ip address :
                  - update source : loopback 0
      - address-family
         - ipv4
            - unicast
               - vrf : Blue
                  - redistribute
                     - protocol : ospf
                     - process id : 20
                  - auto-summary : no ~
         - ipv4
            - unicast
               - vrf : Red
                  - redistribute
                     - protocol : static
                  - redistribute
                     - protocol : connected
         - vpnv4
            - neighbor
               - ip address :
                  - activate : ~
            - neighbor
               - ip address :
                  - send-community : extended
```

Figure 7.7: VPN capsule.

```
B :
   - configure : terminal
      - version : '12.1'
      - hostname : b
      - ip subnet-zero : ~
      - ip cef : ~
      - ip vrf
         - name : Red
            - rd : '65050:1'
            - route target : ~
               - direction : import
               - route target ext community : '65050:1'
            - route target : ~
               - direction : export
               - route target ext community : '65050:1'
         - name : Blue
            - rd : '65051:1'
            - route target : ~
               - direction : import
               - route target ext community : '65051:1'
            - route target : ~
               - direction : export
               - route target ext community : '65051:1'
      + interface
      + router
      - ip classless : ~
      - ip http server : no ~
      - cdp run : no ~
      + line
```

Figure 7.8: B construct, after the 1st operation.

```
B :
   - configure : terminal
      - version : '12.1'
      - hostname : b
      - ip subnet-zero : ~
      - ip cef : ~
      + ip vrf
      + interface
      + router
      - ip classless : ~
      - ip http server : no ~
      - cdp run : no ~
      - address-family
         - ipv4
            - unicast
               - vrf : Blue
                  - redistribute
                     - protocol : ospf
                     - process id : 20
                  - auto-summary : no ~
         - ipv4
            - unicast
               - vrf : Red
                  - redistribute
                     - protocol : static
                  - redistribute
                     - protocol : connected
         - vpnv4
            - neighbor
               - ip address :
                  - activate : ~
            - neighbor
               - ip address :
                  - send-community : extended
      + line
```

Figure 7.9: B construct, after the 2nd operation.

```
B :
  - configure : terminal
      - version : '12.1'
      - hostname : b
      - ip subnet-zero : ~
      - ip cef : ~
      + ip vrf
      - interface
          + Loopback : 0
          - Ethernet : '1/0'
              - ip address : '10.10.20.1 255.255.255.0'
              - ip directed-broadcast : ~
              - shutdown : no ~
              - cdp enable : ~
              - ip vrf forwarding : Red
          + Ethernet : '1/1'
          + FastEthernet : '2/0'
          + FastEthernet : '2/1'
      + router
      - ip classless : ~
      - ip http server : no ~
      - cdp run : no ~
      + line
```

Figure 7.10: B construct, after the 3rd operation.

```
B :
  - configure : terminal
      - version : '12.1'
      - hostname : b
      - ip subnet-zero : ~
      - ip cef : ~
      + ip vrf
      - interface
          + Loopback : 0
          + Ethernet : '1/0'
          + Ethernet : '1/1'
          + FastEthernet : '2/0'
          - FastEthernet : '2/1'
              - ip address : '10.10.60.1 255.255.255.0'
              - ip directed-broadcast : ~
              - shutdown : no ~
              - cdp enable : ~
              - ip vrf forwarding : Blue
      + router
      - ip classless : ~
      - ip http server : no ~
      - cdp run : no ~
      + line
```

Figure 7.11: B construct, after the 4th operation.

```
B :
  - configure : terminal
     - version : '12.1'
     - hostname : b
     - ip subnet-zero : ~
     - ip cef : ~
     + ip vrf
     + interface
     - router
        - ospf : 10
           - network
              - address & wildcard mask : '10.0.0.0 0.255.255.255'
              - area : 0
        - ospf : 20
           - vrf : Blue
              - network
                 - address & wildcard mask : '10.0.0.0 0.255.255.255'
                 - area : 0
              - redistribute
                 - protocol : bgp
                 - process id : 65500
                 - metric type : 1
                 - subnets : ~
        - bgp : 65500
           - synchronization : no ~
           - bgp default ipv4-unicast : no ~
           - neighbor
              - ip address :
                 - remote as : 65050
           - neighbor
              - ip address :
                 - update source : loopback 0
     - ip classless : ~
     - ip http server : no ~
     - cdp run : no ~
     + line
```

Figure 7.12: B construct, after the 5th operation.

```
B :
  - configure : terminal
     - version : '12.1'
     - hostname : b
     - ip subnet-zero : ~
     - ip cef : ~
     + ip vrf
     + interface
     - router
        - ospf : 10
           - network
              - address & wildcard mask : '10.0.0.0 0.255.255.255'
              - area : 0
        - ospf : 20
           - vrf : Blue
              - network
                 - address & wildcard mask : '10.0.0.0 0.255.255.255'
                 - area : 0
              - redistribute
                 - protocol : bgp
                 - process id : 65500
                 - metric type : 1
                 - subnets : ~
        - bgp : 65050
           - synchronization : no ~
           - bgp default ipv4-unicast : no ~
           - neighbor
              - ip address : '10.10.10.3'
                 - remote as : 65050
           - neighbor
              - ip address : '10.10.10.3'
                 - update source : loopback 0
     - ip classless : ~
     - ip http server : no ~
     - cdp run : no ~
     + address-family
     + line
```

Figure 7.13: B construct, after the 6th operation.

```
B :
  - configure : terminal
      - version : '12.1'
      - hostname : b
      - ip subnet-zero : ~
      - ip cef : ~
      + ip vrf
      + interface
      + router
      - ip classless : ~
      - ip http server : no ~
      - cdp run : no ~
      - address-family
          - ipv4
              - unicast
                  - vrf : Blue
                      - redistribute
                          - protocol : ospf
                          - process id : 20
                      - auto-summary : no ~
          - ipv4
              - unicast
                  - vrf : Red
                      - redistribute
                          - protocol : static
                      - redistribute
                          - protocol : connected
          - vpnv4
              - neighbor
                  - ip address : '10.10.10.3'
                      - activate : ~
              - neighbor
                  - ip address : '10.10.10.3'
                      - send-community : extended
      + line
```

Figure 7.14: B construct, after the 7th operation.

```
C :
   - configure : terminal
      - version : '12.1'
      - hostname : c
      - ip subnet-zero : ~
      - ip cef : ~
      - interface
         - Loopback : 0
            - ip address : '10.10.10.3 255.255.255.0'
            - ip directed-broadcast : ~
         - Ethernet : '1/0'
            - ip address : '10.10.30.1 255.255.255.0'
            - ip directed-broadcast : ~
            - shutdown : no ~
            - cdp enable : ~
         - Ethernet : '1/1'
            - ip address : no ~
            - ip directed-broadcast : no ~
            - shutdown : ~
            - cdp enable : no ~
         - FastEthernet : '2/0'
            - ip unnumbered : ~
               - type : loopback
               - number : 0
            - tag-switching ip : ~
         - FastEthernet : '2/1'
            - ip address : '10.10.70.1 255.255.255.0'
            - ip directed-broadcast : ~
            - shutdown : no ~
            - cdp enable : ~
      - router
         - ospf : 10
            - network
               - address & wildcard mask : '10.0.0.0 0.255.255.255'
               - area : 0
      - ip classless : ~
      - ip http server : no ~
      - cdp run : no ~
      - line
         - type : aux
            - line number : 0
         - type : vty
            - line number : 0
            - ending line number : 4
            - password : cisco
            - login : no ~
         - type : console
            - exec timeout
               - minutes : 0
               - seconds : 0
            - transport input : none
```

Figure 7.15: C construct.

```
C :
   - configure : terminal
      - version : '12.1'
      - hostname : c
      - ip subnet-zero : ~
      - ip cef : ~
      + ip vrf
      - interface
         - Loopback : 0
            - ip address : '10.10.10.3 255.255.255.0'
            - ip directed-broadcast : ~
         - Ethernet : '1/0'
            - ip address : '10.10.30.1 255.255.255.0'
            - ip directed-broadcast : ~
            - shutdown : no ~
            - cdp enable : ~
            - ip vrf forwarding : Red
         + Ethernet : '1/1'
         - FastEthernet : '2/0'
            - ip unnumbered : ~
               - type : loopback
               - number : 0
            - tag-switching ip : ~
         - FastEthernet : '2/1'
            - ip address : '10.10.70.1 255.255.255.0'
            - ip directed-broadcast : ~
            - shutdown : no ~
            - cdp enable : ~
      - router
         + ospf : 10
         + ospf : 20
         + bgp : 65050
      + address-family
      - ip classless : ~
      - ip http server : no ~
      - cdp run : no ~
      + line
```

Figure 7.16: C construct, after the 1st operation.

```
C :
   - configure : terminal
      - version : '12.1'
      - hostname : c
      - ip subnet-zero : ~
      - ip cef : ~
      + ip vrf
      + interface
      - router
         - ospf : 10
            - network
               - address & wildcard mask : '10.0.0.0 0.255.255.255'
               - area : 0
         - ospf : 20
            - vrf : Blue
               - network
                  - address & wildcard mask : '10.0.0.0 0.255.255.255'
                  - area : 0
               - redistribute
                  - protocol : bgp
                  - process id : 65500
                  - metric type : 1
                  - subnets : ~
         - bgp : 65050
            - synchronization : no ~
            - bgp default ipv4-unicast : no ~
            - neighbor
               - ip address : '10.10.10.2'
                  - remote as : 65050
            - neighbor
               - ip address : '10.10.10.2'
                  - update source : loopback 0
      + address-family
      - ip classless : ~
      - ip http server : no ~
      - cdp run : no ~
      + line
```

Figure 7.17: C construct, after the 2nd operation.

```
C :
  - configure : terminal
      - version : '12.1'
      - hostname : c
      - ip subnet-zero : ~
      - ip cef : ~
      + ip vrf
      + interface
      + router
      - address-family
         - ipv4
            - unicast
               - vrf : Blue
                  - redistribute
                     - protocol : ospf
                     - process id : 20
                  - auto-summary : no ~
         - ipv4
            - unicast
               - vrf : Red
                  - redistribute
                     - protocol : static
                  - redistribute
                     - protocol : connected
         - vpnv4
            - neighbor
               - ip address : '10.10.10.2'
                  - activate : ~
            - neighbor
               - ip address : '10.10.10.2'
                  - send-community : extended
      - ip classless : ~
      - ip http server : no ~
      - cdp run : no ~
      + line
```

Figure 7.18: C construct, after the 3rd operation.

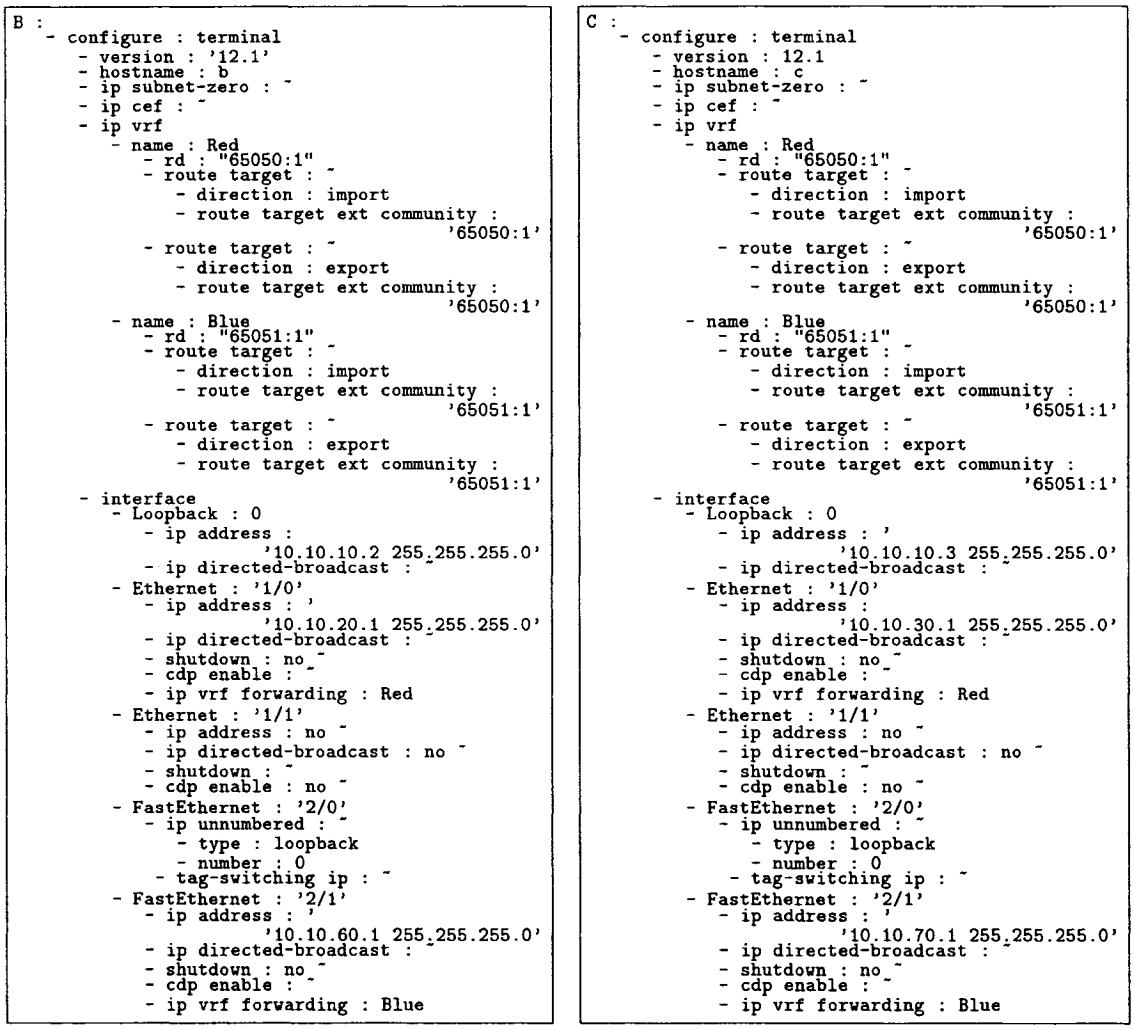

Figure 7.19: B and C constructs, after the compositions.

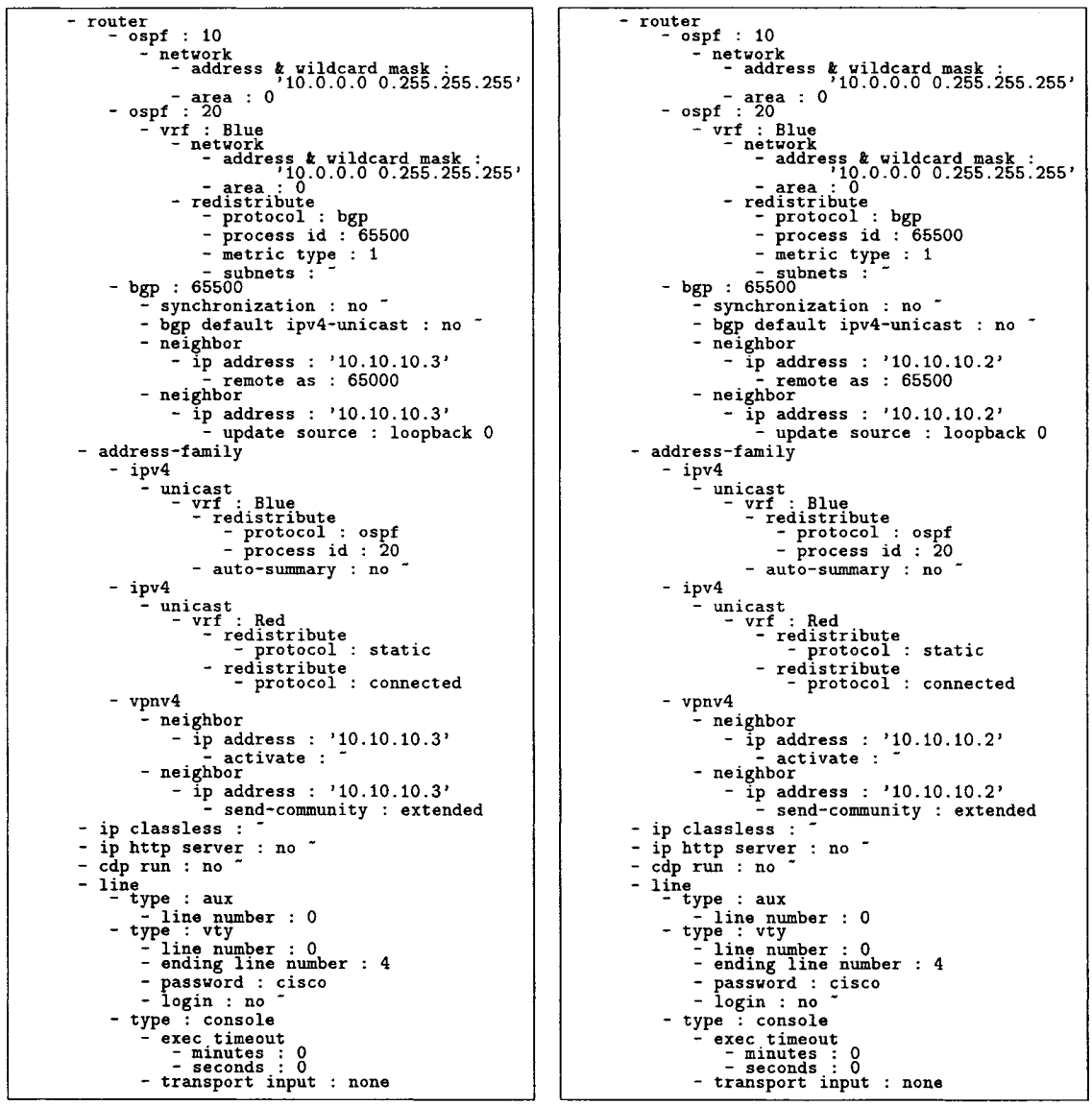

Figure 7.20: B and C constructs, after the compositions (continued).

```
!
version 12.1
!
hostname B
!
ip subnet-zero
ip cef
!
ip vrf Red
   rd 65050:1
   route-target export 65050:1
   route-target import 65050:1
!
ip vrf Blue
   rd 65051:1
   route-target export 65051:1
   route-target import 65051:1
!
interface Loopback0
   ip address 10.10.10.2 255.255.255.255
   no ip directed-broadcast
!
interface Ethernet 1/0
   ip vrf forwarding Red
   ip address 10.10.20.1 255.255.255.0
!
interface Ethernet 1/1
   no ip address
   no ip directed-broadcast
   shutdown
   no cdp enable
!
interface FastEthernet 2/0
   ip unnumbered loopback0
   tag-switching ip
!
interface FastEthernet 2/1
   ip vrf forwarding Blue
   ip address 10.10.60.1 255.255.255.0
!
router ospf 10
   network 10.0.0.0 0.255.255.255 area 0
!
router ospf 20 vrf Blue
   network 10.0.0.0 0.255.255.255 area 0
   redistribute bgp 65500 metric-type 1 subnets
!
router bgp 65500
   no synchronization
   no bgp default ipv4-unicast
   neighbor 10.10.10.3 remote-as 65500
   neighbor 10.10.10.3 update-source loopback 0
!
address-family ipv4 vrf Blue
   redistribute ospf 20
   no autosummary
   exit address-family
!
address-family ipv4 vrf Red
   redistribute static
   redistribute static connected
   exit address-family
!
address-family vpnv4
   neighbor 10.10.10.3 activate
   neighbor 10.10.10.3 send-community extended
!
ip classless
no ip http server
!
no cdp run
!
line con 0
   exec-timeout 0 0
   transport input none
line aux 0
line vty 0 4
   password cisco
   no login
!
end
```

```
!
version 12.1
!
hostname C
!
ip subnet-zero
ip cef
!
ip vrf Red
   rd 65050:1
   route-target export 65050:1
   route-target import 65050:1
!
ip vrf Blue
   rd 65051:1
   route-target export 65051:1
   route-target import 65051:1
!
interface Loopback0
   ip address 10.10.10.3 255.255.255.255
   no ip directed-broadcast
!
interface Ethernet 1/0
   ip vrf forwarding Red
   ip address 10.10.30.1 255.255.255.0
!
interface Ethernet 1/1
   no ip address
   no ip directed-broadcast
   shutdown
   no cdp enable
!
interface FastEthernet 2/0
   ip unnumbered loopback0
   tag-switching ip
!
interface FastEthernet 2/1
   ip vrf forwarding Blue
   ip address 10.10.70.1 255.255.255.0
!
router ospf 10
   network 10.0.0.0 0.255.255.255 area 0
!
router ospf 20 vrf Blue
   network 10.0.0.0 0.255.255.255 area 0
   redistribute bgp 65500 metric-type 1 subnets
!
router bgp 65500
   no synchronization
   no bgp default ipv4-unicast
   neighbor 10.10.10.2 remote-as 65500
   neighbor 10.10.10.2 update-source loopback 0
!
address-family ipv4 vrf Blue
   redistribute ospf 20
   no autosummary
   exit address-family
!
address-family ipv4 vrf Red
   redistribute static
   redistribute static connected
   exit address-family
!
address-family vpnv4
   neighbor 10.10.10.2 activate
   neighbor 10.10.10.2 send-community extended
!
ip classless
no ip http server
!
no cdp run
!
line con 0
   exec-timeout 0 0
   transport input none
line aux 0
line vty 0 4
   password cisco
   no login
!
end
```

Figure 7.21: LSRs B and C config files, after the compositions.

Chapter 8

Conclusions

The Meta-CLI Model provides a simple solution to the text-only configuration commands and files, which confers universality and scalability.

The arduous task is to translate the CLI commands and context dependencies, etc. into Meta-CLI trees, but this overhead might be cost-effective when considering the resources wasted for manual configuration of thousands of interfaces for an *ISP* network and the major risks due to potential configuration errors, when dealing with overlapping domains and policies, multiple, scheduled configurations, etc.

The Meta-CLI Model uses the tree structures, which are a common, universal topology. This means that, *mutatis mutandis*, the model might be adapted for different other applications in various domains, like the *Model Checking* and *Feature Interaction.*

The `tree` data structures are well implemented in some of the main programming languages, like: *C*, *C++*, *Java*, *Python*, etc. This provides a valuable framework for the easy implementation and development of the Meta-CLI Model.

The Meta-CLI Model's algebra has a degree of autonomy and generality that may allow for its adaptation and usage for other domains of application.

Acronyms and Abbreviations

API	Application Programming Interface
BGP	Border Gateway Protocol
BTEF	Branch Target for Execution Failure
CLI	Command-Line Interface
CLIF	CLI File
COPS	Common Open Policy Service
COPS-PR	COPS Usage for Policy Provisioning
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CSPF	Constrained Shortest Path First
DB	Data Base
DCPF	Device Connection Procedure File
DiffServ	Differentiated Services
DEN	Directory Enabled Networks
DENng	Directory Enabled Networks, next generation
DTD	Document Type Definition
EGP	Exterior Gateway Protocol
EIGRP	Enhanced IGRP
FTP	File Transfer Protocol
GUI	Graphical User Interface
IETF	Internet Engineering Task Force
IGRP	Interior Gateway Routing Protocol
IP	Internet Protocol
ISP	Internet Service Provider
IOS	Internetworking Operating System

ISDN	Integrated Services Digital Network
IS-IS	Intermediate System-to-Intermediate System
LSP	Label Switched Path
MIB	Management Information Base
MPLS	Multiprotocol Label Switching
NFA	Non-deterministic Finite Automaton
NVRAM	Non-volatile Random Access Memory
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
OID	Object Identifier
OSPF	Open Shortest Path First protocol
PCIM	Policy Core Information Model
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIB	Policy Information Base
PRID	Policy Rule Instance Identifier
QoS	Quality of Service
RAM	Random Access Memory
RDBMS	Relational Database Management System
RFC	Request for Comments (published by the IETF)
RIP	Routing Information Protocol
RSVP	Resource Reservation Protocol
RTE	Runt-time Exception
SNMP	Simple Network Management Protocol
SPF	Shortest Path First
SPPI	Structure of Policy Provisioning Information
SSH	Secure Shell
TCP/IP	Transmission Control Protocol/Internet Protocol

TE	Traffic Engineering
TED	Traffic Engineering DB
TI	Technician's Interface
VLAN	Virtual Local Area Network
VoIP	Voice over Internet Protocol
VPN	Virtual Private Network
WFQ	Weighed Fair Queuing
WRED	Weighted Early Random Detect
XML	Extensible Markup Language

Bibliography

[1] Craig Zacker. *Networking : The Complete Reference.* Osborne/McGrawHill, 2001.

[2] Toby and Anthony Velte. *Cisco : A Beginner's Guide.* Osborne/McGrawHill, 2001.

[3] *CLI Definition.* http://whatis.techtarget_com/definition/0,,sid9_gci213627,00.html.

[4] Cisco Systems Inc. *Internetworking Terms and Acronyms Book.* http://www.cisco.com/univercd/cc/td/doc/cisintwk/ita/ita_book.pdf.

[5] Cisco Systems Inc. *Using the Command Line Interface.* http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/12cgcr/fun_c/-fcprt1/fcui.htm.

[6] Jakob Nielsen. *Ten Usability Heuristics of a User Interface Design.* http://www.useit.com/papers/heuristic/heuristic_list.html.

[7] Wipro Technologies. *Building Cisco-style Command Line Interfaces.* http://www.bitpipe.com/data/detail?id=1024659156_47&type=RES&x=1753866860.

[8] Douglas E.Comer. *Computer Networks and Internets with Internet Applications.* Third Edition, Prentice Hall, 2001.

[9] International Engineering Consortium. *Multiprotocol Label Switching (MPLS).* http://www.iec.org/online/tutorials/mpls/.

[10] International Engineering Consortium. *Generalized Multiprotocol Label Switching (GMPLS).* http://www.iec.org/online/tutorials/gmpls/.

[11] International Engineering Consortium. *Intranets and Virtual Private Networks (VPNs).*
http://www.iec.org/online/tutorials/int_vpn/.

[12] UC Berkeley - Teaching Library Internet Workshops. *What is the Internet, the World Wide Web, and Netscape?.*
http://www.lib.berkeley.edu/TeachingLib/Guides/Internet/WhatIs.html.

[13] Cisco Systems Inc. *Introduction to Internet.*
http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/introint.htm.

[14] Cisco Systems Inc. *Internetworking Technology Handbook; Introduction to LAN Protocols.*
http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/introlan.htm.

[15] Cisco Systems Inc. *Internetworking Technology Handbook; Introduction to WAN Technologies.*
http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/introwan.htm.

[16] Cisco Systems Inc. *Internetworking Technology Handbook; Routing Basics.*
http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/routing.htm#xtocid2.

[17] Brian Hill. *Cisco : The Complete Reference.* Osborne/McGrawHill, 2002.

[18] Chris Brenton, Bob Abruhoff. *Mastering Cisco Routers*, 2nd edition. Sybex, 2002.

[19] James Boney. *Cisco IOS in a Nutshell.* O'Reilly, 2002.

[20] Joe Habracken. *Practical Cisco Routers.* Que Corporation, 1999.

[21] Harry Newton. *Newton's Telecom Dictionary.* CMP Books, 2002.

[22] Cisco Systems Inc. *Internetworking Technology Handbook; Routing Basics.* http://www.cisco.com/univercd/cc/td/doc/product/software/ios112/sbook/-siprout.htm#xtocid2423731.

[23] Cisco Systems Inc. *Cisco IOS Solutions for Network Protocols.* Macmillan Technical Publishing CMP Books, 2002.

[24] Steve Waldbusser, Jon Saperia, Thippanna Hongal. *Policy Based Management MIB.* http://www.ietf.org/internet-drafts/draft-ietf-snmpconf-pm-11.txt, 2002.

[25] Rajiv Malik, Steve Sycamore, Bill Tracey. *Method and Apparatus for Configuration Management in Communication Networks.* ftp://ftp.snmp.com/snmpconf/CabletronPatents/5832503.pdf, United states Patent : 5,832,503; 1998.

[26] Lundy Lewis, Rajiv Malik, Steve Sycamore, et al. *Method and Apparatus for Defining and Enforcing Policies for Configuration Management in Communication Networks.* ftp://ftp.snmp.com/snmpconf/CabletronPatents/5832503.pdf, United states Patent : 5,832,503; 1998.

[27] Byung-Joon Lee, Taesang Choi, Taesoo Jeong. *X-CLI : CLI-based Management Architecture Using XML.* Asia-Pacific Networks Operations and Management Symposium (APNUMS), 2002, Jeju Island, Korea.

[28] Electronics and Telecommunications Research Institute, Daejeon, Korea. *Wise < TE > : MPLS Traffic Engineering and VPN Management System: Wise < TE/VPN >.* www.etri.re.kr/e_etri/intro/newtech/etri21c_14.html, 2002.

[29] J. Case, M. Fedor, M. Schoffstall, J. Davin. *rfc1157 A Simple Network Management Protocol (SNMP)*, RFC1157.

www.kblabs.com/lab/lib/rfcs/1100/rfc1157.txt.html, 1990.
(www.cis.ohio-state.edu/cgi-bin/rfc/rfcXXXX.html).

[30] J. Case, K. McCloghrie , M. Rose, S. Waldbusser. *Introduction to Community-based SNMPv2*, etc. (RFC1901...RFC1908), 1996.
www.cis.ohio-state.edu/cgi-bin/rfc/rfcXXXX.html.

[31] Carnegie Mellon Software Engineering Institute. *Simple Network Management Protocol.*
http://www.sei.cmu.edu/str/descriptions/snmp.html.

[32] K. Chan, J. Seligson, D. Durham, S. Gai, K. McCloghrie, S. Herzog, F. Reichmeyer, R. Yavatkar, A. Smith. *COPS Usage for Policy Provisioning (COPS-PR)*, RFC3084.
www.faqs.org/rfcs/rfc3084.html, 2001.

[33] J. Saperia, J. Schönwälder. *Policy-Based Enhancements to the SNMP Framework.*
www.ibr.cs.tu-bs.de/vs/papers/policy-tr-00-02.ps.gz, 2000.

[34] A. Westerinen, J. Schnitzlein, J. Strassner, M. Scherling, et al. *Terminology for Policy-Based Management*, 2001 RFC3198.
http://www.faqs.org/rfcs/rfc3198.html.

[35] Intelliden Corporation. *Intelliden > Products.*
http://intelliden.com/product/.

[36] S. Shenker, J. Wroclawski. *Network Element Service Specification Template*, 1997. RFC2216.
http://www.faqs.org/rfcs/rfc2216.html.

[37] Thomas M. Thomas II, Rajah Chowbay, Doris Pavlichek, Wayne W. Downing III, Lawrence H. Dwyer III, James Sonderegger. *Juniper Networks Reference Guide : JUNOS Routing, Configuration and Architecture.* Addison-Wesley, 2002.

[38] James Knapp. *Nortel Networks : The Complete Reference.* Osborne/McGraw-Hill, 2000.

[39] Cisco Systems Inc. *Configuring a Basic MPLS VPN.* http://www.cisco.com/en/US/tech/tk436/tk428/technologies_tech_note-09186a00800a6c11.shtml.

[40] *BONSAI: Dictionary of Latin Botanical Terms.* http://www.cyber-north.com/bonsai/latdic.html.

[41] Intelliden Corporation. http://intelliden.com/news/.

[42] Intelliden Corporation's White Paper. *The Business Case for Business Driven Device Management.* http://intelliden.com/files/WPBDDM.pdf.

[43] Intelliden Corporation's White Paper. *Using an Information Model to Achieve Better Network and System Management.* http://intelliden.com/files/WPInfoMod.pdf.

[44] Intelliden Corporation's White Paper. *The Value of Standards.* http://intelliden.com/files/WPStndrds.pdf.

[45] Intelliden Corporation's White Paper. *The VPN Dilemma.* http://intelliden.com/files/WPVPN.pdf.

[46] Intelliden Corporation's White Paper. *The Power of Directory Enabled Networks (DEN).* http://intelliden.com/files/WPPowerDEN.pdf.

[47] Network Wizards. http://www.nw.com.

[48] Distributed Management Task Force Inc. http://www.dmtf.org.

[49] TeleManagement Forum. http://www.tmforum.org.