INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Université d'Ottawa · University of Ottawa

Université d'Ottawa • University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

ZINE El-ABIDINE, Khaldoune

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

M.A.Sc. (Electrical Engineering)

GRADE - DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

A Development Framework for Efficient Uniform Control of Heterogeneous Communication Networks

Dan Ionescu

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

Nicolas Georganas

Dorina Petriu

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

SIGNATURE

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

A Development Framework for

Efficient Uniform Control

of

Heterogeneous Communication Networks

By

Khaldoune ZINE EL-ABIDINE

A thesis submitted to the
School of Graduate Studies and Research
In partial fulfillment of the requirements for the degree of

Masters of Applied Science

Ottawa-Carleton Institute for Electrical and Computer Engineering
School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa

December 2001

I hereby declare that I am the sole author of this thesis.

I authorize the University of Ottawa to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Khaldoune ZINE EL-ABIDINE

I further authorize the University of Ottawa to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Khaldoune ZINE EL-ABIDINE

Abstract

As new Internet networking technologies appear, there is a growing need for efficient control of Network Elements (NE) (i.e. routers, switches, etc.). However, today's NE control is largely a manual activity, therefore inefficient and unscalable, and the inherent heterogeneous nature of complex networks –that is, the fact that NEs are of different technologies and have different functionality and configuration-management interfaces– makes NE control even more difficult and challenging.

This thesis presents a solution that allows the development of automated, distributed web-based control of heterogeneous NEs through a common interface that hides the underlying NE-specific differences, allowing control of heterogeneous NEs in a transparent way. This solution consists in a software development framework oriented towards reliability –thus service availability–, performance –hence scalability–, and fast extensibility –hence easy integration and rapid adaptability to the ever-changing networking technologies–, without neglecting security.

General design requirements are first discussed. Next, particular attention is given to CLI (Command Line Interface) protocols, where issues related to the automation of CLI interaction are extensively discussed. A real-time approach, using extended finite state machines, is proposed as a key element in this framework. Finally, a working and efficient implementation that supports CLI automation is presented, where emphasis is laid upon the short delays –typically a day– required to augment the application with a new set of NE control operations –Traffic Conditioning operation set for example.

Acknowledgements

I would like to thank my supervisor, Dr. Dan Ionescu, for his guidance, encouragement, optimism and support throughout my research. His ability to set lofty goals and challenge one to achieve them is responsible for the magnitude of this thesis.

I would also like to thank my colleagues in the NCCT laboratory at University of Ottawa for their valuable help and unconditional support throughout my work in general and thesis research in particular.

Finally, I would like to thank my family for their vital moral support, and more particularly my parents for their priceless efforts and sacrifices without which I could never have succeeded.

Table of contents

List of Figures

List of Tables

Glossary of Terms

API	Application Programming Interface
ATM	Asynchronous Transfer Mode
CLI	Command Line Interface
CMIP	Common Management Information Protocol
CORBA	Common Object Request Broker Architecture
CPA	Control Protocol Adapter
DCOM	Distributed Component Object Model
FA	Finite Automaton
FCAPS	Fault, Configuration, Accounting, Performance, and Security Management
FSM	Finite State Machine
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
MIB	Management Information Base
MIME	Multimedia Internet Mail Exchange
NE	Network Element (i.e. routers and switches, etc.)
NM	Network Management
OMG	Object Management Group
QoS	Quality of Service
Reg.Exp.	Regular Expression
REM	Regular Expression Module
RPC	Remote Procedure Call
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
SONET	Synchronous Optical NETwork
SSH	Secure Shell
SSL	Secure Sockets Layer
TCP/IP	Transmission Control Protocol/Internet Protocol
UML	Unified Modeling Language
Web	the Worldwide Web.
XML	Extensible Markup Language

1 Introduction

1.1 Motivation and Research Objectives

Today's communication network comprises many different transmission techniques and technologies. An IP packet might be sent over Ethernet, ATM, SONET, optical, etc. to travel from one end to another. Multi-vendor network management and control, i.e. the management and control of "a network composed of elements from many different suppliers supporting different levels of functionality with different network architectures and protocols" [33], is setting up the grounds for a more performing and efficient QoS-capable network. For the communication industry, the capability of providing real-time control to different vendors' NEs is therefore becoming a crucial task and a difficult challenge at the same time. Indeed, one of the causes of the year 2001 downfall of the global economy was the lack of efficiency in the control of various communications networks. As a consequence, the control of NEs in general and of heterogeneous NEs in particular represents one of the hottest research subjects in communications area. It is related to traffic engineering, bandwidth management, admission control, QoS and other research subjects. These researches are mostly motivated by the need of endowing audio and video stream transmissions with high quality.

The need for vendor-independent solutions emerged at least fifteen years ago. Since then, significant progress in this field has been made. Today, CMIP and SNMP are the results of this progress, although SNMP is considered to be the *de facto* standard [27, 31, 33] in Network Management (NM) given that most vendors offer SNMP-based products. However, SNMP designers took a "minimalist" approach to NM, since "their design had to be effective for a broad range of devices, therefore requiring a low common denominator" [33]. As a result, there is always a remaining non-standard vendor-specific part for each NE, usually called "Enterprise MIB" [1], until a new update of the Standard covers the features supported by that proprietary segment.

Intensive research, briefly reviewed in Chapter 2, has focused on the above problem to allow integrated NM and provide uniform NM interfaces in order to release the full potential of the network. Because a manual type of intervention is still omnipresent in today's world of NM [26, 44], the previous efforts always implicitly assumed that the end manager is a human operator or a small number of collaborating human operators, thus characterized by slow activity. It was also implicitly assumed that the frequency at which a given network's configuration[1] changes today is typically low, that is, no more than a few times per month. Therefore, none of the previous research efforts focused on *efficiency*[2] in network *control* since it would have been rather irrelevant considering the

---

[1] That is, topology, architecture and settings of the different NEs composing the network

[2] In terms of response times for heavy and complex control operations that may involve a large number of end managers, as well as resource utilization implied by such activity.

actual situation. However, state-of-the-art networking technologies tend to provide QoS support, and thus new commercial services –possibly targeted towards mass-public– will eventually appear.

These services will have new and more stringent requirements. In particular, they will require frequent real-time configuration of the network, thus demand efficient and reliable control. With the inherent heterogeneity of the network introduced above, the development of NE control applications that would bridge the gap becomes a tedious task. Meanwhile, the deployment of these commercial services is impeded and the network is not used to its full potential, resulting in important financial losses. As of this writing, no general software development framework for the control of heterogeneous NEs driven by performance and reliability concerns was found. Instead, most efforts in the literature are dedicated to high-level designs and architectures of integrated NM, where design and implementation details of *leaf managers*[1] are usually left unspecified. The development framework in question has to allow NE control which is:

- *Efficient*: in terms of resource utilization and response delays.
- *Scalable*: to be able to handle a large number of NEs simultaneously.
- *Reliable*: it should never leave a NE in an inconsistent or unknown state.
- *Secure*: to allow commercial-grade services.

---

[1] That is, software components communicating directly with NEs via wire management protocols.

- *Uniform and transparent*: using a common control interface to deal with heterogeneous NEs. The underlying NE differences –such as control protocol[1], vendor, version, etc.– have to be hidden behind that common interface, and automatically detected and handled by the implementation.

- *Automated*: it should be easy to automate long interaction sequences through simple operations.

- *Easily and rapidly extensible*: in order to allow rapid integration of new networking software and hardware as they appear, hence minimize the impact time-to-market has on product costs. In particular, implementers of new control procedures need only to focus on network control logic without worrying about other implementation details of the framework.

- *Distributed*: to allow remote control and configuration, preferably from any client in the world. In particular, it should deal appropriately with firewalls and existing security infrastructure.

Note that it should *not* be the purpose of the framework to set a new standard in network management. Instead, it should offer a *complementary* solution for *network element control* rather than a replacement for existing protocols.

This thesis presents a framework that satisfies the above requirements. Although the framework supports extensibility to any control protocol, particular analysis has been

---

[1] We agree to call "control protocol" any protocol used to control a network element.

conducted regarding CLI interaction automation and issues related to its use for NE control. There are several reasons for choosing CLI instead of SNMP or CMIP that will be discussed later in Chapter 5, but which consist mainly in performance and reliability issues: CLI reveals to be more efficient in most scenarios and control with CLI can easily be made reliable.

A functional prototype that supports advanced features of CLI automation is presented as a fundamental starting point of the framework itself. Some of these CLI features are:

a) Complex CLI interaction driven by NE responses. This interaction is modeled by UML extended finite state machines represented graphically.

b) Ability to parse multiple possible NE responses in parallel and make CLI interaction decisions according to the received response.

c) Ability to detect "asynchronous text" (traps) and undertake appropriate actions to handle that "textual event" (typically an asynchronous error message for example).

d) Stream-based text parsing for best performance.

The undeniable power of distributed computing is incorporated to this framework making the uniform control interface securely accessible via the Web to insure firewall-friendliness. SOAP (Simple Object Access Protocol) [3] is the chosen middleware to support web-based secure access. The framework remains, however, easily integrable to any other middleware (CORBA, for example).

1.2 Organization of the Thesis and Contributions

This thesis is organized to provide an incremental understanding of the problem of uniform control of heterogeneous network elements in a distributed environment.

In Chapter 2, related work in the field of Network Management in general and Network Control in particular is briefly reviewed.

Chapter 3 presents an overview of the Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), mathematical results in the field of regular expression matching, and the Unified Modeling Language (UML). Because UML components and state diagrams are both the application environment and the foundation of many concepts proposed in this thesis, UML is reviewed in reasonable detail and emphasis is laid upon UML's real-time extensions. SOAP and WSDL are reviewed to justify their choice as the middleware for the web-based architecture of the framework. Mathematical results related to regular expressions and finite automata are reviewed as well to demonstrate their efficiency since they are used in the implementation of the final prototype.

In Chapter 4, general issues related to the definition of a uniform NE control interface are discussed. Next, design requirements for applications targeting uniform control of heterogeneous NEs are defined. Finally, the architecture of our framework is presented in two steps: The web-based architecture using SOAP and WSDL is presented, and a generic server-side model based on *Control Protocol Adapters* (CPA) is defined.

In Chapter 5, the problem of choosing a control protocol for a given NE is presented. The choice of CLI as a starting point is justified. The particular case of CLI interaction protocols is studied, and specific relevant issues are presented and addressed. A component-based, real-time model is used to design the architecture. A new concept, called *Dynamic Triggering Filter*, is finally introduced as an evolution of regular-expression-based parsing in CLI interaction automation.

An implementation of the architecture introduced in the previous chapter is presented in Chapter 6. As a proof of concept, a limited uniform interface composed of two Traffic Conditioning (TC) operations is implemented. This interface handles two different types of NEs transparently. It is shown how fast and easy the augmentation of the prototype with these two operations was (typically a day), once the development framework was in place.

Finally, in Chapter 7, conclusions are presented on the costs and benefits of the presented framework for the deployment of uniform network control applications. Recommendations are made for future design and implementation of this framework.

This thesis contains several research contributions, which can be summarized as follows:

- Performance issues related to the control of network elements are specifically considered for the first time.

- A new methodology, which consists in uniform CLI interaction automation using a component-based real-time approach, is used for the first time in the field of network management and control.
- The requirements for uniform network control are presented. A generic framework that satisfies those requirements is presented.
- The technical aspects of the UML standard that are relevant to supporting the presented framework are examined. A tool that supports those aspects is briefly presented as well.
- The particular case of uniform CLI automation is studied extensively for the first time in Chapter 5, where the concept of "dynamic triggering filter" is also introduced.
- A fully functional prototype of the presented framework is constructed. A test application is produced and tested.
- Recommendations are made for future research in uniform control of heterogeneous network elements.

2 Related work

Despite the fact that the network control is one of the hottest research subjects in the area of computer communications, it received very little (if not at all) attention from the research and mainly from the industrial community. This fact is astonishing. There is common knowledge, and market analysts [41] confirmed it, that the present trend in the IT business is enhancing the mechanisms that allow providers to deliver real revenue, which means better, faster, easier to operate tools to provision, activate and control the network functionality. These desiderata cannot be accomplished without a complex service and network control platform able to define, create, and activate services in a dynamic way.

The difficulties of approaching the subject are manifold where the main one is related to the fact that there is no prior work done in the area of using feedback principles for controlling computer communications processes. It has to be mentioned, though, that in an invited talk M. Athans raised this issue even in a World Congress of IFAC in 1978 [40] and U. Warrier *et al.* [26] in 1988 say that "the issue of system control is as crucial to network management as system monitoring, yet it has been largely ignored". However, not too much work has been undertaken since for continuing this venue, and it is somehow strange in this research community that conjectures made about twenty years

ago did not have any resonance at all till recently. Of course, there are reasons for this silence. There are a series of conceptual misalignments between the two fields. The computer communication, as whole, relies heavily on the fact that the network is stateless and on the ad-hoc and local resolution of the control decisions that have to be taken in order to eventually propagate the whole stream of data in an end to end fashion without errors. These were the very best assumptions made when designing it. The control system strategy, on the contrary, assumes the presence of the controller which makes centralized decisions for the whole process it controls. The assumptions made for the design of computer communication networks were more than satisfactory for the timeless transfer of data, and started to be a hindering as soon as this transfer related to the transfer of time sensitive data such as audio and video streams or to hard real-time communication processes such data storage in disaster times. Today, the idea of applying control methodology into the area of communications processes and protocols starts to get ground.

2.1 Application control

Works related to the control of application resources can be found in [38], and in [39] one can find the conjecture about the analogy between the resource behavior in a communication network, related to QoS, and an automated control system controlling the level in a tank.

In order to guarantee an appropriate behavior of multimedia applications running over a computer communication network in [38] Baochun Li devised adaptive protocols in order to implement control strategies for compensating the variations of QoS resource requests and consumption per host, per hop, and per network at the same time. A particular feature of the control strategies to be applied to these types of problems which has to be mentioned is that the communications processes do not impose hard real-time constraints. The control strategies explored span from using PID controllers for controlling the resource requests and enabling process, to Kalman filters and eventually to fuzzy processes control techniques. The fuzzy control approach is a real contribution of the work done by the author. However, his research limited the area of control strategies to only OS of the host and to the control of resources managed at the host and at the application level, and although the word *resource* used within the scope of his document covered "network resources" (bandwidth, etc.) it was not specified how the *control* of such attributes on network devices would be materialized.

Another work in this area is a short paper of de Meer [39], where the author suggested an analogy between the level control processes and the allocation of resources in a QoS end-to-end communication. His work was inspired from another work of Alur *et al.* [43] in which the authors developed the paradigm of "water-level monitor" for introducing and reasoning about continuous end-to-end QoS control.

Other works concentrate only on solving a centralized control problem to some local adjustment by using information or signaling from protocols such as RTP [45] or others, though these attempts are not in the area of our exploits.

2.2 Network control

A white paper of CISCO comes to establish a general strategy for implementing a network control infrastructure, especially for providing QoS sensitive applications with the guaranteed parameters and priorities [44]. The white paper argues for a centralized architecture which gives users and network managers the proper tools to interact with the network keeping at the same time the network in a very stable state. Instead of advocating for a software which has the ability to signal QoS priority across the network, i.e. "trusted" QoS controls, where users are responsible for conforming to QoS policy when they launch certain applications, CISCO proposes to centralize QoS implementation, removing the need for end-systems to police themselves and letting network managers set up and enforce the policy within the "trusted" network. It is also mentioned that the trusted and distrusted QoS control policies can be combined such that a consistent policy deployment and enforcement can be accomplished. In this way network managers can control network traffic and deliver mission-critical applications across the enterprise while still enabling traffic delivery for other applications. Despite the very solid positioning, in bringing some general points to the solution of the QoS problem, the White Paper does not specify any methodology of achieving it in an engineering way. Moreover, it targets specific next-generation CISCO elements that are qualified as "intelligent" and thus may not be applied to non-CISCO network elements even though they were QoS-capable.

Another work in the area of network control can be found in [26]. In their paper, U. Warrier *et al.* introduced a language for NM called Network Management Language (NML). This work is a thoughtful approach to network management and control, and is one of the few that explicitly stress the importance of network *control* in network management. Besides the nature and details of the language they introduce (an extension to the Structured Query Language that may be used under an embedded form in a regular programming language), this paper draws attention to the crucial role of an abstraction layer that sits between a MIXP (Management Information Exchange Protocol in the most general sense: SNMP and CMIP are two such MIXPs, which we will also refer to as *control protocols*) and an AU (Application Unit). NML would be such a layer that "reduces the gap between MIXP and application programmers", because the MIXP is a low-level communication protocol from the programmer point of view (even though CMIP is considered as an upper layer protocol in the OSI reference model). Another crucial advantage of the NML is that it can cope with multiple MIXPs transparently. The paper also formulates the requirements for a NML such as grouping of operands, procedural control facility and error handling, and finally a mapping from NML to CMIP is briefly presented. Although we share all the arguments presented in that paper, we find that using CMIP as a final MIXP is restrictive given that "solutions based on CMIP had been few and far between" [33]. NML is intended to use other MIXPs as well but they did not indicate how such mappings would be done. Adding support for another MIXP may be costly and changes in the NML may require cascading and delicate implementation adaptations as well, which may limit the evolution of any NML unless

special care is given to design and implementation methods, for which a development framework for rapid implementation of control procedures is suitable –which we present in this Thesis. Finally, we reproach that performance considerations were completely absent in that paper and we argued previously how essential they are becoming today.

"The Tempest" is a network control framework developed in [46]. The authors of this paper propose a novel network control infrastructure driven by "the need for a single network supporting a large number of diverse services". First, they clarify what they consider as a blurred distinction between network management and network control; "Management will be taken to refer to those functions concerned with the "well-being" of network devices, while control will refer to functions which try to manipulate network devices into doing something useful". Next, they define the paradigm in which they work: design and deploy control *architectures* function of desired services instead of thinking of how to achieve a desired service given the available control architectures. The core idea is the ability to control a switch with multiple (independent) controllers by strictly partitioning the resources of that switch between the controllers. Every such logical partition is called a *switchlet* and the set of switchlets possessed by a controller is called a *virtual network*. Accordingly, their infrastructure contains a component used to divide a physical device into several logical ones (switchlets) so that distinct controllers can run multiple control architectures over the same physical node. Another required component in their infrastructure, called Caliban, adds a level of abstraction to the framework by exposing a small set of primitives which can be mapped to different underlying management protocols transparently. Therefore, controllers communicate

with Caliban via those primitives instead of addressing switchlets directly. Next, a Network builder module runs on top of the previous layers to actually specify and build virtual networks, and finally a control architecture endowed with basic building blocks is provided as a rich template for convenience. Once the infrastructure defined, this paper explains the intent of it by further discussing the concept of *service-specific control architectures* (in particular). In summary, the desired service is the starting point and a control architecture is built to match the requirements of the service using Caliban's primitives and a virtual network. This was proposed as an alternative to the general purpose control architecture paradigm, a sort of one-size-fits-all solution that is arguably being abandoned.

Our understanding of network control[1] and management is compliant with their definition. We also find the service-specific control architecture principle interesting and adhere to it, as outlined in Section 4.1 (although our framework transcends this principle). Their approach has numerous advantages, which include (but are not limited to):

- Incrementable control architectures and architectures dedicated to a single service.
- Fine-grained access control policies can be implemented by the Caliban server.

Unfortunately, it also has some liabilities:

---

[1] We define network element control as the mechanism of sending a series of commands to an element in order to control its state. These commands are verified within a given formalism to be error free, and are endowed with transparent error handling features such that unpredicted NE functional errors produce a roll back of the actions taken upon the NE thus leaving it in a consistent state.

- The client controller is restricted to the set of control primitives offered by Caliban. An overhead in communication time can also be observed due the introduction of an additional indirection (acknowledged in the paper itself).
- The described infrastructure targets ATM networks very specifically and some of the proposed techniques are not (yet) applicable to IP networks.
- The service-specific approach allows service providers to customize their control architecture but requires them to actually write it and deploy it if it cannot be built with the basic building blocks. Therefore, this approach is only suitable for well-defined and common services.

In fact, the paper focuses on control *architectures*, not *interfaces*. More precisely, the control *interface* reflected by Caliban is fixed and later on control architectures are built on top of it. The control *interface* however remains fixed or typically rarely updated (compared to control architectures deployed over it that are intended to change more often). The Caliban thus seems, after all, to be another Tempest-specific form of a standard management protocol (or control interface) such as SNMP or CMIP, and therefore it would encounter the same well-known problems, such as their being limited to a "common denominator" of functionality among heterogeneous network elements. It was explicitly acknowledged in their paper that deploying control architectures that would need specific switch features would be problematic with Tempest. As we argued earlier in this thesis, this inability to fully utilize the capabilities of a given network is an important obstacle.

The problem in Tempest is that only control architectures are customizable per-service whereas Caliban's interface is service-unaware. What would fix the problem is Caliban's interface itself be service-dependant as well so that a different interface can be exposed to different controllers themselves providing for specific services. Each of these control interfaces would still map transparently to underlying NE control primitives. The difference is that since the interface is customizable per-service, the full range of NE capabilities can be reached. Our framework allows flexible control interfaces to be designed seamlessly. Our work remains, however, more modest than Tempest in certain aspects since we do not define a global virtual networking architecture for example.

2.3 Network management

Most work in NM found in the literature deals with *network* management –in contrast with *element control* – and only a few of the previous efforts explicitly refer to NE control.

Artificial intelligence was applied to network management several times in the past. One can find applications of expert systems[1] in introducing semi-automation mechanisms for network management in [24] and [25]. These mechanisms allow intelligent network fault management, network configuration validation, performance monitoring methods,

[1] Expert systems are used in the field of Artificial Intelligence (AI) to separate the knowledge base from the reasoning engine. They are used in the industry as decision makers in situations such as train routing, etc.

etc. It is not our goal to discuss the quality of these papers. However, it has to be mentioned that while the former barely mentioned CMIP as a final control protocol, the latter left it unspecified and both papers do not indicate how abstract operations used in the rule base at the expert system level would be mapped to concrete control commands in a transparent and uniform way. Actually, they both assume the existence of a standard control protocol which makes us believe that SNMP or CMIP were implicitly targeted. Using the latter two protocols for element control (although they are indispensable for management) has at least two limitations: a) performance (see Chapter 5) and b) inability to manage non-standard features residing in Enterprise MIBs (since a common-denominator approach is usually adopted).

Ku *et al.* [36] focus on intelligent NM for gigabit routers. They propose a Java-based architecture (for platform independence) that consists of a complex server acting as a manager on behalf of a human manager GUI applet. The server contains several collaborating engines (FCAPS). In particular, the engine responsible of submitting configuration changes to the router uses import/export to download/upload configuration files from the router. More specifically, in order to perform a configuration task, a) a configuration file residing on the router is downloaded to the server (the manager), b) it is imported to some format, c) configuration is made upon the imported file, d) the file is exported to the router's format and finally e) the configuration file is uploaded to the router. This approach has several liabilities:

- It requires a download/upload for each and every configuration. This operation is typically slow.

- It requires previous knowledge of the internal format in which the router stores its own configuration. This format is not always provided by the manufacturer of the router or NE in general.

- Assuming that the previous format is known, this approach requires implementing a parser for that format, which might be a tedious task. This affects the extensibility of the architecture since it makes the integration of a new router platform a difficult mission.

- Uploading a configuration file to a router may require rebooting the machine. The resulting interruption of service is usually undesirable.

Although we find that the global architecture they propose is interesting, due to the above limitations the resulting performance is poor. It is measured to 12 seconds for each import/export cycle. While configuration processing can generally be done efficiently offline and then submitted to the router, it can be anticipated that in the scenario described in the previous section configuration requests will typically be numerous and involve small incremental NE-configuration changes. Therefore, the benefit that might be obtained from offline processing is diluted by a relatively large delay required to import/export a router's configuration file. Their architecture could have better performance if the import/export module were replaced by an efficient implementation providing the same services to the upper layer in a vendor-independent manner. Our framework is designed to allow this facility.

An original approach is introduced by Do-Hyeon Kim *et al.* in [35]. Their work aims at managing heterogeneous networks with a novel architecture that uses an application program interface and lower layer managers. The proposed architecture is compared to traditional ones such as manager-of-managers and common platform architecture and is arguably interesting. We find, however, that the main contribution of their work resides in the fact that a clear separation between managing LANs and WANs is made. This decision came for taking the "reality" factor into account, that is, the fact that most network management systems employed in LANs and WANs had been developed separately; LAN resources are usually managed in a standard way using SNMP, and WAN resources are managed using proprietary protocols and only a few devices support CMIP. Accordingly, their integrated management architecture uses LAN managers and WAN managers that transparently handle different protocols to provide a consistent interface. However, network *control* is very briefly mentioned regarding LAN managers and SNMP is the used protocol, which have limitations that we mentioned above. Furthermore, no mechanism for mapping higher-level API operations to proprietary APIs is discussed and we find that too much attention is given to graphical user interface details.

Subjects in [27-30] treat mainly of web-based interfaces that have become very trendy in NM. We will not detail much in this section because it would be rather irrelevant. In summary, these documents mostly outline the benefits of web-based NM, and propose different architectures to implement web-based management. However, it is important to mention that they are all concerned with providing an end-user interface that

is suitable for "manual" management. In other terms, these web-based interfaces were not meant to be used by any sort of control automation engine. This is clearly a limitation in the numerous scenarios where automated control is desired.

2.4 Other related work

2.4.1 A pattern system for network management interfaces

Bochmann *et al.* [22] introduced an interesting pattern system for the builders of network management interfaces (NMI). They also developed a framework called "Layla" that supports that system of patterns. Some of the patterns introduced in their paper are the Manager-Agent pattern and the Managed Object pattern. In the Manager-Agent pattern, the Agent (that can also be a manager for other subagents) is usually the one responsible of direct element management, this management being achieved through Managed Objects. The set of Managed Objects contained within a given Agent constitutes the Management Information Base (MIB) of that Agent. One of the major advantages of Agents is that they provide a uniform interface to the Manager so that specific resource details are hidden.

Within the scope of their framework, our development framework would serve as a base to implement NE control operations in Managed Objects. Managed Objects would reflect a set of uniform (vendor-independent) interfaces to Agents. The details hidden by those interfaces are handled automatically by an application implemented within our

framework using CLI interaction. Note that the fact that they use CMIP as a higher-level management protocol is not a problem here. The interface used for network management is orthogonal to our work, although its choice may affect overall runtime performance. Note, finally, that a control portion implemented with our system can coexist with another implementation (say using SNMP) responsible of other management tasks, the whole functionality being hidden behind uniform Managed Object interfaces.

2.4.2 The "Expect" Toolkit

"Expect is a tool for automating interactive applications such as telnet, ftp, passwd, fsck, rlogin, tip, etc." [14]. "Expect" is a popular tool that provides a scripting language for automating terminal-based interactive programs. It basically spawns a process running the program to automate, sends textual commands to it and reads its textual output. That output is "analyzed" and appropriate actions are undertaken if the response is "expected". In fact, it follows the scenario below:

a) Send a textual command to the terminal,

b) Expect a certain number of replies, wait until an "expected" response is received, or timeout.

c) According to the result of the previous action, execute a certain action just like in a) and continue through b), etc.

In step b), the expected responses from the terminal are represented by *regular expressions* [10, 12] to specify text patterns. According to the regular expression matched by the received response, a certain value is returned and the script author decides on the

next action according to that result. A timeout value can also be specified in order to exit the waiting loop if no pattern is matched after a certain time. Timeouts are useful to handle error situations for example.

Expect is powerful and used by many other popular utilities, such as DejaGnu [15]. Unfortunately, it suffers many drawbacks:

- Yet another scripting language to learn.
- Embedded statements become very complex to read and understand. Embedded statements occur when complex interaction –involving loops for example– is to be described in Expect. Many undesirable "goto-like" statements become necessary. The verification of the script becomes tedious and moves the writer away from the main objective: to ensure correct interaction that might already be complicated enough.
- Some advanced –however indispensable– features are not supported by Expect. For example, one cannot keep checking against a certain pattern for a whole session (or generally for more than one send-receive interaction) unless the pattern is re-submitted every time. This can reveal problematic in some cases discussed in Section 5.2. Our approach efficiently addresses this issue, as explained in Section 5.3.
- Expect is limited to regular expressions, which can only parse regular languages [11]. Languages like XML are not regular languages. If the output is XML, it

cannot be parsed with Expect. Our implementation supports extensibility to parse any desired NE output format.

Drawing a state diagram on a paper is natural and much easier and clearer than writing a script. By using graphical design with UML state diagrams, one can see clearly the logic followed by the state machine. This is why we argue that a UML-based approach is more suitable than a scripting language –at least for that reason.

"Expect" was not designed for uniform control of heterogeneous network elements, and reveals insufficient to handle complex and reliable CLI automation with NEs. However, the idea of using regular expressions to trigger subsequent actions is a remarkable starting point. We inspire from this idea and develop a more complete model suitable for uniform NE control using CLI. The presented framework makes extensive use of real-time and object-oriented concepts that greatly facilitate implementation of new automated CLI operations in a graphical environment. [Note: "Expect" is not used in the actual implementation of the presented framework].

We propose a distributed and centralized (from the network point of view the two are not antagonist) architecture which can supervise the network and control according to some control rules the network elements. Our thesis deals with and presents a solution to what in [44] is called a "very complex task due to the many different network elements and to the many parameters required to successfully deploy and implement a QoS policy end to end". We do not define specific services or policies (such as QoS end-to-end). Instead, we define a development framework that would allow rapid and efficient design and implementation of uniform network element control. This is what we call the

horizontal deployment of the network control environment be it for providing QoS or any other service in the network. Our framework provides means for developing element control applications that provide uniform control interfaces. The obtained performance is superior to what has been achieved so far. Flexibility in element control is improved as well because no assumption is made regarding the nature of the uniform interface provided to the upper layer. It can be customized on a per-application basis according to the targeted NE categories and the service to be provided. Enterprise MIBs from multi-vendor NEs can thus be controlled provided that they offer a common desired *functionality* even if the *interface* to that functionality is not standardized by an existing NM protocol.

3 Background:

SOAP, WSDL, Regular expressions and UML

This chapter presents an overview of the Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), mathematical results in the field of regular expression matching, and the Unified Modeling Language (UML). Because UML components and state diagrams are both the application environment and the foundation of many concepts proposed in this thesis, UML is reviewed in reasonable detail and emphasis is laid upon UML's real-time aspects. SOAP and WSDL are reviewed to justify their choice as the middleware for the web-based architecture of the framework. Mathematical results related to regular expressions and finite automata are reviewed as well to demonstrate their efficiency since they are used in the implementation of the final prototype.

This chapter can be skipped if the reader is already familiar with SOAP, WSDL, regular expressions and finite automata theory, and UML real-time concepts and semantics.

3.1 The Simple Object Access Protocol

(*NB*: This section is inspired in part from [4] and [5]. For more about SOAP, please refer to [3-5]).

3.1.1 Introduction

For the most part, software component developers from opposite sides of the tracks stay as far away from each other's technology as possible. This polarity makes it difficult to achieve any level of interoperability. It would be great to find a component technology standard that everyone could agree on. It could be that a subset of minimal technologies is the best answer to this quandary.

XML and HTTP are two such minimal technologies. The Simple Object Access Protocol (SOAP) defines the use of XML and HTTP to access services, objects, and servers in a platform-independent manner. SOAP is a protocol that acts as the glue between heterogeneous software components. If developers can agree on HTTP and XML, SOAP offers a mechanism for bridging competing technologies in a standard way.

The industry has accepted HTTP. It's used everywhere, on all platforms. XML is becoming as ubiquitous as HTTP. This ubiquity makes HTTP a good choice for an interoperable transport mechanism.

XML is a simple and extensible text markup language. Because XML is just text, any application can understand it as long as the application understands the character encoding in use.

Combining HTTP and XML into a single solution provides a whole new level of interoperability. For example, lathered with SOAP, clients written in Microsoft Visual Basic can easily invoke CORBA services running on UNIX boxes, JavaScript clients can easily invoke code running on the mainframe, and Macintosh clients can start invoking Perl objects running on Linux. The list goes on. While some interoperability is achieved today through cross-platform bridges for specific technologies, once SOAP becomes standard, bridges will no longer be necessary.

SOAP is not an entirely new concept; it attempts to codify the main concepts behind existing practices into a simple and generic protocol that can serve as an industry standard.

SOAP can be used in combination with a variety of existing Internet protocols and formats including HTTP, SMTP, and MIME. SOAP does not itself define any application semantics such as a programming model or implementation specific semantics; rather it defines a simple mechanism for expressing application semantics by providing a modular packaging model and encoding mechanisms for encoding data within modules, which allows SOAP to be used in a large variety of systems ranging from messaging systems to RPC.

Also, SOAP doesn't attempt to address the more complicated distributed object protocol services such as object activation, marshaling objects/references, garbage collection, or bi-directional communications. SOAP doesn't prohibit the use of any of these services; they are simply implementation details that can be layered on top of the SOAP protocol.

Finally, although SOAP can be made easier to use through natural language bindings, SOAP does not mandate an API of any kind; a language binding is strictly an implementation that makes SOAP more accessible and easy to use.

[*Note*: Although SOAP can be used in combination with virtually any transport protocol, it was originally designed to be layered on top of HTTP. Actually, the W3C SOAP specification [1] mandates specific SOAP to HTTP bindings to stress the importance of using SOAP over HTTP. This point is so crucial that SOAP is sometimes – abusively– considered to be limited to HTTP. The specification also defines a representation of RPC calls within SOAP. Note that using SOAP for RPC is orthogonal to the SOAP protocol binding (i.e. orthogonal to the transport protocol layered underneath SOAP).]

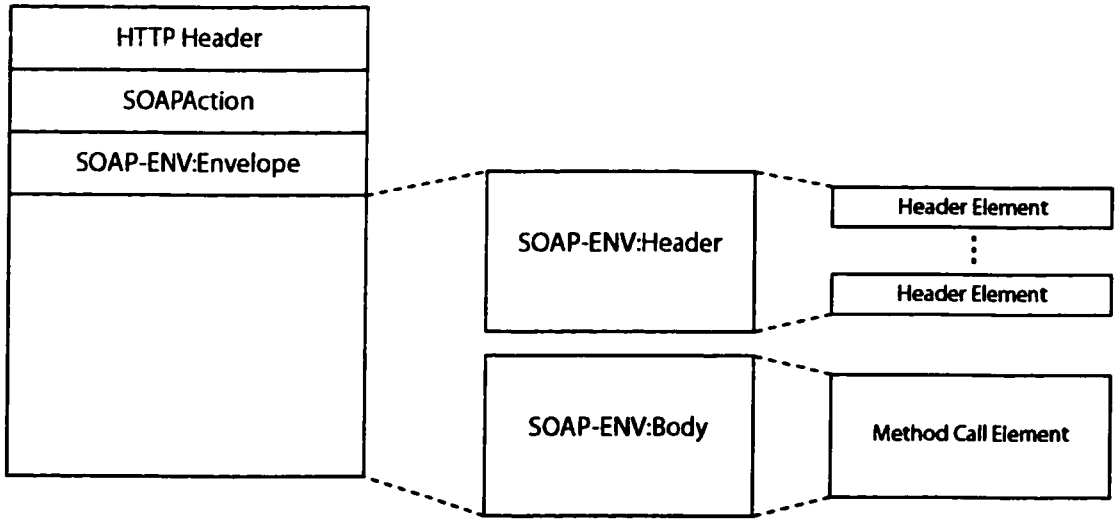

Figure 3-1: simplified SOAP-RPC packet layout (adapted from [5]).

Below is an example of SOAP/HTTP request used for RPC. Note how intuitive and simple is the syntax used to describe the request.

```
POST /bookSeat HTTP/1.1
Host: localhost:8080
Content-type: text/xml; charset=UTF-8
Content-length: XXX

SOAPAction:"/bookSeat"

<SOAP-ENV:Envelope
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

  <SOAP-ENV:Header>
     <Source Value="MIRLab"/>
  </SOAP-ENV:Header>

  <SOAP-ENV:Body>
    <ns1:bookSeat xmlns:ns1="http://localhost:8080/bookSeat">
      <seat xsi:type="xsd:int">
        32
      </seat>
    </ns1:bookSeat>
  </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

The "Body" part (called `SOAP-ENV:Body`) contains the method call element. In the example above, the method invoked is `bookSeat` with an argument called `seat` (of type `xsd:int`, that is, an integer).

3.1.2 Firewall issues

Currently, developers struggle to make their distributed applications work across the Internet when firewalls get in the way. Since most firewalls block all but a few ports,

such as the standard HTTP port 80, all of today's distributed object protocols like DCOM and CORBA suffer because they rely on dynamically assigned ports for remote method invocations. If the network administrator can accept to open a range of ports through the firewall, then it may be possible to get around this problem as long as the ports used by the distributed object protocol are included.

To make matters worse, clients of the distributed application that lie behind another corporate firewall suffer the same problems. If they don't configure their firewall to open the same ports, they won't be able to use the application. Making clients reconfigure their firewalls to accommodate to an application is just not practical.

When SOAP is used in combination with HTTP as the transport mechanism, and since most firewalls allow HTTP to pass through, there will be no problem invoking SOAP endpoints from either side of a firewall.

3.1.3 SOAP and Security

SOAP being a wire protocol, it does not implement security. However, SOAP can use the HTTP protocol, allowing the use of application-level security coupled with secure sockets or HTTPS. SOAP also mandates the use of the SOAPAction HTTP header field (see Figure 3-1 and the SOAP request example), which allows firewalls (or equivalent technology) to filter SOAP method invocations or deny SOAP processing entirely. The firewall would examine the SOAPAction header and filter the SOAP packet based upon the object name, the particular method (remotable or not), or a combination of the two [5].

Besides the firewall security benefits of designing SOAP using extended HTTP headers, the SOAP specification does not define any protocol-specific security features. SOAP may simply utilize any security feature offered by the transport protocol or any endpoint application-specific security feature.

3.2 Web Services Description Language

(*NB*: Sections 3.2.1 and 3.2.2 are mostly extracted from [7]).

3.2.1 Introduction: Web Services Defined

A Web Service is programmable application logic accessible using standard Internet protocols. Web Services combine the best aspects of component-based development and the Web. Like components, Web Services represent black-box functionality that can be reused without worrying about how the service is implemented. Unlike current component technologies, Web Services are not accessed via object-model-specific protocols, such as the Distributed Component Object Model (DCOM), Remote Method Invocation (RMI), or Internet Inter-ORB Protocol (IIOP). Instead, Web Services are accessed via ubiquitous Web protocols and data formats, such as Hypertext Transfer Protocol (HTTP) and Extensible Markup Language (XML). Furthermore, a Web Service interface is defined strictly in terms of the messages the Web Service accepts and generates. Consumers of the Web Service can be implemented on any platform in any

programming language, as long as they can create and consume the messages defined for the Web Service interface.

3.2.2 Web Services related protocols

There are a few key specifications and technologies that are likely to be encountered when building or consuming Web Services. These specifications and technologies address five requirements for service-based development:

- A standard way to represent data
- A common, extensible, message format
- A common, extensible, service description language
- A way to discover services located on a particular Web site
- A way to discover service providers

XML is the obvious choice for a standard way to represent data. Most Web Service-related specifications use XML for data representation, as well as XML Schemas to describe data types.

SOAP (introduced in Section 3.1 above) defines a lightweight protocol for information exchange. Part of the SOAP specification defines a set of rules for how to use XML to represent data. Other parts of the SOAP specification define an extensible message format, conventions for representing remote procedure calls (RPCs) using the SOAP message format, and bindings to the HTTP protocol.

Given a Web Service, it would be useful to have a standard way to document what messages the Web Service accepts and generates—that is, to document the Web Service contract. A standard mechanism makes it easier for developers and developer tools to create and interpret contracts. The Web Services Description Language (WSDL) is an XML-based contract language jointly developed by Microsoft and IBM. It will be further discussed in Section 3.2.3 below.

Developers will also need some way to discover Web Services. The Discovery Protocol (Disco) specification defines a discovery document format (based on XML) and a protocol for retrieving the discovery document, enabling developers to discover services at a known URL.

However, in many cases the developer will not know the URLs where services can be found. Universal Description, Discovery, and Integration (UDDI) specifies a mechanism for Web Service providers to advertise the existence of their Web Services and for Web Service consumers to locate Web Services of interest.

In the context of the present work, service discovery and service provider discovery was not used nor was it necessary anyway. Indeed, the services provided by the application were not meant to be advertised for everyone but rather for users who have previous knowledge of the existence of the service. Therefore, we will only focus on WSDL in the rest of this section. Disco and UDDI were referred-to above for completeness only.

3.2.3 WSDL

Over the past year Microsoft and IBM have proposed several contract languages: Service Description Language (SDL), SOAP Contract Language (SCL), and Network Accessible Services Specification Language (NASSL).

Web Services Description Language (WSDL) is a more mature evolution of all these. It consolidates the efforts and ideas present in those previous attempts; for example, SCL was designed specifically for SOAP whereas WSDL has been designed such that it can express bindings to protocols other than SOAP. The purpose of this section is to introduce WSDL and explain how it contributes in facilitating the use of SOAP.

As outlined previously, a language that describes the capabilities of Web Services (SOAP web services, in particular) –by providing a description of the messages a Web service is able to send and receive– is needed.

It has been argued that SOAP does not really need an interface description language to go with it. If SOAP is a standard for communicating pure content, then it needs a language for describing that content. SOAP messages do carry type information, and so SOAP allows for dynamic determination of type. But a function cannot be called correctly unless its name, the number of parameters and the types of each are known. Without WSDL, one can determine the calling syntax from documentation that must be provided, or by examining wire messages. Either way, a human will have to be involved, and so the process is prone to error.

Other than describing web services to clients, with WSDL, one can *automate* the generation of proxies for Web services in a truly language- and platform-independent way. Like the IDL file for COM and CORBA, a WSDL file is a contract between client and server.

In fact, for every SOAP-RPC call, the call data is marshalled into a SOAP message, sent over the Internet (using HTTP for example), de-marshalled on the server side and finally dispatched to the right entity (the response follows the same logic). In practice, however, this task is not done explicitly by the programmer for each SOAP-RPC call, since it would clearly affect the readability of the code and the scheme is prone to error. Instead, this task is usually handled by a set of *proxy classes* –called *client stubs* and *server skeletons*– that are automatically generated by a WSDL compiler in some programming language such as Java™ or C++. These classes implement the marshalling and de-marshalling functions, which are transparently called every time an RPC call is made.

By doing this, writing the client/server code becomes much easier since a SOAP-RPC call will have exactly the same syntax as a local method invocation. Note also that client stubs and server skeletons may be written using different programming languages. Actually, since SOAP is XML based, it offers a high level of interoperability between heterogeneous software components. SOAP endpoints do not need to assume anything about the programming language other SOAP endpoints are written in. The only binding

between the different endpoints consists in the abstract WSDL interface common to them, reducing coupling to its minimum.

Again, the above scheme is not a new idea specific to WSDL or SOAP; in fact, it has been used in software development for a long time. For example, the CORBA Interface Description Language (IDL) plays the same role as WSDL. Most CORBA implementations contain an *IDL compiler* (code generator) that maps the IDL interface into a natural programming language. Stub and skeleton classes are created by the compiler, and the programmer only has to use them to invoke distributed services transparently.

Note, finally, that the SOAP layer does *not* have to be reflected under the form of a client stub and a server skeleton. It is just one common way, in software development, to implement RPC to SOAP mapping, and more generally services provided by a middleware such as CORBA or DCOM. The SOAP layer can be implemented by any other means provided that the consistency of the SOAP-RPC messages is preserved.

We will intentionally neglect, within this section, other WSDL-related details such as document structure and language bindings since it would make the section unnecessarily heavy. For more about WSDL, please refer to [6-9].

3.3 Regular Expressions and Kleene's Theorem

The purpose of this section is to review mathematical results related to the use of finite automata in regular expression matching, given that the final implementation of our framework relies in part on it.

Classical mathematical definitions about regular languages, regular expressions and finite automata are first reviewed. Next, Kleene's theorem –stating equivalence in a very precise sense between regular expressions and finite automata– is reviewed. Proofs of the reviewed theorems are omitted in order to keep this section as light as possible, but these proofs can easily be found in literature [10-13].

3.3.1 Regular languages

A *language* is a set of strings of symbols. *Formal languages* are characterized by grammars which are essentially a set of rewrite rules for generating strings belonging to a language. *Regular languages* are just a specific kind of formal languages.

3.3.1.1 Basic definitions

An **alphabet** is a finite set of symbols. {0, 1} is an alphabet and {a, b} is another. The set of ASCII characters is also an alphabet. A **word** is a finite sequence of symbols of a given alphabet.

A **language** is a set of words (possibly infinite) over a given alphabet. {a, ab, abba} is a language (over the alphabet {a, b}). The English language is a language in the sense

of the above definition over the alphabet {a, b, ..., z, A, B, ..., Z}. The number of symbols in a string is called the **length of the string**. For a string w its length is represented by |w|. The **empty string** (also called null string) is the string with length 0, that is, it has no symbols. The empty string is denoted by Λ (capital lambda). Thus $|\Lambda| = 0$.

The empty set $\varnothing$ is a language which has no strings. The set $\{\Lambda\}$ is a language which has one string, namely Λ. For any alphabet Σ, the set of all strings over Σ (including the empty string) is denoted by Σ^*. Thus a language over alphabet Σ is a subset of Σ^*.

3.3.1.2 Operations on languages

Since languages are sets, all the set operations can be applied to languages. Thus the union, intersection and difference of two languages over an alphabet Σ are languages over Σ. The complement of a language L over an alphabet Σ is Σ^*- L and it is also a language.

Another operation on languages is **concatenation**. Let L_1 and L_2 be languages. Then the concatenation of L_1 with L_2 is denoted as L_1L_2 and it is defined as $L_1L_2 = \{uv \mid u \in L_1$ and $v \in L_2\}$. That is L_1L_2 is the set of strings obtained by concatenating strings of L_1 with those of L_2. For example {ab, b} {aaa, abb} = {abaaa, ababb, baaa, babb}.

Recursive definition of L^*:

Basis Clause: $\Lambda \in L^*$.

Inductive Clause: For any $x \in L^*$ and any $w \in L$, $xw \in L^*$.

Extremal Clause: Nothing is in L^* unless it is obtained from the above two clauses.

Theorem: L^* is the set of strings obtained by **concatenating zero or more** strings of L. This * is called **Kleene star**. (proof is omitted).

For example if $L = \{aba, bb\}$, then $L^* = \{\Lambda, aba, bb, ababb, abaaba, bbbb, bbaba, ...\}$.

The * in Σ^* is also the same Kleene star defined above.

Definition of L^+: $L^+ = LL^* = L^*L$.

Thus L^+ is the set of strings obtained by **concatenating one or more** strings of L.

For example if $L = \{aba, bb\}$, then $L^+ = \{aba, bb, ababb, abaaba, bbbb, bbaba, ...\}$

Definition of Set of Regular Languages:

Basis Clause: $\varnothing$, $\{\Lambda\}$ and $\{\sigma\}$ for any symbol $\sigma \in \Sigma$ are regular languages.

Inductive Clause: If L_r and L_s are regular languages, then $L_r \cup L_s$, $L_r L_s$ and L_r^* are regular languages.

Extremal Clause: Nothing is a regular language unless it is obtained from the above two clauses.

For example, let Σ= {a, b}. Then since {a} and {b} are regular languages, {a, b} (= {a}$\cup${b}) and {ab} (= {a}{b}) are regular languages. Also since {a} is regular, {a}* is a regular language which is the set of strings consisting of a's such as Λ, a, aa, aaa, aaaa etc. Note also that Σ^*, which is the set of strings consisting of a's and b's, is a regular language because {a, b} is regular.

3.3.2 Regular expressions

Regular expressions are used to denote regular languages. They can represent regular languages and operations on them succinctly.

The set of regular expressions over an alphabet Σ is defined recursively as below. Any element of that set is a **regular expression**.

Basis Clause: $\varnothing$, Λ and σ are regular expressions corresponding to languages $\varnothing$, $\{\Lambda\}$ and $\{\sigma\}$, respectively, where σ is an element of Σ.

Inductive Clause: If **r** and **s** are regular expressions corresponding to languages L_r and L_s, then (**r** + **s**) , (**rs**) and (**r***) are regular expressions corresponding to languages $L_r \cup L_s$, $L_r L_s$ and L_r^* , respectively.

Extremal Clause: Nothing is a regular expression unless it is obtained from the above two clauses.

Conventions on regular expressions

(1) When there is no danger of confusion, bold face may not be used for regular expressions. So for example, $(r + s)$ is used instead of $(\mathbf{r} + \mathbf{s})$.

(2) The operation * has precedence over concatenation, which has precedence over union (+). Thus the regular expression $(a + (b\,(c^*)))$ is written as $(a + bc^*)$,

(3) The concatenation of k **r**'s, where **r** is a regular expression, is written as $\mathbf{r}^k$. Thus for example $rr = r^2$.

(4) We use $(\mathbf{r}^+)$ as a regular expression to represent L_r^+.

Examples of regular expression and regular languages corresponding to them:

- $(a + b)^2$ corresponds to the language {aa, ab, ba, bb}, that is the set of strings of length 2 over the alphabet {a, b}. In general $(a + b)^k$ corresponds to the set of strings of length k over the alphabet {a, b}. $(a + b)^*$ corresponds to the set of all strings over the alphabet {a, b}.

- a^*b^* corresponds to the set of strings consisting of zero or more a's followed by zero or more b's.

- $a^*b^+a^*$ corresponds to the set of strings consisting of zero or more a's followed by one or more b's followed by zero or more a's.

- $(ab)^+$ corresponds to the language {ab, abab, ababab, ...}, that is, the set of strings of repeated ab's.

- With the notation @ = (a+b+...+z+A+...+Z+' ') (i.e. a letter from the English alphabet or a white space), $(@^*\text{hello}@^*)$ denotes the set of all phrases over the English alphabet containing the word 'hello'.

Note: A regular expression is not unique for a language. That is, a regular language, in general, corresponds to more than one regular expression. For example $(a + b)^*$ and $(a^*b^*)^*$ correspond to the set of all strings over the alphabet {a, b}.

Regular expressions are **equal** if and only if they correspond to the same language. Thus for example $(a + b)^* = (a^*b^*)^*$, because they both represent the language of all strings over the alphabet {a, b}.

Note: In general, it is not easy to see by inspection whether or not two regular expressions are equal.

3.3.3 Finite automata

Definition: A model of computation consisting of a *finite* set of states, a start state, an input alphabet, and a transition function which maps input symbols and current states to a next state. Computation begins in the start state with an input string. It changes to new states depending on the transition function. There are many variants, for instance, machines having actions (outputs) associated transitions (Mealy machine) or states (Moore machine), multiple start states, transitions conditioned on no input symbol (a null) or more than one transition for a given symbol and state (non-deterministic finite state machine), one or more states designated as accepting states (recognizer), etc. [12].

3.3.3.1 Deterministic finite automata

Definition: Let Q be a finite set and let Σ be a finite set of symbols. Also let δ be a function from $Q \times \Sigma$ to Q, let q_0 be a state in Q and let A be a subset of Q. We call the elements of Q **states**, δ the **transition function**, q_0 the **initial state** and A the set of **accepting states**. Then a deterministic finite automaton (**DFA**) is a 5-tuple $<Q. \Sigma, q_0, \delta, A>$.

Notes on the definition:

1. The set Q in the above definition is simply a set with a finite number of elements. Its elements can, however, be interpreted as a state that the system (automaton) is in. Thus in the example of a vending machine, for example, the states of the machine such as "waiting for a customer to put a coin in", "have received 5 cents" etc. are the elements of Q. "Waiting for a customer to put a coin in" can be considered the initial state of this automaton and the state in which the machine gives out a soda can be considered the accepting state.

2. The transition function is also called a **next state function** meaning that the automaton moves into the state δ (q, a) if it receives the input symbol a while in state q. Thus in the example of vending machine, if q is the initial state and a nickel is put in, then δ (q, a) is equal to "have received 5 cents".

3. The accepting states are used to distinguish sequences of inputs given to the finite automaton. If the finite automaton is in an accepting state when the input ceases

to come, the sequence of input symbols given to the finite automaton is "accepted". Otherwise it is not accepted. For example, in the example of Figure 3-2, the strings aa and aabb is accepted by the finite automaton. But any other strings such as ab, abb, etc. are not accepted.

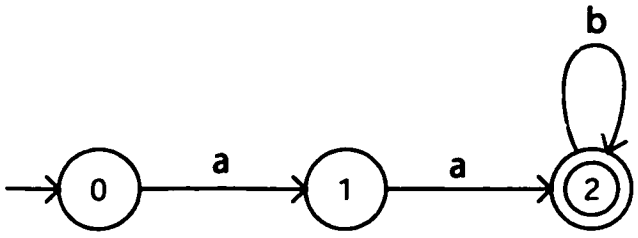

Figure 3-2: an example of a deterministic finite automaton (DFA).

$Q = \{0, 1, 2\}$, $\Sigma = \{a, b\}$, $A = \{2\}$, $q_0 = 0$, δ shown in Table 3-1 below.

State (q)	Input (σ)	Next state (δ (q, σ))
0	a	1
1	a	2
2	b	2

Table 3-1: transition table of the DFA of Figure 3-2.

DFAs are often represented by digraphs called **(state) transition diagrams**. The vertices (denoted by single circles) of a transition diagram represent the states of the DFA and the arcs labeled with an input symbol correspond to the transitions. An arc (p, q) from vertex p to vertex q with label σ represents the transition δ (p, σ) = q. The accepting

states are indicated by double circles. Transition functions can also be represented by tables as seen above. They are called **transition table**.

Note that the behavior of the above DFA is not defined when the DFA is in state 1 and receives the input b, for example. This DFA is said to be **incomplete**. In this case, we consider that the DFA blocks when it receives an unexpected input and subsequent input symbols are not processed. When δ is defined for each couple $(q, \sigma) \in Q \times \Sigma$, the DFA is said to be **complete**.

Definition of δ^*:

Basis Clause: For any state q of Q, $\delta^*(q, \Lambda) = q$, where Λ denotes the empty string.

Inducitve Clause: For any state q of Q, any string $y \in \Sigma^*$ and any symbol $\sigma \in \Sigma^*$,

$$\delta^*(q, y\sigma) = \delta(\delta^*(q, y), \sigma).$$

In the definition, the Basis Clause says that a DFA stays in state q when it reads an empty string at state q and the Inductive Clause says that the state DFA reaches after reading string ya starting at state q is the state it reaches by reading symbol σ after reading string y from state q. This definition will be used below to formally define the notion of "acceptance" by a DFA.

A string w is accepted by a DFA $\langle Q, \Sigma, q_0, \delta, A \rangle$ if and only if $\delta^*(q_0, w) \in A$. That is, a string is accepted by a DFA if and only if the DFA starting at the initial state ends in an accepting state after reading the string.

A language L is accepted by a DFA $\langle Q, \Sigma, q_0, \delta, A \rangle$ if and only if $L = \{ w \mid \delta^*(q_0, w) \in A \}$. That is, the language accepted by a DFA is the set of strings accepted by the DFA.

For example, it can be easily observed that the DFA of Figure 3-2 accepts the language characterized by the regular expression aab^*.

3.3.3.2 Non-deterministic finite automata

Definition: Let Q be a finite set and let Σ be a finite set of symbols. Also let δ be a function from $Q \times \Sigma$ to $P(Q)$[1], let q_0 be a state in Q and let A be a subset of Q. We call the elements of Q **states**, δ the **transition function**, q_0 the **initial state** and A the set of **accepting states**. Then a non-deterministic finite automaton (**NFA**) is a 5-tuple $\langle Q, \Sigma, q_0, \delta, A \rangle$.

Notes on the definition

1. As in the case of DFA the set Q in the above definition is simply a set with a finite number of elements. Its elements can be interpreted as a state that the system (automaton) is in.
2. The transition function δ is also called a **next state function**. Unlike DFAs an NFA moves into **one of the states** given by $\delta(q, \sigma)$ if it receives the input symbol

---

[1] P(X) denotes the set of subsets of X. It is also called the *power set* of X.

σ while in state q. Which one of the states in δ (q, σ) to select is chosen **non-deterministically**.

3. As in the case of DFA the accepting states are used to distinguish sequences of inputs given to the finite automaton. If the finite automaton is in an accepting state when the input ends, i.e. ceases to come, the sequence of input symbols given to the finite automaton is "accepted". Otherwise it is not accepted.

4. Note that any DFA is also a NFA.

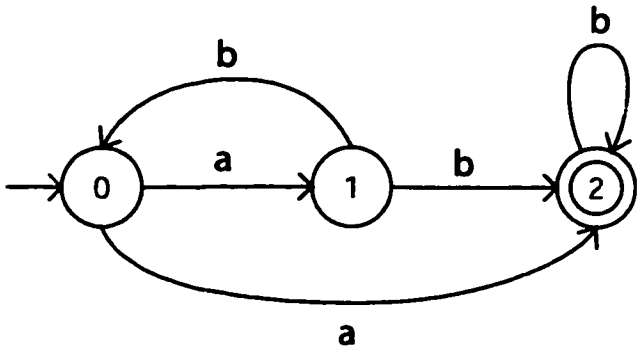

Figure 3-3: an example of an NFA.

$Q = \{0, 1, 2\}$, $\Sigma = \{a, b\}$, $A = \{2\}$, $q_0 = 0$, δ shown in Table 3-2 below.

State (q)	Input (σ)	Next state (δ(q, σ))
0	a	{1, 2}
0	b	∅
1	a	∅
1	b	{0, 2}

2	a	∅
2	b	{2}

Table 3-2: transition table for the NFA of Figure 3-3.

As in the case of DFAs, the definition of language acceptance for NFA relies on the definition of δ^* which is slightly different for NFAs.

For a state q and a string w, $\delta^*(q, w)$ is the set of states that the NFA can reach when it reads the string w starting at the state q. In general an NFA non-deterministically goes through a number of states from the state q as it reads the symbols in the string w. Thus for an NFA $<Q, \Sigma, q_0, \delta, A>$, the function $\delta^* : Q \times \Sigma \rightarrow P(Q)$ is defined recursively as follows:

Definition of δ^*:

Basis Clause: For any state q of Q, $\delta^*(q, \Lambda) = \{q\}$, where Λ denotes the empty string.

Inductive Clause: For any state q of Q, any string $y \in \Sigma^*$ and any symbol $\sigma \in \Sigma$,

$$\delta^*(q, y\sigma) = \bigcup_{p \in \delta^*(q,y)} \delta(p, \sigma)$$

In the definition, the Basis Clause says that an NFA stays in state q when it reads an empty string at state q and the Inductive Clause says that the set of states a NFA *can* reach after reading string $y\sigma$ starting at state q is the set of states it can reach by reading symbol σ after reading string y starting at state q.

We say that a string $x \in \Sigma^*$ is **accepted by an NFA** $<Q, \Sigma, q_0, \delta, A>$ if and only if $\delta^*(q_0, x) \cap A$ is not empty, that is, if and only if it can reach an accepting state by reading x starting at the initial state. The **language accepted by an NFA** $<Q, \Sigma, q_0, \delta, A>$ is the set of strings that are accepted by the NFA.

Example:

Considering the NFA of Figure 3-3, we are going to "run" it with the input string x = aba, that is, calculate $\delta^*(0, x)$.

$\delta^*(0, aba)$ is the union of $\delta(q, a)$ for $q \in \delta^*(0, ab)$ according to the Inductive Clause.

Now $\delta^*(0, ab)$ is the union of $\delta(q, b)$ for $q \in \delta^*(0, a)$ according to the same clause.

Again $\delta^*(0, a)$ is the union of $\delta(q, a)$ for $q \in \delta^*(0, \Lambda)$ according to the same clause.

However $\delta^*(0, \Lambda) = \{0\}$ according to the Basis Clause.

Hence $\delta^*(0, a) = \delta(0, a) = \{1, 2\}$ (see Table 3-2).

Hence $\delta^*(0, ab) = \delta(1, b) \cup \delta(2, b) = \{0, 2\}$.

Hence $\delta^*(0, x) = \delta^*(0, aba) = \delta(0, a) \cup \delta(2, a) = \{1, 2\}$.

The conclusion of this calculus is that the states that can be reached by the NFA when inputting x are 1 and 2. Since $\delta^*(0, x) \cap A = \{2\} \neq \varnothing$, x is accepted by the NFA.

3.3.3.3 Non-deterministic finite automata with Λ-transitions

These FAs are noted NFA-Λ. They are similar to NFAs except that some transitions may have Λ as an associated symbol. Λ-transitions are taken without affecting the symbol input stream, that is, the reading head doesn't move when a Λ-transition is taken.

We will not discuss NFA-Λ in further detail since it would make this section unnecessarily heavy. They have been mentioned here for completeness only. Note, nonetheless, that every NFA is an NFA-Λ.

3.3.3.4 Equivalence between DFA, NFA and NFA-Λ

Theorem: Let L be a language. The following propositions are equivalent:

1. There exists a DFA that accepts L.
2. There exists a NFA that accepts L.
3. There exists a NFA-Λ that accepts L.

In other words, the different sorts of automata introduced previously have the same computation power. Using one or another is thus just a matter of convenience to the situation where they are used. There are also well-known algorithms to convert between any two kinds of FAs (omitted here).

3.3.4 Kleene's theorem

Theorem (part 1 of Kleene's theorem):

Any regular language is accepted by a finite automaton.

Outline of the proof: We are not going to give a detailed proof of the above theorem. Nonetheless, the outline of the proof is worth to be mentioned. Note that the theorem doesn't specify a particular kind of FAs because of the equivalence theorem stated in the

previous section. The proof, however, uses NFA-Λ because of their convenience and flexibility. In fact, the proof follows a recursive logic just like the recursive definition of regular expressions. Indeed, the theorem is easy to prove for the base cases, i.e. when the regular expression in question is either $\varnothing$, Λ, or a symbol $\sigma \in \Sigma$ (basis cases). Next, an inductive construction of the FA is made according to the first-level decomposition of the regular expression r, so whether it is of the form r_1r_2, r_1+r_2, or r_1^*, the built FA is the concatenation of FA_1 (r_1) and FA_2 (r_2), the union of FA_1 and FA_2, or the Kleene star of FA_1, respectively.

Part 2 of Kleene's theorem is the converse of part 1, that is, languages accepted by FAs are regular. We did not emphasize on it because we only use the transformation property from a regular expression to a finite automaton, not its converse.

The advantage of using the above result is that it is very suitable for stream-based parsing, which is what we will need later. Actually, an FA "consumes" the symbols (characters in the case of a Telnet/SSH client reading NE textual response) one by one and never goes back in the stream, and when an accepting state is reached, the string obtained by juxtaposing all past characters consumed to reach that state is a word of the language. Therefore, Kleene's theorem provides a way of matching regular expressions in a *linear time* (i.e. proportional to the number of read symbols if we assume that the processing times of different symbols are approximately equal). This method is thus the optimal in terms of computing complexity. Most programs supporting regular expressions use this result to implement regular expression matching; the famous Unix 'grep' utility is a perfect example.

3.4 UML for Real-time Component-based Design

The Unified Modeling Language (UML) [17], developed under the coordination of the Object Management Group (OMG), is one of the most important standards for the specification and design of object oriented systems. This standard is currently tuned for real time applications in the form of a new proposal, UML for Real-Time (UML-RT). The visual notation of UML-RT is not only intuitive but it also has a deep mathematical foundation [48]. UML-RT is the result of merging the Real-time Object-Oriented Modeling (ROOM) [49] and UML.

UML is a graphical language for visualizing, specifying, constructing, documenting, and executing software systems. Although conventional programming languages are good for expressing different algorithms, they cannot directly show the high-level features of a system. The UML is therefore a language for expressing high-level system properties that are best modeled graphically.

UML provides a base visual modeling language; however, it is not possible for the language to be sufficient for all domains. For this reason the UML has been designed open-ended to make it possible for the language to be extended. Without bloating the base language, new building blocks can be derived from the base to create ones that are specific to a domain (see Figure 3-4).

This section does not attempt to present the UML in its entirety. Rather, its goal to is briefly present the real-time notations to the UML accompanied by a brief overview of some key UML modeling concepts.

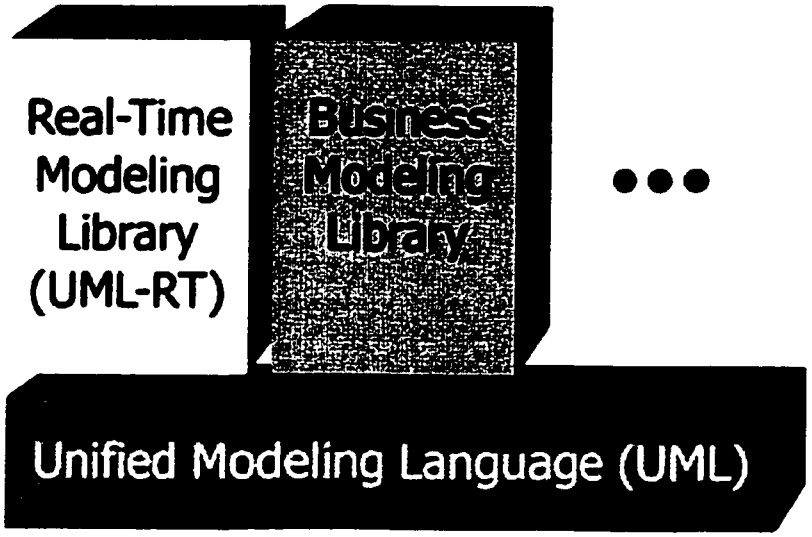

Figure 3-4: Layered UML architecture.

UML-RT has three main building blocks. They include:

Elements: which are the basic object-oriented building blocks of the UML and real-time specializations. They are used to construct models.

Relationships: which are used to join elements together in models.

Diagrams: which help assemble related collections of elements together into a graphical depiction of all or part of a model.

3.4.1 Elements

Elements are the basic object-oriented building blocks of the UML and real-time notations and they are used to construct models. There are four kinds of elements: Structural, Behavioral, Grouping, and Annotational. We will only briefly present the structural and behavioral ones.

3.4.1.1 Structural elements

Desides the base UML elements found in the specification [17] such as Classes and Interfaces, etc. the real-time extension to UML introduces a fundamental modeling element to real-time systems, called *Capsule.*

A capsule represents independent flows of control in a system. Capsules provide a very light weight modeling element for breaking a problem down into multiple logical threads of control. Each capsule instance has its own logical thread of control, though it may share an actual processing thread (known as a "physical thread") with other instances (**roles**). Capsules have much of the same properties as classes; for example, they can have operations and attributes. Capsules may also participate in dependency, generalization, and association relationships. However, they also have several specialized properties that distinguish them from classes.

For example, what differentiates a capsule from a class is how one can formally specify the internal organization of its structure, as a network of collaborating capsule roles. This collaboration is a specialized UML collaboration called a *Capsule Collaboration* (also **capsule structure diagram**). Figure 3-5 shows an example of a capsule's structure. The "ControlSystem" capsule contains three other capsules connected together through ports and connections. These three capsules collaborate to provide an overall service to their containing capsule, itself accessible from the "outside" via one port, namely "operatorPort".

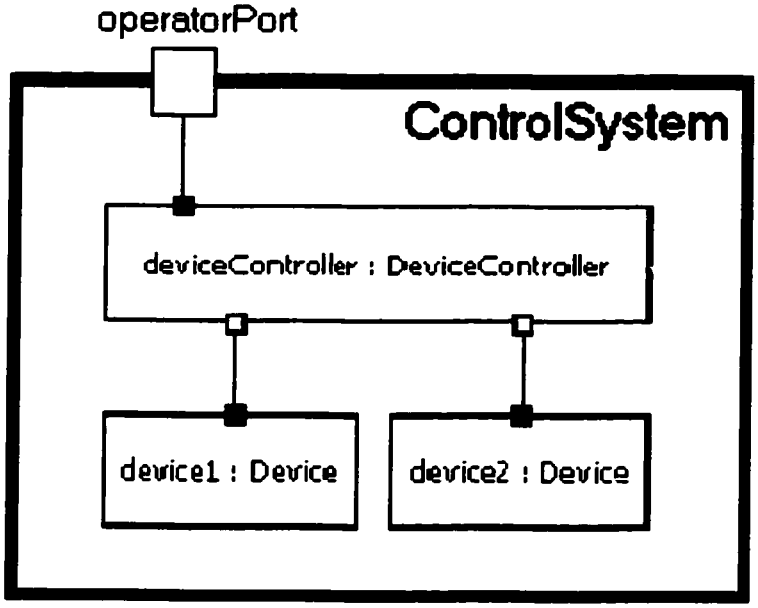

Figure 3-5: a capsule's structure.

Sending **messages** through **public ports** is the only method that capsules can use to communicate with other capsules. In fact, the *only* public attributes of a capsule are its public ports. While the behavior of a class is triggered by the invocation of a public operation on the class, a capsule's behavior is triggered by the receipt of a **signal** event over one of its end ports.

When a capsule receives a message from another capsule a signal event is generated and some response by the capsule is usually required. This typically involves performing some calculations, formulating a response, and sending one or more messages. The optional **state machine** associated with a capsule represents its behavior. It controls the operation of the capsule itself. The state machine is the only element that can access the protected (internal) parts of the capsule. Figure 3-6 shows an example.

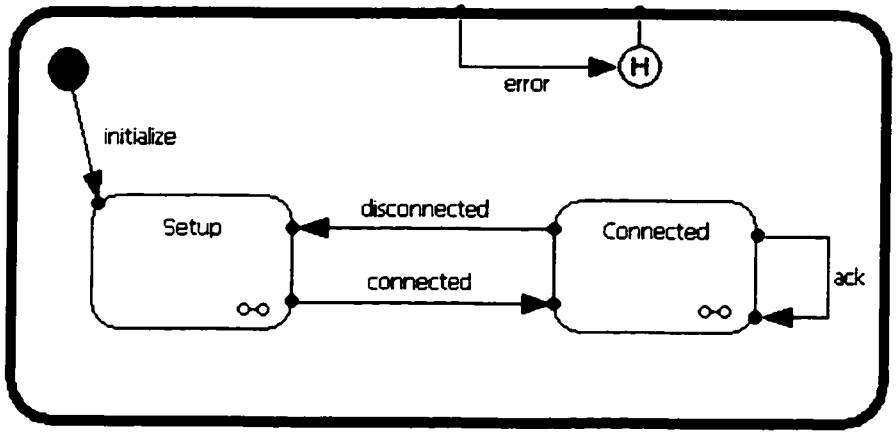

Figure 3-6: an example state machine.

3.4.1.2 Behavioral elements

Behavioral elements represent the dynamic parts of the model that describe the changing state of a system over time. In fact, once the structure of a capsule is defined, one needs to define its behavior (with **state machines**) and illustrate its functionality with execution scenarios (**sequence diagrams**). State machines (that follow Harel's statechart formalism [50]) are found in the base UML standard. UML-RT extends the set of behavioral elements with the concept of **Protocol**.

State machines contain hierarchical states (the state "Setup" in Figure 3-6 is a **composite state** –i.e. containing further internal states– and this is shown with an icon on the bottom-right corner of the state) and transitions that are triggered by events generated upon the arrival of a signal on one of the capsule's ports. To each transition in a state machine, an *action* may be attached. This action is the execution of a piece of code (C++

for example) written by the application developer. As such, there is an equivalence between programs written in a regular programming language and a state machines.

When an event is ready to be processed by a state machine, a search for a candidate transition takes place to determine which one will be taken. A transition is said to be enabled if its trigger is satisfied by the current event, meaning that the transition has the same event and interface specified as the current event.

The search order is defined by the following algorithm:

1. The search begins in the innermost current active state.
2. Within the scope of the innermost current active state, transitions are evaluated sequentially. If a transition is enabled, the search terminates and the corresponding transition is taken.
3. If no transition is enabled in the current scope, the search in step 2 is repeated for the next higher scope, one level up in the state hierarchy.
4. If the top-level state has been reached and no transitions are enabled, then the current event is discarded and the state of the behavior remains unchanged.

As for sequence diagrams, they show interaction scenarios between a set of capsules present in a collaboration.

Finally, the set of messages exchanged between two objects conforms to some communication pattern called a *Protocol*. It is basically a contractual agreement defining

the valid types of messages that can be exchanged between the participants in the protocol. Therefore a protocol comprises a set of participants, each of which plays a specific role in the protocol. *Capsule Ports* are Protocol instances (or protocol roles).

3.4.2 Relationships

Most often model elements must collaborate with other elements in a number of ways. Relationships allow representation of how elements stand in relation to others.

There are four main kinds of relationships in the base UML (association, realization, generalization, dependency). In addition to the base relationships defined in the base UML, UML-RT introduces a new kind called **Connectors**. UML-RT also defines the semantics of the former relationships for the new elements introduced by UML-RT itself (capsules for example). For example, UML-RT defines the semantics of the Generalization relationship for capsules, which basically show how structure *and* behavior can be inherited in a class hierarchy of capsules. A capsule can, indeed, inherit their behavior (represented by state machines) from a parent capsule and extend it and/or override it.

Connectors capture the key communication relationships between capsule roles. They interconnect capsule roles that have similar public interfaces (ports). A key feature of connectors is that they can only interconnect *compatible* ports. Connectors only exist in the context of a capsule collaboration (that is, a capsule's structure diagram). In Figure 3-5 for example, the solid line binding the roles "device1" and "deviceController" is a connector role.

3.4.3 Diagrams

Diagrams allow assembling related collections of elements together into a graphical depiction of all or part of a model. Each diagram provides a view into the elements that make up a model. In this way the user of the model can decide to see only the views of the underlying model that are of interest.

Some diagrams fully specify an element, such as the structure diagram of a capsule or its state machine. Others, such as interaction diagrams, only view particular use scenarios of the elements they involve. The only real-time specialization to UML's diagrams introduced by UML-RT is capsule's structure diagram (which is a special kind of collaboration diagrams).

4 Generic Model

In this chapter, general issues related to the definition of a uniform NE control interface are discussed. Next, design requirements for applications targeting uniform control of heterogeneous NEs are defined. Finally, the architecture of our framework is presented in two steps: The web-based architecture using SOAP and WSDL is presented, and a generic server-side model based on *Control Protocol Adapters* (CPA) is defined.

4.1 The problem of defining a common view of NEs

In order to achieve transparent control of heterogeneous network elements, two main steps are necessary:

- Defining a common abstract view of Network Elements, and
- Transparently mapping that view into NE-specific commands and protocols.

The question of how to define a common abstract view –or model– for all NEs is not a trivial question: NE classes[1] are quite different across vendors and technologies, in

[1] We will say that two NEs belong to the same *class* if and only if they have the *exact* same configuration interface relatively to a given control protocol.

terms of functionality as well as control interfaces. For this reason, defining a common abstract NE model, even for a specific category of NEs (QoS-enabled NEs, for example), may reveal very difficult. An attempt to define a global model will basically encounter the same obstacles as SNMP and CMIP which some have been mentioned previously. As explained earlier in Section 2.2, introducing a service-unaware intermediate abstraction layer (for example Caliban in the Tempest infrastructure case) necessarily limits the range of possible control operations, which we cannot tolerate here. Therefore, we reject the idea of having a generic fixed abstraction model for all NEs and we propose to use a generic control interface that is service-dependant. More precisely, this means that *within the scope of a given service targeting a specific range of NE classes (that support the required functionality), control operations are performed transparently from the controller point of view and NE-specific details are hidden.*

So again, it is not necessary to focus on having a *global* abstract model, that is, a model that covers the widest range of existing NE functionality. As argued in [34], "Traditionally, a bottom-up approach has been adopted with regard to Network Management, i.e. the network itself is the focus rather than the services that are provided, when in reality what is required is a top-down approach". In other words, the abstract model should be defined according to the needs of higher-level services that have been expressed *prior* to the model itself. A concrete example would consist in the case of a service requiring two QoS functional blocks, namely Bandwidth Management and Traffic Conditioning. The NE abstract model in this case would cover (at least) those function blocks and the implementation would achieve transparency among a given set of NE

classes. So, although this "scaled-down" abstract model does not cover all the capabilities of all NE classes, it provides all necessary control procedures required to deliver the higher-level service in a uniform, NE-independent fashion. Also, as future services are needed, the abstract model can be incrementally extended or new models can be added to provide for those needs.

As a matter of fact, we are not concerned with control *architectures* (that is, specific controller designs) but rather with how to efficiently and rapidly implement control procedures for a given control *interface*. The control interface in question, which we called *generic control interface*, is typically targeted for a specific control architecture itself providing for a specific service or a small range of specific services. Since control interfaces are per-service, they do not interfere with each other and a change in one of the control interfaces does absolutely not affect other services that use other control interfaces (via adequate intermediate control architectures). The price to pay for this approach is obviously that new control architectures require implementing new control interfaces, but that is exactly why we are concerned about *rapid and straightforward* implementation of control interfaces. The evaluation of the framework, presented in Section 6.4, shows that implementing new control procedures with our method is particularly fast.

Starting from the latter point, it will be assumed in the rest of this thesis that an abstract NE model is given, and we will not be concerned any more about how to actually define that model. The purpose of this document is to introduce a *flexible framework* for developing support to transparent NE control. The flexibility of this framework is such

that it does not make any assumption about the nature of the exposed interface or the functionality offered by the targeted NEs.

Note that it is not forbidden to expose an operation through the common interface that would provide the user with specific and concrete (i.e. non-generic or non-abstract) information about a certain NE. However, that information is *a priori* not known by the user and should not be needed except for pure "browsing" contexts and other similar tasks. We stress the fact that specific information about NEs sitting behind this generic interface should *never* be needed in order to make NE control decisions. Following this logic, all NEs will be represented in memory by object instances of the same class. That class will implement the common abstract interface to all those NEs.

Note also that it is possible to include support for NE classes that do *not* implement all the functionality exposed by the common interface. In fact, it is assumed that this common interface sits underneath a control layer that comprises a control algorithm designed to provide a specific service to an upper layer. The control layer makes control decisions and issues control commands through the common interface, in turn controlling the NEs transparently. The control logic consistency is thus checked by the control layer, and the common interface is only a means of achieving transparency in performing the actual control functions. Therefore, the control layer is not supposed to send inconsistent commands to the common control interface otherwise it would be synonym of a design weakness in the control layer itself. However, should the control layer request some functionality on a NE that does not support it, a meaningful error message must be returned.

4.2 High-level Design Requirements

This section discusses design requirements for a generic framework for uniform control of heterogeneous network elements. It is inspired in part from [4, 28, 33-35].

4.2.1 Client/Server web-based architecture

Remote objects can give a program almost unlimited power over the Internet. Client/Server architectures have become ubiquitous [27] in most applications. For this reason, one of the main design goals is to allow an internet-wide access to the control module. But since most firewalls block non-HTTP requests, a *web-based* (i.e. HTTP-based) architecture is necessary to get around this limitation.

The Simple Object Access Protocol (SOAP) is the solution chosen to implement the distributed web-based architecture of the system. Section 3.1 introduces SOAP and explains its adequacy –for addressing firewall issues in particular–, whereas section 4.3 explains its role within the global architecture.

4.2.2 Extensibility

As new control protocols and networking hardware appear regularly, the ability to easily add support for new technologies within the system is obviously highly desirable. For this reason, the application has to be easily extensible. By "easily extensible" we mean that extending the system to support new protocols/hardware should require minimal knowledge of the internal architecture from the developer and make extensive

use of reusable components. Therefore, the application has to be modular, and its granularity has to be fine enough to permit easy module replacement and reusability. Also, functional blocks [34] should be used to gather functions with common characteristics; for example the functions used for bandwidth management. This requirement is satisfied by using *control protocol adapters* as discussed in Section 4.4. It would also be suitable that a portion of the code be generated automatically from the framework's design model. This is achieved using code generation tools or IDEs (Integrated Development Environments).

4.2.3 Reliability

Two levels of reliability of the application envisaged in this paper are required:

a) *Completeness*, that is, the application must handle all possible "predictable" errors originating from the NE, such as the failure of a login command due to a wrong password. The application thus never "blocks" and recovering control of the situation is always possible. Most importantly, if a command fails it should return the NE to its original state. This is crucial because if a geographically-remote configuration erroneously puts the NE in an inconsistent state, it can result in important service-interruption times until an administrator physically present on the site fixes the problem.

b) *Correctness*, that is, eliminating inherent design errors from the application, such as wrong memory management. Of course it is a general requirement desired for all types of applications, not only the one we're discussing herein.

But we stress that using IDEs for software development greatly reduces the chances of coding errors, since the generated code has been formally verified once and for all by the designers of the code generator.

The proposed implementation increases the reliability of the application. Indeed, section 5.3 shows how the completeness of the application can be efficiently achieved.

4.2.4 Performance and Scalability

For manual administration tasks, performance is usually not an issue. But for more complex scenarios such as concurrent access for QoS resource reservation, it is important to minimize the processing time on the server.

Performance can be improved at several levels. First, client/server communication times can be minimized by using an efficient and lightweight middleware. SOAP/HTTP latency can be optimized using a scaled-down HTTP server with an embedded SOAP processor (implementation choice adopted by [20]).

Second, NE latency should be minimized by choosing an appropriate control protocol, which is a choice that varies from a NE type to another. In our case, we consider that CLI presents satisfactory performance for most NE classes.

Third, the control server should be multithreaded such that it can handle parallel requests for different NEs simultaneously. This prevents a client requesting a control procedure on NE_A from waiting unnecessarily until a request from another client on NE_B is served, thus reduces processing delays from the client's point of view.

Finally, another possible optimization is the use of *finite automata* to parse the textual output of NEs (only when the control protocol is text-based, of course). Finite automata are mathematically proven to be the most efficient algorithm for regular expression matching (see Section 3.3).

4.2.5 Security

When it is a matter of configuring corporate routers and switches, security –ranging from protection against theft of resources to protection against denial of service attacks– becomes a hot concern. However, the application should use existing security standards in a modular and transparent way, i.e. without affecting the architecture of the non-secured version of the system.

Our design is consistent with this requirement. Client/Server communication security can be achieved by layering SOAP over HTTP, in turn over SSL. On the other hand, communication between the control module and the NEs does not generally need to be fully secured since both components would typically reside on a same security domain (typically a same LAN behind a same firewall). The only security feature that should always be enabled for every NE is access control (through an authentication mechanism that varies from a control protocol to another).

4.3 Web-based distributed architecture with SOAP/HTTP

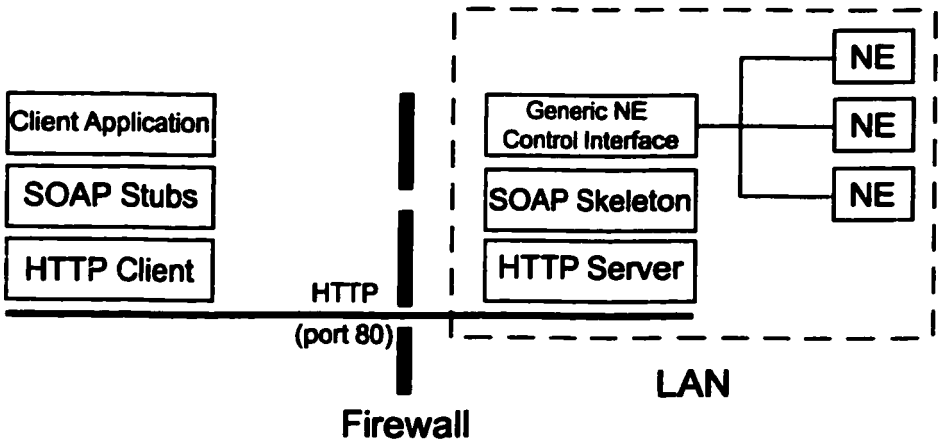

Figure 4-1: SOAP/HTTP for firewall-friendly web-based access.

As outlined previously, SOAP will be used in our system as the middleware layer to support web-based access to the generic control interface. Figure 4-1 shows a layered view of the system emphasizing its web-based aspect. The control module, which services are exposed through the *generic NE control interface*, controls heterogeneous NEs within a same LAN (more generally, NEs situated behind a same corporate firewall). This generic control module is accessed through the firewall using SOAP/HTTP. The SOAP layer consists of the pair SOAP *stubs* (client side) and SOAP *skeletons* (server side). (See Section 3.2.3 for a brief introduction to the roles of stubs and skeletons).

The generic control component exposes its interface –enabling SOAP-based RPC– as a set of *web services*. The Web Services Description Language (WSDL) [6] will be used to describe our interface. Roughly spoken, a WSDL file is an XML-based document

that basically contains the signatures of the available methods (services), which gives the client sufficient knowledge regarding the necessary SOAP structures to make RPC calls. More precisely, "WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint" [6] (See Section 3.2 for an introduction to WSDL).

Chapter 6 discusses implementation details regarding the incorporation of SOAP and WSDL in the application. The next Section discusses the abstract architecture of the generic control module itself.

4.4 Server-side Generic Model

The architecture is two-fold (see Figure 4-2). The *Generic NE Control Interface* is the top layer and represents an abstracted view of a set of NE control functions. This interface is the one that is remotely accessible through SOAP-RPC as previously discussed in section 4.3. Such an interface may expose *any* set of high-level control procedures. Indeed, if an upper layer tries to invoke a generic control function on an NE that doesn't support a control procedure with the same semantics, or if the procedure is simply not implemented for that NE, then an error message will be returned.

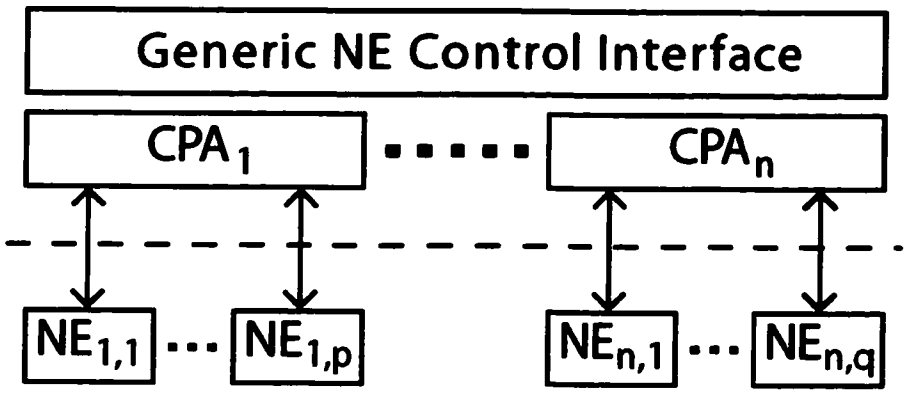

Figure 4-2: Architecture of the generic control system.

The granularity of commands allowed by a NE usually differs from a NE class to another. For example, some NEs will allow individual port configuration, while others will only allow configuration of ranges of ports at once[1]. We can say that the "port configuration command" of the first NE class is more granular than the second's.

The commands included in the generic interface can obviously be at most as granular as the finest grained commands among the supported NE classes. However, they do not necessarily have to be as fine-grained as possible. On the contrary, it is often desired to have access to *conceptually simple operations* that map to complex sets of commands for each NE class. For example, it can be said that "setting the bandwidth for a queue to a certain value" or just "logging in" are conceptually simple requests, although they may have to be mapped to quite long sequences of commands and intermediate tests

[1] For example, some NEs allow configuration of queues and their associated bandwidths on a per-port basis, while other classes of NEs will only allow a same global queues configuration for all ports.

for a given NE class ("logging in" may involve sending several passwords, checking for timeouts, retry the login procedure when an error occurs, and so on).

The second layer is the *control protocol adapters* layer. A Control Protocol Adapter (CPA) is a component that adapts from the generic interface into a specific control protocol for a particular NE class. The CPA thus presents an abstracted view of this NE class to the rest of the system. CPAs are implemented on a per-NE-class basis, which we consider being a sufficiently fine granularity, and each CPA may support one or more control protocols for the NE class that it is bound to.

CPAs are an application of the *Adapter pattern*. The Adapter pattern is one of the structural patterns listed in the reference book *Design Patterns* by the Gang of Four [37]. In software development, an adapter simply maps the interface of one class to that of another. Adapters are used continually throughout the software development process, hence the term Adapter *pattern*.

When a method is invoked through the generic interface, it is dispatched to the appropriate CPA, i.e. *the* CPA of the concerned NE class. An instance of that CPA will then choose one of the supported control protocols –using a predetermined selection scheme– and map the generic method into an appropriate set of commands relatively to the selected protocol. Note that the protocol selection logic is up to the CPA implementer, as long as the end result is adequate and that it doesn't affect critical parameters of the system (performance, security, etc.). Again, a CPA may only implement a subset of the methods exposed by the generic interface, and should return an appropriate error message if a non-implemented/supported method is invoked.

As mentioned in the previous paragraph, when an invocation is made through the generic interface it is *dispatched* (at run time) to the right CPA. In order to achieve "Command dispatching" *transparently*, some knowledge has to be provided to the generic control server about the nature and state of the NEs it is going to control.

Indeed, an implicit knowledge of the composition of the network, that is, the classes of NEs forming that network, is needed. This is necessary because when an operation is invoked upon a certain NE the control server has to choose an appropriate CPA to handle that operation, which can obviously not be done if the class of the NE is not known. Making this information available to the control server can be done by several possible methods as long as it remains transparent to the upper layer. One possible method is the use of a basic network information service. Such a service maintains a database containing specific information about NEs of the network and makes it accessible to the control server. The database in question is therefore populated by the information service and read by the control server. The information provided by this service is typically concise and would consist mainly in vendor, supported protocols and software version information for each NE. As for the network information service itself, its architecture may vary as well. It can simply be inexistent if the database is populated manually (an approach only suitable for small networks), or it can consist in an automated discovery service that would use standard management attributes (with SNMP or CMIP for example) to get the necessary information. Note that the performance of the discovery service is not an issue here because the database is typically filled offline and rarely updated. Also, the control server would access the database directly and hence

performance issues are shifted to the database implementation, which is usually very efficient.

Supplied with information about the controlled network, the control server now can dispatch control requests to the corresponding CPAs. In the most general case, some knowledge about the state of the control system itself –that is, knowing which CPA instances are active and which NEs are being controlled at a given moment– may be needed as well to make dispatching decisions that would insure efficient use of available computing resources. Therefore, given that such knowledge is conceptually global and unique within the system, it would be preferable to relay the dispatching task to a separate component. The dispatcher is not represented in Figure 4-2 because it remains an implementation detail and the latter statement is just an implementation suggestion. It will be further discussed in Chapter 6.

Note that there is no restriction regarding the number of NEs a single CPA instance can handle, as long as they all belong to the same class. Reciprocally, nothing prohibits the use of several CPA instances to control a same NE. Indeed, some NE classes support multiple parallel requests as long as they don't conflict. As outlined in Section 4.1, the consistency of control commands is not checked by the implementation of the generic control interface, but rather by the upper layer. This is why the implementation should not put any restriction on the number of parallel control requests per NE.

Finally, the range of supported NE classes is reflected by the different CPAs implemented within the application. Adding support for another NE class will consist in the implementation of a new CPA for it, which is a modular and scalable way of

extending the application. As we will see later, adding new CPAs is a simple task that doesn't require detailed knowledge about the architecture of the rest of the system, and is performed mainly through sub-classing.

The next Chapter discusses a specific CPA architecture, namely CPAs supporting Command Line Interface (CLI) control protocols.

5 CLI-based CPA design

In this chapter, the problem of choosing a control protocol for a given NE is presented. The choice of CLI as a starting point is justified. The particular case of CLI interaction protocols is then studied, and specific relevant issues are presented and addressed. A component-based, real-time model is used to design the architecture. A new concept, called *Dynamic Triggering Filter*, is finally introduced as an evolution of regular-expression-based parsing in CLI interaction automation.

5.1 Control Protocol Selection

A CPA will require a control protocol to achieve remote control of NEs. This protocol may be SNMP, CMIP, or a login-based Command Line Interface (CLI) mechanism, etc. The NE class itself rather than the designer of a CPA often determines the choice of that control protocol. Several criteria, in fact, have to be considered.

A first criterion would be the availability of the desired functionality. All control protocols that do not support the target functionality are thus eliminated. For example, while most NE classes support SNMP and some of them tend to be CORBA compatible as well, operations specific to the desired control may not always be offered via these

protocols. On the other side, the widest set of control functions in a NE is **always** available via CLI.

A second criterion would be efficiency. This is an optional criterion and depends on upper-layer's requirements. Nonetheless, it can be mentioned that SNMP, for example, has a rather poor performance. In order to configure the state of a NE with SNMP, one has to send UDP packets having the following properties:

- Each packet contains authentication information, which has to be processed by the NE every time. There is no notion of "security session".
- Each SMNP packet can only perform a single and primitive SET operation. Thus many packets need to be sent in order to perform a complex configuration.
- SNMP agents (i.e. embedded in the NE) are typically slow. In fact, the agent usually translates the SNMP data into an internal object representation after undergoing several validity checks. Moreover, some SNMP agent implementations surprisingly translate the SNMP data into internal CLI commands, in turn translated into internal objects.

In comparison to SNMP, CLI has many advantages:

- Authentication information needs to be sent only once, since CLI is used within a "session". Subsequent configuration commands will use an already-opened session to communicate with the NE, thus saving the authentication overhead.

- Complex configuration can be achieved with a few CLI commands (typically 1~3 commands) that map to a much larger number of SNMP SET requests. The traffic necessary to carry that data is obviously low. The generated response is also concise, thus low on bandwidth consumption and faster to process by the controlling entity.

A third criterion would be reliability. In the case of SNMP, since UDP is used the configuration is unreliable. Some acknowledgement mechanism has to be in place to guarantee the successful configuration of the NE. Even when this mechanism exists, it will be required for each SNMP SET request, thus doubling the volume of the traffic necessary to configure the element. CLI is better from that point of view because:

- CLI is usually run over Telnet or SSH, which both use TCP as a transport mechanism. The communication is more reliable than UDP.
- CLI allows a synchronous type of configuration. Each time a command is submitted, the NE responds as soon as it is processed with either a simple prompt or a specific message indicating success/failure of the command. Within this paradigm, it can always be guaranteed that each configuration step is carried properly before issuing the subsequent ones.

A fourth criterion would be ease of implementation. The purpose of this thesis is to present a framework that allows –among other things– easy implementation of automated CLI commands. We argue that the proposed method is easier than using SNMP/CMIP

libraries to implement SNMP/CMIP-enabled CPAs. Implementing software components achieving CLI automation using our method follows the natural interaction that an administrator is used to, and doesn't require much programming skills compared to implementing a standard software component.

Security considerations should not represent an important issue, because the controlling entity usually runs in the same security domain as the controlled NE (typically the same LAN). Therefore, using unencrypted communication over a Telnet session is not problematic. However, security has to be enforced regarding the communication between the controlling agent and a manager acting as a client for that agent, as mentioned in Section 4.2.5.

One of the disadvantages of using CLI is that only control procedures can be accomplished. SNMP and CMIP for example, support traps and thus allow monitoring of events in the network, which is impossible with CLI. Moreover, retrieving network information may reveal more efficient using SNMP/CMIP M-GET commands. This is why we stress the point that our previous argumentation is not meant to show the "superiority" of CLI over other management protocols. Instead, we aimed at justifying why CLI was chosen as the *first* control protocol to study in the subsequent sections of this thesis. We insist on the fact that it should not be used as an exclusive solution for network management, but rather coexist with other solutions in a cooperative fashion. Finally, our proposed method is designed to be extensible to any other control protocol.

Therefore, in the following it is assumed that the underlying control protocol is **CLI**. Issues related to other control protocols will not be covered in this thesis.

5.2 CLI-based CPA design issues

CLI-based CPAs use the command line interface to a box to control it. This is accomplished by opening a login session (via Telnet, SSH or some other means) to the box and then sending *textual commands* to it. Traditionally, this is a task that is performed manually by network administrators to control their NEs. Our purpose is to automate this interaction using a method that satisfies the requirements previously listed in Section 4.2.

The following general CLI characteristics have to be taken into account:

a) CLIs are heterogeneous across vendors and technologies in terms of syntax as well as semantics. CLI-CPAs thus need to map operations exposed by the generic interface into CLI commands on a per-NE-class basis.

b) CLI's output can be viewed as a stream of characters. Accordingly, the CPA should provide a stream-oriented parsing of the NE's textual output.

c) Some CLIs support a basic form of trapping by sending text messages asynchronously. The CPA has to provide a mechanism to parse such messages whenever they are generated by the NE.

d) Some CLIs are implemented such that the prompt may be re-printed prior to an error response being generated. In this case, it may make detection of errors difficult.

All the issues above have to be considered carefully in order to insure reliable NE control via CLI. Our design will address these issues incrementally in a way that reduces the complexity of coping with all the above problems simultaneously.

For the designer of the CPA, being familiar with the Network Element CLI interface is clearly a prerequisite. However, writing a CPA should require little or no knowledge about other implementation details of the framework helping the CPA's writer focus on CPA/CLI interaction logic. The design architecture discussed below takes this requirement into consideration as well.

5.3 FSMs and Regular Expressions for CPA/CLI interaction

5.3.1 Introduction

Commands that are conceptually simple sometimes map to complex sequences of CLI commands. If a command within that sequence fails, some alternatives usually have to be taken. Also, the syntax or arguments of some commands in the sequence may depend upon the result of previous commands within the same sequence or upon the state of the NE.

To model such complex interaction, a high-level command (i.e. a command present in the generic control interface) will be mapped into a *finite state machine* (FSM) that will handle generating the textual commands and parsing the reply coming from the NE (see Figure 5-1 and Figure 5-2). Also, to make the design as modular as possible, the

mission of sending textual commands to the NE and receiving "raw" text replies from it will be undertaken by a separate component (Text-based Control Client) within the CPA.

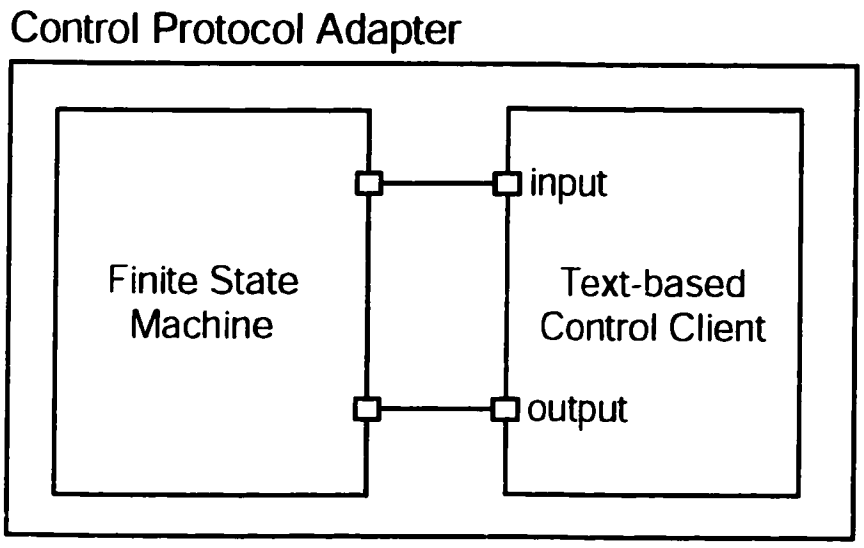

Figure 5-1: General structure of a CLI-CPA.

Figure 5-1 shows the basic structure of a CLI-CPA. The FSM sends text to the Text-based Control Client through the "input" port and receives text from it through the "output" port. The Text-based Control Client is simply a component that hides the details of sending text over a CLI session. It acts as a transport layer, and as such it performs only basic operations (opening sockets, sending and receiving text through them, etc.) and is not aware of the semantics of the CLI commands or the NE reply. The FSM is therefore the engine responsible of controlling CLI interaction.

The CPA will have to use a different FSM for every different generic operation and every different NE class. The CPA also has to infer which FSM to use transparently from

the upper-layer perspective. But the upper-layer doesn't have any knowledge, *a priori*, about the NE class it is invoking the operation upon. The upper-layer only possesses a reference to an object supposed to reflect a common abstract view of a NE. Therefore, the upper-layer cannot communicate the NE class to the CPA simply because it does not know it, which implies that the NE class has to be inferred by the CPA itself. As explained in section 4.4, such information can be obtained from an external network information service. Once the NE class for the target NE is determined, the CPA can finally map the generic operation into *the* appropriate FSM.

The FSM also has to be able to receive parameters from the containing CPA. This makes the design more flexible because it allows parameterization of certain commands, such as the port number parameter in a "port activation" command or the username parameter in a "login" command.

In summary, a CPA dynamically *imports* an appropriate FSM instance (Figure 5-1) into its FSM slot to handle the interaction with the text-based control client. That FSM instance is determined function of the NE class, the latter being obtained from an external information source.

5.3.2 The FSM component

Each FSM follows the same global behavior pattern: After the FSM has sent a textual command to the NE, it enters a new state where a certain number of possible replies from the NE is expected. This new state has a certain number of outgoing transitions, each of them corresponding to a reply. According to the received reply, one

of the outgoing transitions is to be taken and the FSM moves to a new state again. The FSM follows the same pattern until a final state is reached, where a result (optional) is returned and the FSM is subsequently destroyed.

There is, however, a difficulty that stems from the fact that the replies in question can be of a very large number. It is clearly not viable to have an outgoing transition for each expected reply. Instead, one can notice that these replies conceptually fall in a limited number of subsets according to their semantics (e.g. the subset of replies indicating an error of a certain type, etc.). Each one of these subsets can be characterized by a corresponding *regular expression* (see Section 3.3). Once the reply matches one of the expected regular expressions, an associated transition is taken.

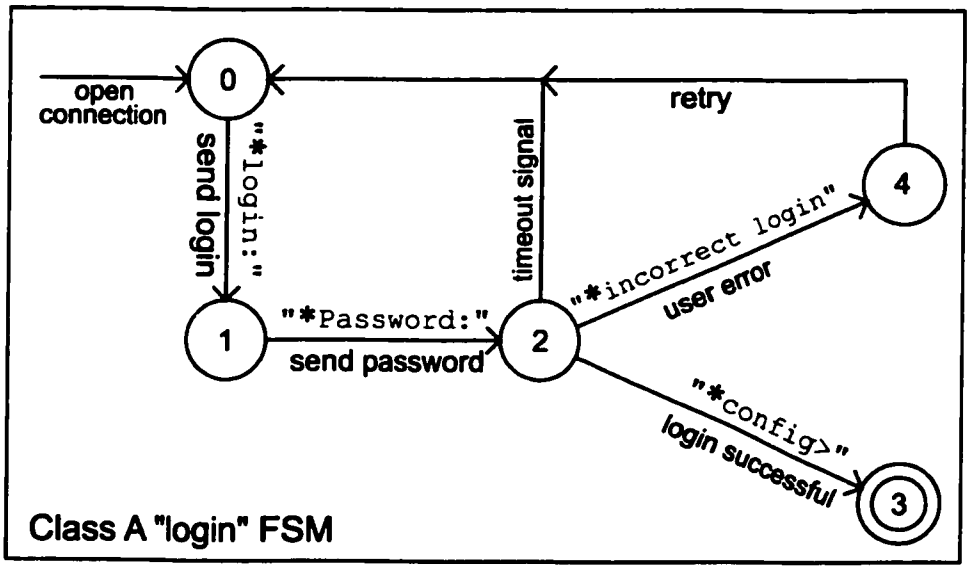

Figure 5-2: an FSM example.

Figure 5-2 shows an example. Strings within quotes and typed in `courier` denote regular expressions (a simple Unix-like notation for regular expressions is used for clarity, such as the '*' that denotes an arbitrary sequence of characters. For the prototype implementation, a standard notation is used instead). Text typed below transitions represents *actions* to be executed when the transition is taken. For example, the transition from state (1) to state (2) expects the password prompt (which is « `Password:` » in the present figure example), and sends the password as soon as that prompt is detected.

The above can be abstracted by saying that the state-transition diagram of the FSM has the following properties:

i) Each transition has an associated action which is executed every time that transition is taken (typically sending a textual command to a NE), and

ii) Each transition can be triggered by « the set of strings matching a given regular expression ».

The latter point is the one on which our efforts will focus the most. Indeed, in most FSM mathematical models, a transition is taken when its associated symbol *equals* the input symbol. Input symbols and transition-attached symbols are, in particular, of the same type (they both belong to the same underlying *alphabet* [12]). In our case, however, input symbols (characters) and transition-attached symbols (regular expressions) are not homogeneous as we would like them to be.

To address the issue of heterogeneity between input symbols and transition-attached symbols, a first solution would consist in changing the "equality operator" (i.e. the

operator used to compare input symbols and transition-attached symbols), such that its left operand be a character and its right operand be a regular expression. The following can be done to achieve the desired behavior: every time a character is received by the FSM, it is stored in a queue forming a string. Then, the current string is checked against the regular expression (the right operand of the operator). If a match is detected, then the equality operator returns *true*, which fires the transition to which the regular expression is attached.

The above workaround bypasses the heterogeneity obstacle. Unfortunately, it raises a more problematic issue. Actually, such an operator is usually expected to be *stateless*. In the most general sense, this means that when comparing any two elements, we expect the comparison result to be the same no matter how many times we repeat this operation, and regardless of the previous elements involved in other comparisons (i.e. the *history* of the operator). It is obviously not the case with the overloaded version of the equality operator introduced herein, since the result depends upon some previously-queued characters. Even if we could tolerate such behavior, it would clearly be a source of bugs since it is hard to observe the behavior of a program involving a stateful operator. In short, the above "solution" is just not practical.

There is also a remaining detail. In classical FSM models, when outgoing transitions have distinct triggering symbols, the fired transition can be resolved unambiguously; there is at most one transition to be taken, and the FSM is said to be –at least locally– *deterministic*. Unfortunately, when the symbols in questions are "regular expressions",

things become slightly subtler. Distinct[1] regular expressions, indeed, do not always recognize disjoint[2] languages. Even worse, given two regular expressions it is not always obvious whether the languages they recognize are disjoint or not, and the answer may require non-trivial computations. In our model, this means that even when outgoing transitions have distinct associated regular expressions, there may still be an ambiguity if at least two of them recognize non-disjoint languages. Such *non-determinism* is obviously undesirable since it can be a source of hard-to-detect bugs. The rest of this section explains the solution chosen to address the previous two issues.

The idea is simple: confine regular expression matching within a separate component, which we will call Regular Expression Module (REM), and define an interaction protocol between the FSM and the REM. The REM should naturally be encapsulated within the FSM component since it is an implementation detail from the point of view of the Text-based Control Client and the CPA.

---

[1] That is, matching (recognizing) different –but not necessarily disjoint– languages.

[2] That is, their intersection is empty.

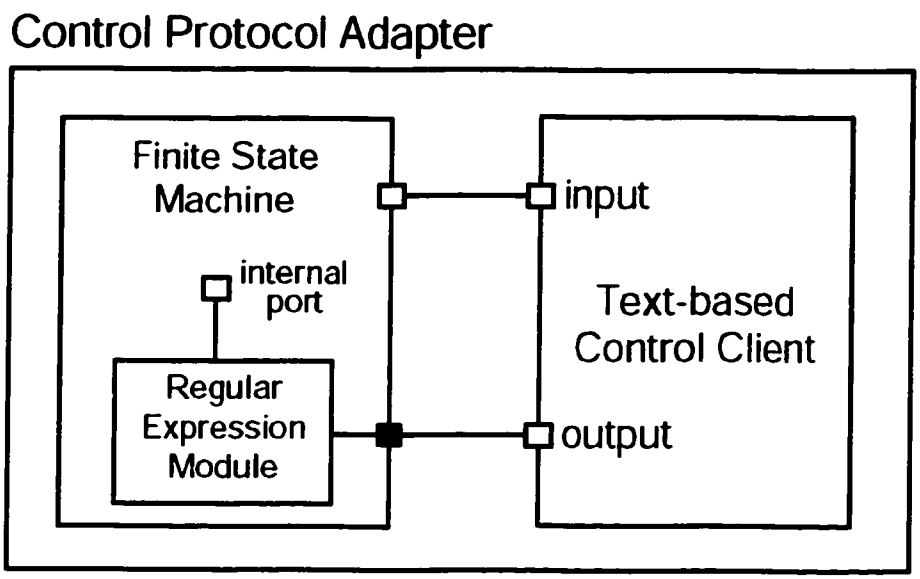

Figure 5-3: FSM with internal REM.

Figure 5-3 shows a component-based structure of the CPA where the internal structure of the FSM is also visible. The FSM component automates:

a) Sending the textual commands via the "input" port of the Text-based Control Client component, and

b) Parsing the output of the NE received via the "output" port of the Text-based Control Client. The parsing is done within the FSM through the Regular Expression Module (REM). The latter communicates with the encapsulating FSM via the "internal port".

Since the mission of regular expression matching has been conferred to the REM, the alphabet on which the FSM operates is no longer "regular expressions" and a new alphabet has to be defined. Now the REM will be responsible of triggering the FSM with symbols from this new alphabet. It has to read input characters –coming from the NE–

and send triggering symbols to the FSM whenever one of the regular expressions is matched.

A property that has to be verified by this new alphabet is having a *partial order.* Indeed, as we mentioned earlier in this section, there is a non-determinism issue due to the use of regular expressions to trigger FSM's transitions. To cope with it, an additional notion of *priority* needs to be attached to each of these regular expressions. The REM will check the character input stream against the regular expressions in their order of priority. The triggering symbol sent to the FSM is the one corresponding to the matched regular expression of the highest priority. By convention, the highest priority will be given to the smallest symbol relatively to the order of the alphabet.

A simple alphabet that satisfies the above requirement is a finite set of *integers.* That is, from now on the FSM will be triggered by integer symbols. Regular expressions attached to FSM transitions are now additional information that is dynamically communicated to the REM. The example presented in Figure 5-2 is now slightly modified into the one of Figure 5-4. Note how the choice of transition-associated integers is totally arbitrary as long as the FSM remains deterministic and the implicit priorities do not corrupt the proper behavior of the FSM.

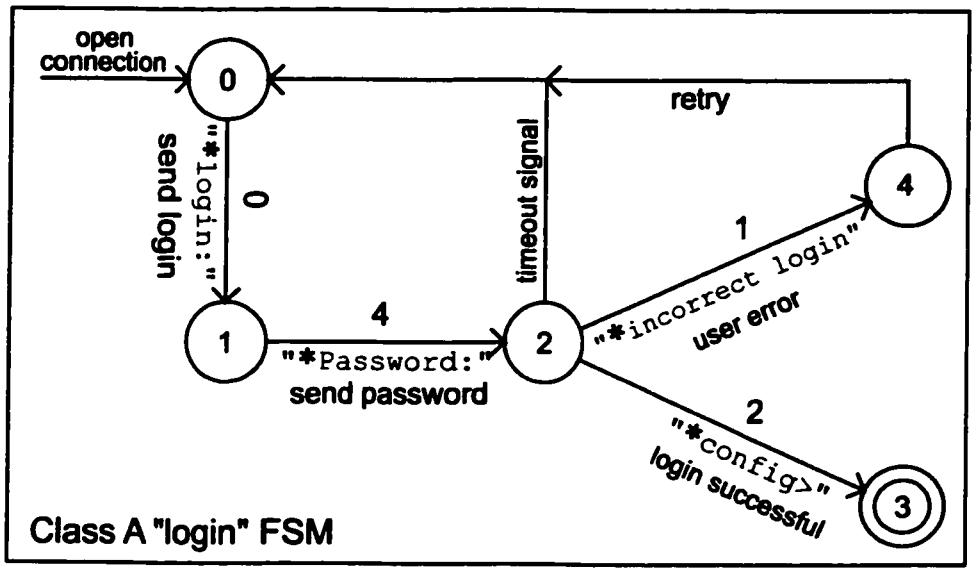

Figure 5-4: an FSM example with integer alphabet and attached regular expressions.

More precisely, every time the FSM enters a new state, it updates the REM data by sending the set of regular expressions it is expecting. The REM reads the characters continuously and sends a symbol (the triggering integer) to the FSM when the sequence of read characters matches the current expression of the highest priority. The FSM takes the transition triggered by the integer received from the REM, updates the REM data again, and so on.

By adopting the previous approach, that is, bringing more structure into the FSM, the task of defining new FSMs for new generic operations is simplified. Actually, if the interfacing between the FSM and the REM is implemented in a base class for all FSMs, it needs only to be inherited by new ones. The FSM defining that FSM/REM interfacing would be the root of the FSM class hierarchy. As a consequence, extending the range of

supported control procedures is done in two simple steps: i) adding an FSM by sub-classing from the base FSM class, and ii) binding it to the generic operation it is meant to implement. In this new FSM, only the state-transition diagram has to be specified, which is obviously the minimal set of information that must be provided by the implementer of the FSM. Re-implementing the default FSM-REM interaction (see section 5.3.2 above) is not needed, and neither is the knowledge of other implementation details, such as the CPA architecture itself. This satisfies the easy extensibility requirement introduced previously in Section 4.2.2. Chapter 6 shows this scheme in great detail using UML's formalism.

Increasing the reliability of the system also becomes easy when using FSMs, because:

a) Complex error-detection and error-recovery schemes may be implemented (which cannot be represented using simple command sequences, for example). We called this feature *completeness* in section 4.2.3 above.

b) The graphical design of the FSM reduces the likeliness of design errors and facilitates debugging, especially if it can be observed at run-time.

5.3.3 The Regular Expression Module

The above can be summarized by saying that the REM acts as a *dynamic filter* on the character input stream. The word "filter" is used herein to denote an active component that transforms a data stream into another data stream, namely a character

stream into an integer stream. It is "dynamic" because its internal parameters are changed dynamically by the FSM every time the FSM enters a new state. Those parameters in question are the set of regular expressions against which the character input stream is checked, in addition to the integer symbols bound to each regular expression in that set.

We purpose now to address the issue b) of Section 5.2 by analyzing its nature and finding necessary design choices for the REM.

Issue b) requires the REM to be stream oriented. There is a classical and powerful algorithm used for regular expression matching based on Kleene's theorem [16] as introduced in Section 3.3.4, which basically states that every regular expression can be mapped to a finite automaton (FA) that recognizes the same language. A finite automaton "consumes" the input symbols once and never reads them again, which makes them a best-performance tool suitable for stream-oriented parsing.

5.4 Further abstraction: Dynamic Triggering Filters

The architecture discussed so far is the one that we adopt for the implementation of our prototype, presented in Chapter 6. The purpose of this sub-section is to discuss further enhancements to the architecture of the FSM that would allow more flexible and thus powerful interaction with the CLI. These enhancements would also allow future extensibility to other types of User Interfaces, namely web-formatted UIs (with HTML or XML for example). These new enhancements will be recommended later in Chapter 6.

Let's first summarize the concepts introduced with the FSM. The first obstacle arose when we needed to trigger a finite state machine with something that is more complex than a simple symbol, namely regular expressions. The proposed solution to that problem was to shift the difficulty of regular expression matching to another component such that the FSM be triggered by simple symbols sent by that new component. The new component that we introduced was the REM. The REM handled regular expressions and acted as a dynamic filter on the character input stream. It transformed the input stream into a stream of integer symbols (or *signals* in UML vocabulary) carrying data (the matching string). The reason why the matching string was sent along with the triggering symbols was to allow the FSM to extract pertinent data and possibly undertake further actions depending on the contents of that string.

By observing the latter scheme, we can notice that what the FSM really needs is a pertinent *feedback* from the parsing module (the REM, so far). As a matter of fact, what is desired from the FSM's perspective is to send text commands to the NE and receive the NE response under a "distilled form", that is, signals carrying data. The FSM specifies the form and content of that distilled information to the parsing module using a certain language (regular expressions, so far). Therefore, the nature of the distilled information is limited by the expressiveness of the language used to describe that same information.

More precisely, using the REM one can only parse text using regular expressions. Although regular expressions are sufficient in most cases, they are limited to a language category called "regular languages" (see Section 3.3). Regular expressions cannot parse

XML for example, because XML is a language that falls in a super set of regular languages called "context-free languages". Moreover, using the REM, data carried by the triggering signals is limited to character strings, as presented previously. For each signal, the attached string is the one that matched the regular expression bound to the same signal. One cannot request more expressive data from the REM, such as the 3rd token of the matching string. Instead, the *entire* matching string is first received by the FSM and has to be further parsed in order to extract any other specific information.

We propose to push the abstraction further in the architecture of the REM, and use a more generic form of it that we call *Dynamic Filters Module* (DFM). The DFM has the same global role of the REM, except that it is extensible to support *any* type of parsers internally. We call these parsers *Dynamic Triggering Filters* (DTF). They are "dynamic" because their properties are settable dynamically by the FSM. They are "filters" because they transform a character input stream into another signal stream. Finally, they are "triggering" because these filters serve to trigger the FSM with the signals they generate.

With the REM, the FSM can only specify one property, which is the text pattern that the response has to match, and is limited to receive the matching string. With a DTF object, the FSM can specify more properties to be verified by the response, and can decide which information to receive when the response verifies those properties. For example, using a DTF the FSM can require the NE response to a) match a given regular expression, b) to have a minimum given number of tokens and c) to contain numerical data. The FSM can further specify that when the NE response satisfies the previous

properties, the signal sent by the DTF carry a list of numbers corresponding to the numerical data contained in the textual response.

DTFs can be incorporated within the FSM using the same logic that was used with regular expressions. Every time the FSM enters a new state, it sends a set of DTF objects to the DFM. These objects will parse the NE response in parallel and send a signal whenever they detect a match. The signal sent by a DTF that detected a match contains data objects that were extracted from the matching string. The algorithm used to extract that data from the matching string is a property of the DTF itself that can possibly be parameterized by the FSM when the DTF object is created.

Using the above idea, the DFM can be extended over time to provide additional parsing functionality by defining new DTF classes offering that functionality as needed. Another advantage of this approach is that the FSM's state-transition diagram is simplified to its maximum, because it is only left to reflect the outline of the CLI interaction and indicates clearly the semantics of the expected NE responses through the use of DTFs.

DTF is an application of the *Factory Method* pattern as defined in [37]: "Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses". In our case, the class `DynamicTriggeringFilter` (see Figure 5-5) contains an object of type `DistilledData`. This object is supposed to be sent to the FSM when the NE response matches the filter in question. Therefore, a `DynamicTriggeringFilter` subclass will

send a data object that is an instance of a subclass of `DistilledData` to the FSM. In summary, object instantiation is deferred to subclasses of `DynamicTriggeringFilter` that will decide which `DistilledData` subclass to instantiate, which is exactly the spirit of the Factory Method design pattern. In Figure 5-5, the object performing regular expression matching (`RegExp_DTF`) becomes a subclass of `DynamicTriggeringFilter` and the data object sent upon the detection of a match (`RegExp_Data`) becomes a subclass of `DistilledData`.

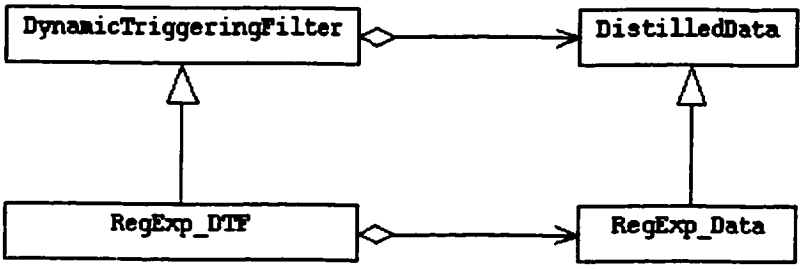

Figure 5-5: the Factory Method pattern applied to DTFs.

6 Prototype implementation

This Chapter presents a fully working prototype that was implemented in order to evaluate the theoretical concepts introduced previously. This involved two main activities:

1. Implementing the generic framework, that is, the infrastructure that provides support for the architecture described in the previous chapters. This includes in particular support for CPAs, FSM and REM, in addition to other necessary components that will be described later. This activity also involved gathering the different third-party libraries necessary to build the prototype.

2. Defining a generic control interface for the prototype, which depends on the desired service as well as the chosen target NE classes.

The implementation part also served as an evaluation of several criteria, such as the required delays to incrementally extend the application and the complexity level of this task. As mentioned before, a big importance was given to the latter criteria –beside the traditional ones such as executable size and processing delays, etc. Indeed, the application was intended to be highly flexible and easily extensible to cope with the ever-changing nature of both networking hardware and software.

Section 6.1 briefly presents the tools and third-party libraries that have been used in the implementation of the prototype. Next, Section 6.2 details the UML design and implementation of the framework. Section 6.3 shows how the previous framework was used to implement a particular control service, namely Traffic Conditioning. Finally, Section 6.4 evaluates the obtained results.

6.1 Third-party libraries and tools

The design was mostly done in UML with real-time extensions, and the implementation in C++. UML design was done using a commercial tool, namely Rational Rose RealTime [19]. This tool supports UML 1.3 with real-time extensions (see Section 3.4 for an introduction to UML's real-time extension) and C++ as an implementation language binding. All UML diagrams shown in subsequent sections are screen captures of diagrams designed using that tool. Figure 6-1 shows an overview of the tool's GUI.

Figure 6-1 also shows a *package diagram*. This diagram emphasizes relationships between packages. Dotted arrows in this diagram represent *dependency relationships*. The root package of this simple graph is the one that contains the capsules and classes that implement the infrastructure supporting CLI automation. All other packages depend directly or indirectly upon it.

The package `<<API>> NE_Abstract_View` contains classes that define the C++ API used to access the infrastructure *locally*. Note how the infrastructure package does *not* depend on the chosen API.

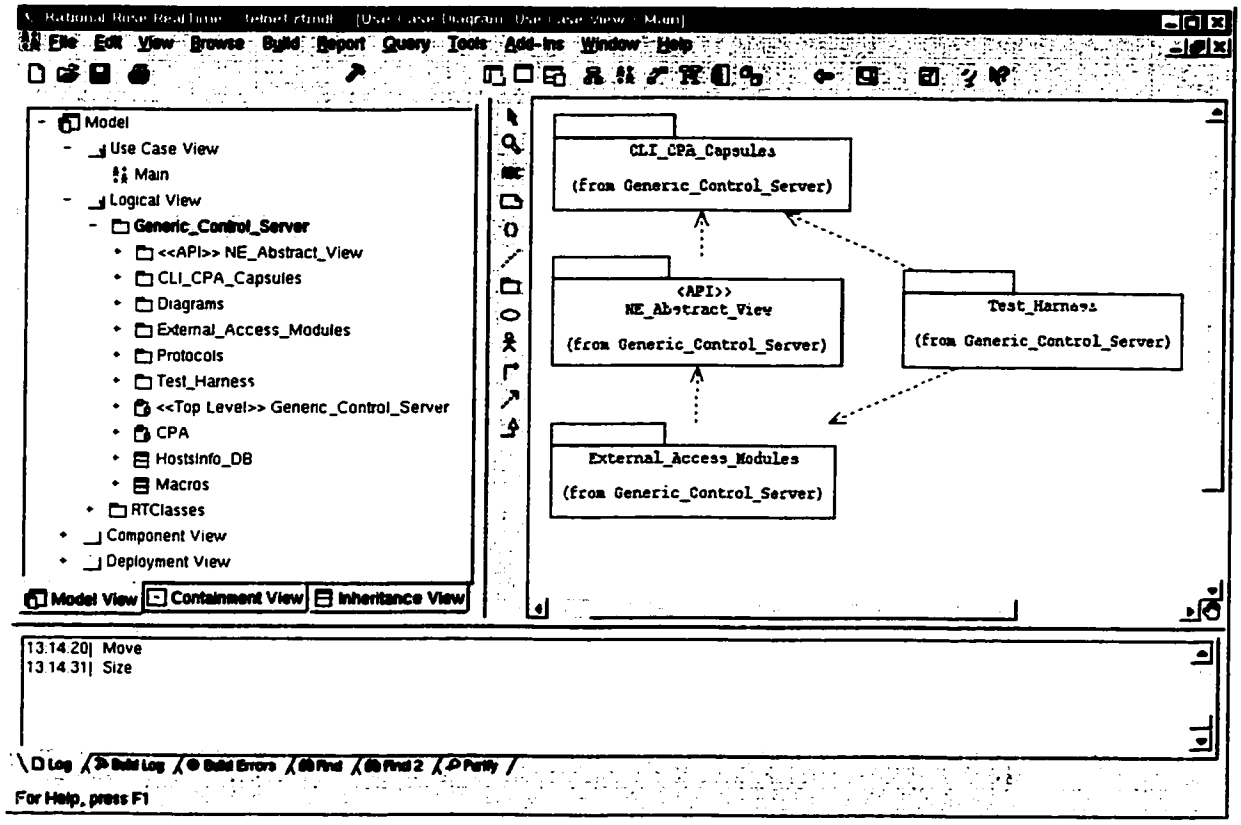

Figure 6-1: a general view of the working environment.

The `External_Access_Modules` package naturally depends on the local API package. This package contains capsules implementing different means of accessing the same local API; through SOAP, command line, sockets, CORBA, etc. It is meant to allow the integration of any middleware without changing the API and thus the API package doesn't have nay knowledge about the existence of this package.

Finally, the package `Test_Harness` contains test capsules that were used to validate other infrastructure capsules individually.

Besides Rational Rose Real-time, other open-source libraries have been used to support several implementation aspects of our framework. For regular expressions, the Grail+ project [21] was used, and finally for SOAP support Systinet's WASP-C++ [20] toolkit was deployed.

6.2 Infrastructure UML Design and Implementation

In this section, we will depict the structure and behavior of the different components out of which this infrastructure is built. We will adopt a top-down approach showing more and more details as we go deeper in the containment hierarchy. Specific UML real-time terminology will be used. For an introduction to this terminology and its semantics, please refer to Section 3.4.

6.2.1 The Generic Control Server component

Figure 6-2 shows the structure diagram of the `Generic_Control_Server` capsule. This capsule is at the top level and thus contains all other capsules within the design, and its structure is meant to reflect the abstract architecture presented in Figure 4-2.

The generic control interface is accessed through the `ExternalAccessLayer` capsule. Only a single instance of this component is necessary within the component. This component receives generic method calls –intended to perform generic NE operations– and forwards them to the appropriate CPA instance using UML signals

(through the port called `cpa_comm`. In fact, the binding –represented on the diagram with a solid line– between the `ExternalAccessLayer` and the `CPA` represents a full-duplex communication channel that carries signals from and to both end ports).

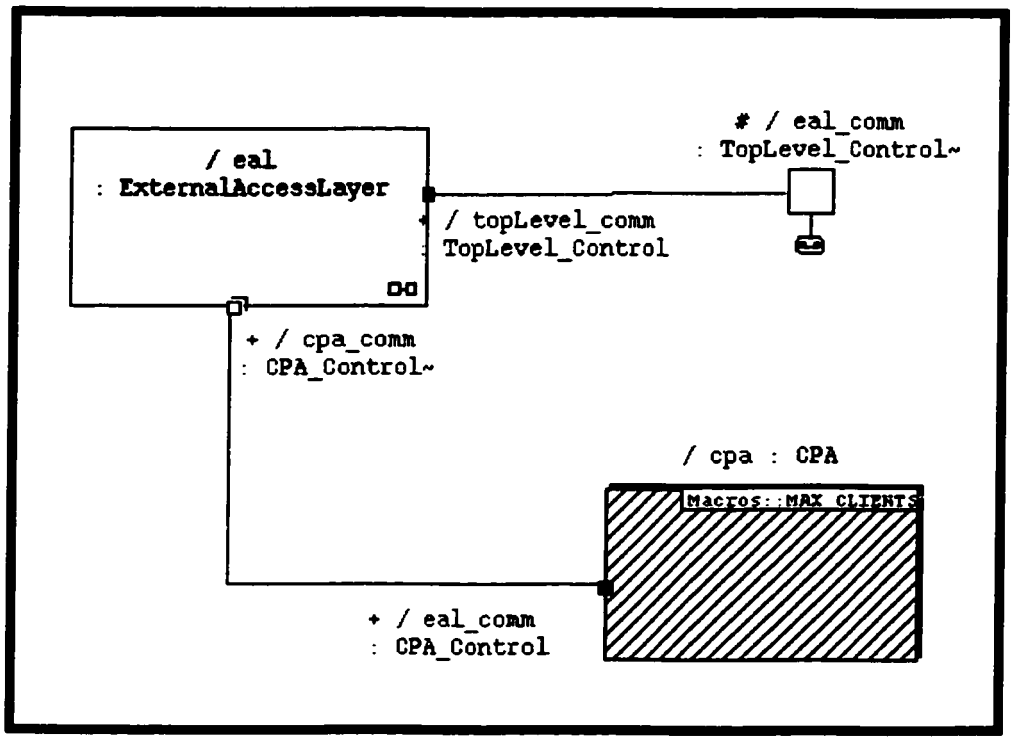

Figure 6-2: Structure diagram of the Generic_Control_Server capsule.

On the other hand, the CPA capsule is *optional*[1] and *replicated*. The fact that it is "optional" means that it is dynamically instantiated at runtime by the capsule that directly contains it –the `Generic_Control_Server` capsule here. The fact that it is replicated

[1] This word is used with UML semantics.

means that it has a multiplicity greater than (or equal to) one, which reflects the fact that more than one NE can be controlled in parallel.

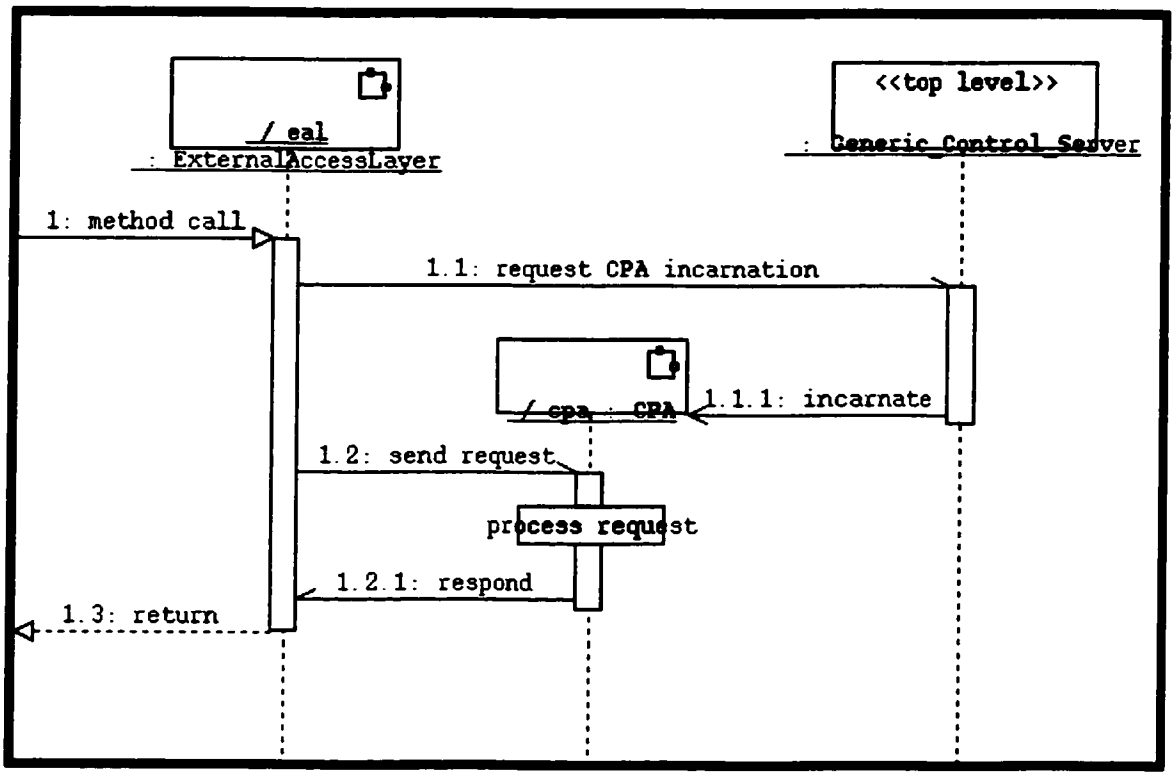

Figure 6-3: Simplified sequence diagram of the `Generic_Control_Server` capsule.

Figure 6-3 shows a simple sequence diagram depicting the basic functionality offered by the `Generic_Control_Server` (GCS) capsule. The `ExternalAccessLayer` (EAL) first receives a synchronous method call from the environment (i.e. from a component that is not represented in the diagram). Next, it sends a signal to the GCS asking it to *incarnate* (i.e. instantiate) a CPA capsule of a certain

CPA sub-class that was determined by the EAL as appropriate for that generic call. The GCS creates a CPA instance that will receive the request from the EAL via signal 1.2. The newly created CPA instance thus handles that request and notifies the EAL upon completion. Finally, the EAL returns a result to the calling method.

6.2.2 The External Access Layer component

This component is intended to separate the logic of the access to the generic control server from the architecture of the server itself. Its architecture allows coexistence of multiple access modules simultaneously (SOAP, CORBA, sockets, etc.).

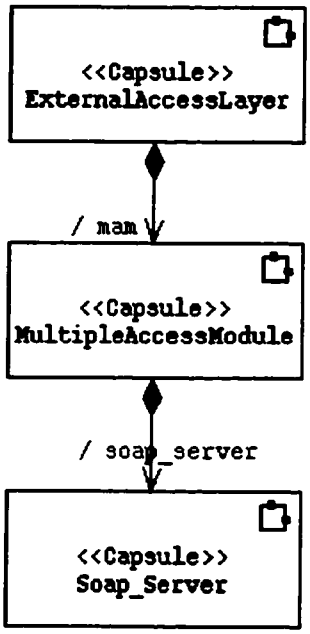

Figure 6-4: `ExternalAccessLayer` containment hierarchy.

Figure 6-4 shows the structure of the `ExternalAccessLayer` (EAL). It contains the `MultipleAccessModule` capsule. The latter, in turn, is intended to contain multiple other capsules used to provide different means of accessing the generic control server. `Soap_Server` for example (the only one implemented in our framework) allows access via SOAP. Capsules contained in the `MultipleAccessModule` component provide different interfaces to clients using different means to access the control server, but they internally use the same access method to inject signals within the control server.

6.2.3 The CPA component

The CPA capsule introduced in the previous section is a quasi-empty capsule (it only contains a single port and no behavior specification). This capsule is intended to serve as a base class for all other CPAs such that CPAs instances of different subclasses can be instantiated at runtime (according to the OO *polymorphism* concept), but the CPA base class itself is never instantiated. Therefore, adding support for new control protocols and/or hardware will be reflected by adding CPA subclasses that would allow adequate control functionality. For example, one can define a CPA subclass that would support both CLI and SNMP and decide dynamically which control protocol to use for each supported NE class.

Figure 6-5 shows a simple CPA class diagram. The class called `CLI_CPA` inherits from the `CPA` base class, and as such it can be instantiated to occupy a `CPA` slot (see Figure 6-2). `CLI_CPA` only supports the CLI control protocol.

In Section 4.4 (Server-side Generic Model), the choice of whether a CPA should handle one NE or multiple ones was left open. For simplicity, we chose to dedicate a `CPA` instance for each NE. Also, it was said in Section 4.4 that CPA classes are defined on a per-NE-class basis. This means that there should be *at most* one CPA class per NE class. Our `CLI_CPA` class will be the same for all NE classes, which is a compliant choice. The distinction between different NE classes will be made *inside* the `CLI_CPA` at the `FSM` level, as explained later in this section.

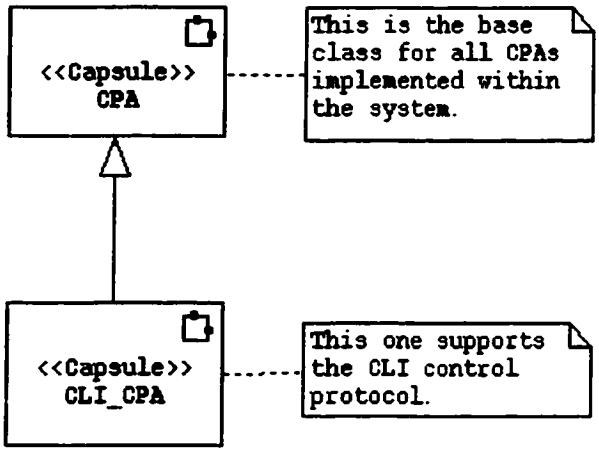

Figure 6-5: CPA class hierarchy.

Figure 6-6 shows the structure diagram of the `CLI_CPA` capsule. The two core components represented in this diagram were presented in Figure 5-1 on page 82, namely the FSM component and the Text-based Control Client. The third component in this diagram, namely the Factory, has not been discussed so far and will be briefly presented later in this section.

The `Text_based_Client` has a simple role: send characters over a text-based session (Telnet or SSH) and asynchronously receive characters from it. Implementation details of this component are irrelevant to this thesis and thus will not be discussed any further.

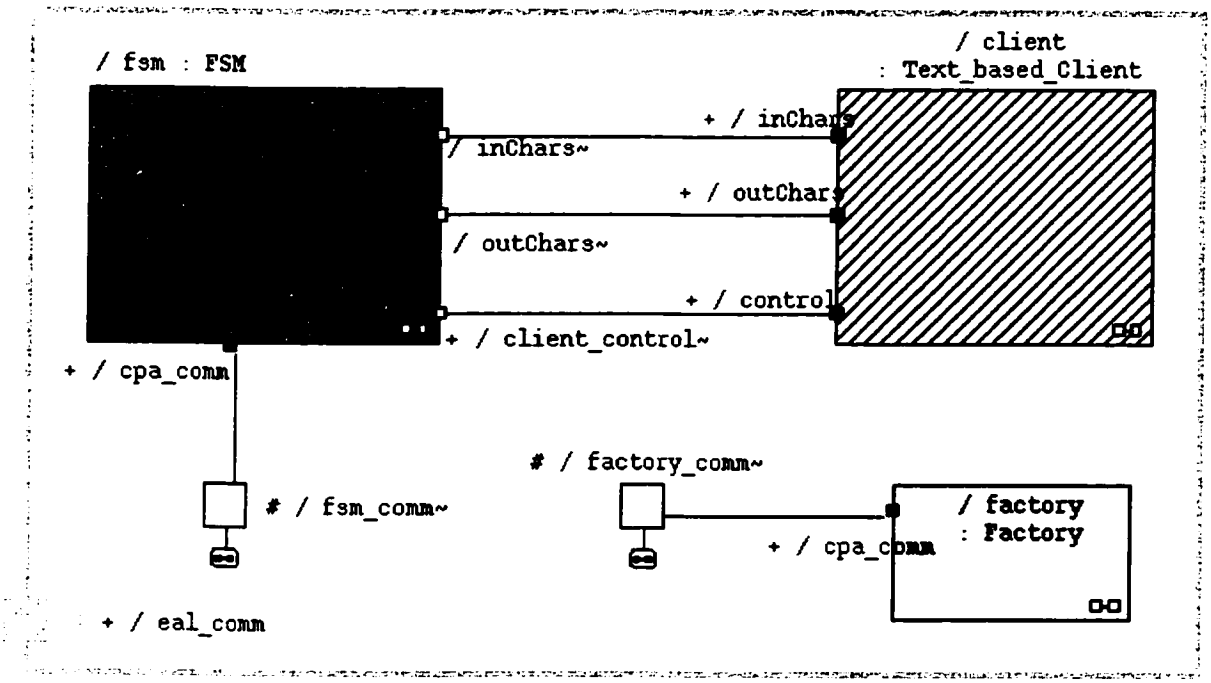

Figure 6-6: CLI_CPA structure diagram.

The `FSM` communicates with the text-based client via three separate communication lines: two lines for input/output characters and a third line for control commands (open/close connection, get connection status, etc.).

Note that the component representing the FSM is drawn in black. This means that the component is a *plug-in* role. Plug-in capsule roles are placeholders for other

compatible[1] capsules, and are used when the capsule class that has to occupy that slot is not known at design time. Remember that the CLI_CPA has to determine which FSM subclass to instantiate function of the received request and the concerned NE class. Once the FSM class determined and instantiated, that FSM instance is *imported* to that plug-in capsule role. The Factory is the component responsible of dynamically determining and instantiating FSM classes that will be later imported to the FSM plug-in role.

6.2.4 The FSM component

The FSM component is intended as a base class for all other FSMs responsible of performing CLI interactions. It provides the necessary structure and tools described in Section 5.3 so that sub-classes do not have to re-implement them.

Figure 6-7 shows the structure of the FSM. All FSM sub-classes thus will have that structure, although enriching it or excluding some components remains possible. The inChars port (which receives characters back from the Text_based_Client) is directly connected to the REM. The REM filters that character stream and sends triggering signals via the port fsm_comm, itself connected to the port regexp_comm. The FSM receives requests from the CLI_CPA through the cpa_comm port and sends textual commands via the outChars port.

---

[1] According to UML, a compatible capsule is a sub-class that doesn't exclude any connected port of the parent capsule. In our example, FSM's compatible capsules are the sub-classes that do not exclude any of the ports inChars, outChars, clientControl and cpa_comm.

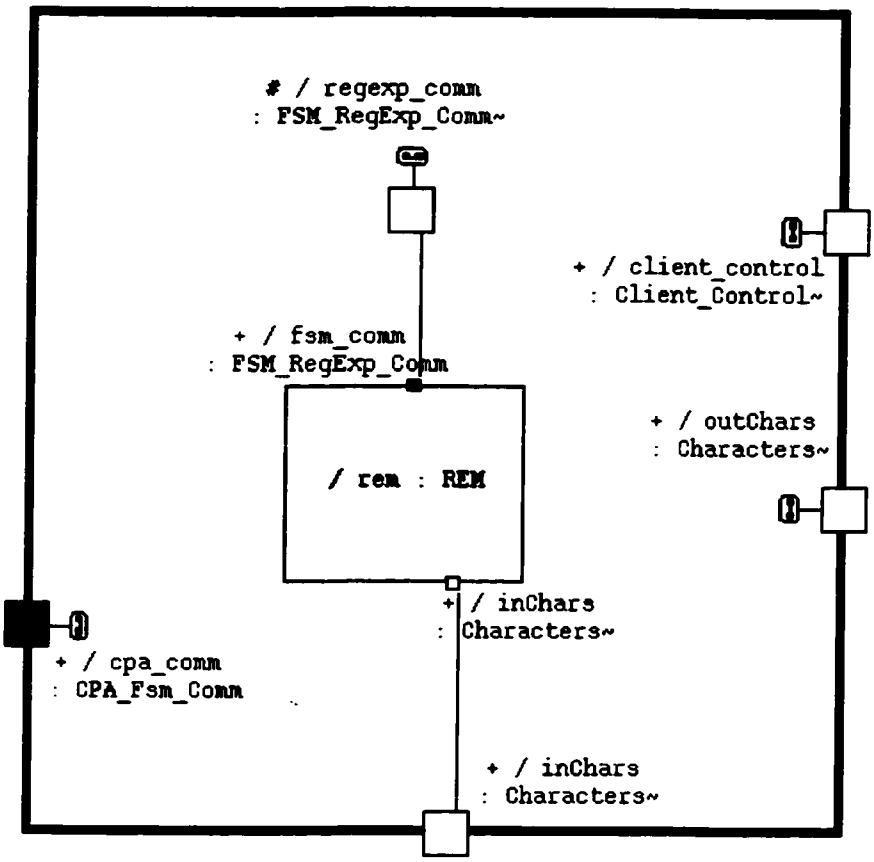

Figure 6-7: Structure diagram of the FSM.

It is unnecessary to present the REM component internal details given that its behavior has been largely discussed in the previous sections. Figure 6-8 is a UML class diagram that summarizes the previous sections and gives a global hierarchical overview of the composition of the framework.

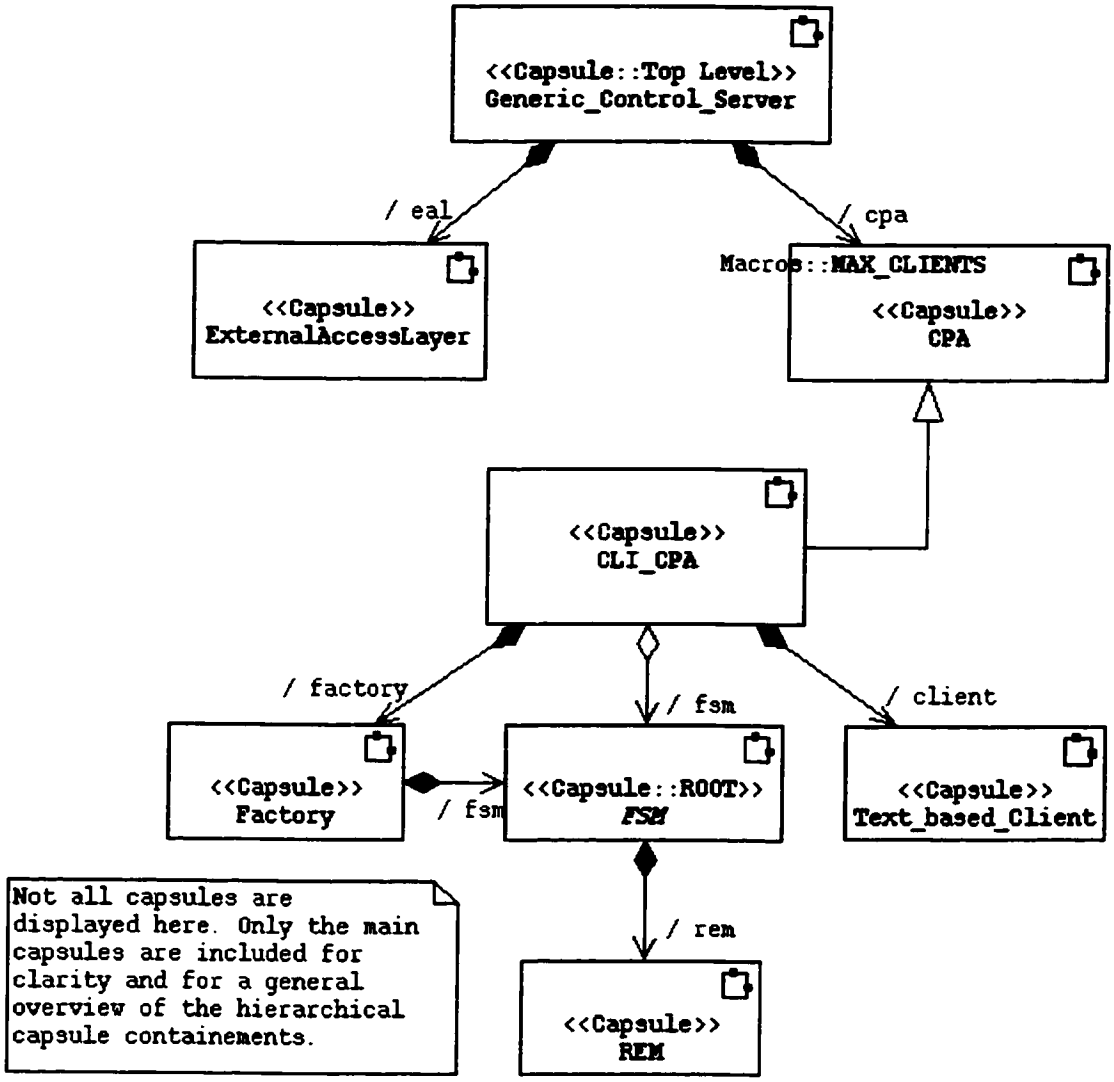

Figure 6-8: UML class diagram emphasizing containment relationships.

6.3 Implementation of a Traffic Conditioning control facility

Now that the infrastructure is in place, implementing a control service will be accomplished mostly through sub-classing.

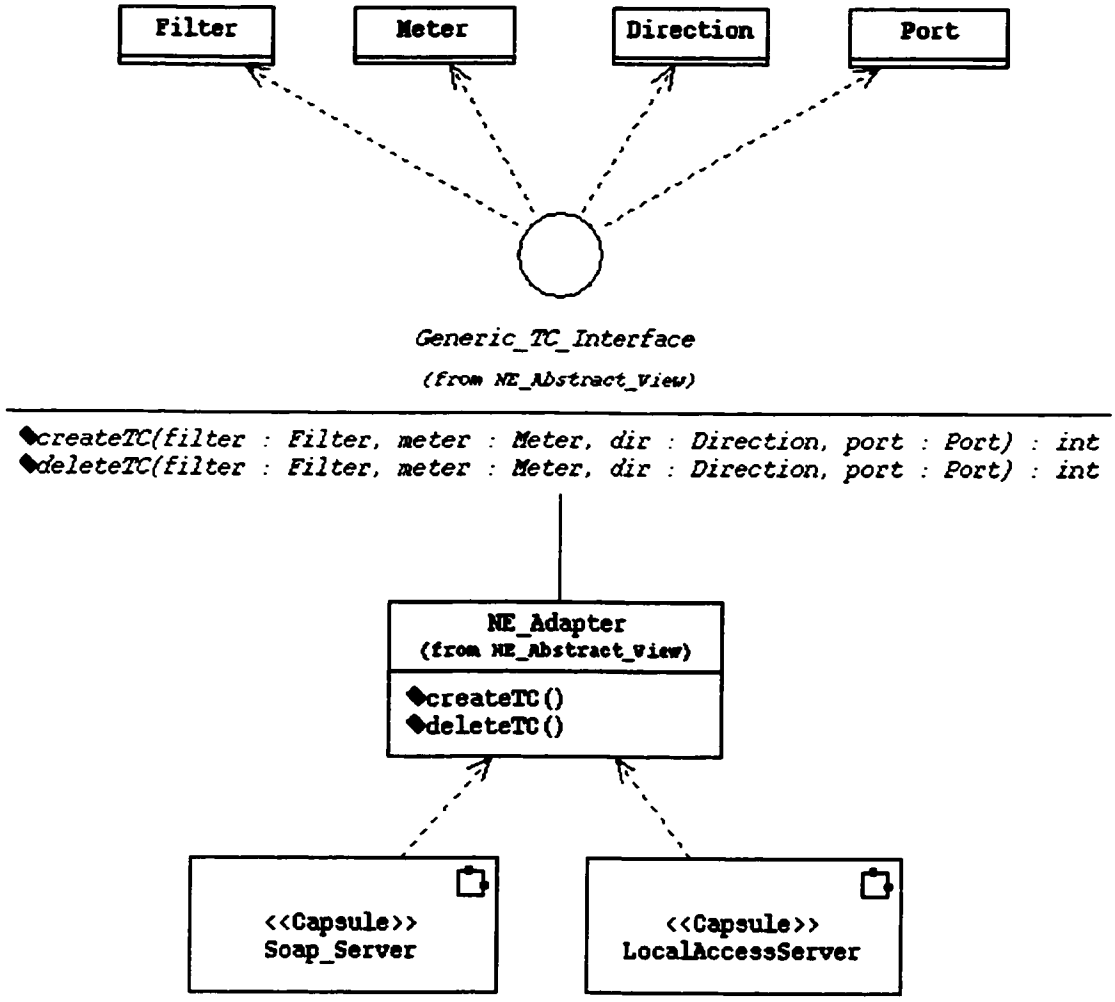

Figure 6-9: A common API for all NE classes.

Since the implemented application serves only as a proof-of-concept prototype, it wasn't necessary to expose an extended set of NE control operations. To prove the concept, the minimal interface would consist in a single operation that targets two different NE classes. In our case, two NE classes are handled, namely "Foundry BigIron 4000" and "Cisco 6505", and two control operations are exposed. These two NE classes are suitable for our example since they possess quite different CLIs, in terms of syntax as

well as interaction scheme. The implemented service will be evaluated in the next section.

Figure 6-9 shows a UML representation of the API, which will later be exposed via a WSDL interface (see Appendix A for the WSDL interface). Instances of the class `NE_Adapter` will be used to perform control operations. The two offered operations are `createTC()` and `deleteTC()`. The former creates a Traffic Conditioning (TC) policy on the element, while the latter deletes it. Since the interface is meant to be generic, the parameters taken by these two operations are generic too, and consist in data structures containing generic QoS-related parameters. Both operations will return appropriate error messages if they fail.

To keep this section as light as possible, implementation details will be given for only for one of the NE classes, namely "Foundry BigIron 4000", and will be quickly mentioned for the other class.

The first step is to define the FSM class hierarchy for each NE class. Figure 6-10 shows a capsule `FDRY_Base` as a root class for all Foundry FSMs. This class has to inherit from the `FSM` base class, and will implement common behavior such as the login sequence and standard error handling mechanisms.

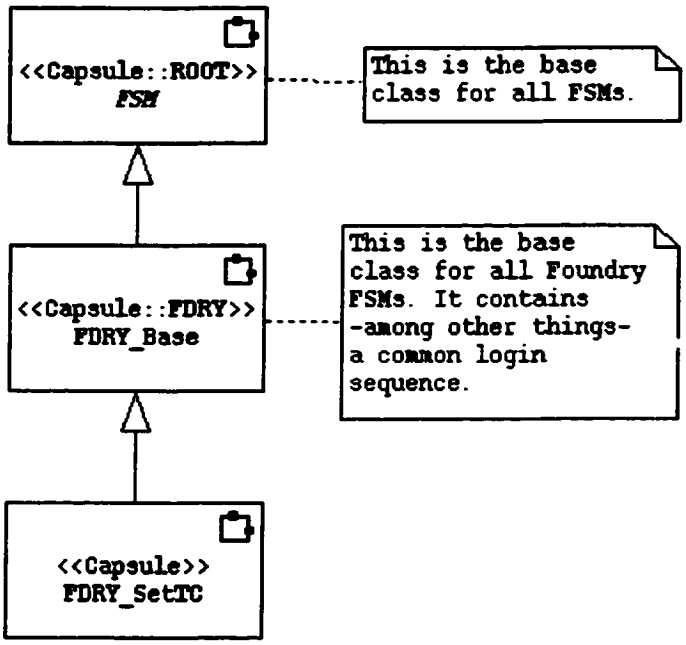

Figure 6-10: FSM class hierarchy for Foundry BigIron 4000.

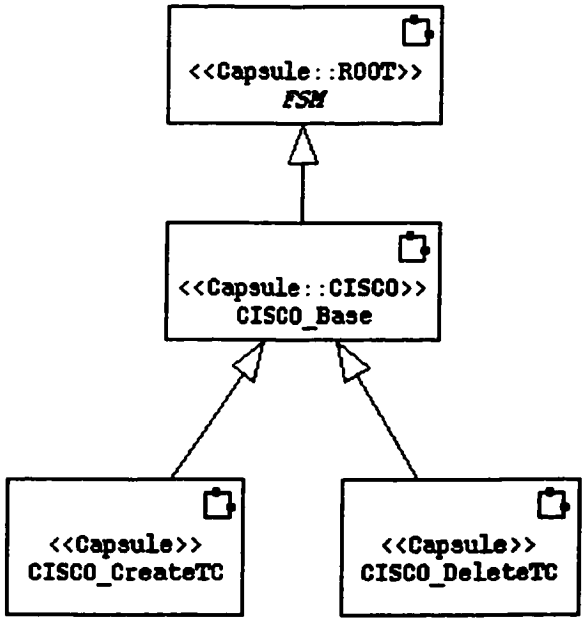

Figure 6-11: FSM class hierarchy for Cisco 6505.

The capsule `FDRY_SetTC` inherits in turn from `FDRY_Base` so that it doesn't have to re-implement common behavior. This is the capsule that is going to be used for both `createTC()` and `deleteTC()` operations. The complete design and testing of the state diagram of the `FDRY_SetTC` capsule was done in about half a day.

6.4 Tests and evaluation

A first evaluation is in regard to the ease of design and implementation of new services. While the design and implementation of the general infrastructure took almost the whole research period for this thesis, augmenting it with the TC service described previously for two heterogeneous NE classes took no more than a day (for a single developer)[1]. Therefore, we consider met the goal of allowing easy and rapid implementation of new services with the presented framework.

As for reliability, the diagrams presented in the previous section show how complex error detection and handling mechanisms can be easily incorporated to the behavior of controlling capsule. Moreover, controlling capsules inherit an important part of their functionality from the FSM base class and use facilities offered by other components of the framework which has already been thoroughly verified once and for all. Also, the simplicity of the code that has to be written in order to implement a new service is such

---

[1] The average time for such a task is estimated to two man-months. If this holds, then our method reduces the effort in a ratio of at least 1:50 in a worst case scenario.

that errors are made less likely. On the other hand the environment used to implement and verify the functionality of the FSMs (Rational Rose Real-Time) being a semi-formal tool for building finite state machines, it is likely that once the FSMs are successfully simulated they are error free. As a consequence, overall reliability is improved.

On the down side, one of the disadvantages of our framework is that it requires a minimum level of training with the Rational Rose RealTime, which is a commercial and rather expensive tool as well. But the principles and the approach introduced in this thesis are not bound to any specific tool. They can be implemented using any other method, even straight coding. However, using an IDE has undeniable advantages.

Another weakness stems from the fact that we did not define any formalism to formally validate the FSMs that perform the control operations. Validation, thus debugging, is done using a traditional testing approach, i.e. testing the application against a set of NEs in a test environment. Nonetheless, in our case debugging and validation were made easier with the "observability" feature available in the runtime library of Rational Rose RealTime, that is, the possibility to visually follow the execution of each and every state machine on a real-time basis.

The rest of this section will present a test scenario that was used to deploy the TC service introduced previously.

Figure 6-12 shows the setup of the testing environment. An MPEG2 video stream is generated by the MPEG2 video server. This stream traverses both the Foundry BigIron 4000 (BI4K) router and the Cisco 6505 (C6505) switch before reaching both video

clients. The traffic generator is here to load the network with heavy traffic. The video flow going to the first video client is always given higher priority and bandwidth, and thus the quality remains perfect during all the test period. This client therefore acts as a reference. The video flow going to the second video client will be configured by controlling BI4K and C6505. The generic control server is running on a host that is connected to the latter two elements on separate ports such that the control traffic doesn't interfere with other traffic (other configurations are also possible). The workstation issues control commands to the control server requesting the same operation (`createTC()` or `deleteTC()`) for both elements and the control server handles them transparently.

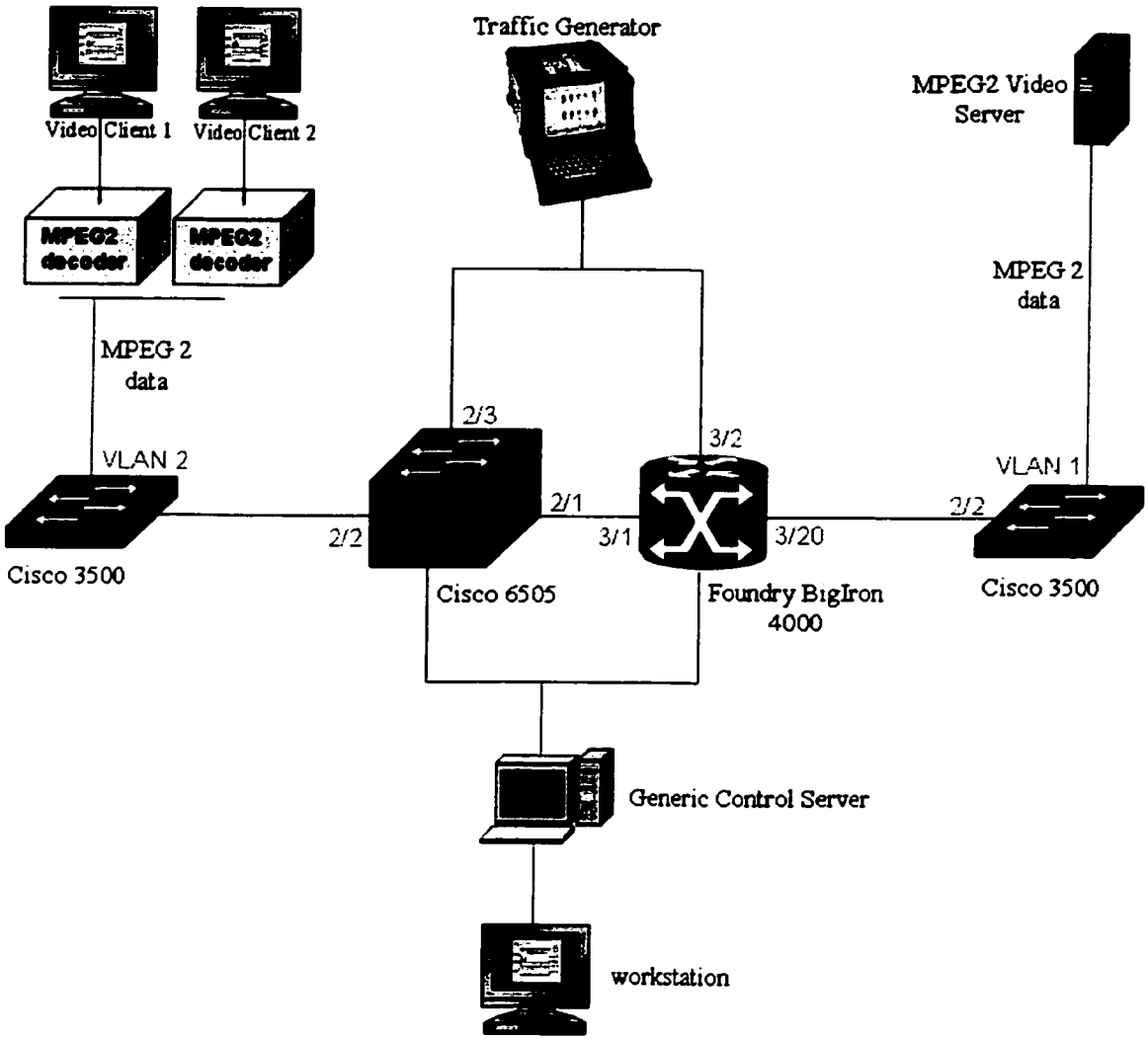

Figure 6-12: Test environment for the Generic Control Server.

Initially, the video flow going to the second client has the same priority as the traffic generated by the traffic generator, and the video quality is visibly poor. When `createTC()` is called, that video flow is given adequate resources and the video quality becomes similar to that of the first client. Reciprocally, when `deleteTC()` is called, the video quality becomes poor again.

Regarding execution times, Table 6-1 summarizes the results for BI4K and Table 6-2 for C6505. These are execution times measured on the server for separate invocations (i.e. for operations invoked upon a single NE at a time). Response delays were found longer for C6505 than for BI4K because the CLI interaction sequence required to carry the `createTC()/deleteTC()` commands for C6505 are slightly more complex than BI4K's and the CPU on the C6505 is also slower. Also, the execution of the first call is always slower because it involves spawning a Telnet client and undertaking a login sequence. Subsequent calls use an already open session to the element and thus are naturally faster. Also, the `deleteTC()` operation has a same type of interaction with the CLI as `createTC()` for both elements thus similar execution times.

Operation	First call	Subsequent calls
`createTC()`	~ 210 ms	~ 15 ms
`deleteTC()`	~ 215 ms	~ 19 ms

Table 6-1: response delays for `createTC()/deleteTC()` for BI4k.

Operation	First call	Subsequent calls
`createTC()`	~ 314 ms	~ 82 ms
`deleteTC()`	~ 295 ms	~ 64 ms

Table 6-2: response delays for `createTC()/deleteTC()` for C6505.

When run in parallel (i.e. when requesting the control on both elements "simultaneously"), it was found that the response delays were approximately unchanged. In fact, performance is limited by the speed of the Telnet session. Characters are processed at a much faster rate than their arrival rate. Therefore, when the control server is running two threads in parallel to parse two response streams, it is fast enough to switch from a stream to another without affecting the overall response delay for each invocation. It can be anticipated, however, that response delays will begin to be perceptibly affected when the load of the host running the control server will reach a certain threshold. It can also be anticipated (although it wasn't verified by concrete tests in our case because of the non availability of a large testing environment) that this threshold in *inversely proportional* to the processing power of the host. Therefore, the proposed solution is scalable because response delays *can* be improved by adding processing power to the server.

The server was running on a single processor Sun workstation, 440 MHz 64 bit-UtlraSparc 2I with 512 Mbytes of RAM, SunOS 5.7. The client was running a java application on a Windows NT 4.0 PC, K6 III/450 processor with 256 Mbytes of RAM.

7 Conclusions and Future Research

Network control will play an increasingly important role in Information Technology of the 21st century. However, element control today is inefficient and unscalable and, paradoxically, has not received enough attention from previous research. The need for a flexible, efficient and reliable solution for the problem of uniform control of heterogeneous network elements has led to the research of this thesis.

This document presents a software development framework that is intended for *straightforward implementation of any range of network element control procedures in a uniform fashion*. The proposed solution is driven by realistic concerns: time-to-market, simplicity, as well as performance and reliability.

7.1 Contributions of this Thesis

Overall, this thesis examines the problem of uniform control of heterogeneous network elements based on the theory of finite state machines.

An abstract semi-formal solution has been developed addressing different aspects of this problem. More specifically, several relevant aspects were considered explicitly, including more particularly performance, reliability and extensibility issues.

First, the *Adapter pattern* [37], incarnated in the use of Control Protocol Adapters (CPA), was considered in Chapter 4 as a generic solution to allow uniform control. CPA design was discussed in a generic fashion, i.e. without any assumption upon the control protocol or other network element-specific attributes.

Next, control automation via Command Line Interface (CLI) was considered as a solution presenting interesting characteristics. Specific issues related to CLI automation for network element control were stated. A design pattern for CLI-oriented CPAs that uses finite state machines and regular expressions was discussed, and the previously-stated issues were incrementally addressed.

A more powerful concept incarnating the *Factory Method pattern* [37], which we called Dynamic Triggering Filters, was also introduced to maximize CLI control flexibility.

Finally, UML design and implementation of a functional prototype were presented and evaluated. It was found that implementing new control procedures was particularly fast and that runtime performance results were satisfactory.

This thesis made several research contributions, which can be summarized as follows:

- The requirements for uniform control of heterogeneous network elements were presented. A generic framework that satisfies those requirements was presented.
- A new methodology, which consists in uniform CLI interaction automation using a component-based real-time approach, was used for the first time.
- The particular case of uniform CLI automation is studied extensively for the first time in Chapter 5, where the concept of "dynamic triggering filter" was also introduced.
- The technical aspects of the UML standard that are relevant to supporting the presented framework were examined. A tool that supports those aspects and extends them with real-time attributes was briefly presented as well.
- A fully functional prototype of the presented framework is constructed and a test application is produced and tested. Traditional performance measures revealed satisfactory and other criteria, such as the ease of implementing new control procedures, were evaluated and found significantly interesting.

7.2 Future Research

The development of a complete control facility, with an extended uniform interface and support for several control protocols, in beyond the scope of this thesis. Work in this

area should be continued in order to enrich this framework with more formal characteristics. This would include the following:

- Implementing the dynamic filtering module (DFM) presented in Section 5.4. In addition, investigating which facilities are most needed when parsing a network element's response via CLI.
- Extending the framework with other management protocols, such as SNMP and CMIP. This would allow augmenting the framework with *management* facilities.
- Investigating performance issues and determining the performance of different control protocols in different scenarios. By doing this, CPAs can decide which control protocol to use for each operation if they have previous knowledge of performance attributes.
- For CLI automation (at least), defining a mechanism that would allow formal or semi-formal validation of finite state machines used to control network elements' CLIs.
- Investigating other applications of the presented design patterns.

Bibliography

[1] IETF, RFC 2570, *Introduction to Version 3 of the Internet-standard Network Management Framework*, April 1999. http://www.ietf.org/rfc/rfc2570.txt

[2] IETF Internet Draft, *An Informal Management Model for Diffserv Routers*, February 2001. http://www.ietf.org/internet-drafts/draft-ietf-diffserv-model-06.txt

[3] W3C Note. *Simple Object Access Protocol (SOAP) 1.1*. May 2000. http://www.w3.org/TR/SOAP/

[4] A. Skonnard. *SOAP: The Simple Object Access Protocol*. MIND®, January 2000. http://www.microsoft.com/mind/0100/soap/soap.asp

[5] K. Scribner and M. C. Stiver. *Understanding the Simple Object Access Protocol*. Earthweb® article. http://softwaredev.earthweb.com/article/0,,10455_641321,00.html

[6] W3C Note. *Web Services Description Language (WSDL) 1.1*. http://www.w3.org/TR/wsdl

[7] M. Kirtland. *A platform for web services*. MSDN library, January 2001.

[8] C. C. Tapang. *Web Services Description Language (WSDL) Explained*. MSDN library, July 2001.

[9] Y. Shouhoud. *Introduction to WSDL*. Devxpert. http://www.devxpert.com/tutors/wsdl/wsdl.asp

[10] A. Jung. *Regular expressions and finite automata*. Lecture notes. University of Birmingham, January 2001. http://www.dubal.co.uk/Work/ModComp/handout1.pdf

[11] A. Jung. *Regular Languages.* Lecture notes. University of Birmingham, January 2001. http://www.dubal.co.uk/Work/ModComp/handout2.pdf

[12] P. E. Black. *Dictionary of Algorithms, Data Structures, and Problems*, NIST. Sep. 1998. http://www.nist.gov/dads/terms.html

[13] P. K. Anumula, A. Di Fabio, J. Zhu. CS390 Web Course Study Materials. December 2001. http://www.cs.odu.edu/~toida/nerzic/390teched/web_course.html

[14] National Institute of Standards & Technology (NIST). *Expect.* http://expect.nist.gov/

[15] GNU Software. *DejaGnu.* http://www.gnu.org/software/dejagnu/dejagnu.html

[16] A. Simpson. *Regular expressions and Kleene's theorem.* Lecture notes. University of Edinburgh, Oct. 2000. http://www.dcs.ed.ac.uk/teaching/cs2/online/Lectures/CS2Ah/LangProc/lp4.pdf

[17] Object Management Group (OMG). *Unified Modeling Language Specification, version 1.3.* June 1999. http://www.omg.org/

[18] Rational Software. *Rational Rose RealTime Online Help.* http://www.rational.com/

[19] Rational Software. *Rational Rose RealTime.* http://www.rational.com/

[20] WASP C++. *Systinet WASP C++.* 2001. http://www.systinet.com/

[21] University of Western Ontario. *The Grail+ Project.* February 2000. http://www.csd.uwo.ca/research/grail/

[22] R. K. Keller , J. Tessier , G. V. Bochmann. *A pattern system for network management interfaces.* Communications of the ACM September 1998, Volume 41 Issue 9.

[23] C. A. Joseph , A. Sherzer , K. Muralidhar. *Knowledge based fault management for OSI networks.*Proceedings of the third international conference on Industrial and engineering applications of artificial intelligence and expert systems. June 1990.

[24] T. L. Janssen. *Network expert diagnostic system for real-time control.* Proceedings of the second international conference on Industrial and engineering applications of artificial intelligence and expert systems. June 1989.

[25] W. Fuller. *Network management using expert diagnostics.* International Journal of Network Management. August 1999. Volume 9, Issue 4.

[26] U. Warrier , P. Relan , O. Berry , J. Bannister. *A network management language for OSI networks.* ACM SIGCOMM Computer Communication Review, Symposium proceedings on Communications architectures and protocols. August 1988. Volume 18, Issue 4.

[27] S. Papavassiliou. *Network and service management for wide-area electronic commerce networks.* International Journal of Network Management March 2001. Volume 11, Issue 2.

[28] J. Kurose. *Future directions in networking research.* ACM Computing Surveys (CSUR) December 1996.

[29] Ching-Wun 'Bo' Tsai , Ruay-Shiung 'Bo' Chang. *SNMP through WWW.* International Journal of Network Management. March 1998. Volume 8. Issue 2.

[30] Hong-Taek Ju , Mi-Joung Choi , James W. Hong. *An efficient and lightweight embedded Web server for Web-based network element management.* International Journal of Network Management. September 2000. Volume 10, Issue 5.

[31] N. J. Muller. *Web-accessible network management tools.* International Journal of Network Management. September 1999. Volume 7 Issue

[32] L. H. Deri. *Desktop versus web-based network management.* International Journal of Network Management. December 1999. Volume 9, Issue 6.

[33] J. A. Gutiérrez. *A connectionless approach to integrated network management.* International Journal of Network Management. July 1998. Volume 8, Issue 4.

[34] L. J. G. T. van Hemmen. *Models supporting the network management organization.* International Journal of Network Management. November 2000. Volume 10, Issue 6.

[35] Do-Hyeon Kim , You-Ze Cho. *Design and implementation of network management systems for integrated management of LANs and WANs.* International Journal of Network Management. May 2000. Volume 10, Issue 3.

[36] H. Ku, J. Forslow, J. Park. *Web-based configuration management architecture for router networks.* IEEE Symposium Record on Network Operations and Management Symposium 2000. IEEE, Piscataway, NJ, USA. p. 173-186. NOMS 2000.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1994.

[38] Baochun Li. *AGILOS: A Middleware Control Architecture for Application-Aware Quality of Service Adaptations.* Ph.D. thesis, UIUC, July 2000.

[39] J.B. de Meer. *On the Specification of EtE QoS Control.* Proceedings of the IWQoS97, May 21-23, 1997, New York

[40] M. Athans. *Optimization of Communication Processes; An Analysis.* Proceedings of the IFAC World Congress, Helsinki, July 23-30, 1997.

[41] The Strategis Group. *U.S. CLEC Financial Benchmarks.* July 2001.

[42] Goldman-McKinsey & Company. *US Communications Infrastructure at a Crossroads: Opportunities Amid the Gloom.* August 2001

[43] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P-H Ho, X. Nicolin, J. Sifakis, S. Yovine. *The Algorithmic Analysis of Hybrid Systems.* Theoretical Computer Science 138, pp. 3-34. 1995.

[44] CiscoAssure Policy Networking End to End Quality of Service- CISCO White Paper, http://www.cisco.com/warp/public/cc/pd/nemnsw/cap/tech/caqos_wp.html

[45] Ingo Busse, Bernd Deffner, Henning Schulzerine. *Dynamic QoS Control of Multimedia Applications Based on RTP.* GMD-Fokus, http://www.fokus.gmd.de/step/acontrol/ac.html

[46] J.E van der Merwe, S. Rooney, I.M. Leslie and S.A. Crosby. *The Tempest – A Practical Framework for Network Programmability.* IEEE Network. v. 12 n 3 May/June 1998. pp. 20-28.

[47] Wohlstadter, E. Jackson, S. Devanbu, P. *Generating wrappers for command line programs: The Cal-Aggie Wrap-O-Matic project.* Conference Paper. Proceedings - International Conference on Software Engineering 2001. p 243-252.

[48] R. Grosu, M. Broy, B. Selic, G. Stefanescu. *What is Behind UML-RT?* Behavioral specifications of businesses and systems, Kluwer Academic Publishers, 1999.

[49] B. Selic, G. Gullekson, P. Ward. *Real-time Object-Oriented Modeling.* Wiley. 1994.

[50] D. Harel. *Statecharts: a visual formalism for complex systems.* Science of Computer Programming. July 1987.

Appendix A: WSDL interface for the TC service

```
<?xml version="1.0"?>
<wsdl:definitions name="NE_Adapter" targetNamespace="http://kluane.genie.uottawa.ca/qos/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://idoox.com/wasp/tools/java2wsdl/output/"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/">
    <wsdl:types>
        <xsd:schema targetNamespace="http://kluane.genie.uottawa.ca/qos/">
            <xsd:complexType name="Filter">
                <xsd:sequence>
                    <xsd:element name="destAddress" type="xsd:string"/>
                    <xsd:element name="filterType" type="xsd:int"/>
                    <xsd:element name="protocolType" type="xsd:int"/>
                    <xsd:element name="tos" type="xsd:short"/>
                    <xsd:element name="srcAddress" type="xsd:string"/>
                    <xsd:element name="destMask" type="xsd:string"/>
                    <xsd:element name="srcMask" type="xsd:string"/>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="Meter">
                <xsd:sequence>
                    <xsd:element name="conformingAction" type="xsd:string"/>
                    <xsd:element name="meterType" type="xsd:int"/>
                    <xsd:element name="nonConformingAction" type="xsd:string"/>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="Direction">
                <xsd:sequence>
                    <xsd:element name="direction" type="xsd:int"/>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="Port">
                <xsd:sequence>
                    <xsd:element name="port" type="xsd:string"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:schema>
    </wsdl:types>
    <wsdl:message name="NE_Adapter_createTC_Request">
        <wsdl:part name="p0" type="xsd:string"/>
        <wsdl:part name="p1" type="tns:Filter"/>
        <wsdl:part name="p2" type="tns:Meter"/>
        <wsdl:part name="p3" type="tns:Direction"/>
```

```
        <wsdl:part name="p4" type="tns:Port"/>
    </wsdl:message>
    <wsdl:message name="NE_Adapter_deleteTC_Request">
        <wsdl:part name="p0" type="xsd:string"/>
        <wsdl:part name="p1" type="tns:Filter"/>
        <wsdl:part name="p2" type="tns:Meter"/>
        <wsdl:part name="p3" type="tns:Direction"/>
        <wsdl:part name="p4" type="tns:Port"/>
    </wsdl:message>
    <wsdl:message name="NE_Adapter_deleteTC_Response">
        <wsdl:part name="response" type="xsd:int"/>
    </wsdl:message>
    <wsdl:message name="NE_Adapter_createTC_Response">
        <wsdl:part name="response" type="xsd:int"/>
    </wsdl:message>
    <wsdl:portType name="NE_Adapter">
        <wsdl:operation name="createTC" parameterOrder="p0 p1 p2 p3 p4">
            <wsdl:input name="createTC" message="tns:NE_Adapter_createTC_Request"/>
            <wsdl:output name="createTC" message="tns:NE_Adapter_createTC_Response"/>
        </wsdl:operation>
        <wsdl:operation name="deleteTC" parameterOrder="p0 p1 p2 p3 p4">
            <wsdl:input name="deleteTC" message="tns:NE_Adapter_deleteTC_Request"/>
            <wsdl:output name="deleteTC" message="tns:NE_Adapter_deleteTC_Response"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="NE_Adapter" type="tns:NE_Adapter">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
        <wsdl:operation name="createTC">
            <soap:operation soapAction="" style="rpc"/>
            <wsdl:input name="createTC">
                <soap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://kluane.genie.uottawa.ca/qos/NE_Adapter"/>
            </wsdl:input>
            <wsdl:output name="createTC">
                <soap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://kluane.genie.uottawa.ca/qos/NE_Adapter"/>
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="deleteTC">
            <soap:operation soapAction="" style="rpc"/>
            <wsdl:input name="deleteTC">
                <soap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://kluane.genie.uottawa.ca/qos/NE_Adapter"/>
            </wsdl:input>
            <wsdl:output name="deleteTC">
                <soap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://kluane.genie.uottawa.ca/qos/NE_Adapter"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="JavaService">
        <wsdl:port name="NE_Adapter" binding="tns:NE_Adapter">
            <soap:address location="http://kluane.genie.uottawa.ca/qos/tc/"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```