Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

João André Martins (26464)

Lisboa
(2009)

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

SmART: An Application Reconfiguration Framework

Joao Andre Martins (26464)

Orientador: Prof. Doutor Hervé Paulino
Co-orientador: Prof. Doutor João Lourenço

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para a obtensão do Grau de Mestre em Engenharia Informática.

Lisboa
(2009)

To my family and friends, who supported me through the best and worst.

Summary

Among the information technology sectors is the virtualization sector, which stands out by being a very active area, with many collaborations and advances. Virtualization's benefits are plenty. Some of them are the massive server consolidation, reduced cooling, structural and electrical costs and the isolation between virtual machines.

Virtual appliance is a concept resulting from the advancements on virtualization, and defines an alternative software distribution model from those existing by then, such as the traditional CD-ROM distribution, software as a service and hardware appliances. This model allows customers to have their IT solution fully specified and optimized for the task it must perform.

With the virtual appliance outbreak, many servers using different applications tend to gather on the same physical machine. The applications, in turn, may implement plenty configuration formats, from widely adopted standards to proprietary formats. The configuration of these applications is typically carried by busy system administrators, who must dedicate some of their time coping such a vast configuration format range.

This thesis contributes for the nourishing of the virtual appliance concept and proposes the creation of a tool which automatically configures applications inside virtual appliances, regardless of the application being configured. The contributions on this area are, therefore, very rare, which elevates this dissertation to a pioneer of the area.

The approach for the problem of automatic application configuration explores the frequently found patterns in configuration files. Typically, those files use similar concepts, such as parameter definitions, parameter blocks and comments. Besides this, only some files were found to implement other concepts, although on a punctual basis. This dissertation proposes a framework for the automatic configuration of applications based on this characteristic. A configuration file is transformed to a structured and application-independent language, like the eXtended Markup Language, which is then modified and reverted to its original syntax.

Keywords: Virtual appliance, automatic application configuration

Sumário

Entre os sectores de tecnologias de informação, o sector da virtualização distingue-se por ser uma área muito activa, que conta com muitas colaborações e desenvolvimentos. As utilidades da virtualização são inúmeras. Entre elas estão a consolidação maciça de servidores, redução nos custos de arrefecimento, estruturais e de electricidade e o isolamento entre máquinas virtuais.

Fruto dos avanços da virtualização é o conceito de *virtual appliance*, que define um modelo de distribuição de *software* alternativo aos até então existentes, tais como distribuição tradicional (CD-ROM), *software as a service* e *hardware appliances*. Este modelo permite aos clientes terem a sua solução informática específica e optimizada para a tarefa que se propõe desempenhar.

Com o surgimento das *virtual appliances*, muitos servidores que recorrem a diversas aplicações tendem a juntar-se em máquinas físicas. As aplicações, por sua vez, podem implementar vários formatos de configuração, desde padrões amplamente adoptados a formatos de proprietário. A configuração destas aplicações é tipicamente feita por administradores de sistemas ocupados, que têm assim necessidade de canalisar o seu tempo para lidar com a variedade de formatos de configuração.

Esta dissertação vai no encontro dos avanços no ramo das *virtual appliances*, propondo a criação de uma ferramenta que faça a configuração automática de aplicações, independentemente da aplicação em causa. São muito poucos os contributos nesta área, pelo que esta dissertação pode ser considerada pioneira na área.

A abordagem para o problema da configuração automática de aplicações explora os padrões recorrentes em ficheiros de configuração de aplicações. Tipicamente, estes ficheiros implementam conceitos semelhantes, como definições de parâmetros, blocos de parâmetros e comentários, existindo ainda outros ficheiros que implementam padrões próprios, tratando-se, no entanto, de casos pontuais. Esta dissertação propõe uma *framework* baseada nesta característica, onde um ficheiro de configuração é transformado para uma linguagem estruturada e independente da aplicação, como a eXtended Markup Language (XML), e que depois de alterado é revertido na sua síntaxe original.

Palavras-chave: Virtual appliance, configuração automática de aplicações

Contents

List of Figures

List of Tables

List of Listings

1

Introduction

This dissertation presents a framework for the automatic configuration of applications in virtual appliances. This framework, entitled Smart Application Reconfiguration Tool (SmART), configures virtually any kind of application by transforming the application's configuration files into a generic structure, dettached from the configuration file original syntax. This abstraction allows for a configurator to traverse configuration files in original syntax from any application and modify them to reflect a desired behaviour on the target application. As soon as the configurator manages to apply the changes, the information in the configuration files on generic syntax is used to reproduce them in their original syntax. This framework is very innovative on the grounds that it approaches the area of automatic application configuration, an area which benefits from few contributions and currently does not offer many solutions for that endeavour.

This chapter is structured on the following way: Section 1.1 explains the motivation for the resolution of the problem, Section 1.2 defines the problem to be solved, Section 1.3 reveals how that problem can be solved, Section 1.4 enumerates the contributions of this dissertation for the problem solution and Section 1.5 presents the organization of the remainder of this dissertation.

1.1 Motivation

The virtualization area is a very hot and fast-moving area of Information Technology (IT). Although virtualization is not an entirely new concept, only recently it has been noticed, developed and studied carefully and in depth, so it is prone to more advancements and should be considered as a mainstream software deployment model for the future [VMwb].

Despite the term and ideas of virtualization being quite old (1960's), only recently it has been given increased attention [Sin04]. Companies which recur to IT services are increasingly aware of virtualization due to the crucial changes it brings along. Virtualization makes it possible to gather several underutilized servers into fewer ones (possibly just a single server) by virtualizing the hardware where each server runs. This reduces not only the companies' electrical bills, but also cooling, space and hardware costs. Virtualization brings along other benefits such as the fast replication and/or migration of virtual machines (VMs), which allows for high availability of the services provided by the VMs.

But not only the final users of virtualization woke up to the trend. Some IT giants such as Intel, AMD and Microsoft already entered the virtualization market, with different solutions such as hardware-assisted virtualization or hypervisor based virtualization. This huge trend also got into the consumers market, with popular solutions such as VMware Workstation or Microsoft Virtual PC, and even in the gaming market. Anyone who runs games on MAME or WINE is using virtualization technology. Currently, there are a lot of virtualization solutions in the market. These include VMware [VMwa], XenSource [Cit], Microsoft [Mica], Sun [Micc], Parallels [Par] products, and so on.

With the advancement of virtualization technology, the new Virtual Appliance (VA) concept emerged. VAs came to change the way of looking to software deployment. Unlike the existing software deployment models, presented ahead in Section 2.1, such as the traditional CD-ROM way, Software as a Service or Hardware Appliances, VAs allow the costumers to have their own IT platform, fully optimized for the task being and completely under their domain. The blending of virtualization benefits with VAs' performance is bringing software deployment to a new level.

This shift in the IT sector motivated project VIRTU [Sola] which is a collective effort by the consortium composed of Evolve Space Solutions, Universidade Nova de Lisboa, Universidade de Coimbra, HP Labs and the European Space Agency. The project's goal is to develop an open-source vendor-independent virtualization platform, including support for the management and configuration of applications.

There are currently two emerging opportunities on the IT market. On one side, virtualization has changed the way in which companies deploy software, on the other hand, application configuration is still mostly a manual task. VIRTU exploits these opportunities by defining a virtualization tool which manages application stacks. In other words, VIRTU allows for the creation of optimized virtual machines assembled with existing applications which can be automatically configured.

VIRTU project's use cases range from software testing in complex distributed systems, including the European Space Operations Center's Mission Control System (MCS) and Ground System Tests and Validation infrastructure (GSTVi), to IT infrastructural management, like Portugal Telecom's Application Service Provider or Novabase's Enterprise Applications Environment. VIRTU also makes it possible for a customer to have its solution self-provisioned within his domain, but also to maintain different application versions to achieve legacy compatibility.

The advent of virtual appliances makes it possible to gather a great amount of applications on the same machine. This results in many different ways to represent configurations, which, in turn, makes the management of those configurations complicated and time-consuming.

This dissertation proposes the creation of a tool, entitled Smart Application Reconfiguration Tool (SmART), that is able to configure any application, regardless of how it represents its configuration.

1.2 Problem Description

The problem of automatic application configuration is caused by software applications which employ different means of representing their configurations. In some cases, it was noticed that the same application implemented different configuration formats from one version to another. This lack of standardization reveals that the configuration aspects of the applications are put in the backburner by application developers. Currently, the solutions on the field of automatic configuration of applications are seldom. This work goes in a way as to explore the mentioned opportunity.

VAs may have different formats and may contain all sorts of applications, so it would be very helpful if a tool could configure any kind of application inside a VA. For this matter, it is essential that the tool interprets each configuration file independently of the application being configured, rather than having a limited set of known configurations and trying to match them with the configuration file.

Besides the differences in the representation of configurations between two disparate applications, once can also identify divergent configurations in the same applications. Typically, whenever an application is upgraded, so are its configuration files, even if the differences in both situations come down to one or two extra lines containing new parametrizable elements than before. This motivates the abstraction from the configuration representation, which will allow any application to be configured based on what settings it defines, not how they are defined.

Two ways to configure applications in VAs can be identified. The first sets up the configuration files before the application is installed on the VA, whereas the second searches for the configuration files in the VA and changes them. Given the fact that the we are dealing with text configuration files which need to be read and parsed in both approaches, applications whose configuration files are in the binary format may require special parsers. This is often the case in closed-source applications.

An example of the typical usage of this tool is the case where many VAs containing the same applications (e.g., webserver) need to be deployed on an intranet. Although these VAs have the same software, their configurations must be different (e.g., machine name, IP address, etc.). Following is a brief description of both approaches and the way in which each approach deals with the example case.

1.2.1 Pre-installation Configuration

Before the application is installed on a VA, there are two elements: the application installation package and the VA. In order to configure an application prior to its installation, the configuration files must be read, parsed and changed, to reflect the desired settings, outside the VA. After the configuration file is properly modified, the application package must be reconstructed to be ready for installation.

This approach has two significant limitations. First, whenever a different configuration for another VA is required, a new package containing that configuration must be built. This results in the proliferation of packages for the same applications, differing only on their configuration files.

Also, since this approach deals with uninstalled packages, it does not allow the same VA to be configured more than once. If an application is to be configured again, the process goes back to the beginning and the application has to be re-installed.

1.2.2 Post-installation Configuration

On the post-installation configuration, the configuration file is retrieved from an already built VA. The file is then parsed, changed outside the VA and will later replace the older version inside the VA. Getting the configuration file from inside the VA may be the toughest part of the process because it requires knowledge of the VA format, as well as the software inside it. For instance, in Fedora Core Linux distributions the Apache HTTP Server configuration files are usually stored in the */etc/httpd/* directory whereas in Ubuntu Linux distributions they are located at */etc/apache2/*.

In the example case, only one template VA, containing the applications, must be built. It is then replicated enough times to match the desired number of VAs. After all VAs are created, their applications can be configured with the tool any number of times without the need to reinstall the application. This clears the need of having to proceed with multiple installations.

1.2.3 Application Configuration in General

One way to treat the configuration of different applications equally is to look for similarities in their structure. The fact is that the majority of configuration files are represented similarly. Although the formats or the languages of the configuration files tend to differ, different configuration files end up by implementing the same concepts, or patterns. The majority, if not all configuration files, use the notion of *parameter* to represent the setting of a value to an application variable.

In order to determine the patterns that configuration files are likely to implement, the applications of some VIRTU project usage scenarios were analysed. These applications include:

- Apache
- Eclipse
- MySQL
- PostgreSQL

Listing 1.1: Apache configuration file excerpt

```
# ServerRoot: The top of the directory tree under which the server's
# configuration, error, and log files are kept.
ServerRoot "@@ServerRoot@@"

Allow from all

<IfModule !mpm_netware_module>
<IfModule !mpm_winnt_module>
#
# If you wish httpd to run as a different user or group, you must run
# httpd as root initially and it will switch.
User daemon
Group daemon
</IfModule>
</IfModule>

!includedir /etc/mysql/conf.d/
```

- Mantis

Following are presented some configuration file excerpts of the presented applications. Each excerpt is analysed to identify the commonly found patterns in each of them.

Apache

The patterns in the Apache HTTP Server [Foua] configuration files (Listing 1.1) are *parameters* formed as *id - separator - value*, where the separator is a space character and the value may have multiple instances; *comments*, which are text lines starting with the '#' character; and *blocks*, which are sets of patterns delimited by a header and a footer. Besides these patterns, there are also *special* instructions, which start with a '!' character, followed by a command key and further arguments.

Eclipse

The Eclipse [Foub] configuration files (Listing 1.2) are in a format similar to the eXtensible Markup Language (XML) format [Con]. It mainly contains *elements* and *blocks*. The *blocks* may be empty or may contain other *blocks*. In case the *blocks* are empty, they are represented solely by a tag, whereas if a *block* contains other *blocks*, it is delimited by a starting and an ending tag. Additionally, a *block* may contain attributes, corresponding to *parameters*.

Being XML, it also supports comments, of the form `<!--this is a comment-->`.

Listing 1.2: Eclipse configuration file excerpt

```
<workbenchAdvisor/>
<fastViewData fastViewLocation="1024">
<orientation view="org.eclipse.ui.views.ContentOutline"
position="512"/>
</fastViewData>
```

Listing 1.3: MySQL configuration file excerpt

```
[mysqld]
#
# * Basic Settings
#

user            = mysql
pid-file        = /var/run/mysqld/mysqld.pid
socket          = /var/run/mysqld/mysqld.sock
port            = 3306
```

MySQL

The MySQL [Micb] configuration files (Listing 1.3) are in a format similar to the INI file format (ahead in Section 1.3). A MySQL configuration file implements *parameters* formed as *id - separator - value*, where the separator is the '=' character; *comments* which are text lines starting with the '#' character; and *blocks* identified by an initial header and which may contain multiple parameters and/or comments.

PostgreSQL

The PostgreSQL [Grob] configuration files (Listing 1.4) are an amalgam of *parameters* and *comments*. A *parameter* is formed as *id - separator - value*, where the separator is the '=' character and the value may be a keyword or a string delimited by the apostrophe character as the value. A *comment* may appear anywhere in the file and is identified by the '#' character. Unlike the previous examples, the PostgreSQL configuration files do not implement the *block* pattern, although there seems to be a followed convention in which a set of *parameters* is preceded by a comment, which is similar to a *block*.

Mantis

The Mantis PHP Bug Tracker [Groa] configuration file (Listing 1.5) implements *parameters* of the form *$ - key - separator - value*, where the separator is the '=' character and the value is a keyword or a string delimited by apostrophes or quotes. It also implements *comments*, which are text lines that start with the '#' character.

Listing 1.4: PostgreSQL configuration file excerpt

```
#------------------------------------------------
# CLIENT CONNECTION DEFAULTS
#------------------------------------------------

datestyle = 'iso, mdy'
timezone = unknown                     # actually, defaults to TZ
#timezone_abbreviations = 'Default'
```

Listing 1.5: Mantis configuration file excerpt

```
# --- database variables ---------

# set these values to match your setup
$g_hostname      = "localhost";
$g_db_username   = "mantisdbuser";

$g_webmaster_email       = 'webmaster@example.com';

$g_allow_file_upload     = ON;
```

1.2.4 Conclusion

The analysis carried to this set of applications allows us to conclude, with a reasonable degree of certainty, that the patterns passible to be found on the average configuration file are reduced to the following:

Parameters
This pattern basically consists on a pair of *id*, *list of values*, separated by a *separator* symbol. Although on the majority of applications, a *parameter* is simply structured as *id - separator - value*, there are some which associate many *values*, separated by *separators* to an *id*.

Blocks
This pattern is composed of a *header* delimiter, followed by other patterns, finally delimited by a *footer*. On some applications (e.g., MySQL), blocks are only delimited by an initial *header*.

Comments
This pattern consists on a line which starts with a reserved *comment symbol*, followed by any *text content*, which may be void.

The three presented patterns define the basis of the majority of applications. Nonetheless, a special case of a pattern was found in the Apache application, where a *special* instruction, composed of a reserved *instruction symbol* and subsequent *keywords*, defines a pattern, very much alike the *comment* pattern, but not the same. Since this pattern was only found on the referred application, we must not expect that every application strictly implements the presented patterns, but may rather implements patterns of its own.

1.3 Problem Approach

Section 1.2 identified two ways to configure applications in virtual appliances: pre and post-installation. The pre-installation approach has no outstanding requirement and operates by generating as many application packages as different required configurations for that application, and does not allow for the re-configuration of an application in a practical fashion (i.e., without resorting to that application re-installation). On the other hand, the post-installation approach requires previous knowledge of the software running inside the VA but, contrary to the previous approach, each different configuration only requires a new configuration file version, and allows for the practical re-configuration of an application. Given the benefits and drawbacks of each approach, the post-installation approach requirement was found tolerable and moreover, allows for better functionality than the pre-installation approach and, therefore, SmART will follow this discipline.

The adopted approach operates by conducting a lexical analysis on an application's configuration files with the aim of extracting the relevant information for that application configuration from them. This raises a significant requirement which says that the configuration files must be in text format. A consequent implication of this requirement is that special formats, like the Windows OS registry or binary files, may not be configurable using the approach. However, there exist dedicated parsers for the treatment of those formats and these can be used by the framework to provide support for the mentioned kinds of formats.

Still in the scope of the applications supported by SmART, Section 1.2.3, the VIRTU project usage scenarios analysis concluded that overall, configuration files either follow three formats:

- *Apache-like format*;
- *INI-like format*;
- *XML-like format*.

The *Apache-like* format is characterized by defining *parameters* as key/value pairs separated by spaces or a similar character, *blocks* delimited by a header and a footer, containing other patterns, and *comments*.

The *INI-like* format defines *parameters* as key/value pairs separated by an equal, space or a similar character, *blocks* delimited by a header and comments.

The *XML-like* format defines *blocks* as XML elements delimited by a header and a footer, if they contain other *blocks*, or a single tag if they do not contain other *blocks*, *parameters* as XML attributes contained in an XML tag, and *comments*.

SmART should provide built-in support for these three configuration file categories, seen as most of the configuration files fit in one of these categories. However, the tool must also be extensible to support new configuration file formats.

Finally, one way of tackling this problem is to reduce the configuration files of any application to a generic structure, dettached from its application specifics, which contains the relevant elements for its application configuration. This structure can then be modified without any knowledge of the application to which it belongs. Once altered, that structure is again converted to a configuration file in its original syntax, containing the applied modifications.

1.4 Contributions

This dissertation contributions for the automatic application configuration are:

- Identification of commonalities amidst configuration files structures;
- Proposal of a framework for the rebuilding of configuration files of an application;
- Implementation of the proposed framework;
- Provision of the VIRTU project with a state-of-the-art and innovative tool for application reconfiguration, independent of its vendor.

1.5 Document Organization

The rest of the dissertation is structured as follows: the next chapter covers the state of the art of the related subjects to this thesis, Chapter 3 describes the proposed architecture of the SmART tool, Chapter 4 explains the tool implementation decisions, Chapter 5 provides the validation process carried to the tool, Chapter 6 approaches the integration of SmART with the VIRTU virtualization platform, and Chapter 7 reveals the final considerations about this thesis.

2

State of the Art and Related Work

This section aims at presenting the state of the art in automatic application configuration, as well as studying the tools which may be used to tackle the problem identified in Chapter 1.

To better understand the outlines of the problem, this section starts by explaining the context of this dissertation. Namely, the virtual appliances theme is explored with the purpose of making it more natural.

Afterwards, the existing techniques to deal with the problem are analysed. There is, nevertheless, a particularity regarding this part. As previsouly mentioned in Chapter 1, the configuration of applications is still mostly a manual task. As such, the existing references considerably relevant to this theme are very few. Therefore, this dissertation consists on one of the first contributions for the problem of automatic configuration of applications.

Finally, the adopted concepts and tools to mitigate the problem are analysed. The automatic file recognition theme is studied by identifying two parser generators and describing them. Finally, it is seen how the file can be represented in an abstract, independent from its original language, way.

2.1 Virtual Appliances

A Virtual Appliance (VA) is a minimal virtual machine (VM, Section 2.1.1) composed of pre-configured and pre-installed applications plus an optimized operating system called Just Enough Operating System (JeOS) (Figure 2.1). VAs are normally created to perform a specific task in the most effective way, therefore they only contain the essential and necessary resources for the execution of that task, contrary to the regular VM where all of the kernel OS features are present, even those whose use is seldom.

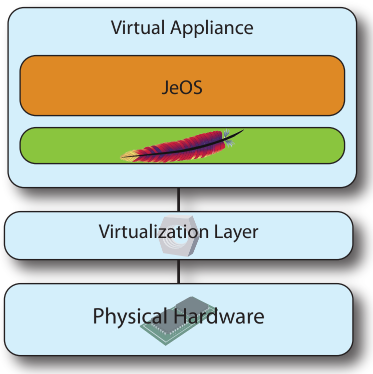

Figure 2.1: Virtual appliance

2.1.1 Virtualization

Virtualization is a very broad term and refers to something which does not exist *physically* but appears to exist. Think of it in the context of *virtual reality*. With virtual memory, for example, computer software gains access to more memory than is physically installed, via the background swapping of data to disk storage. Similarly, virtualization techniques can be applied to other IT infrastructure layers such as networks, storage, laptop or server hardware, operating systems and applications.

The concept of platform virtualization refers to the technique for hiding the physical characteristics of computing resources from the way in which other systems, applications or end users interact with those resources. In other words, virtualization allows multiple logical computing units to run on a single physical computing unit, like in the Figure 2.2. This is done with the aid of a software layer (alias virtualization layer, virtualization manager, virtual machine monitor or hypervisor) which provides the illusion of a "real" machine to multiple instances of "virtual machines".

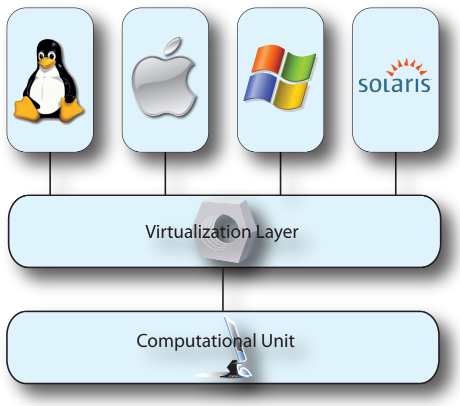

Figure 2.2: Physical unit running four VMs

The virtualization concept goes back to the 1960s when corporate mainframes were in a severe underutilization state [oV08]. By then, some IBM research personnel were working on the CP-40 OS designed to run on the System/360 time-sharing system. It was the first OS to completely virtualize a system (namely System/360) so that other S/360 OSes could be tested and executed. This system paved the way for virtualization as known nowadays and concepts pioneered by CP-40 are still actual, like the case where an abstraction layer catches traps and simulates them. Curiously, CP-40 is still supported by IBM mainframes (System z10). Presently, mainframes came to suffer from the same underutilization problem and so virtualization is *en vogue* once again.

2.1.2 Motivation

Nowadays, deployment via download or DVD defines the typical software distribution model. Despite its simplicity, this model features limitations such as deployment and configuration complexity. Other flaws that might be pointed are the huge number of application versions to maintain, which leads to IT resources draining; having to cope with a myriad of hardware platforms, which leads to intensive time-consuming testings; Independent Service Vendors (ISVs) might see their revenues decreased since if the customer is content with the product, he will only upgrade when a groundbreaking feature comes out; and so forth.

As a response to these challenges, some alternative deployment models arose. Such is the case of Software as a Service (SaaS), hardware appliances and virtual appliances.

Software as a Service (SaaS)

This approach consists on providing a service by hosting a software application that customers over the Internet can use. In this way, the need for install and configure on the customer's computer is eliminated, so problems as software maintenance and support are no longer customer's responsibility but software vendor. The revenue becomes different: customers pay when they want to use the service, rather than having a single expense when purchasing the application.

However, this will turn the application vendor into a service provider that has to maintain its infrastructure. Also, given the variety of existing open-source solutions and the inexpensive hardware, it appeals economically to the customer to keep the application running under its control.

Hardware Appliances

Another approach to software distribution is the creation of hardware appliances bundling the application with hardware capable of running it. The customer just needs to plug in the appliance and turn it on. This approach also clears the need for the customer to install and configure the application, but this forces the application vendor to enter a different market, the hardware market.

Virtual Appliances

The existing hardware virtualization techniques allow the leveraging of the hardware appliances advantages without the hardware requirement. The virtual appliance vendor packs in a set of required applications with a JeOS on a virtual machine that can be executed on the vast majority of existing hypervisors given the appearance of virtualization format standards (OVF). The deployment of the appliance is simple and the appliance comes pre-configured. The appliance vendors are able to customize the applications and guest OS as much as they want, thus leading to unseen efficiency. Updates are vendor's responsibility and just require him to address the flaws identified in the former version, produce an upgraded VA and deploy it in the customer's hardware. Management is made dramatically simpler by the use of management tools created by the appliance vendors [Sta07]. Moreover, VAs supply customers with all the benefits of virtualization: increased hardware use, reduction of physical resources, etc.

2.1.3 Open Virtual Machine Format

The Open Virtual Machine Format (OVF) is a joint effort of various virtualization solution vendors such as XenSource, VMware and Microsoft as a part of the Distributed Management Task Force Inc. It consists on an open, secure, portable, efficient and extensible format for packaging and distribution of VMs [VX]. With OVF, VMs which are created on one platform can also be managed and used on other platforms, providing platform independence, although it enables platform-specific enhancements to be captured.

OVF differs from other formats such as VMware's VMDK and Microsoft's VHD since these are run-time VM images. Although they are currently used for VM transportation, they do not address problems such as multi-tiered VMs consisting of multiple independent VMs, or VMs with multiple disks.

Another important aspect in the scope of OVF is its integrity. OVF was designed so that the VM configures itself, without the need for the virtualization platform where it is installed to recognize the VM's file system. This is particularly useful since it allows VMs to run on any OS and virtualization platform which supports the OVF format.

OVF is extremely useful in the Virtual Appliance case since it provides essential foundations for VA's such as efficiency, portability, ease of use and distribution, etc.

An OVF package can be stored as a single file using the TAR format under the .ova extension (open virtual appliance or application). It consists of the following [DMT08]:

- one OVF descriptor file (descriptor file or .ovf file) containing the package metadata and its contents;

- zero or one OVF manifest file (manifest file or .mf file) containing the SHA-1 digests of individual files in the package;

- zero or one certification file (certification file or .cert file) containing the OVF package signature digest along with the base64-encoded X.509 certificate.

- zero or more disk images files
- zero or more additional resource files, such as ISO images

OVF specification v1.0.0 has been released on September, 2008 and is prone to intensive development.

2.1.4 Virtual Appliances in Real Life

A good example of an application of virtual appliances is the Amazon Elastic Compute Cloud [Ama], also known as Amazon EC2. It is a service which allows customers to rent hosted computing capacity in order to run their applications. The customers can create a VA comprised by applications, libraries, data and associated configuration settings or use a preconstructed image and then upload it in the Amazon Machine Image format to the Amazon storage service, Amazon Simple Storage Service (S3). VAs provide the means for elastic computing by enabling a fast and efficient way for service expansion.

Whereas previously SaaS solutions failed to succeed due to the lack of control of the service from the customer and the maintenance costs involved, Amazon EC2 uses multiple Xen hypervisor nodes so that customers can build VAs containing their applications and host them outside their domain. In this way, the vendor is able to provide the service desired by the customer without the need to maintain the solution, focusing on keeping a secure, reliable, efficient and inexpensive environment for the VAs to run on. On the other hand, the customers now have the control over their services and may update them any time necessary without having to wait for the vendor. Furthermore, customers are only billed for what they consume (i.e., instance uptime or data transferred), opposed to paying an established fee even if the service usage is low or non-existant.

2.2 Automatic Configuration of Applications

To the best of our knowledge, our approach is the first to exploit the similarities among configuration files to allow for automatic, vendor-independent and on-the-fly application reconfiguration. Similar existing projects, such as AutoBash [SAF07] or Chronus [WCG04], take on automatic application configuration as a way to assist the removal of configuration bugs. AutoBash employs the causality support within the Linux kernel to track and understand the actions performed by a user on an application and then recurs to a speculative execution to rollback a process, if it moved from a correct to an incorrect state. Chronus, on the other hand, uses a virtual machine monitor to implement rollback, at the expense of an entire computer system. It also focuses a more limited problem: finding the exact moment when an application ceased to work properly.

Two other projects, better related to this work, are Thin Crust [Cru] and SmartFrog [GGL$^+$09]. Both projects aim at automatic application configuration, but take a different approach than ours. Following is a brief summary of both.

2.2.1 Thin Crust

Thin Crust is an open-source set of tools and meta-data for the creation of VAs. It features three key components: Appliance Operating System (AOS), Appliance Creation Tool (ACT) and Appliance Configuration Engine (ACE). The AOS is a minimal OS built from a Fedora Kickstart file [1], which can be cut down to just the required packages to run an appliance. The user may download a ready-to-run AOS image or may create his own through the ACT. The ACE is ran at VA boot time and loads the appliance recipe. The appliance recipe contains VA metadata and the modules used by the VA. If it does not match the appliance configuration, the latter is changed to reflect the changes. Finally, Thin Crust supports VMware, KVM and EC2 (see Section 2.1.4) by providing conversion tools from and to the mentioned formats.

2.2.2 SmartFrog

SmartFrog (Smart Framework for Object Groups) is a framework for the creation of configuration-based systems. Its objective is to make the design, deployment and management of distributed component-based systems simpler and more robust. It defines a language to describe component configurations and a runtime environment to activate and manage those components.

One of the major causes for the problems in large distributed systems design is identified by SmartFrog as the *ad-hoc* way in which such systems are designed. This results in application configuration data scattered by the system, often causing its repetition.

In SmartFrog, there is the notion of a SmartFrog *component* and a SmartFrog *system*, which is composed by various such components. A SmartFrog *component* is constituted by three parts: the lifecycle manager, which applies a configuration on a component; configuration data, which defines the behaviour of a component; and the managed component, which may be a piece of software implementing any functionality-specific interface. Each component is monitorized by its lifecycle manager and must report any failure to the other components of the system. The components can also be distributed on a network.

2.3 Automatic Recognition of Configuration Files

The lexical analysis of the file with the objective of producing that same file in a different structure provides a way for the tool to abstract itself from each configuration file language specifics, refining the configuration file down to just the essential for configuration (i.e., the elements needed to modify the application settings, such as parameter keys and attributes). The lexical analysis of a file is accomplished by a parser, which recognizes the file original syntax and produces a version of the same file in a different structure.

Configuration file parsers are produced by grammar compilers (or parser generators). A grammar compiler reads a grammar that recognizes a certain configuration file language and

[1]

compiles it, resulting in the generation of a parser for a specific configuration file language. Following are presented some parser generators:

2.3.1 SableCC

SableCC [GH98] is an object-oriented framework that generates compilers, or parsers, in the Java programming language. This framework is based on two fundamental design decisions. Firstly, the framework uses object-oriented techniques to automatically build a strictly-typed abstract syntax tree that matches the grammar of the compiled language and simplifies debugging. Secondly, the framework generates tree-walker classes using an extended version of the visitor design pattern which enables the implementation of actions on the nodes of the abstract syntax tree using inheritance.

SableCC specification files do not contain any action code. Instead, SableCC generates an object-oriented framework in which actions can be added by defining new classes containing the action code. This leads to a shorter development cycle, allowing for the rapid prototyping of SmART.

Given a grammar, SableCC's produces a set of packages, namely:

- *analysis*: contains different tree walkers (e.g., depth first, reversed);
- *lexer*: contains the elements needed for the lexical analysis of the grammar;
- *node*: classes containing node representations;
- *parser*: contains the parser class.

None of the automatically produced classes ought to be modified. Instead, all of the user generated code should be put in classes extending those created by SableCC. By doing this, every time a problem arises (e.g., error in a grammar), it can be easily located and corrected by modifying only a small portion of the code. Another aspect of discussible importance is that SableCC is distributed under the GNU Lesser General Public License.

Notwithstanding, SableCC also reveals some significant shortcomings. SableCC does not support error productions. In other words, when a generated parser comes upon an unrecognizable token in a configuration file, it does not allow the parse to advance further beyond that token, consequently halting the parse on that position. Since error productions are a common feature on parser generators, this lackage was not noticed upon the choice. This, in turn, severed the ability to produce multiple recognized file blocks to just one. The parsing statistics also become unreliable, since a parser might not recognize the first token a configuration file and recognize the rest of the file, but without error productions, parsing will inevitably stop in the beginning of the file and that parser will be reported to have parsed 0% of the file, when in fact it would manage to parse nearly 100% of the file.

2.3.2 JavaCC

The Java Compiler Compiler (JavaCC) [Proa] is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building, actions, debugging, etc.

JavaCC generates top-down parsers (i.e., start at the root of the derivation tree, picks a production and tries to match the input tokens [oM]), declares the lexical and grammar specifications in one file, making grammars easier to read and maintain, supports semantic lookahead, uses the Extended Backus-Naur Form (i.e., operators *, + and ?) and supports error recovery in two forms: shallow recovery and deep recovery. In shallow recovery, if an input token does not match any production, it is possible to move to the next desired token (e.g., semicolon). Deep recovery is similar to the shallow recovery, but also allows to recover from an error inside a production, which is impossible resorting to shallow recovery alone.

2.4 Representation of Configuration Files in a Generic Syntax

The concept of generic syntax refers to a language-independent syntax used in the process of automatic application configuration. The generic syntax is used to abstract the configuration files from their original languages. When a configuration file is represented in a generic syntax, another independent entity which recognizes that syntax is able to manipulate the file without having knowledge of what the original syntax was, or what application uses that file. The chosen generic syntax must provide a practical way to traverse and manipulate files. Furthermore, it must be widely supported and well documented. Its software license must also allow it to be used freely.

Considering the prievous requirements, the chosen syntax was the eXtensible Markup Language (XML) since it is a broadly adopted and supported standard. XML allows for the structuring of configuration files in a way that makes them easily manipulable and traversable.

Some available implementations of XML parsers were identified in many programming languages. The criteria to choose from an XML parser implementation includes factors such as the programming languages it is available on and what it allows the user to do resulting data structure from the parse.

Since the programming language used to develop the tool is going to be Java, two major XML parser implementations in Java were considered: the Simple API for XML (SAX) and the Domain Object Model (DOM). Following is a brief description of both.

Simple API for XML

The Simple API for XML (SAX) is an Application Programming Interface (API) for XML in Java, although it is available in many other programming languages. It is event-driven: it reads from a stream and reacts when processing instructions, elements, comments and text.

Being event-oriented, SAX does not implement a representation of the document in memory and, therefore, memory accesses are seldom in quantity. This leads to some memory independence and fairly good memory consumption ratings, also excelling in large files parsing. SAX parses in an uni-directional way since it does not allow previously read data to be read again, without restarting the parser. This consists on a problem when a whole XML document must be in memory in order to be properly parsed, as SAX requires the user to start the parsing again to get values previously obtained.

Domain Object Model

DOM is a cross-platform and language-independent interface for creating and manipulating XML documents in memory. It is a World Wide Web Consortium (W3C) standard.

It loads an XML document to memory and allows for dynamic access to it, allowing for bi-directional parsing. DOM represents an XML document in memory in a tree-like structure. Furthermore, it is very well supported in many programming languages, including Java.

Since DOM keeps a document in memory after it is parsed, a drawback is the time spent to parse a big file on the grounds that many memory allocation requests might sum up to a considerable time.

3 An application reconfiguration framework

Chapter 1 identified the problem of automatic application configuration, along with the concept of pattern in a configuration file, whereas Chapter 2 studied the existing tools and technologies that can help to mitigate the problem. Using that knowledge, this chapter presents the framework for automatic application configuration, the major contribution of this dissertation.

In truth, the proposed framework reconfigures applications (possibly fresh installations or deployments of an application) instead of configuring them at deployment time, but for the sake of clarity, the problem will be addressed in the remainder of the dissertation as the automatic configuration of applications. The proposed framework requires three steps to configure an application automatically: conversion from original to generic syntax, file modification and conversion back to original syntax. Only the first and third belong to the scope of this work. The configuration resorts to the lexical and syntatic analysis of the configuration file and subsequent production of a data structure which is equivalent to the original file. After the data-structure is properly modified, it is reconverted back to the original file form.

We start by identifying the functional and non-functional requirements of the tool in Section 3.1, Section 3.2 provides a broad view over the solution proposal, Sections 3.3 and 3.4 detail the required components in each step and their interfaces, Section 3.5 approaches the file modification part by proposing some ways to modify the intermediate file in generic syntax and providing some information about the generic syntax and Section 3.6 deals with the support for file formats not recognizable by the framework.

3.1 Architectural Requirements

Before approaching the proposal of a framework for the automatic application configuration, the tool requirements must be identified. This section presents the functionalities which the tool is expected to provide, in the form of use cases. A use case defines a goal-oriented set of interactions between external actors and the tool. Following is a list of functional and non-functional requirements:

1. **The user must be able to convert a configuration file syntax from its original syntax to a generic one, independently of the application.**

   Description: The user must be able to try parsing the configuration file with the available parsers.

   Non-functional requirements:

   Performance: The generated file with the generic syntax must be as simple as possible.

2. **The user must be able to define grammars for configuration file languages.**

   Description: If there are no suitable parsers for a given configuration file, the user must be able to define a grammar that recognizes the new configuration file language.

   Non-functional requirements:

   Usability: The grammar definition syntax should be a broadly adopted one.

   Usability: To ease the parser generation process, the user must be able to iterate through the previously built grammars so as to rollback any change made on a grammar.

3. **The user must be able to produce a parser from a grammar.**

   Description: When the user builds a grammar for a new configuration file language, there must be a means to compile that grammar in order to generate a parser.

   Non-functional requirements:

   Usability: The user must be able to add parsers built outside the tool.

4. **The user must have access to the parser compilation trace.**

   Description: When a grammar is compiled, the user should be able to check on the parser compilation trace to see if it was successfully compiled or whether any error persists in the grammar declaration.

Non-functional requirements:

Usability: The error messages must clearly identify the source of the error.

5. **The user must receive information relative to the grammar fitness with a given configuration file.**

   Description: When the user attempts at parsing a configuration file with a given parser, he/she must be informed if the parsing was successful or not, as well as the quantity of file recognized by that parser.

   Non-functional requirements:

   Usability: The sections of the configuration file that were parsed and those that were not must be clearly identified.

6. **The user must be able to store a functional parser generated by the tool, in order to be used on later tool runs.**

   Description: When the user checks a new parser to parse a configuration file entirely, he/she must be able to store it in a repository so it can be reused.

   Non-functional requirements:

   Security: The user must be able to see the parsing outcome in order to check if the parser is indeed operating as intended to.

7. **The user must be able to reconvert a configuration file syntax from generic syntax to the original one.**

   Description: Given a configuration file in generic syntax, the user must be able to convert it into its original syntax, keeping its functionality unharmed.

8. **File conversion must not eliminate comments.**

   Description: On the original to generic syntax conversion, the comments should be stored in order to show up in the final file.

9. **The user must be able to halt the tool execution at some point and continue the configuration process later.**

   Description: In the middle of the configuration process (i.e., right after the original to generic syntax conversion), the administrator should be able to halt the execution flow, have the necessary data stored, such as the file in the generic syntax or the tentative parsers, and continue from the same state later.

10. **The user must be able to manually delete parsers from the parser repository.**

    Description: The user must be able to manually delete parsers from the parser repository so that the user may intervene if the parser repository becomes to big, or if a certain parser becomes obsolete.

3.2 Automatic Application Configuration

Having identified the requirements for the tool, it is now possible to tackle a possible solution for the problem. This section presents a framework which is the solution proposal for the automatic configuration of applications.

The application configuration files are read by the tool, which parses them in order to convert their syntax from the original to a generic, application-independent, one. This allows for the configuration files to be manipulated and altered in the same way, for any application, through an agent which recognizes the generic syntax.

Being in generic syntax, the file can be efficiently modified by an agent which operates on that kind of syntax, such as an automated script. Ensuing the file modification, the tool is responsible to do the inverse conversion, back to the file original form. Once the file has been converted to the generic syntax, the characteristics of the original syntax have been aprehended and can be used to reconvert the file back to its original syntax.

The configuration process (Figure 3.1) can be divided into three main stages:

1. Configuration file extraction;
2. Original to generic syntax conversion;
3. File modification;
4. Generic to original syntax conversion.

The second stage is accomplished through an Original to Generic syntax Converter (OGC) and the fourth by a Generic to Original syntax Converter (GOC). This dissertation points ways in which OGC and GOC might evolve. The first and third stages are not on the original work scope and therefore are not tackled in detail. Nevertheless, Section 3.5 approaches the file modification area superficially, hinting at some possible ways of dealing with the subject and identifying possible operational constraints.

3.3 Original to Generic syntax Converter

The Original to Generic syntax Converter (OGC) is structured like a three-tier architecture [Ram00]. The three-tier architecture consists of a software design pattern which

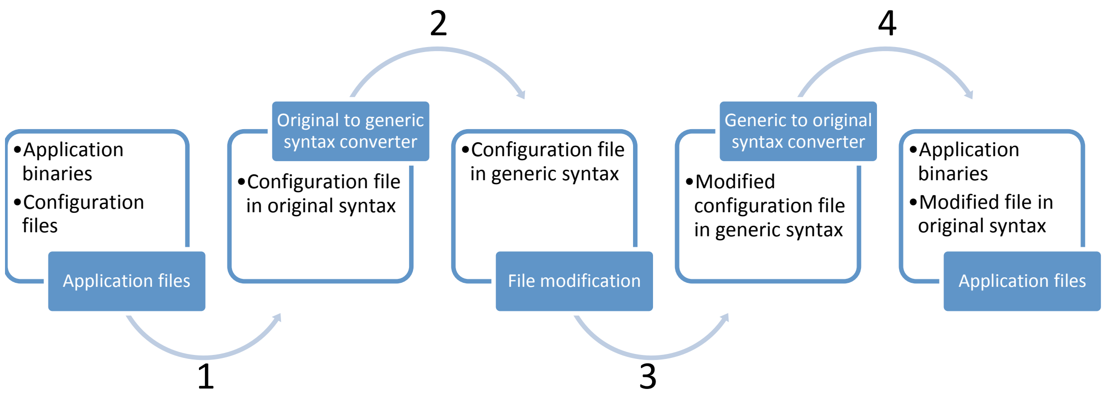
2
4
•Application binaries
•Configuration files
Application files
Original to generic syntax converter
•Configuration file in original syntax
•Configuration file in generic syntax
File modification
Generic to original syntax converter
•Modified configuration file in generic syntax
•Application binaries
•Modified file in original syntax
Application files
1
3

Figure 3.1: SmART Work Flow

Component	Description
User Interface	The User Interface makes the link between the user and the tool. It can be thought of as a command interpreter, which reacts to user orders by calling other tool components. Additionally, it is used to simplify some otherwise complex tasks, such as building grammars for configuration files.

Table 3.1: Presentation Layer Components

defines three inter-related tiers, each with its own responsibilities. In this work context, the tiers are:

Presentation Layer. Contains the presentation logic, including simple control and user input validation.

Logic Layer. Contains the processing logic and the data access.

Storage Layer. Provides the data storage.

In this topology, each layer is modifiable without interfering with the others, increasing modularity.

The three-tier architecture is depicted in Figure 3.2. However, the OGC does not follow a strict three-tier architecture, as later on, it is seen that some components on the presentation layer interact with those on the storage layer.

The components employed by the OGC are now briefly described. Table 3.1 presents the components used by the Presentation Layer, Table 3.2 contains the description of the Logic Layer components and Table 3.3 introduces the components used by the Data Layer.

3.3.1 Component Description

This section provides a description for each component required by OGC. First, the task of each component is briefly analysed and then, the operations of each component are more thoroughly

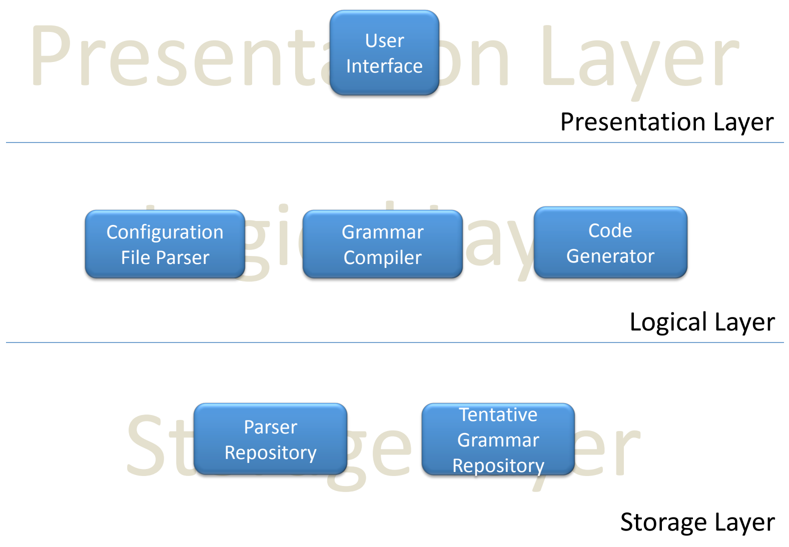

Figure 3.2: Components overview

Component	Description
Configuration File Parser	The Configuration File Parser recognizes the diversity of configuration files and parses them in order to change their structure to an application-independent one;
Code Generator	The Code Generator reads the parsed output from the Configuration File Parser and generates the generic syntax of the file;
Grammar Compiler	The Grammar Compiler is the component which generates new parsers from grammars.

Table 3.2: Logic Layer Components

Component	Description
Parser Repository	The Parser Repository is a parser database which holds all the functional parsers generated to the date;
Tentative Grammar Repository	The Tentative Grammar Repository stores the grammars built by the user on the process of creating a new parser for a configuration file language.

Table 3.3: Data Layer Components

Listing 3.1: User Interface proposed interface

```
interface UI {
  void processFiles(String[] fileNames);
  void submitGrammar();
  byte[] getTentativeGrammar();
  void approveParser();
  boolean importParser();
  void haltSession();
  void recoverSession();
}
```

described. The interfaces that every component should implement are also presented, in a *Java-like* syntax. Enforcing every component to implement an interface not only allows any component to be completely remade without having any effect on other components, it also aids the integration process of SmART with other tools.

User Interface

The User Interface (UI, Listing 3.1) is the tool's front-end, through which the user interacts with the tool. It is invoked by the user and receives an array of configuration file locations at boot time, although it only processes one at a time.

For each file, `UI` summons the `Configuration File Parser` to make an attempt at parsing a configuration file with the parsers in the `Parser Repository`. Two outcomes are possible: either a parser manages to recognize the totality of a file or none does.

The first case is the ideal case, where at least one of the parsers in the `Parser Repository` manages to fully recognize the configuration file. In this situation, `UI` displays the representation of the generic syntax, as returned by the `Configuration File Parser`, to let the user validate the generated structure. At this point, the configuration file in generic syntax is stored in memory, so `UI` must save it to a file to let file modification take place.

The second case, where the configuration file was not completely parsed by any available parser, requires user intervention. If the file could not be totally recognized by any parser, another parser which recognizes this new language must be generated. For this endeavour, `UI` provides the user with a friendly interface which allows him to define a new grammar that recognizes the configuration file language. The user must also be allowed to build a new grammar from an existing one. This is useful in the case where a parser managed to parse a configuration file almost entirely. In this case, the new parser might be ready just by tuning an existing one, instead of having to build it from scratch.

As soon as a grammar is defined, the user triggers its compilation with the `Grammar Compiler`'s `submitGrammar` call. Any grammar compilation errors should be displayed to the user. To ease parser generation, the user must have access to the previously tested grammars. This allows for the rollback of changes in grammars in an easy way. To accomplish this, every time the user submits a new grammar, `UI` sends it to the `Tentative Grammar`

Listing 3.2: Configuration File Parser proposed interface

```
interface CFP {
  Map<Parser,ParsingData> doParse(String fileName);
  ParsingData doParse(String fileName, Parser parser);
}
```

Repository. Then, the user can fetch the previsouly built grammars through the getTentativeGrammar call.

The parsers generated by the Grammar Compiler must be validated by the user to check if the new parser is able to parse the configuration file, or if the generic syntax produced by the parser is indeed the desired one. UI must handle this information and show it to the user in a user-friendly optic (e.g., graphically). Then, if the user approves the parser, he can summon the approveParser method to store the parser with the remaining parsers in the Parser Repository. Finally, UI stores the generic syntax in memory on a file.

Besides the presented basic functionality, UI offers some more utilities to the user. Namely, the importParser method allows the importation of external parsers if the user is already in possession of a functional parser for a new configuration file. Additionally, the user may suspend the tool's execution and resume it later through the haltSession method. This method stores the tentative grammars to disk, as well as other volatile data in memory, such as the produced file in generic syntax, if it was already produced. The recoverSession method obtains all the stored data to resume a previously saved session.

Configuration File Parser

The Configuration File Parser (CFP, Listing 3.2) can either parse a configuration file with every parser in the database or just with a single one. The parsers analyse the configuration files in their original syntax, declared in a grammar which defines the tokens and productions of that syntax, and produce the abstract syntax tree (AST) of the file.

The ASTs generated by the parsers must follow the typing indicated in Section 1.3. Therefore, nodes composing an AST are either *parameters*, *blocks*, *comments* or nodes corresponding to new patterns. By assuring that the ASTs are typed in such a way, it is possible to conceive a uniform Code Generator, which is able to traverse the AST of any application. Furthermore, strictly typed ASTs guarantee that the generic syntax is constant for every application. In other words, the generic syntax will be composed of elements like *parameters*, *blocks*, *comments* and *specials*, which is a standard recognized by the file modification agent.

CFP may iterate the available parsers in the Parser Repository and attempt to parse the input file with each one of them. Alternatively, CFP may also be called to parse a file with a newly generated parser if the file was not entirely recognized with any of the existing parsers.

When a parser is tested, it returns the configuration file intervals which were recognized, along with the AST of each interval. The ASTs are sent to the Code Generator to be converted into generic syntax, in order to help the user analyse and validate the output produced

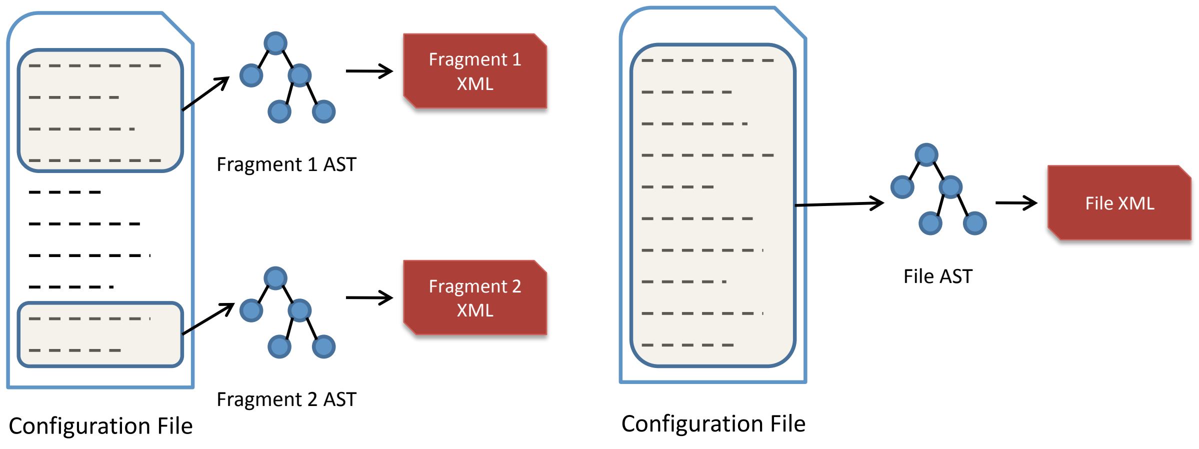

Figure 3.3: Partial and totally parsed configuration file stages

Listing 3.3: Code Generator proposed interface

```
interface CG {
  Document doTranslate(List<Node> ast);
}
```

by a parser. If a file has been completely recognized, the interval returned by the parser is the same as the overall original file length and the only returned AST will correspond to the whole file. The stages of the file in both scenarios are depicted in Figure 3.3. Since the `CFP` tests a file with every parser in the `Parser Repository`, it returns a map of every parser and the data regarding the parsing attempt with that parser. On the other hand, if the `CFP` tests a file with a single parser, it returns the parsing data with that parser.

Code Generator

The Code Generator (CG, Listing 3.3) traverses the ASTs created by the parsers and produces the generic syntax for a configuration file or a configuration file fragment. Besides that, `CG` must also capture the properties of the configuration file original syntax and save them in the generic syntax in order to ease the opposite conversion process, from generic to original syntax.

`CG` is able to correctly traverse any AST, provided that this follows the typing described in Section 1.3, with *parameters*, *comments*, *blocks* and other types meanwhile defined. Moreover, the generic syntax produced by `CG` must be the same for any kind of configuration file so as to detach the Original to Generic syntax Converter from the Generic to Original syntax Converter. More considerations on the generic syntax produced by `CG` can be found in Section 3.6.

Grammar Compiler

To allow the generation of new parsers for the recognition of new configuration file languages, the Grammar Compiler (GC, Listing 3.4) was created. `GC` produces parsers from user-built

Listing 3.4: Grammar Compiler proposed interface

```
interface GC {
  CompilationData doCompile(byte[] grammar);
}
```

Listing 3.5: Parser Repository proposed interface

```
interface PR {
  Parser storeParser(ParserID parserID, ParserInt parser, byte[]
  grammar);
  Parser importParser(ParserID parserID, ParserInt parser);
  Iterator<String, Parser> getIterator();
  void deleteParser(String parserID);
  void saveState();
}
```

grammars. For this, CG may make use of an outside parser generator, such as Bison [FSF], SableCC [GH98], JavaCC [Proa], etc., by invoking it whenever a grammar is received. Also, when a grammar is not valid, CG must return an error message containing information about the problem.

Parser Repository

The Parser Repository (PR, Listing 3.5) is a parser database which manages the parsers used by the tool to that date. It maintains a map of `ParserID`, which is the parser name and must uniquely identify it, and `Parser`, used to call a given parser. `PR` provides a number of functionalities to the user. These functionalities are now enumerated.

The parsers generated by the `Grammar Compiler` may be stored by the `PR` if the user has approved them. The `storeParser` method saves a parser in the database, given its name, which uniquely identifies it, the parser and its grammar definition.

If a user possesses an already built parser for a new configuration file, it is possible to import that parser into the tool. The `importParser` method saves an external parser in the database. The importation of external parsers is, nevertheless, conditioned by a pre-defined interface which forces external parsers to implement an invocable method. This matter is further discussed in Section 3.6.

Besides inserting parsers into the database, it is also possible to obtain an iterator for the existing parsers in the parser repository through the `getIterator` method, to delete a parser from the database with the `deleteParser` method and to save the state of the database by calling the `saveState` method.

Listing 3.6: Tentative Grammar Repository proposed interface

```
interface TentativeGrammarRepository {
  Grammar store(GrammarID grammarID, byte[] grammar);
  Grammar next();
  Grammar prev();
  Grammar get(int grammarNumber);
  void discard(int grammarNumber);
  void discardAll();
  void saveState();
}
```

Tentative Grammar Repository

To respond to the architectural non-functional requirement which says that the user must be able to iterate through the previsouly tested grammars, the Tentative Grammar Repository (TGR, Listing 3.6) was conceived. `TGR` is a database which maintains a list of the grammars, functional or not, created by the user so far, in the process of building a new parser. Each grammar in the database knows what grammar was built before and after it.

To store a grammar, the `store` method receives a `GrammarID` with the grammar name and a byte array containing the grammar declaration.

The grammars are retrievable in two ways. The user can obtain grammars on an *undo/redo* fashion using the `next` and `prev` methods, which return the grammar built immediately before or after. However, if the user desires to get a specific grammar in time, it can be obtained using the grammar serial number which is unique to each grammar.

The `discard` and `discardAll` methods delete grammars from the database. The `discard` method discards a grammar and those based on it, while the `discardAll` method discards every grammar in the database.

Finally, the state of the database can be saved, with the `saveState`, whenever the tool execution is halted.

3.3.2 Original to Generic Syntax Converter Operation Scenarios

Depending on whether a parser for a configuration file exists in the `Parser Repository`, OGC may have two different execution scenarios. For a clearer perception of the architecture and to give a better insight on the interactions between components, both scenarios are now briefly summarized.

Considering that there is a parser for a given configuration file in the database, the original to generic conversion unfolds as follows (Figure 3.4):

1. The `User Interface` calls the `Configuration File Parser` to parse the configuration file;

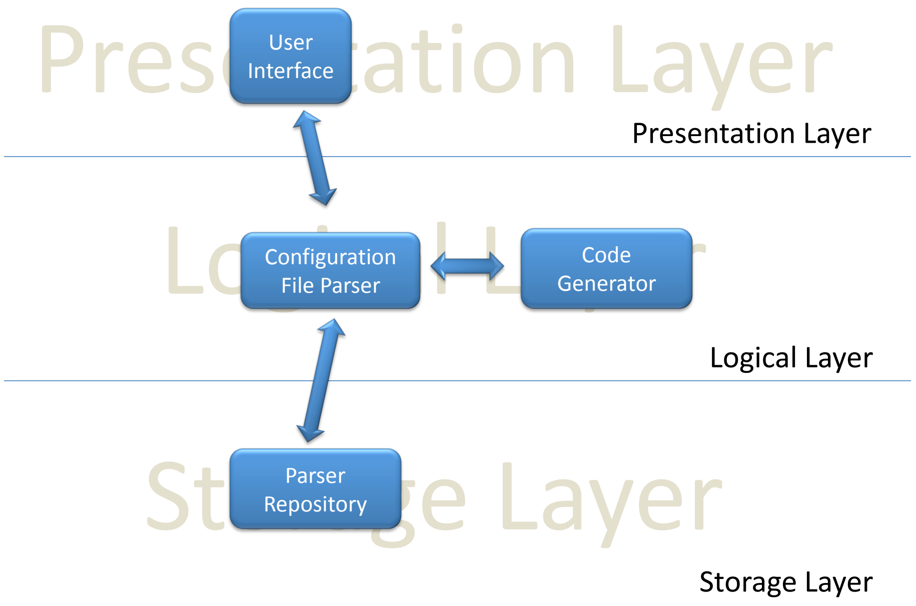

Figure 3.4: Interactions on the successful file recognition scenario

2. The `Configuration File Parser` gets the parsers from the `Parser Repository` and tests the configuration file with all of them. By assumption, at least one parser is able to parse the configuration file entirely, therefore, at least one valid AST for the entire configuration file is produced;

3. The `Configuration File Parser` sends the generated ASTs to the `Code Generator` and returns `User Interface` the generic syntax of the file for each parser;

4. Finally, the `User Interface` chooses a valid file in generic syntax and stores it to disk.

On the other hand, if a configuration file is new to the tool, the process requires more components to operate. The previous process is performed normally until step 2, where no parser was found to be able to parse the configuration file by the `User Interface`. Afterwards, the action flow is as follows:

1. The user must build a grammar through the `User Interface` so that the `Grammar Compiler` generates a parser from it. If a grammar declaration is not valid or if the generated parser still cannot parse the configuration file, another grammar must be built until the configuration file is parsed;

2. Meanwhile, every grammar produced by the user is stored in the `Tentative Grammar Repository` for the user to easily rollback any change made to a grammar;

3. When a parser successfully recognizes a configuration file, the user has the last word as he may approve or disapprove the structure produced by that parser;

4. If the user does not approve of a parser, another one must naturally be generated;

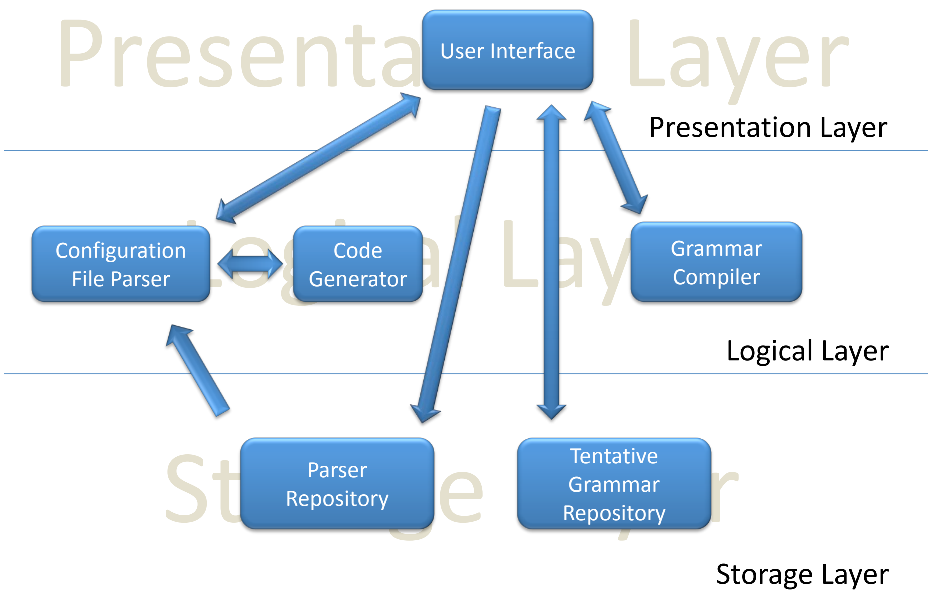

Figure 3.5: Interactions on the failed file recognition scenario

Listing 3.7: Printer proposed interface

```
interface Printer {
  void print(String file);
}
```

5. If a parser is approved, it is stored in the `Parser Repository` and the `Tentative Grammar Repository` may be emptied.

Afterwards, the course of action returns to normal: the AST produced by the parser is sent to the `Code Generator` and the resulting file in generic syntax is stored to disk by the `User Interface`. The interactions between components on this process are depicted in Figure 3.5.

Annex A.1 is a sequence diagram showing how components interact with each other, and in what order.

3.4 Generic to Original syntax Converter

The conversion from generic syntax back to the original one is accomplished solely by one component, the `Printer` (Listing 3.7).

The `Printer` is able to convert files in generic syntax to any configuration file format. It does so by using the information on the file's original syntax, captured by the `Code Generator` in Section 3.3.

The outcome of the `Printer` is the modified file in its original syntax, which remains 100% functional and is ready to be re-merged with the other application files, causing the application to become configured.

3.5 Modifying the Configuration File

Despite not being part of this work, some possible ways to deal with the configuration file modification theme were identified.

In the presence of a configuration file, structured in an XML document, one can either manually modify it, using a regular text editor or a similar XML editor, or automatically modify it, employing an XML script which applies the changes to a configuration file without the need for any user intervention.

The first way does not ask for any special requirement, other than having a text editor at hand, and therefore, consists on the easiest way to modify a configuration file in XML. On the other hand, the second way requires a script to be previously built. This script may, nevertheless, be reusable and also allows for a completely automatic application configuration process. However, one must first know how the generic syntax is structured in order to build a script that operates on it.

Therefore, it is very important that all the files in generic syntax follow the same structure and nomenclature, regardless of their original format. For example, assuming there is a script which goes through an entire configuration file searching for the block “Network” to change the “address” parameter value to “192.168.0.100”, it is vital that every file in generic syntax represents block parameter patterns in the same way (i.e., a generic syntax element called “Block” with a “value” element assigned to it) so that the script recognizes those patterns when it sees them. The same goes for anyone who wants to manually make that change. The person who applies the modification should not be obliged to know multiple generic syntaxes for various configuration file languages, but rather a homogeneous syntax which might slightly diverge from language to language, depending on that language idiosyncrasies.

3.6 External Parser Addition

In response to the non-functional requirement in Section 3.1 which says that the user must be able to add parsers built outside the tool, Section 3.3.1 indicates that the `Parser Repository` implements the `importParser` method. This utility is especially useful when the user wants to configure applications whose configuration files cannot be parsed by parsers created by the chosen parser generator (e.g., binary files) or if the user already possesses a parser for a configuration file.

Nevertheless, in order for an external parser to be added to the tool, it must implement the interface defined in Listing 3.8. This interface solely defines one method, which is the one called by the `Configuration File Parser` when parsing a file with an external parser. Furthermore, the method must return the parsing statistics and the parsed blocks in generic syntax.

Listing 3.8: External Parser Proposed Interface

```
interface ExternalParserInterface {
  public ParsingData parse(String fileName);
}
```

4 An Implementation

Chapter 3 approached the application configuration framework by identifying the requirements to be met and defining a tool workflow and component architecture. This chapter describes the implementation of that framework, developed in the context of this thesis.

First, implementation decisions of several orders are presented. Section 4.1 presents the adopted programming language, Section 4.2 presents the adopted parser generator and Section 4.3 presents the adopted generic syntax format. Then, Section 4.4 shows the implementation of the created data structures in terms of the motivation for their use and their functionality. Afterwards, Section 4.5 presents the details of the Original to Generic syntax Converter component implementation, Section 4.6 shows how the files in generic syntax are printed back to original syntax and last, Section 4.7 displays the Generic to Original syntax Converter component implementation.

4.1 Programming Language

As it is well known, programming languages differ from each other not only in terms of supported programming paradigms, but also expressive power, documentation, popularity, etc. Among the requirements that the programming language was expected to meet, two had a special weight in the time of decision: the chosen programming language should be object-oriented due to the modular nature of the tool and high-level to easily and quickly provide programming tools such as data types, data structures, etc.

Given the previous considerations, the decision fell on the Java[1] programming language, considering it supports the object-oriented programming paradigm, among many other programming paradigms, is high-level and its API (Application Programming Interface) is well

[1]http://java.sun.com/

Listing 4.1: Node interface

```
interface Node {
        void translate();
}
```

documented. Moreover, a great number of parser generators provide support for this programming language.

4.2 Parser Generator

The choice of a parser generator took into account the two parsers studied earlier in Section 2.3, SableCC and JavaCC.

SableCC is defined by automatically generating all nodes of an AST from a grammar, implementing a fairly simple and intuitive grammar definition interface and strictly separating all the user-generated code from the parser generator-generated code.

On the other hand, JavaCC makes it possible for the insertion of Java code in the grammar definition, which may cause the grammar definition interface a bit more complex. Unlike SableCC, JavaCC does not automatically generate the nodes of the AST, leaving that task to the user. The separation of parser generator and user-generated code is relative and depends on the degree of detachment defined in the grammar.

Initially, SableCC's characteristics led us to consider it as the best option, since it allowed for the rapid prototyping of the tool. However, after used and tested, its mechanics were found to be more complex over time considering that, by having each parser creating its own AST, it requires specific walkers for each AST. The direct impact that this characteristic had on SmART is that for each different configuration file type is a different grammar, which leads to a different AST, which leads to one AST walker for each configuration file type. Adversely, since JavaCC requires the nodes of the AST to be manually defined, it is possible to conceive a structure where every node in it implements a method that can be called by the structure walker (see Listing 4.1). In this way, only one walker is required for any kind of configuration file, which is able to visit the AST of any configuration file type and produce the file in generic syntax by calling the method defined in the interface of Listing 4.1.

Another severe limitation of SableCC is that it currently does not offer the error recovery functionality, whereas JavaCC provides two types of error recovery: shallow and deep error recovery. Error recovery consists on continuing the parsing of a configuration file even after an unrecognized token has been found. This allows for better accuracy when calculating the fitness of a parser with a configuration file.

Given the previous considerations, the initial beliefs in SableCC were turned in favour of JavaCC.

Listing 4.2: Parser data structure

```
interface Parser {
  String getParserID();
}
```

Listing 4.3: InternalParser data structure

```
interface InternalParser {
  Method getParseMethod();
  byte[] getGrammarDefinition();
}
```

4.3 Generic Syntax

Section 2.4 indicated that the chosen format for the representation of the configuration files in generic syntax would be XML (eXtended Markup Language). Furthermore, that Section saw that there are two XML parser implementations for the Java programming language, DOM [W3S] and SAX [Prob]. It was seen that the first keeps a tree in memory, representing the XML document, while the second is event-driven.

By balancing the benefits and drawbacks of each parser, DOM was preferred to SAX on account of representing an XML document in memory as a tree structure. This allows for a quick transition of an AST to a DOM tree.

4.4 Data Structures

This section presents the data structures created to cope with various aspects regarding the passing of data between components. For each data structure, an interface is revealed and an explanation on the data structure functionality and the methods declared in the interface is presented.

4.4.1 Parser

`Parser` objects are used as references for the parsers used by the tool. This implementation considers that the parsers are identified by their name and, therefore, the `ParserID` field consists on a `String` instead. However, the `Parser` class cannot be instanciated since it is an abstract class. To distinguish the internal parsers, built by the tool, and external parsers, built outside the tool and imported by a user, two classes extending `Parser` were created: `InternalParser` and `ExternalParser`. The two classes are now explained.

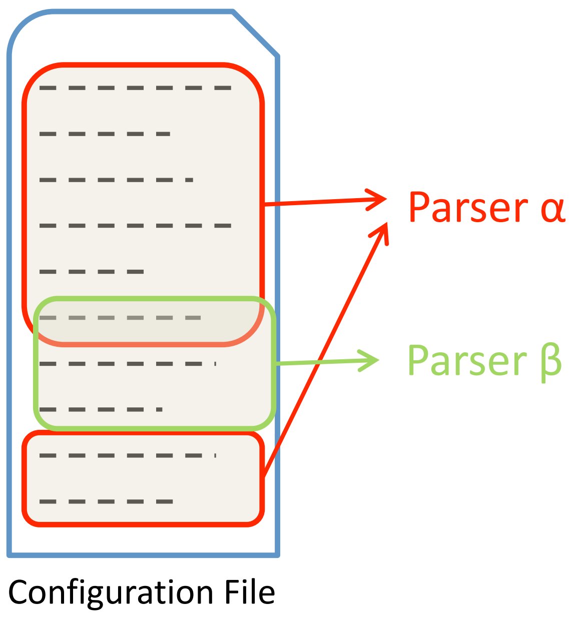

Figure 4.1: Recognition of a configuration file by two parsers

InternalParser

`InternalParser` (Listing 4.3) objects represent parsers generated by the tool, through the `Grammar Compiler`. An `InternalParser` object contains a pointer to the parse method and the grammar used to construct the parser.

Regarding the parse method, the latter is invoked resorting to the Java reflection API and returns a list of `Node` objects (Listing 4.1) due to be transformed to XML by the `Code Generator`.

Concerning the grammar used for the generation of the `InternalParser`, it is kept with the parser in order to aid users on the generation of new parsers. Consider the case where a user tries to configure a file which is new to the tool and two parsers are stored. The result of the parsing attempt is depicted in Figure 4.1.

In the example, two parsers are tested on a configuration file with 100 lines. The first parser, α, manages to recognize the file from line 0 to 59 and then from line 70 to 99, resulting in a 90% of parsed file. On the other hand, parser β recognized 20% of the file, from line 50 to 69.

In this situation, the user who would build the new parser for the file might want to access the grammar of the parser α, since it parsed almost the entire file, and base the new parser on the old instead of starting from scratch. Also, seen as the second parser recognized the part of the file that the first did not, the user might want to check the grammar of the second parser.

Listing 4.4: ExternalParser data structure

```
interface ExternalParser {
  ExternalParserInterface getParseMethod();
}
```

Listing 4.5: Grammar data structure

```
interface Grammar {
  int getSerialNumber();
  String getGrammarID();
  byte[] getGrammarDefinition();
}
```

ExternalParser

`ExternalParser` objects are used to represent parsers built outside the tool. Since these parsers might not be built with the parser generator used by the tool, there is no knowledge of the content they produce when parsing a file. Therefore, an `ExternalParser` has an attribute of the `ExternalParserInterface` type, which contains the invocable method declared in Listing 3.8 to parse a file and return the file in generic XML.

4.4.2 Grammar

`Grammar` objects (Listing 4.5) stand for tentative grammars. They are created in the `User Interface` and stored the `Tentative Grammar Repository` when a new grammar is built.

A `Grammar` contains a serial number, which is used by the `Tentative Grammar Repository` to have a means of identifying each grammar, since the name of all the tentative grammars for a parser tends to be equal because it represents the name of the parser being built. In this implementation, the `getGrammarID` returns the grammar name in a `String`. The grammar name is used to name the `Grammar`, but also to name the parser which it was used to generate, as already mentioned. Finally, a `Grammar` also contains an attribute for the grammar declaration, accessible through the `getGrammarDefinition` method.

4.4.3 ParsingData

`ParsingData` objects (Listing 4.6) contain pieces of a configuration file, in generic syntax, and information regarding the parsing of that file with a parser and are returned by the `Configuration File Parser`.

Although the `ParsingData` objects do not contain any reference to the corresponding parser, they always refer to the parsing result of a parser. This relation is explicit when the `Configuration File Parser` returns objects of this kind. When a single parser is tested, the return is a `ParsingData` object, obviously referring to the parse with the tested parser.

Listing 4.6: ParsingData data structure

```
interface ParsingData {
  boolean wasParsed();
  int getParsingDone();
  List<Block> getParsedBlocks();
  void dump(String fileName);
}
```

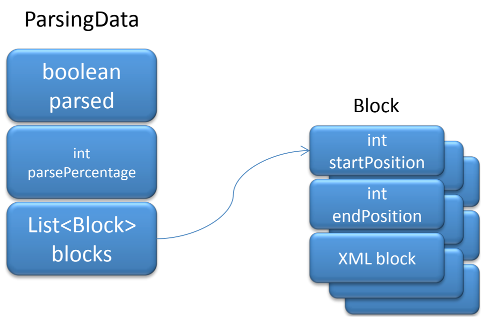

Figure 4.2: Graphical representation of ParsingData

Whenever multiple parsers are tested, the return is a map of the tested parsers and their `ParsingData`.

By having a `ParsingData` contain the parts of a file recognized by a parser, the case where a file is completely parsed is a special case of when a file is not totally parsed, considering that in the first case, a ParsingData will only contain one part, which is the whole parsed file. However, a parser may return only one part of a file but not recognize it entirely. Therefore, additional control is required to check if a file was well parsed or not. The `wasParsed` method tells whether a file was entirely recognized, or not.

In case a file was not completely parsed, its parsing statistics are logged. This is an indicator of how fit the parser was found to be with the given configuration file and can be measured in terms of percentage of parsed file.

Additionally, the `ParsingData` objects also contain a list of Block objects. A Block object has no relation with the *block* pattern, but rather contains the generic syntax of a parsed file part as well as the positions of the first and last character that Block represents. The representation of a `ParsingData` object, and the contained Block object, is shown in Figure 4.2.

Whenever the file is due to get modified in order to be configured, or the user halts the tool execution, the `dump` method is called in order to store the blocks in a given file in persistent storage.

4.4.4 CompilationData

`CompilationData` objects (Listing 4.7) are produced by the `Grammar Compiler` to send information regarding the compilation of a grammar to the `User Interface`. A `CompilationData` object tells if a grammar was successfully compiled in the `wasCompiled`

Listing 4.7: CompilationData data structure

```
interface CompilationData {
  boolean wasCompiled();
  String getErrorMessage();
  String getParserLocation();
}
```

method. If the compilation ended without any occurred error, the parser location is given by the `getParserLocation` method. Alternatively, if an error ocurred during the compilation, it is accessible through the `getErrorMessage` method.

4.5 Original to Generic syntax Converter

After the components required in the Original to Generic syntax Converter were identified and the behaviour of each one was described, their implementation took place. Following is shown how the components identified in Section 3.3 were implemented.

4.5.1 User Interface

The `User Interface` is passed multiple configuration file locations upon its invocation by the user. The parsing of multiple configuration files is sequential, or, in other words, `UI` configures one file at a time and only when that file is configured should the next file configuration start.

To configure a file, `UI` sends its location to the `Configuration File Parser` in order to parse it with every available parser. The `Configuration File Parser` returns a map of `ParsingData` objects which tells if the parsing was successful with any parser, as well as the percentage of parsed file, the parsed file blocks in generic syntax and the position of the first and last characters of the corresponding block.

When a file is successfully parsed, the corresponding AST is sent to `Code Generator` and the returned file in XML is placed in a temporary file folder, with the same name as the original configuration file, plus the suffix *".xml"*.

If a file was not entirely parsed, `UI` displays the parts of the file which were recognized by the parser with the best configuration file parse percentage and the grammar used on the respective parser. Then, `UI` prompts the user for a file containing a grammar to be compiled and sends the grammar to `Grammar Compiler`. The `Grammar Compiler` returns a `CompilationData` object which tells whether there was any error compiling the grammar. If an error ocurred, the user must alter the grammar in the file to correct the error(s) and then trigger another grammar compilation until the grammar is successfully compiled.

When a grammar is successfully compiled, `UI` attempts to parse the configuration file with the newly generated parser by invoking the `Configuration File Parser`'s `doParse` method which tests a file with a single parser. If the file was still not entirely recognized, the user must alter the grammar to recognize the text parts which failed to be parsed, until a parser which recognizes the whole file is generated.

The initial plan was for a graphical interface to be launched whenever a file could not be parsed by any parser in the database. This interface would display useful information to the user such as the graphical representation of the configuration file in XML, grammar navigator and highlighted configuration file based on its parsed and not parsed parts. `UI` would also allow the user to define a grammar without having to manually program it, resorting to the selection of configuration file pieces and assigning meanings to them, in an interactive way. However, the implementation of the graphical interface will only take place during the integration of the tool with VIRTU.

4.5.2 Configuration File Parser

When the `Configuration File Parser` is summoned for the first time, it initiates the Parser Repository.

The `doParse(String fileName)` method calls the `doParse(String fileName, Parser parser)` method for every available parser in the `Parser Repository` and logs the resulting `ParsingData` objects in a map, where the entry key is the used `Parser`. This method returns the map generated in this way.

The `doParse(String fileName, Parser parser)` method, in turn, treats a configuration file with a parser at a time. If `parser` is an `InternalParser`, its parse method is invoked using the Java reflex API and the return, a list of `Node` objects (Listing 4.1), is sent to the `Code Generator` to be passed to generic XML. Thrown exceptions during parse time mean that a file was not entirely recognized. The exceptions caught at parsing time contain the position where the recognized block starts and ends. These exceptions are analysed to retrieve that information. The percentage of recognized file is calculated based on the limits of every parsed block. In the end, a `ParsingData` object containing the parsing percentage, the recognized blocks, characterized by their start and end points, together with the XML of the respective blocks, is returned.

4.5.3 Parser Repository

The `Parser Repository` holds a map of `Parser` objects corresponding to the available parsers. When `PR` is initiated, it gets the last saved state by loading the parser map to memory.

The `storeParser` method creates a `Parser` object from a string containing the name of the parser, and the `ExternalParserInterface` which contains the parser. An optional `grammarFile` can be passed, in case the user wants to merge the grammar with the `Parser` so that it is available in the future. If the `grammarFile` argument is not `null`, the grammar must be stored in the grammar directory, with the same name as the parser. The `Parser` is then inserted in the `Parser` map. The `importParser` does a similar job in inserting a parser in the database, but for external parsers.

`PR` also implements methods for getting the parser map iterator, `getIterator` and to delete a parser from the repository, `deleteParser`, which consists on deleting the map entry with the parser name in argument.

4.5.4 Grammar Compiler

The `Grammar Compiler` is implemented as a class which receives a grammar definition and produces a parser from it. After JavaCC generates the parser classes, these must be compiled before the `Parser Repository` is able to load them.

In the end, the `Grammar Compiler` returns a `CompilationData` object, which tells that a grammar failed to compile if an exception was caught, or that it compiled successfully otherwise.

4.5.5 Code Generator

The `Code Generator` traverses structures composed of nodes implementing the interface defined in Listing 4.1 and invokes the method defined the interface upon visiting each node in order to generate the generic XML of the pattern associated to that node. The reason why every node in structure implements that interface was already explained in Section 4.2 and consists on forcing each node of the AST to implement a method which can be called by `CG`. In this way, the information on how to print the nodes to generic XML is stored in the nodes and `CG` gains the ability to successfully traverse ASTs of any configuration files.

On an early stage of the tool, the code that generates the generic XML of a node in the AST must be generated manually, for new patterns. However, with the creation of a graphical interface for the definition of grammars for new types of configuration files in the `User Interface`, it should be possible to use the information received from the user to automatically generate that code.

`CG` is responsible for logging the details of each configuration file language, such as the used delimiters on patterns, or how a pattern is formatted. For this, `CG` resorts to a *Metadata* special field, which stores the information regarding the delimiters used by the patterns, and another *FStr* special field to log how each pattern is formatted. Details regarding the generation of original syntax details are further described in Section 4.6. A set of functions to manipulate the *Metadata* and the *FStr* fields are provided.

The generic XML must be the same for every configuration file types, following a structure similar to the presented in Section 1.3, with *blocks*, *comments*, *parameters* and other special elements. This is important since the generic XML document might become unrecognizable to the configurator which modifies it if the user decides, for example, to denote blocks by *"blockx"*. The configurator will afterwards find a *"blockx"*, when in fact its operation was defined on a *"block"*.

4.5.6 Tentative Grammar Repository

The `Tentative Grammar Repository` implements a linked list of `TGRNode` objects. A `TGRNode` represents a tentative grammar in `TGR` and contains a `TGRNode prev` attribute with

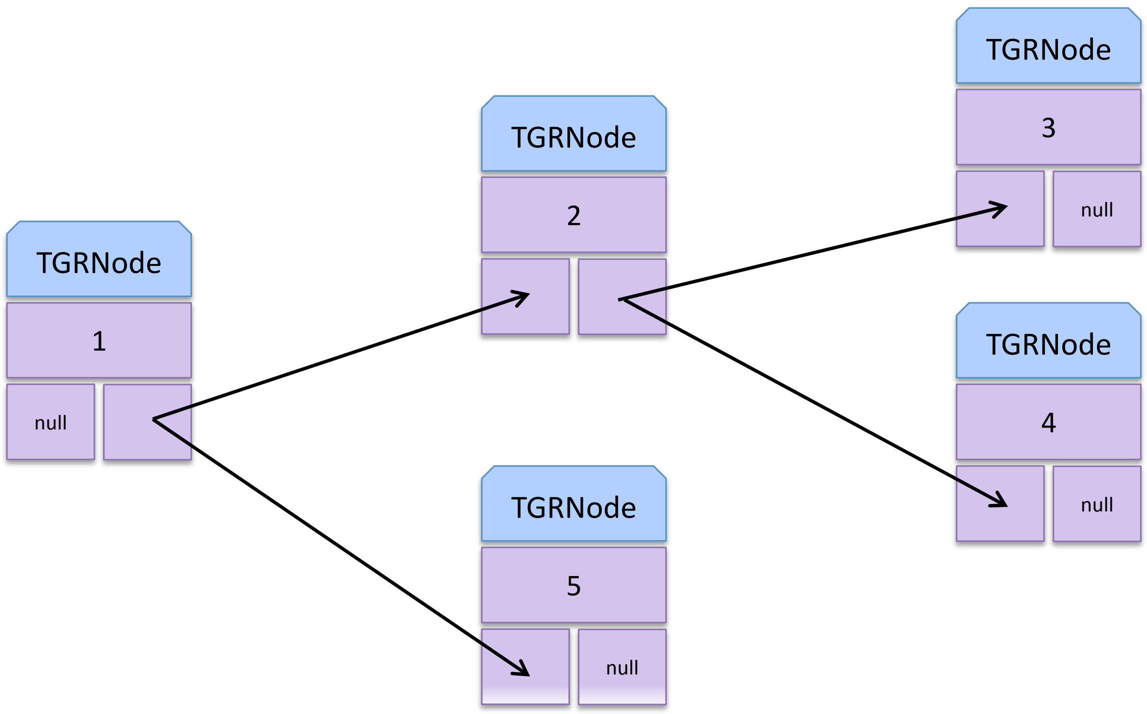

Figure 4.3: List of TGRNodes

the grammar where the current was based on and a `TGRNode` list `next` containing the grammars based on the current. An example is represented in Figure 4.3.

To allow the user to traverse all the built grammars, the `prev` and `next` methods resort to each grammar serial number to iterate the grammars backwards/forwards. Each method may also return null if no grammar was built before/after the current one, respectively. The `get` method receives an integer representing the desired grammar serial number and returns the respective Grammar.

The `discard` method deletes a grammar from the database, and every grammar based on that one. The `discardAll` empties the list of tentative grammars. To save the state of the database, the gramar list is stored to disk.

4.6 Generation of Original Syntax

The generation of original syntax consists on printing a file on generic syntax back to its original syntax by means of a `Printer` component. When a file is transposed from its original syntax, say *INI*, to the generic syntax (XML), the language-specific details, like the '[' and ']' header delimiters, are discarded to allow for application-independent configuration. Furthermore, the format of each pattern is also lost. Only the relevant information for the configuration process, such as parameter keys and values, is kept.

When the printer tries to print the file in generic syntax back to its original form, it will

Listing 4.8: Example Metadata

```
<Metadata>
  <Block>
    <LBra>[</LBra>
    <RBra>]</RBra>
  </Block>
</Metadata>
```

not know what application is being configured at that moment. Therefore, it cannot determine what delimiters to print on a given pattern (e.g., *INI "[]"* or *XML* block delimiters), or how to print them (e.g., *INI* single header or *XML* header and footer). So, to print a file to its original syntax, there should be enough information regarding the original syntax, in the generic syntax, for the `Printer`.

One way to know what delimiters should be print is to store them in a field in the beginning of the file in generic syntax. As the pattern delimiters are immutable throughout the configuration file (i.e., comments always start with # in an Apache configuration file), information regarding them can be unified in a special *Metadata* field in the beginning of the file. This makes it possible for the `Printer` to query *Metadata* to know what to print on a certain pattern.

The format of a pattern is of great importance as well. The same pattern might have different formats in two different configuration file languages. For instance, an *INI* block has only one block header, whereas an *XML* block (i.e., element) has a header and a footer. Again, since the Printer does not know what application it is configuring at the moment, it does not know whether to print a pattern one way or the other.

To solve this problem, a format string which describes how a pattern looks like is associated with each pattern, forming an *FStr* field. In this way, the `Printer` gets the *FStr* from a pattern and prints it accordingly. A format string may be composed of the following:

%a.name The value of the attribute named *name* in the current element is printed;

%e The value of the current element is printed;

%m.name The value of the element named *name* in *Metadata* is printed;

%c The format string is in the next child of the current element;

%s A space is printed;

%n A new line is printed;

Let's consider the following header in an INI configuration file:

```
[Engine]
```

The corresponding *Metadata* is depicted in Listing 4.8 and the *FStr* will look like Listing 4.9.

Listing 4.9: Example FStr

```
<Block name="Engine">
  <FStr>
    %m.LBra%a.name%m.RBra%n
  </FStr>
  ..(other patterns)..
</Block>
```

Listing 4.10: Apache parameter with a single value

```
ServerType standalone
```

Note that in the example, the *FStr* is assigned to the *Engine* block and not to the block pattern. This happens because there are cases where the same pattern has different formats in a configuration file language. For example, a parameter in an Apache configuration file might have one (Listing 4.10) or multiple values (Listing 4.11), or an XML element might have one (Listing 4.12) or multiple sub-elements (Listing 4.13), which leads to multiple format strings. Therefore, *FStr* cannot be factorized in a single field, like *Metadata*.

4.7 Generic to Original Syntax Converter

The Generic to Original syntax Converter contains two classes, `PrinterMain` and `Printer`. The first is used to interact with the user, while the second contains methods for the interpretation and printing of the file in generic syntax (i.e., XML). Following is a deeper explanation of both classes.

The PrinterMain class implements a `main` method which receives the location of the XML file in argument. It imports the XML document from the file to memory using the Java XML parsing API. Then, PrinterMain sends the document pointer and a stream to the `Printer` and the latter prints the final configuration file in original syntax to the stream in argument.

The Printer class manages the printing of XML elements inside an XML file. It implements a single visible method, `print`.

The `print` method is called by `PrinterMain` to print the entire XML document into a configuration file in its original syntax. It receives a stream and a `NodeList` object as arguments. `NodeList` refers to a DOM interface for the representation of lists of `Node` objects, whereas each `Node` object stands for XML components, such as elements, attributes or text nodes. Since the XML standard defines XML documents as having a single root node, `PrinterMain` calls

Listing 4.11: Apache parameter with multiple values

```
DirectoryIndex index.html index.htm index.shtml
default.htm default.html index.php
```

Listing 4.12: XML element with a single subelement

```
<perspectiveBar>
  <itemSize x="196"/>
</perspectiveBar>
```

Listing 4.13: XML element with multiple elements

```
<coolbarLayout locked="0">
  <coolItem id="group.file"
  itemType="typeGroupMarker"/>
  <coolItem id="org.eclipse.ui.workbench.file"
  itemType= "typeToolBarContribution" x="104" y="30"/>
</coolbarLayout>
```

the `print` method with the root node's children, which is, in turn, a `NodeList`.

The first element in the XML files generated by the tool is always *Metadata*, containing the information about the original syntax. This element aids the printing process and is not part of the final configuration file, therefore it must not be printed. The remainder of the elements are then printed. XML elements generated by the tool always contain an *FStr* field, corresponding to the format string of that element, explained in Section 4.6. All elements in the document are visited and printed according to their format string. If the current element has other child nodes, they must be printed as well.

5

Framework Evaluation

Chapter 3 presented the proposed application configuration framework and Chapter 4 covered the process of implementing the framework. This chapter provides us with the validation of the tool, carried after the implementation phase had ceased.

First, Section 5.1 shows the functional validation of the tool by explaining how the requirements identified in Section 3.1 are met. Then, Section 5.2 presents the operational validation by checking the integrity of the tool and examining the configuration process step by step. Finally, Section 5.3 provides the performance validation by testing the tool behaviour in different scenarios (i.e., configuration of a very big and very small configuration file).

5.1 Functional Validation

In Section 3.1, some functional and non-functional requirements of the tool were identified. This section presents the ways in which these requirements are met.

1. The user must be able to convert a configuration file syntax from its original syntax to a generic one, independently of the application.

The user is able to summon the tool from a terminal and pass it multiple configuration file locations. The tool, in turn, produces abstract trees for the configuration files, and then the configuration files in a structured, generic format.

Non-functional requirements:

Performance: The generated file with the generic syntax must be as simple as possible.

The information regarding the original syntax is factorized by the code generator in a *Metadata* field.

2. The user must be able to define grammars for configuration file languages.

After the user submits a number of files to the tool, they are processed and, if not fully recognized, the user is able to manually define a grammar, or pass the tool a file containing the grammar for the generation of new parser.

Non-functional requirements:

Usability: The grammar definition syntax should be a broadly adopted one.

JavaCC grammar interpreter supports the Extended Backus-Naur Form and allows Java code, for more commodity.

Usability: To ease the parser generation process, the user must be able to iterate through the previously built grammars so as to roll back any change made on a grammar.

Every time the user tries a newly created parser on a configuration file, the grammar used for it is stored in a repository and can be recovered at any time.

3. The user must be able to produce a parser from a grammar.

As soon as the user finishes declaring a grammar, the latter is passed to the tool, which compiles it on an external parser generator.

Non-functional requirements:

Extensibility: The user must be able to add parsers built outside the tool to support other configuration file paradigms (e.g., Windows registry files, binary files, etc.).

The parser repository implements the `importParser` method which allows the user to store an external parser. However, the parser must implement the interface presented in Listing 3.8.

4. The user must have access to the parser compilation trace.

When a grammar is compiled, its compilation trace is shown in case there any error ocurred, or a success message is displayed, otherwise.

Non-functional requirements:

Usability: The error messages must clearly identify the source of the error.

The error messages contain the source of the compilation error.

5. The user must receive information relative to the grammar fitness with a given configuration file.

Any time a parser is tested with a configuration file, the percentage of parsed file with that parser is displayed.

Non-functional requirements:

Usability: The sections of the configuration file that were parsed and those that were not must be clearly identified.

When a parser is tested with a configuration file, the regions of the file recognized by it are highlighted in a graphical environment.

6. The user must be able to store a functional parser generated by the tool, in order to be used on later tool runs.

The user can trigger a `storeParser` command, which stores a parser in the parser repository definitively.

Non-functional requirements:

Security: The user must be able to see the parsing outcome in order to check if the parser is indeed operating as intended to.

The graphical user interface displays a user-friendly representation of the generic syntax so the user can confirm that the parser is indeed operating in the intended way.

7. The user must be able to reconvert a configuration file syntax from the generic syntax to the original one.

The user is able to summon the tool and pass it a tool-generated XML file to be converted into the original syntax, keeping its functionality.

8. File conversion must not eliminate comments

The built-in parsers preserve the comments of the original file. For new parsers, their grammars must contemplate the comments and not ignore them.

9. The user must be able to halt the tool execution at any point and continue the configuration process later.

When the user halts the tool execution, the parser repository and tentative grammar repository states are stored in order to be recovered later. If a valid version of the configuration file has been produced before the user halted the execution, the file is stored and can be recovered later.

10. The user must be able to manually delete parsers from the parser repository.

The parser repository implements the `delete` method which allows the user to delete an existing parser from the database, given its name.

5.2 Operational Validation

This section presents some tests carried on the tool, using various configuration file types, with the objective of specifying the configuration file intermediate states and determining the validity of the final configuration file, produced by the tool.

The framework provides built-in support for three major configuration file categories, identified in Section 1.3:

- Files composed of parameter blocks delimited only by a header, and comments (i.e., *INI*-like);
- Files composed of parameters, blocks delimited by a header and a footer and comments (i.e., *Apache*-like);

Listing 5.1: MySQL configuration file snippet

```
[mysqldump]
quick
quote-names
max_allowed_packet      = 16M

# The MySQL database server configuration file.

!includedir /etc/mysql/conf.d/
```

- Files composed of blocks delimited by dynamic headers and footers, and comments (i.e., *XML*-like).

Three tests were conducted, one with a file of each category. All the tests were ran on a system with the Intel Pentium Dual T3200 processor, 2 GB DDR2 main memory and Ubuntu Linux 9.04 operating system.

Each test consisted on configuring an application by invoking SmART with the application configuration file. The configuration files were converted in XML and then converted back to their original format. Between the tests, the file was not modified, since Section 3.2 showed that file modification is not part of this dissertation scope.

Due to the considerable length of each file, the tested configuration files are not integrally displayed in this section. Instead, they can be found in the SmART webpage (`http://asc.di.fct.unl.pt/SmART`). The examples refer to parts of the configuration file which were handpicked to display the tool behaviour in the presence of each pattern implemented by the file.

To check the differences between the original and final file, the *diff*[1] UNIX utility was used. *diff* is a file comparison utility that shows the differences between two text files, in terms of different lines. Whenever lines from both files differ, the result shows the divergent lines of the original configuration file, and then those from the generated configuration file.

5.2.1 *INI*-like Configuration Files

This test consisted on configuring the MySQL database management system. The MySQL configuration file format is a slightly altered version of the INI format, but is still recognized by the tool. The configuration file (Listing 5.1) implements parameters with zero or one assigned values, contained in blocks delimited by a static header (lines 1-4), comments (line 6) and special instructions (line 8). The complete configuration file is located at `http://asc.di.fct.unl.pt/SmART/my.cnf`. The file containing the code generated by the tool for this configuration file can be found at `http://asc.di.fct.unl.pt/SmART/ini.xml`.

The tool produced the following XML for the block (Listing 5.2), comment (Listing 5.3) and special instruction (Listing 5.4):

[1]`http://www.gnu.org/software/diffutils/diffutils.html`

Listing 5.2: MySQL block in XML

```
<Block name="mysqldump">
        <FStr>%m.start%a.name%m.end%n%c%c%c%c</FStr>
        <Parameter>
                <FStr>%e%n</FStr>
                <Key>quick</Key>
        </Parameter>
        <Parameter>
                <FStr>%e%n</FStr>
                <Key>quote-names</Key>
        </Parameter>
        <Parameter>
                <FStr>%e%m.equal%e%n</FStr>
                <Key>max_allowed_packet</Key>
                <Value>16M</Value>
        </Parameter>
        <EOL>
                <FStr>%n</FStr>
        </EOL>
</Block>
```

Listing 5.3: MySQL comment in XML

```
<Comment>
        <FStr>%m.start%e%n</FStr>
        <Text> The MySQL database server configuration file.</Text>
</Comment>
```

Listing 5.4: MySQL special instruction in XML

```
<Special>
        <FStr>%m.start%e%n</FStr>
        <Value>includedir /etc/mysql/conf.d/</Value>
</Special>
```

Listing 5.5: MySQL Metadata

```
<Metadata>
        <Comment>
                <start>#</start>
        </Comment>
        <Parameter>
                <equal>=</equal>
        </Parameter>
        <Block>
                <start>[</start>
                <end>]</end>
        </Block>
        <Special>
                <start>!</start>
        </Special>
</Metadata>
```

The *Metadata* generated by the tool is represented on Figure 5.5.

The file was re-converted to its original syntax. *diff* showed that the differences between the original and generated files resume to a few absent spaces from the first to the second. This happens since the parser ignores the space characters outside strings. As MySQL ignores the spaces in configuration files, the file functionality is maintained.

5.2.2 *Apache*-like Configuration Files

This test consisted on configuring the Apache HTTP server. The Apache configuration file (Listing 5.6) implements parameters with a single value (line 1), parameters with multiple values (lines 3-4), blocks (lines 6-8), nested blocks (10-14) and comments (line 16). The complete configuration file is located at `http://asc.di.fct.unl.pt/SmART/httpd.conf`. The file containing the code generated by the tool for this configuration file can be found at `http://asc.di.fct.unl.pt/SmART/blox.xml`.

The tool produced the following XML for the parameter with a single value (Listing 5.7), parameter with multiple values (Listing 5.8), block (Listing 5.9), nested (Listing 5.10) and comment (Listing 5.11):

The *Metadata* generated by the tool is represented on Figure 5.12.

The file was re-converted to its original syntax. *diff* showed that the differences between the original and generated files are some absent new lines and tabulations. Once again, this is due to the parser which ignores some new lines and tabulations. However, this does not affect in any way the functionality of the new file, so the result is valid.

Listing 5.6: Apache configuration file snippet

```
ServerType standalone

DirectoryIndex index.html index.htm index.shtml default.htm
default.html index.php

<IfModule mod_mime_magic.c>
  MIMEMagicFile /usr/local/apache/conf/magic
</IfModule>

<IfModule mod_alias.c>
  <Directory "/usr/local/apache/icons">
    Options Indexes MultiViews
  </Directory>
</IfModule>

# httpd.conf -- Apache HTTP server configuration file
```

Listing 5.7: Apache parameter with a single value in XML

```
<Parameter>
        <FStr>%e%m.equal%e</FStr>
        <Key>ServerType</Key>
        <Value>standalone</Value>
</Parameter>
```

Listing 5.8: Apache parameter with multiple values in XML

```
<Parameter>
        <FStr>%e%m.equal%e%m.equal%e%m.equal%e%m.equal%e%m.equal%e%m.
           equal%e</FStr>
        <Key>DirectoryIndex</Key>
        <Value>index.html</Value>
        <Value>index.htm</Value>
        <Value>index.shtml</Value>
        <Value>default.htm</Value>
        <Value>default.html</Value>
        <Value>index.php</Value>
</Parameter>
```

Listing 5.9: Apache block in XML

```
<Block>
        <FStr>%c%c%c%c%c</FStr>
        <Header name="IfModule mod_mime_magic.c">
                <FStr>%m.start%a.name%m.end</FStr>
        </Header>
        <EOL>
                <FStr>%n</FStr>f
        </EOL>
        <Parameter>
                <FStr>%e%m.equal%e</FStr>
                <Key>MIMEMagicFile</Key>
                <Value>/usr/local/apache/conf/magic</Value>
        </Parameter>
        <EOL>
                <FStr>%n</FStr>
        </EOL>
        <Footer name="IfModule">
                <FStr>%m.start%a.name%m.end</FStr>
        </Footer>
</Block>
```

5.2.3 *XML*-like Configuration Files

This test consisted on configuring the Eclipse workbench. The Eclipse workbench configuration (Listing 5.13) file is originally on XML format and implements empty elements/blocks (line 1), elements with attributes/blocks containing parameters (line 3-4), nested blocks (lines 6-8) and comments (line 10). The complete configuration file is located at `http://asc.di.fct.unl.pt/SmART/workbench.xml`. The file containing the code generated by the tool for this configuration file is found at `http://asc.di.fct.unl.pt/SmART/xml.xml`.

The tool produced the following XML for the empty block (Listing 5.14), block containing parameters (Listing 5.15), nested blocks (Listing 5.16) and comments (Listing 5.17):

The *Metadata* generated by the tool is represented on Listing 5.18.

Then, the file was re-converted to its original syntax. *diff* showed that the differences between the original and generated files are some extra new lines in the generated file. This is due to the original file having a dual criteria for new lines. In some places, an XML tag is followed by a new line, where in other places they are not.

Still, the tool is able to recognize the whole file, and it produces a file which is completely functional and well defined.

5.2.4 Parsing the Generated XML

In order to make sure that the XML file generated by the tool was indeed valid, an experiment was carried: parsing the file generated by the tool. Beforehand, the tool should be able to parse

Listing 5.10: Apache nested block in XML

```
<Block>
        <FStr>%c%c%c%c</FStr>
        <Header name="IfModule mod_alias.c">
                <FStr>%m.start%a.name%m.end</FStr>
        </Header>
        <EOL>
                <FStr>%n</FStr>
        </EOL>
        <Block>
                <FStr>%c%c%c%c%c</FStr>
                <Header name="Directory "/usr/local/apache/icons"">
                        <FStr>%m.start%a.name%m.end</FStr>
                </Header>
                <EOL>
                        <FStr>%n</FStr>
                </EOL>
                <Parameter>
                        <FStr>%e%m.equal%e%m.equal%e</FStr>
                        <Key>Options</Key>
                        <Value>Indexes</Value>
                        <Value>MultiViews</Value>
                </Parameter>
                <EOL>
                        <FStr>%n</FStr>
                </EOL>
                <Footer name="Directory">
                        <FStr>%m.start%a.name%m.end</FStr>
                </Footer>
        </Block>
        <Footer name="IfModule">
                <FStr>%m.start%a.name%m.end</FStr>
        </Footer>
</Block>
```

Listing 5.11: Apache comment

```
<Comment>
        <FStr>%m.start%e%m.end</FStr>
        <Text># httpd.conf -- Apache HTTP server configuration file</
           Text>
</Comment>
```

Listing 5.12: Apache Metadata

```
<Metadata>
        <Comment>
                <start>#</start>
                <end>\n</end>
        </Comment>
        <Parameter>
                <equal> </equal>
        </Parameter>
        <Header>
                <start><</start>
                <end>></end>
        </Header>
        <Footer>
                <start></</start>
                <end>></end>
        </Footer>
</Metadata>
```

Listing 5.13: Eclipse configuration file snippet

```
<workbenchAdvisor/>

<coolItem id="org.eclipse.ui.workbench.file"
itemType="typeToolBarContribution" x="104" y="30"/>

<perspectiveBar>
  <itemSize x="196"/>
</perspectiveBar>

<!--this is a comment-->
```

Listing 5.14: Eclipse empty block in XML

```
<Block name="workbenchAdvisor">
        <FStr>%m.start%a.name%m.end</FStr>
</Block>
```

Listing 5.15: Eclipse block with parameters in XML

```
<Block name="coolItem">
        <FStr>%m.start%a.name%s%c%s%c%s%c%s%c%m.end</FStr>
        <Parameter>
                <FStr>%e%m.equal%e</FStr>
                <Key>id</Key>
                <Value>"org.eclipse.ui.workbench.file"</Value>
        </Parameter>
        <Parameter>
                <FStr>%e%m.equal%e</FStr>
                <Key>itemType</Key>
                <Value>"typeToolBarContribution"</Value>
        </Parameter>
        <Parameter>
                <FStr>%e%m.equal%e</FStr>
                <Key>x</Key>
                <Value>"104"</Value>
        </Parameter>
        <Parameter>
                <FStr>%e%m.equal%e</FStr>
                <Key>y</Key>
                <Value>"30"</Value>
        </Parameter>
</Block>
```

Listing 5.16: Eclipse nested blocks in XML

```
<Block>
        <FStr>%c%c%c</FStr>
        <STag name="perspectiveBar">
                <FStr>%m.start%a.name%m.end</FStr>
        </STag>
        <Block name="itemSize">
                <FStr>%m.start%a.name%s%c%m.end</FStr>
                <Parameter>
                        <FStr>%e%m.equal%e</FStr>
                        <Key>x</Key>
                        <Value>"196"</Value>
                </Parameter>
        </Block>
        <ETag name="perspectiveBar">
                <FStr>%m.start%a.name%m.end</FStr>
        </ETag>
</Block>
```

Listing 5.17: Eclipse comment in XML

```
<Comment>
<FStr>%m.start%e%m.end</FStr>
<Text>this is a comment</Text>
</Comment>
```

Listing 5.18: Eclipse Metadata

```
<Metadata>
        <Prolog>
                <start><?xml</start>
                <end>?></end>
        </Prolog>
        <Parameter>
                <equal>=</equal>
        </Parameter>
        <STag>
                <start><</start>
                <end>></end>
        </STag>
        <ETag>
                <start></</start>
                <end>></end>
        </ETag>
        <Comment>
                <start><!--</start>
                <end>--></end>
        </Comment>
        <Block>
                <start><</start>
                <end>/></end>
        </Block>
</Metadata>
```

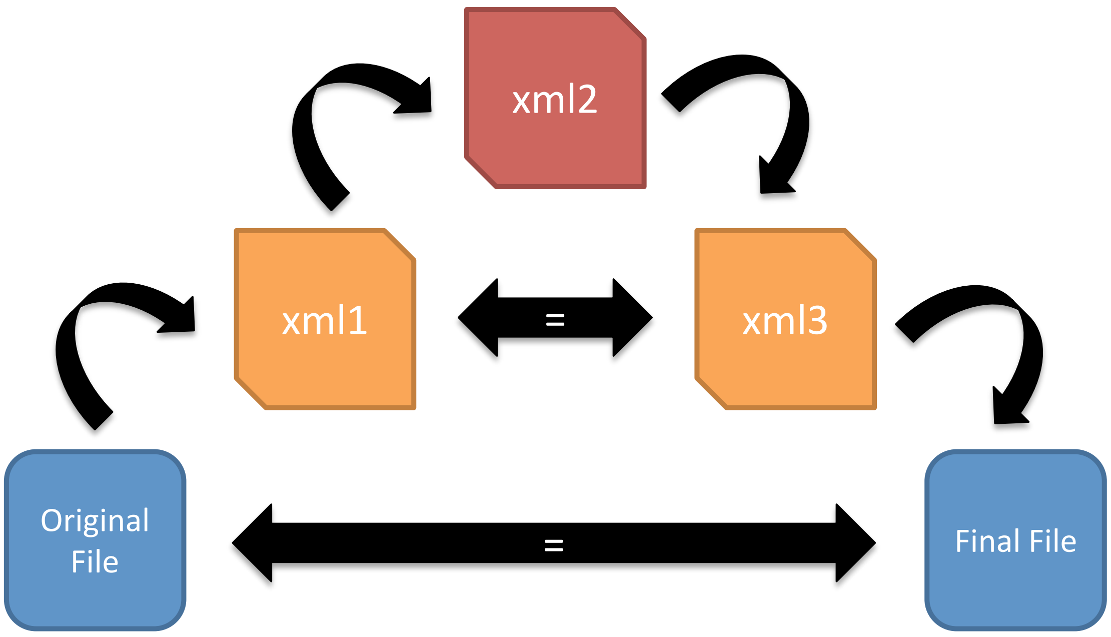

Figure 5.1: Parsing the generated XML

the generated XML file since it is able to parse files resemblant to XML.

First, OGC (Original to Generic syntax Converter) was called to parse a configuration file. From this parse resulted the file *xml1*. Then, OGC was called again to parse *xml1* into *xml2*. Then, GOC (Generic to Original syntax Converter) was called to print *xml2* into *xml3*, and finally GOC was called again to print *xml3* into the *final* file. This is depicted in Figure 5.1.

Before the test, *xml1* was expected to be equal to *xml3* and the original file to be equal to the *final* file. *diff* showed that *xml1* was exactly equal to *xml3* and that the original file only differed from *final* in the sense that only a few new lines were omitted from the original file, analogously to the test in Section 5.2.3. Still, the final file remains recognizable by XML parsers, since they also ignore new lines. Moreover, the tool was shown to produce valid XML files since it is capable of parsing them over again.

5.3 Performance Validation

This section studies how the tool behaves in the presence of configuration files with different sizes. Despite not having been designed to excel performance-wise, obtained performance results by the tool might turn out to be interesting.

The tested configuration files are from PostgreSQL and are all recognized by an *INI*-like parser. These are:

- file α with 1,4 KB (`http://asc.di.fct.unl.pt/SmART/pg_ident.conf`)
- file β with 3,5 KB (`http://asc.di.fct.unl.pt/SmART/pg_hba.conf`)

- file γ with 16,6 KB (http://asc.di.fct.unl.pt/SmART/postgresql.conf)

These files were selected to verify if there is any relation between the file size and the time it takes to configure each one. The final results are represented as the sum of two times: the time to convert the file into generic XML, step 1, and the time to convert the file from XML into its original syntax, step 2.

File α:

Step 1: 0,143 s

Step 2: 0,073 s

Total time: 0,216 s

File β:

Step 1: 0,191 s

Step 2: 0,086 s

Total time: 0,277 s

File γ:

Step 1: 0,463 s

Step 2: 0,262 s

Total time: 0,725 s

This test shows that for short configuration files, the elapsed times to convert a file to XML and back to its original format are similar. An interpretation of these results might be that the configuration work done by the tool is very small, whereas the majority of the elapsed time was spent doing computational tasks such as memory allocations, etc. However, as the files become bigger, so does the time to convert them, although in a disproportionate way. γ is almost 5 times bigger than β, nevertheless, γ only takes 2.5 times more than β to be converted in both ways.

Taking into account the average file size of VIRTU's use case application configuration files, which is nearly 20KB, we can also conclude that the time spent to configure a configuration file is sustainable. Nevertheless, the tool may require considerable amounts of time to carry certain tasks, like parser creation. During implementation, we realised that some parsers took a while to compile while others were compiled almost instantly. Following is an analysis to the JavaCC parser generator which, in spite of approaching elements that fall out of this dissertation scope, aims at comprehending the impact of the parser generation process in the tool's performance. In this analysis, we compile the three parsers that the tool supports by default and present the elapsed times for each (Table 5.1).

INI	*Apache*	*XML*
0,270s	0,280s	0,300s

Table 5.1: Elapsed times for parser compilation in JavaCC

An interpretation of the obtained times might tell us that the needed time to compile a parser is also sustainable. However, the compiled parsers are not too complex and basing any conclusion exclusively on these timings may be misleading. To assess more accurately the required time for parser generation tasks, we compiled a parser which recognizes a more complex language, the Java language, and the elapsed time was 0,550s. This allows us to conclude that, even for fairly complex languages, the parser generation time is very bearable. Nonetheless, these timings refer to the JavaCC parser generator, as we also generated some parsers with SableCC and, although most parsers took as long as JavaCC's to compile, parsers for very complex languages can take up to minutes to compile.

6 Integration with VIRTU

The last chapter validated the framework, defined in Chapter 3, in terms of how it met the requirements, its functionality and also its performance. We now find SmART suitable to be integrated with the VIRTU virtualization platform.

This chapter describes how SmART can be integrated within the VIRTU tool. First, Section 6.1 gives the background of the VIRTU project, together with the motivation for its creation and a brief description. Then, Section 6.2 presents the VIRTU workflow with the objective of giving a better perspective of the operations allowed by the tool. The VIRTU architecture is detailed in Section 6.3, with the description of each component and its interactions with the other components. Finally, the localization of the integration, both in the workflow and in the architecture, is indicated and explained in Section 6.4.

6.1 Project Description

A consortium of enterprises and universities, composed of Evolve Space Solutions, Universidade Nova de Lisboa, Universidade de Coimbra, HP Labs and the European Space Agency, funded by QREN/ADI, came together to develop a virtualization platform known as project VIRTU.

In Section 2.1.1 we saw that virtualization is a technology for the abstraction of computing systems which has recently received a considerable number of contributions, despite having emerged for the first time several years ago. The changes that virtualization allows for in the IT sector, such as the consolidation of many servers into a single physical machine and the resulting lowering of hardware, cooling, power consumption and facility costs, short system maintenance time, great isolation level between virtual machines, etc., have revolutionized the way IT-recurring companies look at software and hardware deployment.

Aware of this market shift, Evolve Space Solutions proposed VIRTU, a platform for the creation and management of virtual machines. Before that, the GoVI virtualization tool was developed for the European Space Agency in order to deal with some outstanding problems in their IT infrastructure, such as the large number of underutilized machines, great demand for isolation and a myriad of different hardware platforms. The development on VIRTU initially took on the GoVI infrastructure as a starting point, but eventually evolved to become an independent project, dettached from the former.

VIRTU explores interoperability between virtualization solutions, such as Xen [Cit] or VMware [VMwa]. One way to achieve portability of VMs among different virtualization platforms is to resort to virtualization standards. Open Virtual Machine Format (see Section 2.1.3), which is being developed by the biggest virtualization players, is one of such standards and VIRTU might evolve to support it.

Another big opportunity in the virtualization area is the fact that application configuration is still largely a manual task. The complexity of configuring a system with manifold applications, employing different configuration formats, can be frightening for a system administrator. This complexity is only magnified when dealing with a huge system, composed of hundreds, or even thousands of such systems. To ease the system maintenance burden, VIRTU proposes the generic automatic application configuration or, in other words, the ability to configure any application, regardless of its vendor or configuration representation, on the most autonomous and automatic manner.

The framework presented by this dissertation, SmART, is the Universidade Nova de Lisboa contribution for project VIRTU. Once SmART was completed, it was integrated with the whole VIRTU tool.

6.2 VIRTU Top-Level Analysis

The VIRTU tool [Solb] implements the concept of `Assembly Instance`, which refers to one, or a set of virtual machines that the users will use and manage directly. An `Assembly Instance` results from the instantiation of an `Assembly Configuration`, which contains the information used to generate one VM, or a set of VMs. To each VM in an `Assembly Configuration` corresponds a `Virtual Machine Configuration` and when an `Assembly Configuration` is instanced, its `VM Configurations` are also instanced into `VM Instances`. Each `Virtual Machine Configuration` is a combination of `Building Blocks` and `Publication Files`. A `Building Block` is a template element, such as an operating system, application or virtual machine, whereas a `Publication File` contains the desired configuration of a `Building Block`. In order to re-use previously existing `Assembly Configurations`, it is also possible for `Assembly Configurations` to contain other `Assembly Configurations`. The VIRTU `Assembly Configuration` model is depicted in Figure 6.1.

The configuration of an `Assembly Instance` is carried at the level of its `VM Instances`. Inside a `VM Instance` resides a process which is ran at every VM boot and, for each

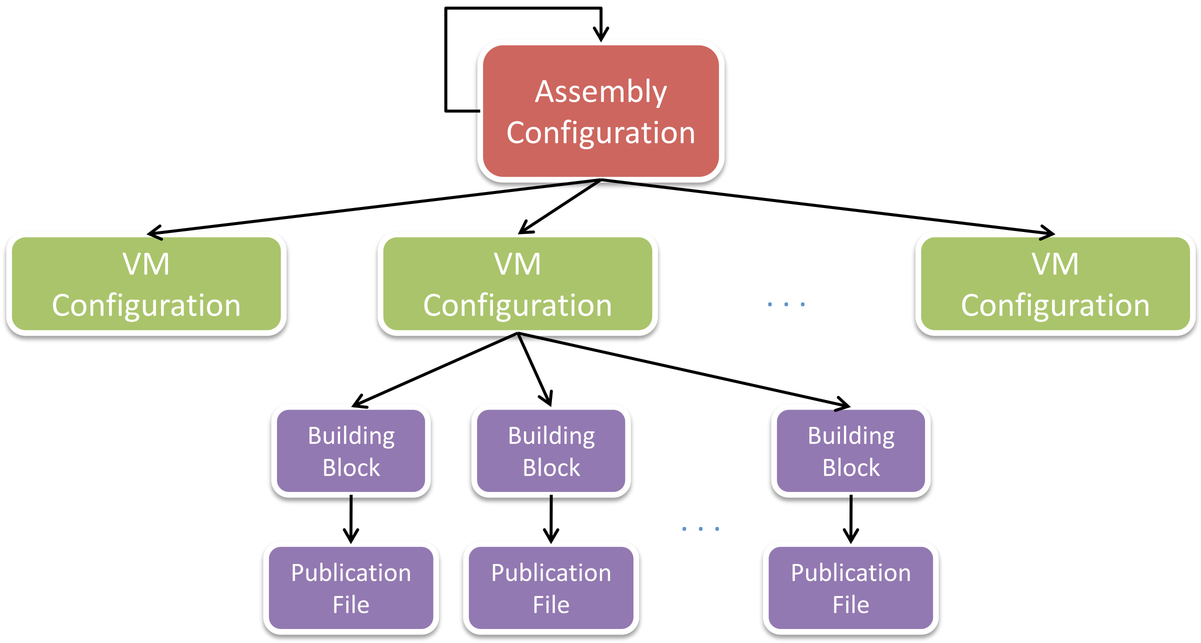

Figure 6.1: VIRTU Assembly Configuration

`Publication File`, checks a flag that tells if that `Publication File` was altered since the last boot. If not, no action is taken for that `Building Block`. Otherwise, if a `Publication File` is indeed different, it means the `Building Block` associated to that `Publication File` has altered variables or different operating modes and, consequently, the configuration files for that `Building Block` must be rebuilt. This rebuilding is done by the process itself and once it is finished, the flag is reset.

The actors in the VIRTU tool can be administrators or users. Each role is allowed to perform different actions that may be divided in two categories: virtual machine management and user management.

An administrator has the permission to manage (i.e., create, edit and delete) the tool's `Building Blocks` and `Publication Files`, to construct `Assembly Configurations`, to accept `Assembly Instance` requests from the user and deploy those `Assembly Instances` and to manage the running `Instances`, with the option of halting their execution. The administrator may also create and manage the users of the tool.

On the other hand, the actions available to users are to configure `Publication Files`, to request `Assembly Instances`, to use the virtual machines and control their lifecycle, to share `Assembly Instances` with other users and to organize them. Beyond that, the user may also change his password.

On a broad view, VIRTU may be divided into a handful of logical modules. This division aims at increasing agility and flexibility in the maintenance and optimization of the system resources. A scheme containing the modules is shown in Figure 6.2.

The VIRTU `Configuration Database` stores all information regarding the tool, such as user accounts, the locations of the available `Building Blocks`, `Assembly`

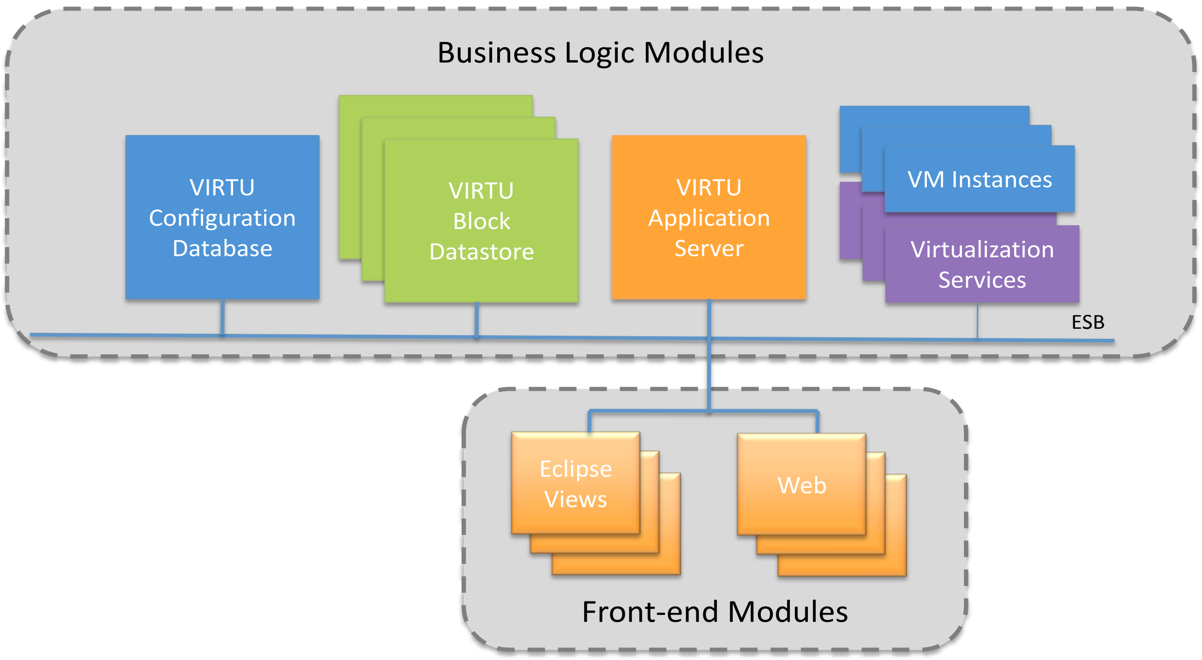

Figure 6.2: VIRTU Logical View

`Configurations` and `Instances` and users reports and suggestions. The VIRTU `Block Datastore` is the physical storage for the available `Building Blocks`. The VIRTU `Application Server` provides access to the tool for users and administrators, together with the processing logic. The `Virtualization Services` is where the virtualization is handled. It contains the virtualization servers and the existing virtual machines. Finally, the `Eclipse Views` and `Web Front-End` are interfaces used to allow interaction with the user through an application or a typical web browser.

6.3 VIRTU Architecture

The VIRTU tool arcitecture is structured in a three-tier architecture, as depicted in Figure 6.3. Following is description of each layer and the respective components.

6.3.1 Data and Resources Layer

The `Data and Resources Layer` provides the storage utility for the system and hosts the system's virtualization layer. It is composed by the `Block Datastore`, the `Configuration Database` and `Virtualization Services`.

Block Datastore

The `Block Datastore` is an abstraction for the physical storage of the `Building Blocks`. It may have multiple instances and can be distributed over different physical locations. It allows `Building Blocks` from another system to be uploaded, to create, modify and delete

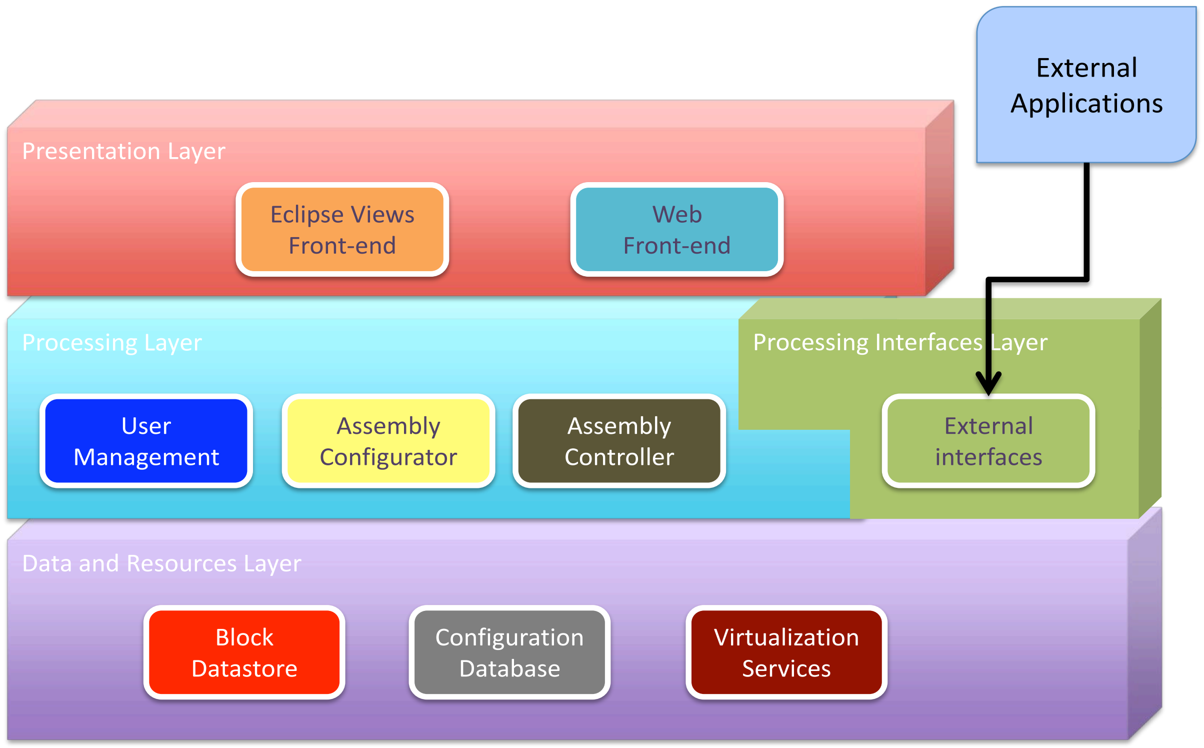

Figure 6.3: VIRTU Architectural Design

`Building Blocks` and to query for `Building Blocks`, although the `Building Block` locations are stored in the `Configuration Database`.

The `Building Blocks` stored by the `Block Datastore` may contain:

- information relative to the configuration of the virtual machine;
- a virtual hard disk containing a fresh installation of an operating system and a first-run script;
- the virtual machine hard disk for the storage of data;
- an application block, which can be a deployed application or an installer.

Configuration Database

The `Configuration Database` is a relational database which contains information relative to the VIRTU tool, such as the existing users, the existing `Assembly Configurations` and `Assembly Instances`. It is comprised of a database server to support the relational database and VIRTU DB, an Eclipse Rich Client Platform (RCP) [Fouc] plugin used to establish a bridge between the `Configuration Database` and the `Processing Layer` by directly handling database operations. The Eclipse Rich Client Platform is a set of tools which provides tested features such as a lifecycle manager, text editors and a workbench with views, perspectives, wizards, etc., and which can be used to start building an application.

It provides a number of features for the management of the tool information by interacting with other components. The User Management performs user (creation, edition, deletion), password (change) and role (set user, administrator) management on it, the `Assembly Configurator` uses it to manage the available `Publication Files`, `Building Blocks`, `VM Configurations` and `Assembly Configurations` and, finally, the `Assembly Controller` accesses the `Assembly Instances` in run-time from it.

Virtualization Services

The `Virtualization Services` contain the system's virtualization layer which provides the virtualization foundations for the running virtual machines, together with each virtual machine instance hosted by the tool. It provides the basic VM operations such as start, stop, pause, create, delete, etc., and the support to configure and install applications at first-run time.

6.3.2 Processing Layer

The `Processing Layer` is where all the tool operations are ran and the tool workflow is implemented and managed.

User Management

The `User Management` is the component which controls the user-oriented operations. It handles operations like login checking, password changes, role attributions and creation/deletion of users. The access to the `User Management` is provided by the `Eclipse Views Front-End` and the `External Interfaces`, in case of a stand-alone application or a web-service access, respectively. The effects of the operations are reflected in the `Configuration Database`, under the user related tables.

Assembly Configurator

The `Assembly Configurator` allows the administrators to create, edit or delete `Assembly Configurations` and the `Building Blocks` and `Publication Files` that compose those Assemblies. As for the user, he may search through the existing `Assembly Configurations`, request instances of some of them and edit `Publication Files` associated to those instances. Finally, the `Assembly Configurator` also reports bugs to the administrator.

The access to the `Assembly Configurator` is provided by the `Eclipse Views Front -End` and the `External Interfaces`. The `Assembly Configurator` interacts with the `Configuration Database` to manage the Assembly definitions and the `Block Datastore` to manage the `Building Blocks`.

Assembly Controller

The Assembly Controller allows the administrator to control Assembly Instances in run-time and accept or decline user requests for new Assembly Instances, as well as delete the existing ones. The user may, in turn, request new Assembly Instances, share Instances with other users and organize them, access the Instances (i.e., through Remote Desktop) and control them.

Interaction with the user resorts to the Eclipse Views Front-End and the External Interfaces. The Assembly Controller accesses the Configuration Database to get any Assembly Instance state on any time, the Block Datastore to request physical Building Block deployment and the Virtualization Services to perform basic VM management operations.

External Interfaces

The External Interfaces allow the interaction of the tool with other existing infrastructures. It defines a set of components for the communication with the Processing Layer . The External Interfaces interactions cover every component in the Processing Layer since this is the component which allows the integration with third-party elements. The User Management is used for password, login and role management, the Assembly Configurator is used for the management of Assembly Configurations, the Assembly Controller is used for VM lifecycle management and remote connection operations and the Web Front-End and the External Applications are used to allow the access to the tool from a web-browser or a third-party infrastructure.

6.3.3 Presentation Layer

The Presentation Layer contains the elements which allow the interaction with users and administrators.

Eclipse Views Front-End

The Eclipse Views Front-End provides access to the tool through a stand-alone application, based on the Eclipse RCP, which supplies views and perspectives corresponding the functionalities available to users and administrators. This component interacts with all other components in the Processing Layer to reflect the users' and administrators' requests.

Web Front-End

The Web Front-End objective is to allow interaction with the tool by means of a regular web browser, requiring no special applications to be installed in order to communicate with the tool and allowing access from devices other than typical computers, such as smart-phones or thin-clients. Unlike Eclipse Views Front-End which communicates directly with the

`Processing Layer`, the `Web Front-End` uses the `External Interfaces` component as an interface to the tool.

6.3.4 Other Notable Sectors

Apart from the presented layers and components, VIRTU makes use of other sectors that are not included in the tool architecture, but their relevance is considerable enough for the integration of SmART with VIRTU.

The VIRTU tool employs some third-party software which is also integrated with the tool, as is the case of MySQL for the provision of a relational database management system or Eclipse for the linkage of Java objects to the database. These components are COTS (Commercial, off-the-shelf), software built by other entities which can be integrated with the objective of quickly providing functionalities to the tool, possibly saving time by removing the need to build those functionalities from scratch, but at the cost of some integration work. VIRTU keeps these third-party components in a CVS (Concurrent Versions System) repository, from where they can be accessed.

Another important data sector is located inside the virtual machines created by the tool. This sector hosts the first-run and configuration scripts which are ran automatically by each virtual machine so that they get the required data from the tool autonomously.

6.4 SmART integration

The VIRTU workflow and architecture were described in the previous sections. Using that knowledge, this section describes how SmART fits in the VIRTU workflow and the whereabouts of the SmART component placement in the VIRTU architecture. The integration takes on both SmART major components, the Original to Generic syntax Converter (OGC) and the Generic to Original syntax Converter (GOC, see Section 3.2), and employs them on separate VIRTU components. OGC is used in the creation of configuration file templates when adding new Building Blocks to VIRTU while GOC is used by the virtual machines to generate configuration files from templates.

In the VIRTU workflow, SmART will be called for the first time when an administrator adds new `Building Blocks`. As it was seen in Section 6.3.1, these can either be applications, operating systems or virtual machines. In this case, a `Publication File` needs to be associated to that `Building Block`. For this endeavour, SmART is used to transpose the `Building Block` configuration files, whose parameterization is relevant, into XML documents, generating a `Publication File`. This `Publication File` is then stored in the `Configuration Database`, from where it is obtained when its usage is required (i.e., when a user requires the instantiation of an `Assembly Configuration` composed by that `Building Block`). There might be the case where one or more configuration files of an application are unknown to the tool. If such is the occasion, the administrator must define a grammar which recognizes the new configuration file in order to build a parser for it. In the end, a new `Building`

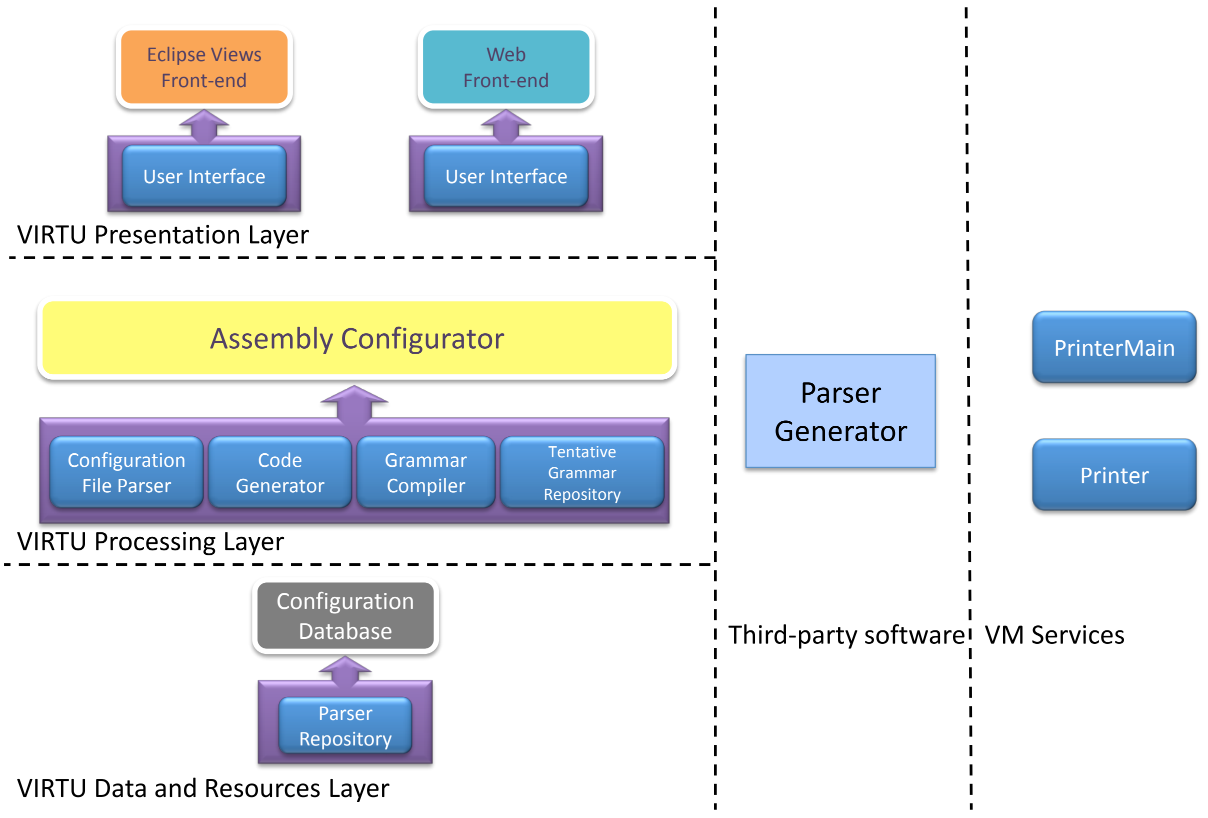

Figure 6.4: SmART integration points with VIRTU

`Block` is stored in the `Block Datastore` and a `Publication File` is associated to it in the `Configuration Database`.

The second usage of SmART in the VIRTU tool is for the generation of configuration files from `Publication Files`. This is the case when users request `Assembly Instances` of existing `Assembly Configurations`. After the request is made, the user may parameterize the application by choosing the variable values or mode of operation of a `Building Block` by means of a graphical interface which triggers a script that alters the `Publication File` associated to that `Building Block`. When a `VM Instance` boots, it runs a script which checks if the configurations are up to date by retrieving the `Publication Files` it uses from the `Configuration Database` and reading their flag values. If a `Publication File` was meanwhile altered, SmART is called to produce a configuration file from it.

Figure 6.4 sketches the integration of SmART from the point of view of the VIRTU tool. As Section 3.2 points out, SmART is composed by two major components, the OGC and the GOC. Section 3.3 shows that the sub-components of the OGC are:

- `User Interface;`
- `Configuration File Parser;`

- `Parser Repository`;
- `Code Generator`;
- `Grammar Compiler`;
- `Tentative Grammar Repository`.

and the GOC is composed by `Printer` and `PrinterMain`.

As previously mentioned, the two components perform disparate functions. The subcomponents in OGC will be merged with the `Assembly Configuration` creation part whereas GOC will integrate the configuration file generation part. Following is deeper explanation of the integration of the SmART components in the VIRTU tool.

6.4.1 SmART Original to Generic syntax Converter

SmART Parser Repository

The lowest level integration will be that of the `Parser Repository`. After the parsers are created, they are inserted in the `Configuration Database`. This calls for the alteration of the relational database by adding parser-related relations to it and the alteration of VIRTU DB in order for the latter to support the necessary database operations for the creation, edition and deletion of parsers. The concept of `Parser Repository` will, therefore, refer to a set of database relations which can be accessed by other components through the VIRTU DB software layer.

SmART Processing Layer and Tentative Grammar Repository

Most of SmART's core will be integrated with the VIRTU `Assembly Configurator`. Namely, SmART OGC's `Configuration File Parser`, `Code Generator`, `Grammar Compiler` and also `Tentative Grammar Repository` are due to be located under the `Assembly Configurator`. The reason for the localization of this merge is, as it was seen earlier in this section, the fact that SmART OGC is only accessed by the VIRTU tool on the process of creating new `Assembly Configurations`.

In this way, the normal SmART workflow remains unchanged, apart from some small adjustments. For instance, the `Configuration File Parser` will now receive parse requests from the VIRTU `Presentation Layer` and get the parsers by issuing parser queries to the `Data Layer`, by means of VIRTU DB. The `Code Generator` will remain intact, since it only interacts with the `Configuration File Parser`. The `Grammar Compiler` will now access the parser generator from the VIRTU third-party software sector. As for the `Tentative Grammar Repository`, its integration with the VIRTU tool was found to be more suitable to the VIRTU `Processing Layer`, opposed to the SmART `Storage Layer`, since it will not store permanently any grammar, but rather act as a grammar browser, which saves the final grammars with the corresponding parsers and discards the rest. Besides that, the `Tentative Grammar Repository` will now interact with the `Eclipse Views` and the `Web Front-Ends`.

SmART User Interface

The `User Interface` will be integrated in the `Presentation Layer` with the `Eclipse Views Front-End` and the `Web Front-End`, so as to provide a way for the administrator to perform the tasks of creating the `Publication Files`, when new `Building Blocks` are added to VIRTU, and building new grammars when parsers capable of recognizing the configuration files of the `Building Blocks` do not exist in the database.

6.4.2 SmART Generic to Original syntax Converter

As for `Printer` and `PrinterMain`, they will form an autonomous process in the boot sector of each virtual machine, which transforms a `Publication File` on a configuration file, if that `Publication File` modification flag is set.

6.4.3 Parser Generator

Much like in SmART, the `Parser Generator` component will be considered as third-party software, therefore, this component will reside as an external package in the third-party software repository. It is accessed solely by the `Grammar Compiler`.

6.5 Summary

At the time of elaboration of this dissertation, the integration of SmART with VIRTU was only commencing. This chapter presents the result of project meetings where we analysed ways in which both tools can be merged, from the point of view of the VIRTU tool. The taken approximation consists on joining the SmART components with the existing VIRTU ones.

By merging the prototype of a framework on a bigger tool whose development is well under way, the adaptation of some prototype components should be expected. This is the case of the SmART `Parser Repository`, which will be tranformed from a simple list of parsers into a set of relations in the VIRTU `Configuration Database`. This adaptation is not straightforward though, as with most of the available parser generators, the produced parsers will consist on a set of Java classes. The classes that compose the parser can be stored in the database without any hassle, nevertheless, they cannot be executed from there. Three alternatives for the execution of the parser classes were identified: to copy the classes locally to a temporary folder and execute them; to copy the classes to memory and run them from there; to execute a database method which allows the parser classes to be ran from the database. After the project meeting, another solution for this problem was identified, which consists on gathering the parser classes in a .JAR container [Micd] and storing the packages in the database.

In this way, each class in the package knows the exact location of the others and is always able to call them.

There are still two components left to implement: the graphical interface for the `User Interface` and the configurator.

The first consists on a graphical interface which allows the administrator to define new grammars, in a practical and efficient way. It should provide the administrator with functionalities such as configuration file syntax highlight, new pattern definition, etc. The SmART prototype offers the necessary functionalities for the implementation of this component, therefore, the remaining work consists on designing the graphical interface and linking it with the other SmART components.

The implementation of the configurator involves the design of an interface from where the user can alter the configuration file parameters. This interface should also generate a script which recognizes the XML generic syntax. That script is used to traverse a Publication File, applying the required changes as it comes upon the right parameters, consequently generating a new Publication File containing the desired parameterization.

Apart from the previous considerations, the integration process should unfold in a seamless way, providing the VIRTU tool with the ability to automatically configure applications, regardless of their vendors.

7 Conclusion and Future work

The concepts of virtualization and, more particularly, virtual appliance, rose to significant importance in the last few years. Virtualization allows for the consolidation of many underutilized servers into fewer machines, increased infrastructural utilization, easy replication and replication of virtual machines, among many other benefits. Virtual appliances are a consequence of the virtualization advancements and refer to VMs composed solely of a minimal OS and a set of applications, optimized to perform a specific computational task.

The VIRTU project is a collective effort by a consortium of enterprises and universities, whose objective is to develop a platform for the creation and management of virtual appliances. Nowadays, the offering of virtualization solutions is increasing each day, but the configuration of virtual appliances is still a mainly manual task. Since a single machine tends to garner many VAs, configuring them in an effective way has become a problem. VIRTU tries to exploit this opportunity by offering on-demand configuration of applications, independently of their vendor.

This thesis tackles the problem of automatic configuration of applications, regardless of their vendors, in virtual appliances. The automatic configuration of applications is important since it allows for the reduction of required time to configure applications. It assumes a special importance if we consider a business scenario, where much time is wasted in configuration duties. A hurdle for the application configuration, regardless of the vendor, is that nowadays, applications already represent their configurations on widely adopted standards, although many of those standards exist and the majority of the applications resort to vendor-specific formats. The aim of this work consists on creating a framework that allows the abstraction of the used formats, allowing for the configuration process to be the same for every application, as well as automating that process from it results the least human interaction.

At first, two possible ways to deal with this subject were approached: configure the application pre or post installation on the virtual appliance. Configuring the application before its installation on the virtual appliance consists on altering the configuration file in the application package and then install it on the virtual appliance. This approach was seen to lead to package redundancy, since the number of required different configuration will equate to the number of application packages to be installed. Moreover, in case of further configurations of an application, this approach requires a package to be re-installed on another VA.

A better approach is the configuration post installation. In this approach, an application configuration file is extracted from inside a VA and configured by the tool, only to be merged with the VA in the end of the configuration process. Although this approach requires knowledge of the application location, it allows for easy configuration of any number of virtual appliances and, additionally, it does not force an application to be reinstalled in case of a reconfiguration.

The Smart Application Reconfiguration Tool (SmART) is the major contribution of this thesis for the automatic configuration of applications problem. It consists of a framework with several components that interact between themselves in order to configure applications by abstracting configuration files from their format details onto a generic structure. This framework is extensible on the grounds that the interfaces for each component are presented and detailed. This makes it possible for the rebuilding of any of the tool's components without any effect on the remaining components.

Still on this dissertation scope, an implementation of the framework was carried and its documentation is also made available through this dissertation. The resulting prototype proved that taking advantage of the frequently found patterns present in configuration files is a valid concept. This proof assumes the form of a validation in three criteria: requirement, functionality and performance. The fact that no other similar tools to this one are available not only made it harder to pioneer some concepts, it also meant that, much to our dismay, no comparative validation could be made.

Our interpretation of the validation results is a positive one. On one side, the framework is shown to fulfill every requirement towards which it was designed. In terms of functionality, SmART is proven to be able to configure files that fall on three categories: INI-like, XML-like and Apache-like files. These categories, in turn, cover the vast majority of existing configuration files. Regarding the other files that are currently not supported by the tool, we believe that the subsequent work with the framework by the other Project VIRTU players may be very interesting since it will reveal the framework's power to be fully extensible to other file formats. Finally, the performance results, although assuming a lesser importance in the framework validity assessment, reveal that the necessary time to configure a file using SmART is beyond tolerable and, furthermore, show that our choice of parser generator implies no big overhead to the tool.

With the concept proved and having developed a functional prototype, the integration of the SmART tool with the VIRTU tool proceeded. This process is intended to provide VIRTU with the ability to automatically configure applications, regardless of their vendors. To complement this dissertation with reliable information about VIRTU, we spent two weeks with the Evolve team. The integrational process only started on the final term of the elaboration of the dissertation. Therefore, we can only offer a suggestion of how to execute the integration process, meanwhile identifying the possible obstacles that might occur.

7.1 Future Work

From the integration with VIRTU will result an important improvement to the tool: a graphical user interface. This interface aims at making the grammar definition process more natural to the user, sparing him from knowing any JavaCC syntax. The framework implementation already offers the functionalities for an implementation of such interface, therefore, this task will revolve around the design and linkage of the interface with the tool.

A suggestion for a future improvement on the tool is for it to infer the meanings of different patterns in new configuration file languages on its own, using grammar inference. Grammar inference [Hon], also known as automata induction, grammar induction and automatic language acquisition, is a field of research with the purpose of learning grammars and languages from data. It can be applied on the fields of syntatic pattern recognition, adaptative intelligent agents, diagnosis, computational biology, systems modelling, prediction, natural language acquisition, data mining and knowledge discovery. According to [PH01], it is possible to learn grammars from simple examples, like sparse configuration files.

There is active work on this subject, namely the GenParse project, which is a collaboration among the University of Alabama-Birmingham, U.S.A. and the University of Maribor, Slovenia. It aims at easing the development of Domain-Specific Languages and also at developing renovation tools for legacy systems. It makes use of the Genetic Programming paradigm to infer Context-Free Grammars.

A

Appendix - Architecture

A.1 OGC sequence diagram

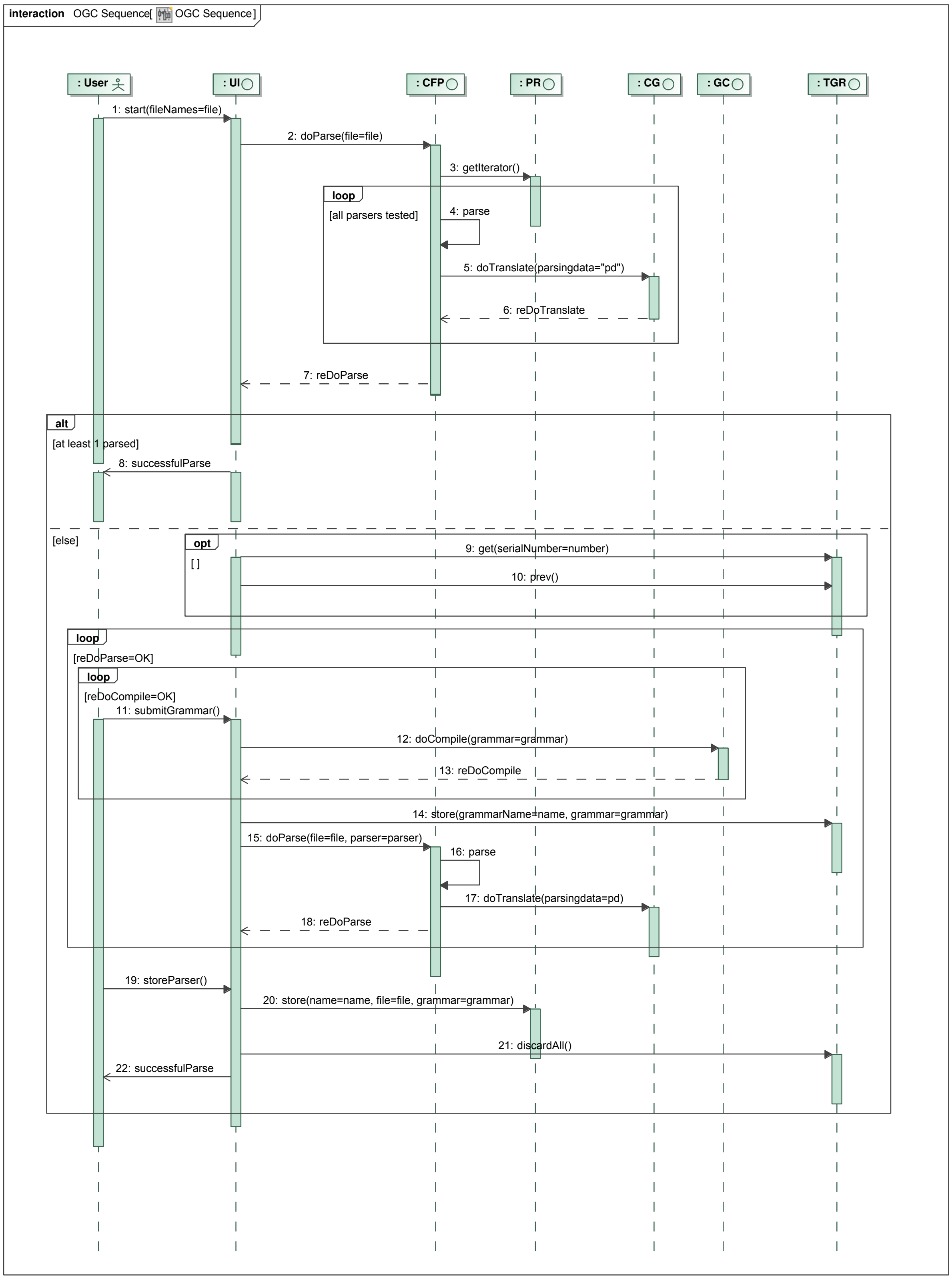

Figure A.1: Original to generic syntax converter sequence diagram

Bibliography

[Ama] Amazon. Amazon elastic compute cloud (amazon ec2). World Wide Web, http://aws.amazon.com/ec2/.

[Cit] Citrix. Xen source: Open source virtualization. World Wide Web, http://www.xensource.com.

[Con] World Wide Web Consortium. Extensible markup language. World Wide Web, http://www.w3.org/XML/.

[Cru] Thin Crust. Thin crust main page. World Wide Web, http://www.thincrust.net/.

[DMT08] DMTF. Open virtualization format specificiation. World Wide Web, http://www.dmtf.org/standards/published_documents/DSP0243_1.0.0.pdf, 2008.

[Foua] The Apache Software Foundation. Apache homepage. World Wide Web, http://www.apache.org/.

[Foub] The Eclipse Foundation. Eclipse homepage. World Wide Web, http://www.eclipse.org/.

[Fouc] The Eclipse Foundation. Rich client platform. World Wide Web, http://wiki.eclipse.org/index.php/Rich_Client_Platform.

[FSF] Inc. Free Software Foundation. Bison - gnu parser generator. World Wide Web, http://www.gnu.org/software/bison/.

[GGL$^+$09] Patrick Goldsack, Julio Guijarro, Steve Loughran, Alistair Coles, Andrew Farrell, Antonio Lain, Paul Murray, and Peter Toft. The smartfrog configuration management framework. *SIGOPS Oper. Syst. Rev.*, 43(1):16–25, 2009.

[GH98] Etienne M. Gagnon and Laurie J. Hendren. Sablecc, an object-oriented compiler framework. *Technology of Object-Oriented Languages, International Conference on*, 0:140, 1998.

[Groa] Mantis PHP Bug Tracker Development Group. Mantis bug tracker homepage. World Wide Web, http://www.mantisbt.org/.

[Grob] PostgreSQL Global Development Group. Postgresql homepage. World Wide Web, http://www.postgresql.org/.

[Hon] Vasant Honavar. Automata induction, grammar inference, and language acquisition. World Wide Web, http://www.cs.iastate.edu/~honavar/ailab/projects/grammar.html.

[Mica] Microsoft. Microsoft virtualization. World Wide Web, http://www.microsoft.com/virtualization/default.mspx.

[Micb] Sun Microsystems. Mysql homepage. World Wide Web, http://www.mysql.com/.

[Micc] Sun Microsystems. Sun virtualization. World Wide Web, http://www.sun.com/solutions/virtualization/index.jsp.

[Micd] Sun Microsystems. Using jar files. World Wide Web, http://java.sun.com/developer/Books/javaprogramming/JAR/basics/.

[oM] University of Maryland. Lecture: Top-down parsers. http://www.cs.umd.edu/class/fall2002/cmsc430/lec4.pdf.

[oV08] History of Virtualization. Vmware. World Wide Web, http://www.vmware.com/technology/history.html, 2008.

[Par] Parallels. Parallels homepage. World Wide Web, http://www.parallels.com/.

[PH01] Rajesh Parekh and Vasant G. Honavar. Learning dfa from simple examples. *Mach. Learn.*, 44(1-2):9–35, 2001.

[Proa] JavaCC Project. Javacc homepage. World Wide Web, https://javacc.dev.java.net/.

[Prob] Sax Project. Sax. World Wide Web, http://www.saxproject.org/.

[Ram00] Ariel Ortiz Ramirez. Three-tier architecture. World Wide Web, http://www.linuxjournal.com/article/3508, 2000.

[SAF07] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. Autobash: improving configuration management with operating system causality analysis. *SIGOPS Oper. Syst. Rev.*, 41(6):237–250, 2007.

[Sin04] Amit Singh. An introduction to virtualization. http://www.kernelthread.com/publications/virtualization/, 2004.

[Sola] Evolve Space Solutions. Virtu product line. World Wide Web, http://www.evolve.pt/index.php?option=com_content&view=article&id=82&Itemid=85.

[Solb] Evolve Space Solutions. Virtu software design document.

[Sta07] James Staten. The case for virtual appliances, November 2007.

[VMwa] VMware. Vmware home page. World Wide Web, http://www.vmware.com.

[VMwb] VMware. Vmware virtualization. http://www.vmware.com/technology/virtualization.html.

[VX] Vmware and Xensource. The open virtual machine format whitepaper for ovf specification. White Paper, Version 0.9.

[W3S] W3Schools. Xml dom tutorial. World Wide Web, http://www.w3schools.com/dom/default.asp.

[WCG04] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.