Leveraging Feature Models to Configure Virtual Appliances

Clément Quinton, Romain Rouvoy, Laurence Duchien

Leveraging Feature Models to Configure Virtual Appliances

Clément Quinton
Inria Lille - Nord Europe
LIFL UMR CNRS 8022
University Lille 1, France
clement.quinton@inria.fr

Romain Rouvoy
University Lille 1, France
LIFL UMR CNRS 8022
Inria Lille - Nord Europe
romain.rouvoy@lifl.fr

Laurence Duchien
Inria Lille - Nord Europe
LIFL UMR CNRS 8022
University Lille 1, France
laurence.duchien@inria.fr

ABSTRACT

Cloud computing is a major trend in distributed computing environments. Software virtualization technologies allow cloud *Infrastructure-as-a-Service* (IaaS) providers to instantiate and run a large number of virtual appliances. However, one of the major challenges is to reduce the disk space footprint of such virtual appliances to improve their storage and transfer across cloud servers. In this paper, we propose to use a *Software Product Line* (SPL) approach and describe the virtual appliance as a set of common and variable elements modeled by means of *Feature Model* (FM). We describe a solution to reverse engineer a FM from a virtual appliance and we show how we take advantage of the SPL configuration mechanisms to significantly reduce the size of a virtual appliance.

1. INTRODUCTION

Cloud computing has emerged as a major trend in distributed computing for "*enabling convenient, on–demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*" [11]. One key technology to enable such on-demand flexibility consists in applying software virtualization [4]. Commonly adopted by cloud *Infrastructure-as-a-Service* (IaaS) providers, virtualization technologies provide many benefits such as resource isolation, security and flexibility. IaaS providers can dynamically instantiate and host virtual appliances on physical machines depending on the users requirements. A virtual appliance is a software image containing a software stack usually composed of an *Operating System* (OS), libraries, applications servers, and applications. This software image is designed to run on virtual machine platforms such as Xen hypervisors (*e.g.*, Amazon EC2[1]).

Nevertheless, the success of virtualization has lead to the deployment of a huge number of pre-configured virtual appliances (*e.g.*, Amazon EC2 has 6521 *public* virtual appliances [8], which makes the choice difficult and leads customer to configure their own appliance, thus increasing the number of custom appliances). IaaS providers today face many problems such as virtual appliances storage, low-latency retrieval of virtual appliances and slow transfer of virtual appliances data across cloud servers. Current researches show that reducing the amount of virtual appliances data to store and transfer (*e.g.*, using virtual appliance deduplication) is one key factor to improve the performance of cloud management systems. In addition to deduplication [8, 13], our contribution provides tools to cope with all the above issues when configuring the virtual appliance.

One way to handle these issues is to use a *Software Product Line* (SPL) approach. SPL engineering consists in the description, management and implementation of the commonalities and variabilities existing among the members of the same family of software products [5, 14]. A well-known approach to variability modeling is by means of *Feature Models* (FM) introduced as part of *Feature Oriented Domain Analysis* (FODA) [9] back in 1990. We introduce in this paper a tool–supported approach for reverse engineering a FM from a package based OS distribution (*e.g.*, Debian, Ubuntu), assuming that the package dependency language can be considered as a variability language [7]. The key challenge in this task is the construction of a FM representing the OS in order to configure it to fit exactly the user requirements and reduce its disk space footprint. For example, packages such as *eclipse-platform* or *junit* are useless for an application server deployed in production, and not selecting them in the configuration process reduces the disk space footprint of the virtual appliance. Such an approach can be extended to the whole software stack and provides a way to configure a PaaS *à la carte* (*e.g.*, libraries and application servers).

The paper is structured as follows. In Sec. 2 we describe the Debian packages format and the associated metadata and we give an overview of FD and SPL techniques. We then describe in Sec. 3 how such techniques can be applied to reverse engineer a FM and configure a virtual appliance in order to reduce its disk space footprint. Sec. 4 evaluates the benefits using our approach compared to a manually configured virtual appliance. Related work are studied in Sec. 5 while Sec. 6 concludes the paper.

2. BACKGROUND

This section gives a brief overview of the package metadata that can be retrieved from package repositories and

[1] `http://aws.amazon.com/ec2/`

CloudCP'12 April 10 2012, Bern, Switzerland.

introduces the main concepts of state-of-the-art SPL approaches.

2.1 Packages Metadata Overview

Our solution focuses on *Free and Open Source Software* (FOSS) distributions (*e.g.*, Debian, Suse, Red Hat). In FOSS distributions, a package is composed of a component (that contains executable scripts, documentation, etc.), a set of files used to configure the package and metadata, which describe its attributes (*e.g.*, package size) and the inter-package dependencies. For the sake of simplicity, we focus here on the DEB[2] format as used in the Debian distribution but this solution can also be applied on RPM[3] packages. FIG. 1 depicts an excerpt of the `tomcat6` package metadata:

Package: *tomcat6*
Priority: *optional*
Section: *web*
Installed-Size: *308*
Maintainer: *Ubuntu Developers*
Original-Maintainer: *Debian Java Maintainers*
Architecture: *all*
Version: *6.0.32-5ubuntu1*
Depends: *tomcat6-common (>= 6.0.32-5ubuntu1), ucf, adduser, debconf (>= 0.5) | debconf-2.0*
Recommends: *authbind*
Suggests: *tomcat6-docs (>= 6.0.32-5ubuntu1), tomcat6-admin (>= 6.0.32-5ubuntu1)...*
Size: *30412*

Figure 1: Excerpt of the `tomcat6` package metadata

Such information can be retrieved using commands like `apt-cache show "<package_name>"` where `<package_name>` refers to the package name (*e.g.*, `tomcat6`). This gives us information like the name, the priority, the size and the version of the package. The latter is very important when specifying relationship between packages. Indeed, the above example shows that the `tomcat6` package can be installed only if the `tomcat6-common` package in a version at least equal to `6.0.32-5ubuntu1` is installed too (see `Depends` relationship). These relationships are divided into:

- `Depends`. For a package $P1$ to work properly, a package $P2$ requires to be installed. In this case we say that $P1$ depends on $P2$. $P1$ `Pre-Depends` $P2$ means that the installation of $P2$ has to be completed before starting $P1$ one.
- `Suggests`. The suggested packages of a package $P1$ are related to $P1$ and usually enhance its usefulness. The installation of these packages is optional. `Recommends` has a similar meaning than `Suggests`. It is strongly recommended but still optional to install packages listed under `Recommends` dependency.
- `Conflicts`. A package $P1$ cannot be installed if a package $P2$ is already installed. $P2$ has to be uninstalled in order to resolve the conflict and allow $P1$ to be installed. If a package $P1$ `Breaks` a package $P2$, it will be impossible to unpack $P1$ unless $P2$ is unconfigured first. Both relationships are not necessarily reciprocal.
- `Replaces`. This relationship means that a package $P1$ should overwrite files in package $P2$, or completely replace package $P2$.

Notice that for a given package, there are as many descriptions as there are versions (*e.g.*, for the `tomcat6` package, you can find such descriptions for versions `6.0.32-5ubuntu1` or `6.0.32-5ubuntu1.1`).

2.2 SPL and Feature Modeling

SPL engineering aims at generating specific products from the requirements expressed by customers by composing a set of complementary features. Therefore, each product of the SPL shares commonalities with other products from the same SPL and owns a variable part that makes it specific. FMs are used to specify the features of a SPL (thus describing commonalities and variabilities of a product family) and their valid combinations.

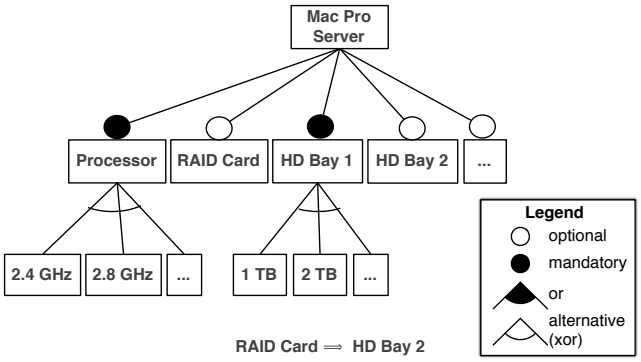

Figure 2: Excerpt of the computer feature model

A *Feature Diagram* (FD) (see FIG. 2 that depicts an excerpt FM of a Mac Pro Server consists of a hierarchy of *features* (typically a tree), which may be *mandatory* (commonality) or *optional* (variability) and may form *Xor* or *Or*-groups. Constraints (*e.g.*, *implies* or *excludes*) can also be specified using propositional logic to express inter-feature dependencies. In the above example, the `processor` (as well as the `memory`), which can be either `2.4 GHz` or `2.8 GHz`, is a mandatory feature as the computer cannot operate without it. Configuring the `Mac Pro Server` to have a `RAID card` implies such a computer to own a second `hard drive` (`HD Bay 2`). Such a relation is described as a constraint between features and is associated to the FD. We consider that a FM consists of a FD and the associated set of constraints.

Every feature can be realized by one or more assets (*e.g.*, aspect, component, model, piece of code, documentation). Assets of different features are combined to obtain a software product. In this context, features can be seen as a way of configuring a software product. In the remainder of this paper, we consider that assets are distribution packages and the product derived from the SPL is a configuration (*i.e.*, a combination of all selected packages). By means of constraints, we assume that the resulting configuration is fully functional.

[2] `http://www.debian.org/doc/debian-policy/ch-binary.html`
[3] `http://rpm.org/`

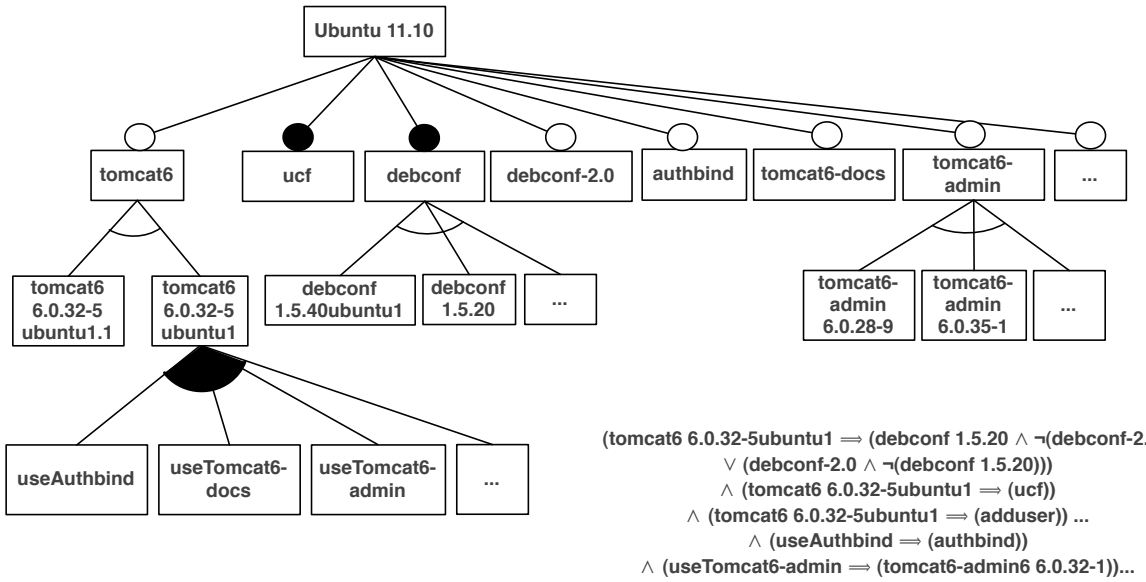

Figure 3: Excerpt of the extracted `tomcat6` FM

3. VIRTUAL APPLIANCE A LA CARTE

In this section, we propose a mapping from package dependencies language to FM and we explain how we used it as an input to reverse engineer a FM from a package description. We then show how such a procedure can be combined with existing FM algorithms to build the FM of a Linux distribution.

3.1 Reflecting Software Packages as Features

To enable the FM reverse engineering for a package-based distribution, we loop into the package repository and extract the description for each package from its metadata, as described in Sec. 2.1. We propose several rules to reason on these metadata: *(i)* the name given by the `Package` attribute is the name of the package we want to reverse engineer in the FM and is mapped into a feature, *(ii)* the kind of relationship between a parent feature and its child features is specified by their *priority* level (*e.g.*, `optional` and `mandatory` for `authbind` and `ucf`, respectively) and *(iii)*, each package whose name is listed behind `Depends`, `Pre-Depends`, `Suggests`, `Recommends`, `Replaces`, `Conflicts` or `Breaks` attribute is mapped into a feature. For a given package, merging version FM together yield the complete FM of the package. Furthermore, considering two packages $P1$ et $P2$, we propose the following mapping (cf. Table 1) from package dependencies language relationships to propositional formula:

Dependencies Language	Propositional Formula
$P1$ `Depends` $P2$	$P1 \Rightarrow P2$
$P1$ `Pre-Depends` $P2$	
$P1$ `Conflicts` $P2$	$P1 \Rightarrow (\neg P2)$
$P1$ `Breaks` $P2$	

Table 1: Mapping package dependencies to feature constraints

We consider `Depends` and `Pre-Depends` equivalent in terms of propositional formula (the same applies to `Conflicts` and `Breaks`), even if there is a semantic difference between them, according that FM does not support feature ordering. These formula are then used as the set of constraints associated to the FD. `Suggests` and `Recommends` relationships are not mapped into propositional formula but are used to extract information that describes the variability of the package. For example, the `tomcat6` package `recommends` and `suggests` `authbind`, `tomcat6-docs (>= 6.0.32-5ubuntu1.1)` or `tomcat6-admin (>= 6.0.32-5ubuntu1.1)` to be installed. Such packages are considered as *optional* features since it is not mandatory to install them for `tomcat6` to work properly.

After extracting information from package metadata, our tool provides mechanisms to generate FM representations for several FM languages (*e.g.*, Familiar [2], S.P.L.O.T.[12]) to manipulate and reason about FMs. Fig. 3 depicts an excerpt of the extracted FM of `tomcat6` based on metadata described in Fig. 1. `tomcat6-docs (>= 6.0.32-5ubuntu1)`, `authbind` and `tomcat6-admin (>= 6.0.32-5ubuntu1)` are recommended or suggested packages (*i.e.*, features according to our proposed mapping) while `ucf` and `adduser` are required packages as specified by the FM constraints.

3.2 Reflecting Configurations as Feature Models

Once every FM is extracted from package metadata, we need to merge them together in order to yield an exhaustive FM of the distribution. We rely on Familiar (*FeAture Model scrIpt Language for manIpulation and Automatic Reasoning*) for merging extracted FMs. Familiar is a *Domain Specific Language* (DSL) dedicated to the management of FMs that supports manipulating and reasoning about FMs. The main reason we used Familiar is that it provides dedicated operations to manipulate FMs and a *merge* operator [1] in particular. Package FM are given as input to the Familiar merge operator to produce the *merged* FM.

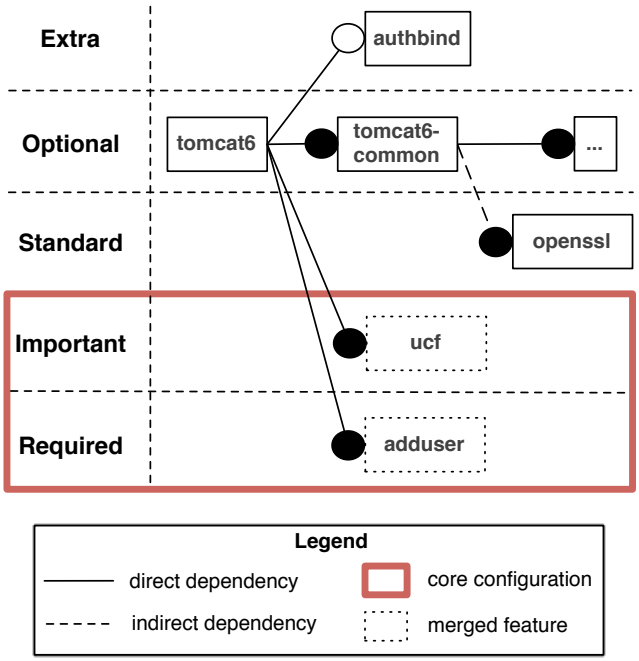

Figure 4: Excerpt of the `tomcat6` packages merged with the minimal configuration

In order to get the minimal disk space footprint for the virtual appliance we need to merge as little packages as possible. In the case of FOSS distribution, such as the Debian one, package metadata provide information about the priority given to a package by the distribution maintainers (`Required`, `Important`, `Standard`, `Optional`, `Extra`). Considering that `Required` packages are necessary for the proper operation of the system and `Important` packages should be found on any Unix-like system[4], we set the associated features as mandatory. In consequence, we consider them to be part of the common part of the SPL and we define the set of `Required` and `Important` packages (*i.e.*, the merge of corresponding features) as the *core* valid configuration, as depicted in Fig. 4. Other kind of packages (*i.e.*, `Standard`, `Optional` and `Extra`) introduce variability in the SPL and allow the distribution to be configured *à la carte*. When such a package (*e.g.*, `tomcat6`) is added on top of the core configuration, features (*i.e.*, packages) that are in the `Depends` and `Pre-Depends` relationships of this package and that are `Required` or `Important` (*e.g.*, `ucf` or `adduser`) are merged with the ones already installed in the core configuration.

4. PRELIMINARY VALIDATION

4.1 Implementation

We developed a prototype implementation script that loops into the Ubuntu package repository[5] and parses more than 33,000 package metadata. Parsed information can be used to sort packages by priority level, estimate the disk space footprint of the core configuration and foresee the footprint of a package that is going to be installed. Once package metadata retrieved, our tool uses a mapping procedure developed in Java to generate the package FM into several target languages such as Familiar or S.P.L.O.T..

[4] `http://www.debian.org/doc/manuals/debian-faq/ch-pkg_basics.html`

[5] `http://packages.ubuntu.com/oneiric/allpackages`

4.2 Estimation

By exploiting the extracted information, we can estimate the disk space footprint of a core configuration (as described in Sec. 3.2). Table 2 reports on the number of packages available (in a Ubuntu distribution) for each priority level and the estimated disk space footprint.

Priority Level	Footprint (MB)	Number of Packages
`Extra`	33,614	8,140
`Optional`	63,658	24,947
`Standard`	86	106
`Important`	45	53
`Required`	88	81

Table 2: Size & number of packages in the Ubuntu repository

We used the `Installed-Size` metadata attribute to compute the estimated disk space footprint. This size, measured in KB, gives only an estimate of the total amount of disk space required to install the named package. Then, we compared the disk space footprint of two virtual appliances based on Ubuntu 11.10 *oneiric* `Desktop` and `Server` edition respectively, with a `Debian minimal` install and the `Ubuntu core` configuration automatically generated from the FM built by our tool (Fig. 5).

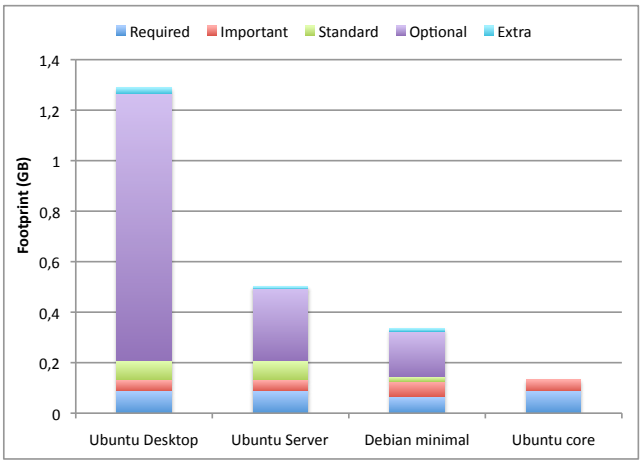

Figure 5: Estimated footprint of different configurations

All three Ubuntu configurations share a common part (*i.e.*, the core configuration) that has a disk space footprint of about 132MB, that represents the footprint of `Required` packages (88MB) added to the footprint of `Important` packages (44MB). We define this set of packages as the base configuration for our virtual appliance. We thus avoid the configuration of a lot of `Standard`, `Optional` and `Extra` packages that can be found in the `Desktop` and `Server` edition, as

reported in TABLE 3. Such a configuration is about 89% and 73% lighter than `Desktop` and `Server` editions, respectively.

	Footprint & (Nb packages)	
Priority Level	Desktop Edition	Server Edition
`Extra`	22.6 MB (60)	1.7 (5)
`Optional`	1,058.7 MB (744)	290.1 (75)
`Standard`	76.4 MB (105)	76.3 (104)
`Important`	44 MB (53)	44 (53)
`Required`	87.7 MB (81)	87.7 (81)

Table 3: Size & number of packages for the Ubuntu Desktop and Server editions

On top of the `Ubuntu core` virtual appliance, we then added the `tomcat6` package to the core configuration in order to provide a web application server (cf. FIG. 6, `Ubuntu Tomcat`). Installing a package that is not in the core configuration (*e.g.*, `tomcat6` whose level is `optional`) requires the installation of packages that are in the `Depends` and `Pre-Depends` relationships of this package and that are not `Required` or `Important` (in this case they are part of the core configuration and are already installed), *e.g.*, `tomcat6` depends on `openssl` et `tomcat6-common` that are `standard` and `optional`, respectively.

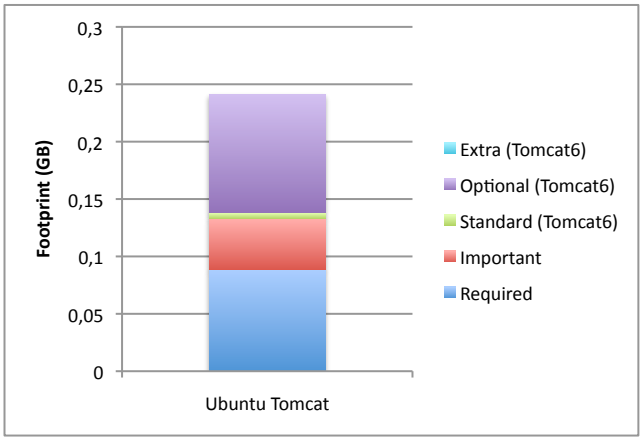

Figure 6: Disk space footprint of a configured virtual appliance providing the Tomcat 6 application server

The whole `tomcat6` package has a disk space footprint of about 160MB that is divided into 103MB, 6Mb and 51MB for `optional`, `standard` and `{important + required}` packages respectively, and the global `Ubuntu Tomcat` footprint is estimated to 235MB. The `tomcat6` package appears to be quite heavy but looking in depth, `tomcat6` depends on the `openjdk-6-jre-headless` package (minimal OpenJDK Java runtime) that is about 100MB and that is shared with all Java-based application (*e.g.*, Eclipse, whose total footprint – without `required` and `important` packages – is estimated at about 424MB). Besides, a recent work [17] showed that crosscutting features can significantly influence the footprint of many other features (*i.e.*, several packages can share dependencies to the same package that only has to be installed once to resolve all dependencies, thus reducing the total footprint). Such a configuration can be seen as a way to configure a PaaS *à la carte* and can easily be extended to other functionalities (*e.g.*, adding a database server such as MySQL to provide database support).

5. RELATED WORK

REVERSE ENGINEERING FM. The tool described in this paper aims at reverse engineering a FM from the Debian package repository. She *et al.* [16] also propose to reverse engineering a FM, but for the Linux kernel. Using heuristics with feature names, feature description and feature dependencies as input data, they identify parent feature candidates of each feature to retrieve the whole feature diagram. Their procedure do not detect OR-GROUPS. Our approach uses one source of information (package metadata) to build the FM of a given package. In [10], Mancinelli *et al.* propose a tool, *Ceve*, to extract package metadata information. They do some analysis on these data to detect errors and inconsistencies to help the distribution editors, on the server side, to maintain the package base. We also use a tool to extract package metadata information, but while we just need to loop into the repository and use package names as arguments, *Ceve* needs the concrete .deb or .rpm file to work properly.

SPL & PACKAGES. Previous work has been done to associate package dependencies and SPL. In [7], they suggest that Debian dependency language can be considered as a variability language and propose a mapping from this language to propositional formulas. While we use the reverse engineered FM to configure a virtual appliance, they show how this mapping can be used do some analysis operations (*e.g.*, detection of inconsistencies). However, the automatic extraction of FM from a Debian repository is not implemented. Di Cosmo *et al.* [6] propose using Debian dependencies management tools to analyse FM. While our approach aims at describing packages dependencies as FM, they go the opposite way and describe feature models using Debian packages dependencies.

SPL & LINUX KERNEL. Several interesting works have dealt with issues related to SPL and the Linux OS, but focused on the Linux kernel. In [18], the authors explain that the Linux kernel achieves some of the goals that the SPL guidelines also aims at and thus show that it can be considered as a SPL. She *et al.* [15] extract a variability model from the Linux kernel code base (Kconfig) and propose a mapping from Kconfig concepts to FM ones. A similar work has been done in [3] where they also propose a mapping from Kconfig concepts and *Component Description Language* (CDL) concepts to FM ones. Sincero *et al.* [19] go the way around by proposing a mapping from feature relations to Linux kernel configurator language.

6. CONCLUSION AND PERSPECTIVES

In this paper, we have presented a tool-supported approach for reverse engineering a Feature Model (FM) from a package based OS distribution (*e.g.*, Ubuntu), considered as the base of a virtual appliance. For every package in the distribution repository, we extract the corresponding package FM, taking package metadata information as input sources. In this purpose, we propose a mapping from package dependencies to propositional formula. Once each package FM

extracted, we are able to yield the complete FM (*i.e.*, the distribution FM) by reusing dedicated tools developed by the SPL community.

Our approach, based on SPL engineering, faces the growing challenge of reducing the disk space footprint of virtual appliances. Indeed, IaaS providers have to manage a huge amount of VMs and deal with many problems such as VM images storage or low-latency transfer of these VM images. By configuring the OS of the virtual appliance using SPL approach, we provide support for configuring a VM as light as possible. The preliminary result of our experimentation shows that our approach reduces at least 73% the size of the virtual appliance by only configuring the embedded OS to fit the user requirements.

For future work, we think that our approach can be extended to face the challenge of package version conflicts resolution and package installation ordering. Indeed, our configuration tool provides a configuration that is valid regarding SPL engineering but that needs more accurate information to work properly. For a given package and its set of constraints, we need to know the correct version of a package to be used and the installation order, what is not supported with SPL mechanisms. Although we focus on the OS level in this paper, we advocate that to maximize the benefits and obtain the lighter footprint for the VM such a configuration has to be applied on the whole software stack (*i.e.*, OS, database, application servers and applications).

Acknowledgments

A special thanks to Aurélien Bourdon and his development skills. This work is supported by Ministry of Higher Education and Research, Nord–Pas de Calais Regional Council and FEDER through the Contrat de Projets Etat Region Campus Intelligence Ambiante (CPER CIA) 2007-2013.

7. REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. France. Composing Feature Models. In *2nd International Conference on Software Language Engineering (SLE'09)*, LNCS, page 20. Springer, Oct. 2009.

[2] M. Acher, P. Collet, P. Lahire, and R. France. A Domain-Specific Language for Managing Feature Models. In *Symposium on Applied Computing (SAC)*. Programming Languages Track, March 2011.

[3] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In *ASE*, pages 73–82. ACM, 2010.

[4] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Gener. Comput. Syst.*, 25:599–616, June 2009.

[5] P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[6] R. Di Cosmo and S. Zacchiroli. Feature Diagrams as Package Dependencies. In *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, pages 476–480, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] J. Galindo, D. Benavides, and S. Segura. Debian Packages Repositories as Software Product Line Models. Towards Automated Analysis. In *Proceeding of the First International Workshop on Automated Configuration and Tailoring of Applications (ACOTA)*, 2010.

[8] K. R. Jayaram, C. Peng, Z. Zhang, M. Kim, H. Chen, and H. Lei. An Empirical Analysis of Similarity in Virtual Machine Images. In *Middleware Industry Track*. ACM, 2011.

[9] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) - Feasibility Study. Technical report, The Software Engineering Institute, 1990.

[10] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the Complexity of Large Free and Open Source Package-Based Software Distributions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 199–208, Washington, DC, USA, 2006. IEEE Computer Society.

[11] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology, 2009.

[12] M. Mendonca, M. Branco, and D. Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 761–762, New York, NY, USA, 2009. ACM.

[13] C.-H. Ng, M. Ma, T.-Y. Wong, P. P. C. Lee, and J. C. S. Lui. Live Deduplication Storage of Virtual Machine Images in an Open-Source Cloud. In *Middleware*, volume 7049 of *Lecture Notes in Computer Science*, pages 81–100. Springer, 2011.

[14] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[15] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. The Variability Model of The Linux Kernel. In *VaMoS*, volume 37 of *ICB-Research Report*, pages 45–51. Universität Duisburg-Essen, 2010.

[16] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse Engineering Feature Models. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 461–470, New York, NY, USA, 2011. ACM.

[17] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable Prediction of Non-functional Properties in Software Product Lines. In *SPLC*, pages 160–169. IEEE, 2011.

[18] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is The Linux Kernel a Software Product Line? In *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*, 2007.

[19] J. Sincero and W. Schröder-Preikschat. The Linux Kernel Configurator as a Feature Modeling Tool. In *SPLC (2)*, pages 257–260. Lero Int. Science Centre, University of Limerick, Ireland, 2008.