KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

A framework for integrated configuration management of distributed systems

Bart Vanbrabant

Supervisor:
Prof. dr. ir. W. Joosen

Dissertation presented in partial
fulfillment of the requirements for the
degree of Doctor in Engineering

June 2014

A framework for integrated configuration management of distributed systems

Bart VANBRABANT

Examination committee:
Prof. dr. A. Bultheel, chair
Prof. dr. ir. W. Joosen, supervisor
Prof. dr. ir. C. Huygens
Prof. dr. ir. E. Duval
Prof. dr. ir. F. Piessens
Prof. dr. D. Hughes

Prof. dr. ir. F. De Turck
(University of Ghent)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

June 2014

Uitgegeven in eigen beheer, Bart Vanbrabant, Celestijnenlaan 200A box 2402, B-3001 Heverlee (Belgium)

ISBN 978-94-6018-863-3

D/2014/7515/86

Acknowledgements

This work would not have been possible without the support and collaboration of many. Therefore I would like to thank:

- My jury for the interesting discussion and the valuable feedback on this dissertation.
- Wouter for giving me the opportunity to do this research in all freedom and further explore this research domain.
- Thomas for introducing me into the domain of configuration management both as my master thesis advisor and later as a colleague.
- The Agency for Innovation by Science and Technology in Flanders (IWT) for providing the funding for four of the six years of this research.
- ULYSSIS, the years as a volunteer in this organisation provided me with the valuable experience in system administration to support this research.
- Everyone from industry that I met at various venues for the discussions, feedback and inspiration.
- My colleagues for the collaboration over the years and the fun we had. In particular: Bert, Dimitri, Maarten, Jasper, Steven and last but not least my fellow musketeers: Stefan and Wouter!
- Each of the seven master thesis students for the collaboration (and for not causing to much trouble).
- Family and friends for the support and the interest in my work.
- Most of all my wife Britt for the incredible journey up to now and our son Daan for always being enthusiastic. Thank you both for the unconditional support, without it, this research would not have been possible.

Abstract

Our society relies on software applications both in our private and professional lives. Many of these software applications are interconnected to create large-scale distributed systems. Unfortunately software applications fail frequently and the cost of the resulting downtime is high. Configuration errors cause many of these failures. Moreover, the services of a distributed system interact with and depend on many other services of the distributed system and the deployment environment, increasing the impact of configuration errors.

The configuration of software, and especially distributed systems, consists of many configuration parameters that need to be consistent in the entire configuration: from the parameters in an end-user application, to the parameters in a network switch of the underlying infrastructure. Every update to the configuration needs to keep all configuration parameters consistent, otherwise failures and thus downtime occurs. Configuration management tools automate the configuration and reconfiguration of software applications and their execution environment. These tools use an input specification that models the desired state of the configuration, including interdependencies between configuration parameters.

Unfortunately the current state of the art in configuration management cannot model an entire distributed system. Either the configuration is managed per device or per subsystem. Therefore, system administrators still need to duplicate configuration parameters with the risk of introducing inconsistencies.

This dissertation introduces a framework for integrated configuration management. The input specification of the framework is an integrated modular configuration model that contains the entire configuration of an infrastructure: all devices, all subsystems and their interdependencies. The framework uses refinements to capture all levels of abstraction, including low-level configuration artifacts such as a configuration file on a machine, as well as architectural concepts such as high-availability services with failover and replication. The integrated configuration model supports capturing all relevant relations between configuration parameters in order to keep all parameters consistent after each change. The framework generates configuration artifacts and can enforce them on real infrastructures. System administrators can use the framework to port their current ad-hoc scripts to structured, repeatable and maintainable scripts. Developers of a configuration model can use software engineering methods such as modularity, reuse and refinement. The framework approach as well as these supported software engineering methods offer a principled approach to describe and enforce complex configuration updates.

A prototype implementation and three case studies validate the feasibility of the framework. Additionally, the case studies demonstrate that the framework can: (a) fully automate the configuration of a complex distributed system and its execution environment, including provisioning the underlying virtual machines, (b) manage a heterogeneous infrastructure from an integrated configuration model, such as network equipment and servers, and (c) automate domain-specific allocation of configuration parameters such as dual stack IP subnet allocation.

Beknopte samenvatting

In onze samenleving zijn we in ons privé en professioneel leven steeds meer afhankelijk van software programma's. Veel van deze software programma's zijn onderling met elkaar verbonden om zo een gedistribueerd systeem te creëren. Helaas falen software programma's frequent en is de resulterende onbeschikbaarheid erg kostelijk. Fouten in de configuratie van deze software programma's liggen vaak aan de basis van dit falen. Daarenboven wordt de impact van configuratiefouten versterkt door het feit dat softwareprogramma's in een gedistribueerd systeem afhankelijk zijn van andere programma's in het gedistribueerd systeem, en afhankelijkheden hebben op de uitvoeringsomgeving waarin ze opereren.

De configuratie van software, en in het bijzonder van gedistribueerde systemen, bestaat uit vele configuratieparameters. De waarde van deze parameters moet consistent zijn met alle andere parameters in de configuratie: van de configuratieparameters van een programma dat gebruikt wordt door een eindgebruiker, tot de parameters in een netwerkswitch in de onderliggende infrastructuur. Elke aanpassing moet zorgvuldig uitgevoerd worden zodat alle configuratieparameters consistent blijven met elkaar. Indien dit niet gebeurt kan de aanpassing falen veroorzaken en wordt de software onbeschikbaar. “Configuration management tools” laten toe om de configuratie en herconfiguratie van software programma's en hun uitvoeringsomgeving te automatiseren. De invoer van deze gereedschappen is een model dat de gewenste toestand van de configuratie beschrijft, inclusief afhankelijkheden tussen configuratieparameters om ze zo onderling consistent te houden. Helaas is de huidige generatie van beheersgereedschappen niet in staat om een volledig gedistribueerd systeem in zijn geheel te beschrijven. De configuratie wordt doorgaans per apparaat of per subsysteem beheerd. Ondanks de automatisatie moeten systeembeheerders dus nog steeds configuratieparameters manueel consistent houden tussen verschillende beheersgereedschappen.

Deze thesis introduceert een raamwerk voor geïntegreerd configuratiebeheer. De invoer voor het raamwerk is een geïntegreerd, modulair configuratiemodel. Dit model bevat de configuratie van de volledige infrastructuur: alle apparaten, alle

subsystemen en hun afhankelijkheden. Het raamwerk gebruikt verfijning om alle abstractieniveaus te vatten in één model, inclusief concepten van laag abstractieniveau zoals configuratiebestanden op een machine, maar ook architecturale concepten zoals programma's met hoge beschikbaarheid en replicatie. Het geïntegreerd configuratiemodel kan alle relaties tussen configuratieparameters modelleren zodat bij aanpassingen elke parameter consistent gehouden kan worden. Het raamwerk genereert configuratieartefacten en kan ze toepassen op echte infrastructuren: zoals configuratiebestanden aanmaken en installeren of software installeren op machines. Systeembeheerders kunnen het raamwerk gebruiken om hun huidige ad-hoc scripts over te zetten naar gestructureerde, herhaalbare en beheerbare scripts. Ontwikkelaars van een configuratiemodel kunnen gebruikmaken van software engineering technieken zoals modulariteit, hergebruik en verfijning. Het raamwerk en het gebruik van technieken uit software engineering biedt een rigoureuze en gedisciplineerde aanpak aan voor het beschrijven en uitrollen van complexe configuratieaanpassingen.

Een prototype en drie gevalstudies valideren de haalbaarheid van het raamwerk. Bovendien demonstreren de gevalstudies dat het raamwerk in staat is om: (a) de configuratie van complexe gedistribueerde systemen en hun uitvoerinsomgeving volledig te automatiseren, inclusief het genereren van de onderliggende virtuele machines, (b) heterogene infrastructuur, zoals netwerkapparatuur en servers, te beheren vanuit een geïntegreerd configuratiemodel, en (c) om domein specifieke configuratieparameters automatisch te alloceren, bijvoorbeeld subnetwerken in een "dual stack" IP netwerk.

Contents

Chapter 1

Introduction

The importance of large-scale distributed systems in the current digital society is obvious. Companies such as Microsoft, Google, Facebook or Twitter provide large scale software services on which people rely extensively to organise their day to day professional and social lives. On the other hand companies such as Oracle, Amazon, SAP or SalesForce.com also provide large scale software products and services on which many enterprises rely for their day to day operations. Additionally almost all enterprises rely on software systems to function for their day to day activities, such as banks, retailers, government and even production companies. Their software systems have become ubiquitous; they are also becoming more complex as they depend on many other software and services.

Unfortunately software products and services fail frequently and the cost of the resulting downtime is high [67, 91]. Analysis of these failures reveals that many are caused by configuration errors [39, 86, 87]. This is a class of errors to which distributed systems are especially prone due to their complex interactions and interdependencies between the services and the deployment environment. These interdependencies occur at all layers of the software stack, ranging from network services through middleware services to application components.

1.1 Configuring and operating distributed systems

In enterprises either development organisations deliver in-house applications, or software applications are acquired from a third party. This application is then deployed, configured and managed by the team that governs the operations, often called the system administrators or operators. This process consists of the following tasks:

Deployment of the application by installing all its files at the correct locations defined by the developers and the vendor of the operating system and installing all the dependencies of application. On most operating systems this process is automated with package management tools that can automatically manage application dependencies (e.g. yum, apt, ports or msi).

Configuration of the application by setting its configuration parameters to the desired values. The system administrator determines the desired values based on how the application will be used and by the environment where the application is deployed. Configuration files or databases (e.g. Windows registry) hold the values for the configuration parameters of an application.

Installation of updates at runtime (updates include: improving security, fixing bugs or adding features) and evolve the configuration parameters to change how the application functions.

The configuration of an application includes the selection and installation of software applications, as well as setting configuration parameters of the installed applications and the underlying infrastructure (operating system, firmware of devices, etc.). Configuration parameters provide system administrators with the means to adapt the functionality of the software application to the actual deployment environment. For example, the configuration parameters to configure an IP address on a network interface such as the IP address and netmask. Operators of an application determine correct values for these parameter to deploy a functional application. However, the system administrator can only determine a fraction of these parameters freely. All other parameters are either a duplicate of another configuration parameter or they are derived from one or more other configuration parameters. Whenever an operator changes one configuration parameter he needs to ensure that all dependent parameters are updated to keep the configuration consistent. Additionally, operators not only manage the configuration parameters of the application but also the configuration parameters of the application's execution environment: operating system, network and storage equipment, printers, etc.

An example from IP address allocation illustrates these dependencies: An IPv4 address is a 32-bit integer that is assigned to each machine connected to a network (e.g. the Internet) and should be unique within that network. As a result a machine can have 2^{32} different values for its IPv4 address. A second machine can have 2^{32}-1 different values, and so on. However in reality the degrees of freedom are significantly lower. Each IPv4 subnet has two configuration parameters: the network-address and the netmask that determine how many machines can be connected to the network. For a typical home or office network this is for example 134.58.39.0/24. This means that this network can have only 2^8 different IP addresses of which the 0 is reserved for the network-address and 255 for the broadcast-address. This leaves us with only 2^8-2 different values for the IPv4 address of a machine in that subnet and makes the IPv4 address configuration parameter dependent on the network-address and netmask configuration parameters: e.g. each IP address has the same prefix 134.58.39.

A distributed software system is a specific type of application that includes components deployed on multiple machines and that communicate over the network. This distribution increases the interdependencies between configuration parameters because all distributed components need to function as a single application and thus all configuration parameters need to be consistent, e.g. a client-server relation between an application and the database server it uses.

The ongoing rise of the cloud computing paradigm increases the management challenge because it allows distributed systems to quickly scale by deploying additional nodes. They can evolve dynamically to very large sizes and in addition they dynamically scale down when load decreases to reduce operational costs. Cloud computing increases the scale and the dynamics of distributed software systems in many ways, as well as the pace of configuration updates operators need to keep up with. Moreover, recently large cloud providers [56, 77] started billing by the minute. This pushes the rate of change in the management of a distributed system to unprecedented frequencies.

The configuration and management of distributed systems is further complicated by the gap in abstraction level between development and operation of software. In the past 30 years software engineering has been gradually raising the abstraction level in which applications and distributed systems are described and developed (from procedural through object oriented to component based software development). The increased abstraction level hides complexity and allows software engineers to build larger and more complex applications. Software engineers describe and reason about a distributed system in function of component interactions and deployment on nodes, in contrast to how operators deploy it as binaries and configuration files on a file system. Operator practices have not been able to keep up with this increase in abstraction and therefore the configuration of a distributed system is often characterized by low level artifacts such as files, system services and software packages. Advances in the field of software engineering enable us to build more complex systems but the operational side of these systems has not advanced sufficiently to operate them reliably [6, 46].

The work of operators to manage large distributed systems can be tedious and repetitive because similar operations and changes need to be carried out on multiple machines. Operators try to automate their work using ad-hoc scripts. Scripts automate certain aspect of the configuration and management of the machines and the software that runs on them (operating system and applications). Operator scripts are often custom for their environment, ad-hoc and very brittle because they evolved over many years.

Configuration management tools offer a more structured approach to system administration automation. These tools exist in a broad range of automation and abstraction capabilities. At one end of the spectrum are tools that merely provide a framework to distribute, schedule and execute custom scripts and generate reports from the execution. On the other end of the spectrum are tools that offer a desired state configuration model of the managed infrastructure. Each of these tools have a similar reference architecture.

Figure 1.1 shows a simplified diagram of this architecture specifically for desired state tools. A central repository stores an input specification that describes the desired configuration of the managed distributed system. One or more translation agents generate the desired state of each of the resources the tool manages on various devices. A deployment agent on each managed device retrieves, through pull or push, the desired state of each of the resources it manages. The deployment agent then enforces the desired state of each of its managed resources.

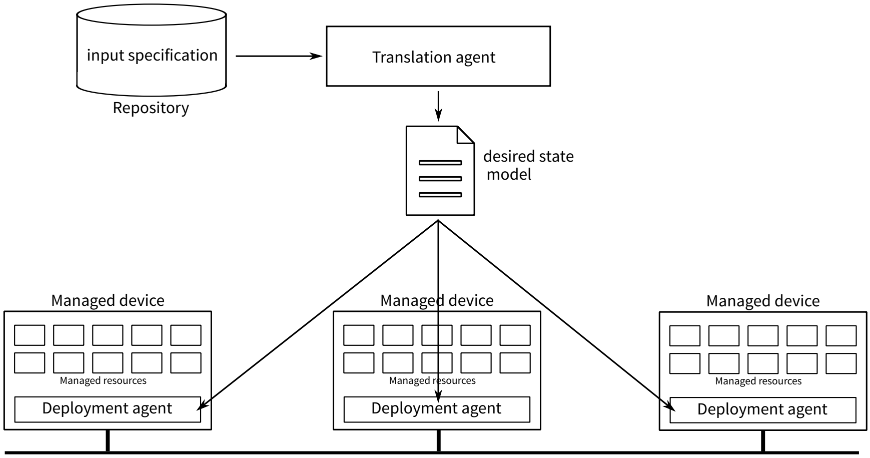


Figure 1.1: A reference architecture for configuration management tools.

1.2 Integrating development and operations

Industry has been slowly adopting configuration management tools for nearly three decades now. The tools that enterprises use are often a system to distribute and schedule scripts on many devices in a network. Cfengine [19] was one of the first systems to offer a new type of tool that uses a desired state model of the intended configuration of a single machine. Many new tools emerged that use a desired state model [8, 25, 34], but offer different input languages and deployment properties. However, each of them only offer limited abstraction capabilities. A second generation of desired state model based tools emerged with Puppet [96], Cengine 3 [22] and Chef [24]. They offer mechanisms to abstract away heterogeneity and complexity but only for a single machine at a time.

The focus on the desired state of a single machine at a time makes it impossible for these tools to keep up with the increase in the speed of configuration changes and the increase in complexity of distributed applications, both trends which are largely driven by the increased importance of cloud computing and delivering software as a service. Operators address this lack of integrated management by (again) adding ad-hoc scripting to these configuration management tools.

Recently industry has an increased attention to the changes required to more closely integrate the development and operations of software. This movement is labelled with the term DevOps. A key change they advocate is adopting configuration management tools to automate configuration management. This is often referred to as infrastructure-as-code [90].

An integrated desired state configuration model of a distributed system and the entire underlying software stack and infrastructure can ensure that all configuration parameters have a consistent value. To achieve a configuration parameter set that is consistent, all dependencies between configuration parameter need to be represented in the configuration model. This is only possible when the configuration model contains a representation of all concepts in the infrastructure. These concepts can be of any abstraction level, such as use asynchronous replication between data-center A and B, to Apache should serve requests for http://example.com from the directory /var/www/example.com.

An integrated approach to configuration management is required to address the challenges that the management of contemporary distributed systems pose. Integrated in the sense of: (1) managing all configuration of a distributed system from a single integrated configuration model and (2) integrating operations and development. An integrated configuration model for a distributed system needs to be designed and develop in the same way as the software for the distributed system. Furthermore, in this section we elaborate on our vision on how the development of an integrated configuration model should be included in the development process of the distributed system (Figure 1.2).

The configuration model of an application determines how the application and infrastructure it runs on are provisioned, deployed and configured. An integrated configuration model for contemporary distributed systems is by definition a large configuration model. From the field software engineering it has been long established that the most efficient way to develop large software project is by modularising it [88]. Many stakeholders develop modules of the configuration model. In the next paragraphs we discuss the role of each stakeholder in an ideal development model:

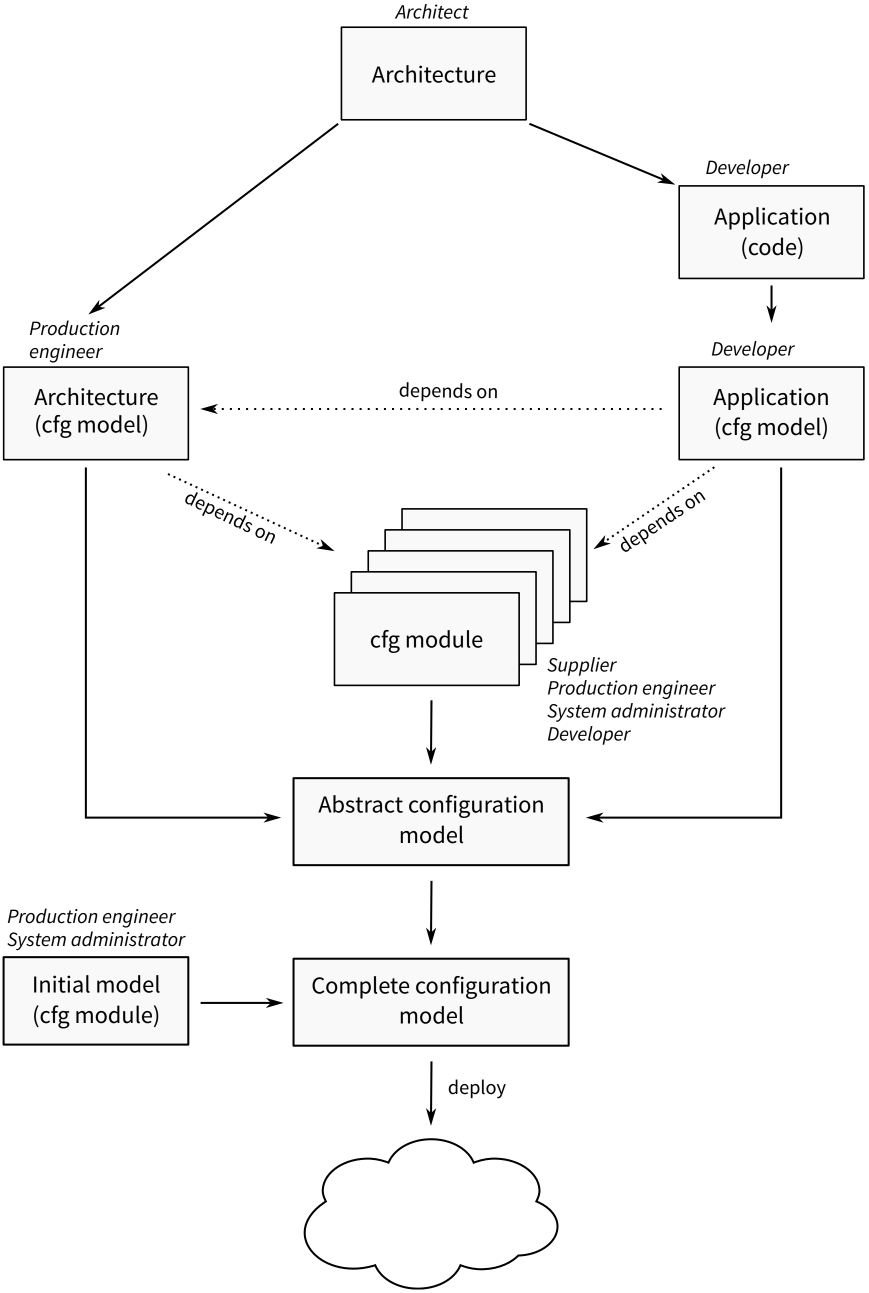


Figure 1.2: The application lifecycle and the development of the integrated configuration model.

Architect The architect creates the architecture of the application. The architecture contains a design of the execution environment (software stack, infrastructure, etc.) for the application, which is often represented in the deployment view of the architecture.

Developer The developer develops the application components mainly based on the logical view of architecture. Additionally the developer should also develop the configuration module(s) to deploy and configure the application components on the execution environment. During the development of this configuration module the developer reuses concepts (entities) from other modules that manage technology that is used to develop the application components. For example, application components developed on JEE technology require a JEE application container. The application configuration modules expose the configuration parameters of the application in a consistent and stable manner to the operators that will manage the application in production.

Production engineer The production engineer designs the execution environment for the application components. The configuration model of this execution environment is packaged in a architecture configuration module. It defines the concepts from the architecture and implements the deployment views. This implementation refines the high level configuration model (architecture level) in function of concepts defined in existing modules. These existing module come from the supplier that delivers the hardware and software used to implement the execution environment. The architecture module should however still be parametrised to the target infrastructure (e.g. datacenter, IaaS provider, IP range, DNS domain, etc.) and the scale (e.g. the replication factor of application components).

System administrator The system administrator deploys and operates the application that is in use. During the operation of the application the system administrator takes care of changing configuration parameters, installing updates, etc. through the configuration model.

Supplier The supplier delivers the hardware and software that is procured to deploy and run the application on. Ideally the supplier also provides configuration modules with their hardware and software. These modules encapsulate the configuration knowledge of experts in the configuration of a particular type of hardware, software or subsystem. If such a module is not available it needs to be developed in house.

The architecture, application and other configuration modules define how each component should be deployed and configured, but lack the parameters to make the deployment specific. These parameters will often be determined by the developers and production engineers and operated by the system administrators that operate the execution environment once it is deployed. They instantiate an initial model with these configuration parameters.

1.3 Problem statement

The process explained in the previous section is our vision on how an integrated configuration model should be developed and used. Based on this process and a study of the current state-of-the-art we identified the following problems:

1. Configuration management tools need to offer an environment to describe a distributed system, its configuration and configuration changes at a high level of abstraction. This reduces the mismatch between reasoning about the system in terms of its architecture and configuration. Such a tool will be more cost effective in terms of manpower and reduces the risk of configuration errors.

2. Configuration management tools have to integrate the management of all layers in the infrastructure in one tool, from low level network equipment to components deployed in application containers. Only then can maximal automation be achieved. Additionally, a tool has to support dependency management between interdependent configuration parameters to reduce duplication of configuration parameters and enable configuration reuse.

3. Users of a configuration management tool have to be able to refine a high-level configuration description of an application to the configuration of a functional application, equipped with all the necessary configuration details for a target environment.

4. A configuration management tool must be applicable to real-world systems and infrastructures both physical and virtualised. Moreover, it should be flexible enough to allow its users to incrementally adopt it: employ it to manage only a single service, manage all subsystems in an distributed system or everything in between. However only fully integrated management can capture all relations between all relevant configuration parameters. If only a subset of the distributed system is managed: duplicated and thus possibly inconsistent configuration parameters are inevitable.

1.4 Contributions

In this dissertation we introduce IMP, a framework for integrated configuration management that offers configuration management technology to address the above mentioned challenges. IMP focuses on modeling the configuration of the entire distributed system and its execution environment, and on the generation of a configuration description that is ready-to-deploy. IMP contributes the following to achieve this:

- A modeling language to create an integrated configuration model of the distributed system, supporting high-level descriptions that also cover dependency management. Users of IMP can develop the integrated configuration model in a modular fashion and reuse existing configuration modules. The configuration model is an exact description of the configuration of the distributed system, leaving no degrees of freedom to the framework. This ensures that the mental model [106] of system administrators matches with the configuration of the distributed system.
- Generation of configuration artifacts by refining high-level and partial configurations iteratively in function of lower level configurations. Thus, enabling users of IMP to express the configuration of all managed resources in a single integrated configuration model.
- A platform that manages real infrastructures from an integrated configuration model. Moreover, system administrators can leverage this platform to port their existing ad-hoc scripts. These scripts can then use the interface provided by the configuration model, which makes them less brittle. Because of the integration in the framework, system administrators can fully automate the allocation of configuration parameters for specific services or subsystems, possibly based on monitoring information to close the control loop.
- Initial work on a principled approach to describe and enforce complex configuration updates in the integrated configuration model. A principled approach for the development of the configuration model, allows a configuration management tool to also take state transitions and application data into account. The IMP enforcement agent provides a modest step in this direction by enforcing inter-host service dependencies.

Additionally this dissertation provides an extended and up to date version of the configuration management framework introduced by Delaet et al. [31].

The prototype implementation of IMP shows that it is feasible to build the technology to provide an integrated configuration management platform. IMP is validated in three cases of which one a large distributed system that is generated, deployed and configured on an IaaS platform. Additionally, one of the case studies and its evaluation show that IMP can fully automate deployment and configuration management of a large infrastructure with; for example, 92 virtual machines, 3006 managed files, 926 packages installed and 1007 services managed. These 3006 files contain in total 4977 configuration parameters when the storage systems uses its built-in peer-to-peer discovery and 20115 configuration parameters when the storage system has a 100% static configuration. All this variation is controlled in the configuration model by four integer parameters.

1.5 Other work

System configuration tools are a powerful tool to manage a large and complex distributed system in an integrated fashion. The integrated configuration model determines entirely what is deployed on an infrastructure and how it is configured. This means that there is a single point of access to the configuration of your infrastructure. It makes access control and audit of who does what easier, but it may require a workflow with multiple people involved that review changes before they are included in the configuration model.

Due to the organisation of the input in files for such a configuration model, file based access control does not offer access control that is fine-grained enough. In *Federated access control and workflow enforcement in systems configuration* (received best student paper award as LISA 2009) [111, 112] we proposed a fine-grained language level access control mechanism. This mechanism performs access control on language level changes in the input specification and enforces update workflows including reviews by additional operators. We validated this access control and workflow mechanism on a small configuration language [111] and implemented the access control mechanism [113] on the Puppet [96] system configuration tool. Both implementations show that is feasible to raise the abstraction level access control policies to the level in which a configuration management tool describes its configuration model.

The initial prototype [111] was implemented on a simple and small, but functional, configuration management tool. The experience of designing and implementing such a tool served as inspiration for the design and prototype of IMP.

1.6 Structure of this dissertation

In chapter 2 we propose a reworked and updated version of our comparison framework for configuration management tools [31]. We applied this framework to 11 tools and distill six areas for improvement for the current state-of-practice in configuration management tools. In chapter 3 we propose IMP, an integrated configuration management platform. We discuss its architecture, configuration modeling capabilities, how it enforces configuration changes and how IMP refines high level configuration concepts to a deployable configuration. Chapter 4 discusses the IMP prototype we developed, how we applied IMP on three case studies. These case studies show that IMP can indeed integrate network and server management in a single tool. Additionally, although IMP does not have generic support for automatically finding configuration parameter values, it can do this for specific domains. Finally, it shows that IMP can fully bootstrap and manage a complex distributed system on an IaaS platform. We conclude the evaluation chapter with an evaluation of IMP using our evaluation framework and discuss how IMP tackles the areas of improvement from chapter 2. In chapter 5 we give a summary of related work concerning configuration management tools both from academia and industry, management standards and meta-models and last of related work concerning human factors that provided valuable input for IMP. We conclude the dissertation in chapter 6 with lessons learned and future work both in research and to further mature IMP. Appendix A gives an extensive language overview of IMP's modeling language. Appendix B contains a tutorial to get started with IMP to manage a two machine distributed system and finally appendix C contains tables with the survey results of chapter 2 summarized.

Chapter 2

A survey of configuration management tools

This chapter provides a comparison framework for configuration management tools. This framework is a reworked and extended version of the framework of Delaet et al. [31]. This comparison framework was applied to tools from both industry and academia in 2010. The results of this survey are both useful for practitioners to objectively select a configuration management tool and to determine areas of improvement in the state-of-the-art and state-of-practice. These areas of improvement and a study of related work allowed to set clear research goals and provide a contribution to the field in this dissertation. Although the survey was conducted in 2010 the identified areas of improvement are still relevant. Some tools have become less relevant while others appeared, however we believe that the areas of improvement have not been sufficiently addressed both by research and industry.

The adoption of configuration management tools in industry is still very limited even though tools that improve over ad-hoc scripting exist for over 20 years. Adopting a configuration management tool implies a significant investment in time and/or money. Before making such an investment, an informed choice based on objective criteria is the best insurance that an enterprise has picked the right tool for its environment.

This chapter introduces a comparison framework for configuration management tools that allows users of it, to make a systematic and objective comparison. The framework consists of four categories of properties:

1. **Specification properties** related to the input of the configuration management tool.

2. **Deployment properties** related to deploying the input specification.
3. **Specification management properties** related to the configuration management process.
4. **Tool support** properties related to what support is available to users of a tool.

The framework is used to evaluate 11 existing open-source and commercial configuration management tools. This set of 11 tools was in 2010 at the time of the evaluation a representative set of configuration management tools. This chapter contains a summary of these evaluations. The full evaluations are available on a website[1].

The remainder of this chapter is structured as follows: Section 2.1.starts with the description of the framework. Next, Section 2.2 summarizes the findings for the evaluated 11 tools. A summary in table form is presented in Appendix C. Section 2.3 determines areas for improvement in the state-of-the-art based on the framework and the evaluation. Section 2.4 concludes this chapter.

2.1 The comparison framework

The comparison framework contains properties for both the specification of the input and the enforcement phase. The third type of properties in the comparison framework are meta-specification properties: related to how a tool deals with managing the input specification itself. The final type of properties deal with user support.

2.1.1 Specification properties

The specification properties determine how the input of the tool is specified and managed, including the mechanisms available to handle complexity and heterogeneity. The framework groups the properties in four categories related to: the specification paradigm, abstraction and modularization mechanisms , and modeling of relations.

2.1.1.1 Specification paradigm

The specification paradigm evaluates whether the language is declarative or imperative and the user interface to define the specification: command-line or graphical.

---

[1] http://distrinet.cs.kuleuven.be/software/sysconfigtools

Tools that use a declarative input language express the desired state of the computer infrastructure. The runtime of the tool compares the desired state with the configuration on every managed device and derives a plan to move to the desired state. In the configuration management literature, this process is described as *convergence* [30]. A configuration management tool that supports *convergence* has the additional benefit that divergences from the desired state are automatically corrected [46].

Tools that use an imperative input language distribute, schedule and deploy scripts on the managed device. System administrators write these scripts in an imperative input language. For an imperative script to work reliable, all possible states of the managed devices need to covered and checked in the script. Moreover, the configuration management tool must also keep track of what scripts are already executed on every device. An alternative is to make all the operations in the script idempotent [61].

An example clarifies the practical differences between an imperative and a declarative language. Suppose a system administrator does not want file `/etc/hosts_deny` to be present on a device:

- In a declarative language, the system administrator must ensure that the file is not included in the model or explicitly define that the file must not exist.
- In an imperative language, the system administrator must first write a test to verify if `/etc/hosts_deny` exists. If the file exists, another instruction is needed to remove the file. If the system administrator does not write the first test, the action fails if the file was already removed.

Orthogonal on the choice of declarative or imperative specification language is the choice of user interface: Command-line interfaces typically have a steeper learning curve than graphical interfaces but, once mastered, can result in higher productivity. Command-line interfaces also have the advantage that they can be integrated with other tools through scripting. In contrast, system administrators are typically quicker up to speed with graphical interfaces [57].

2.1.1.2 Abstraction mechanisms

A successful configuration management tool is able to make abstraction of the complexity and the heterogeneity that characterises IT infrastructures where hardware and software of several vendors and generations are used simultaneously [9]. Making abstraction of complexity and heterogeneity is very similar to what general purpose programming languages do.

Abstraction from complexity is an important concept in a programming paradigm such as object orientation. In object orientation, implementation details are encapsulated behind a clearly defined interface. Encapsulation is a concept that is valuable for modeling configurations as well. Responsibilities and expertise in a team of system administrators are not defined on machine boundaries, but based on subsystems or services within the infrastructure. For example: DNS or the network layer. Encapsulation enables experts to model an aspect of the configuration and expose a well documented API to other system administrators.

Modern IT infrastructures are very heterogeneous environments. Multiple generations of software and hardware of multiple vendors are used in production at the same time. These heterogeneous resources need to be configured to work together in one infrastructure.

Six-levels of abstraction [9] can classify tools, based on how the language of a configuration management tool deals with complexity and heterogeneity. These levels range from high-level end-to-end requirements, to low-level bit-configurations.

1. **End-to-end requirements**: End-to-end requirements are typical non-functional requirements [98]. They describe service characteristics that the computing infrastructure must achieve. Figure 2.1 shows an example of a performance characteristic for a mail service. Other types of end-to-end requirements deal with security, availability, reliability, usability, ...
2. **Instance distribution rules**: Instance distribution rules specify the distribution of instances in the network. An instance is defined as a unit of configuration specification that can be decomposed in a set of parameters. Examples of instances are mail servers, DNS clients, firewalls and web servers. A web server, for example, has parameters for expressing its port, virtual hosts and supported scripting languages. In Figure 2.1, the instance distribution rule prescribes the number of mail servers that need to be activated in an infrastructure. The need for such a language is expressed in literature [7, 9].
3. **Instance configurations**: At the level of instance configurations, each instance is an implementation independent representation of a configuration. An example of a tool at this level is Firmato [15]. Firmato allows modeling firewall configurations independent from the implementation software used.
4. **Implementation dependent instances**: The level of implementation dependent instances specifies the required configuration in more detail. It describes the configuration specification in terms of the contents of software configuration files. In the example in Figure 2.1 a sendmail.cf file is used to describe the configuration of mail server instances.

5. **Configuration files**: At the level of configuration files, complete configuration files are mapped on a device or set of devices. In contrast with the previous level, this level has no knowledge of the contents of a configuration file.

6. **Bit-configurations**: At the level of Bit-configurations, disk images or diffs between disk images are mapped to a device or set of devices. This is the lowest level of configuration specification. Bit-level specifications have no knowledge of the contents of configuration files or the files itself. Examples of tools that operate on this level are image cloning systems or snapshots on IaaS systems.

1. End-to-end requirements
Configure enough mail servers to guarantee an SMTP response time of X seconds
↓
2. Instance distribution rules
Configure N suitable machines as a mail server for this cluster
↓
3. Instance configurations
Configure machines X, Y, Z as a mail server
↓
4. Implementation dependent instances
Put these lines in sendmail.cf on machines X, Y, Z
↓
5. Configuration files
Put configuration files on machines
↓
6. Bit-configurations
Copy disk images onto machines

Figure 2.1: An example of different abstraction levels of configuration specification for an email setup.

Figure 2.1 shows the six abstraction levels for system configuration, illustrated with an email setup. The illustration is derived from an example discussed in [9]. The different abstraction levels are tied to the context of system configuration. In the context of policy languages, the classification of policy languages at different levels of abstraction is often done by distinguishing between high-level and low-level policies [78, 115]. The distinction of what exactly is a high-level and low-level policy language is rather vague. In many cases, high-level policies are associated with the level which are called end-to-end requirements, while low-level policies are associated with the implementation dependent instances level. A classification tied to the context of configuration management gives a better insight in the different abstraction levels used by configuration management tools.

In conclusion, a configuration management tool automates the deployment of configuration specifications. At the level of bit-configurations, deployment is simply copying bit-sequences to disks, while deploying configurations specified as end-to-end requirements is a much more complex process.

2.1.1.3 Modularization mechanisms

One of the main reason system administrators automate the configuration of their devices is to avoid repetitive tasks. Repetitive tasks are not cost efficient. Moreover, they raise the chances of introducing errors. Repetitive tasks exist in a computer infrastructure because there are large parts of the configuration that are shared between a subset (or multiple overlapping subsets) of devices [9]. For example, devices need the same DNS client configuration, authentication mechanism, shared file systems, ... A configuration management tool that supports the modularization of the configuration specification reduces repetition in the configuration specification.

In its most basic form, modularization is achieved through a grouping mechanism: a device A is declared to be a member of group X and as a consequence inherits all system configuration chunks associated with X. More advanced mechanisms include query based groups, automatic definition of groups based on environmental data of the target device and hierarchical groups.

An additional property of a modularization mechanism is whether it enables third parties to contribute partial configuration specifications. Third parties can be hardware and software vendors or consultancy firms. System administrators can then model their infrastructure in function of the abstractions provided by the third-party modules and reuse the expertise or rely on support that a third party provides on their configuration modules.

2.1.1.4 Modeling of relations

One of the largest contributors to errors and downtime in infrastructures are erroneous configurations [86, 87, 91] due to human error. The root cause of errors in a configuration are often inconsistent configuration parameters. For example, a DNS service that has been moved to an other server or moving an entire infrastructure to a new IP range. Explicitly modeling relations that exist in the network helps keeping a configuration model consistent.

Modeling relations is, like the modularization property of Section 2.1.1.3, a mechanism for minimizing redundancy in the configuration specification. When relations are made explicit, a tool can automatically change configurations that depend on each other. For example, when the location of a DNS server changes and the relation between the DNS server and clients is modeled in the configuration specification, a configuration management tool can automatically adapt the client configurations to use the new server. Again, modeling relations reduces the possibility of introducing errors in the configuration specification.

To evaluate how well a tool supports modeling of relations they are described in function of two orthogonal properties of relations: granularity and multiplicity.

1. **granularity**: Section 2.1.1.2 defined an instance as a unit of configuration specification that can be decomposed in a set of parameters. Examples of instances are mail servers, DNS clients, firewalls and web servers. A web server, for example, has parameters for expressing its port, virtual hosts and supported scripting languages. Based on this definition relations can be classified in three categories: (1) relations between instances, (2) relations between parameters and (3) relations between a parameter and an instance.
    (a) **Instance relations** represent a coarse grained dependency between instances. Instance dependencies can exist between instances on the same device, or between instances on different devices. An example of the former is the dependency between a DNS server instance and the startup system instance on a device: if a startup system instance is not present on a device (for example: /etc/init.d), the DNS server instance will not work. An example of dependencies between instances on different devices is the dependency between DNS servers and their clients.
    (b) **Parameter relations** represent a dependency between parameters of instances. An example of this is a CNAME record in the DNS system: every CNAME record also needs an A record.
    (c) **Parameter - instance relations** are used to express a relation between an individual parameter and an instance. For example a mail server depends on the existence of an MX record in the DNS server.

   Note that it depends on the abstraction level of a tool which dependencies it can support. The two lowest abstraction layers in Figure 2.1, configuration files and bit-configurations, have no knowledge of configuration parameters and as a consequence, they can only model instance dependencies.

2. **multiplicity**: Relations can range from one-to-one to many-to-many relationships. A simple one-to-one relationship is for example a middleware platform depending on a language runtime. A many-to-many relationship is for example the relation between all DNS clients and DNS servers in a network. A configuration management tool can provide support to query and navigate relations in the configuration management tool's specification. For example, a webservice runs on a machine in the DMZ. This DMZ has a dedicated firewall that connects to the Internet through an edge router in the network. The webservice configuration has a relation to the host it is running on and a

relation to the Internet. The model also contains relations that represent all physical network connections. With these relations, a firewall specification can derive firewall rules for the webservice host, the DMZ router and the edge router [15].

An additional feature is the ability of the tool to support the modeling of constraints on relations. Two types of constraints can be distinguished: validation constraints and generative constraints.

1. **validation constraints** are expressions that need to hold true for your configuration. Because of policy or technical factors, the set of allowable values for a relation can be limited. Constraints allow to express these limitations. Examples of such limitations are:
    - The configuration management server can only handle 100 deployment agents.
    - Clients can only use the DNS server that is available in their own subnet.
    - Every server needs to be configured redundantly with a master and a slave server.
2. **generative constraints** are expressions that leave a degree of freedom between the instantiation of a managed resource and the device on which this resource is located. Languages without support for generative constraints need a one-to-one link between the managed resource and the device on which is needs to be applied in the configuration specification. Languages with support for generative constraints leave more degrees of freedom for the tool. An example of a generative constraint is: One of the machines in this set of machines needs to be a mail server.

2.1.2 Deployment properties

A configuration specification should result in configuration artifacts, a state model or scripts that are deployed on a real world system. In this section tools are again subdivided in four categories based on their: scalability, update workflow, deployment architecture and platform support.

2.1.2.1 Scalability

Large infrastructures are subject to constant change in their configuration. Configuration management tools must deal with these changes and be able to quickly enforce the configuration specification, even for large infrastructures. Large infrastructures typically get more benefit of using a higher level specification (see Figure 2.1). However, the higher-level the specification, the more processing power is needed to translate this high level specification to enforceable specifications on all managed devices.

2.1.2.2 Configuration update workflow

Configuration update workflow management deals with planning and execution of (composite) changes in a configuration specification. Changes can affect services distributed over multiple machines and with dependencies on other services [9, 85].

One aspect of workflow management is state transfer. The behavior of a service is not only driven by its configuration specification, but also by the data it uses. The mail spool and mailboxes is the data of a mail server, while the web pages are the data of a web server. When a service is upgraded or transfered to another device, the configuration management tool has to take care that the state (collection of data) remains consistent in the face of changes.

Another aspect of workflow management is the coordination of distributed changes. This has to be done very carefully as not to disrupt operations of the computing infrastructure. A change affecting multiple machines and services has to be executed as a single transaction. For example, when moving a DNS server from one device to another, one has to first activate the new server and make sure that all clients use the new server before deactivating the old server. For some services, characteristics of the managed protocol can be taken into account to make this process easier. For example, the SMTP protocol retries for a finite span of time to deliver a mail when the first attempt fails. A workflow management protocol can take advantage of this characteristic by allowing the mail server to be unreachable during the change.

A last aspect of workflow management is non-technical: if the organizational policy is to use maintenance windows for critical devices, the tool must understand that changes to these critical devices can influence the planning and execution of changes on other devices.

2.1.2.3 Deployment architecture

The typical deployment of a configuration management tool is illustrated in Figure 1.1. A configuration management tool starts from a central specification for all managed devices. Next, it (optionally) processes this specification to device profiles and distributes these profiles (or the full specification) to every managed device. An agent running on the device then enforces the device's profile.

Configuration management tools differentiate their deployment architecture along two properties: 1. the architecture of the translation agent and 2. whether they use pull or push technology to distribute specifications.

1. **architecture of translation agent**: Possible approaches for the architecture of the translation agent can be classified in three categories, based on the number of translation agents compared to the number of managed devices: centralized management, weakly distributed management and distributed management [75].
    (a) **centralized management** is the central server approach with only one translation agent. In large networks, the central server becomes a bottleneck.
    (b) **weakly distributed management** is an approach where multiple translation agents are present in the network. This approach can be realized by replicating the server and providing a shared policy repository for all servers. Another possible realization of this approach is organizing translation agents hierarchically.
    (c) **distributed management** systems use a separate translation agent for each managed device. The difficulty with this approach is enforcing inter-device relations because each device is responsible for translating its own configuration specification. As a consequence, devices need to cooperate with each other to ensure consistency.
2. **push or pull**: A pull based mechanism, means that the deployment agent needs to contact the translation agent to fetch the translated configurations. In a push based mechanism, the translation agent contacts the deployment agent. Configurations often contain sensitive information like passwords or keys and exposing this information to all deployment agents introduces a security risk, there an authentication and authorisation mechanism is required.

2.1.2.4 Platform support

Modern infrastructures contain a variety of computing platforms: Windows/Unix/Mac OS X servers, but also desktop machines, laptops, tablets, smartphones and network equipment. Even in relatively homogeneous environments, a tool cannot assume that all devices run the same operating system: operating systems running on network equipment are fundamentally different than those running on servers/desktops and smartphones are yet another category of operating systems.

Good platform support or interaction with other tools is essential for reducing duplication in the configuration specification. Indeed, many relations exist between devices running different operating systems. For example: a server running Unix and a router/firewall running Cisco IOS. If different tools manage the server and router, relations between the router and server need to be duplicated in both tools which in turn introduces consistency problems if one of the relations changes. An example of such a relation is the the firewall rule on a Cisco router that opens port 25 and the SMTP service on a Unix server.

2.1.3 Specification management properties

The larger and more integrated the specification for a tool becomes the harder it becomes to develop and evolve the specification. The evaluation framework defines seven categories related to specification management: usability, support for version control, documenting the configuration, interaction with existing management interfaces and databases, handling conflicts, update workflow definition and access control and authorisation of updates.

2.1.3.1 Usability

The usability of a configuration management tool can be split up in: 1. ease of use of the language, 2. support for testing specifications and, 3. monitoring the infrastructure.

1. **ease of use of the language**: The target audience of a configuration management tool are system administrators. The language of the configuration management tool should be powerful enough to replace their existing tools, which are mostly custom tools. But it should also be easy enough to use, so the average system administrator is able to use it. Good system administrators with a good education [60] are already scarce, so a configuration management tool should not require even higher education.

2. **support for testing specifications**: To understand the impact of a change in the specification, the configuration management tool can provide support for testing specifications through something as trivial as a dry-run mode or more complex mechanisms like the possibility to replicate parts of the production infrastructure in a (virtualized) testing infrastructure and testing the changes in that testing infrastructure first [14].

3. **monitoring the infrastructure**: A configuration management tool can provide an integrated (graphical) monitoring system and/or define a (language-based) interface for other tools to check the state of an infrastructure. A language-based interface has the advantage that multiple monitoring systems can be connected with the configuration management tool. A monitoring

system enables the user to check the current state of the infrastructure and the delta with the configuration specification.

2.1.3.2 Versioning support

Some configuration management tools store their specification in text files. For those tools, a system configuration specification is essentially code. As a consequence, the same reasoning to use a version control system for source code applies. It enables developers and system administrators to document their changes and track them through history. In a configuration model this configuration history can also be used to rollback configuration changes and it makes sure an audit trail of changes exists.

The configuration management tool can opt to implement versioning of configuration specification using a custom mechanism or, when the specification is in text files, reuse an external version control system and make use of the hooks most generic version control systems provide.

2.1.3.3 Specification documentation

Usability studies [14, 57] show that a system administrator spends much of its time on communication with other system administrators. These studies also show that a lot of time is lost because of miscommunication, where discussions and solutions are based on wrong assumptions. A configuration management tool that supports structured documentation can generate documentation from the system configuration specification itself and thus remove the need to keep the documentation in sync with the real specification.

2.1.3.4 Integration with environment

The infrastructure that is managed by the configuration management tool is not an island: it is connected to other networks, is in constant use and requires data from other sources than the system configuration specification to operate correctly. As a consequence, a system administrator may need information from external databases in its configuration specification or information about the run-time characteristics of the managed nodes. A configuration management tool that leverages on these existing sources of information integrates better with the environment in which it operates because it does not require all existing information to be duplicated in the tool.

2.1.3.5 Conflict management

A configuration specification can contain conflicting definitions, so a configuration management tool should have a mechanism to deal with conflicts. Despite the presence of modularization mechanisms and relation modeling, a configuration specification can still contain errors, because it is written by a human. In case of such an error, a conflict is generated. In the framework two types of conflicts exist: application specific conflicts and contradictions in the configuration specification, also called modality conflicts [72].

1. **application specific conflicts**: An example of an application specific conflict is the specification of two Internet services that use the same TCP port. In general, application specific conflicts can not be detected in the configuration specification. For example, application specific protocols for conflict management for IPSec and QoS policies [23, 51].
2. **modality conflicts**: An example of a modality conflict is the prohibition and obligation to enable an instance (for example a mail server) on a device. In general, modality conflicts can be detected in the configuration specifications.

When a configuration specification contains rules that cause a conflict, this conflict should be detected and acted upon.

2.1.3.6 Workflow enforcement

In most infrastructures a change to the configuration will never be deployed directly on the infrastructure. A policy describes which steps each update need to go through before it can be deployed on the production infrastructure. These steps can include testing on a development infrastructure, going through Q&A, review by a security specialist, testing on a exact copy of the infrastructure and so on. Exceptions on such policies can exist because not every update can go through all stages, updates can be so urgent that they need to be allowed immediately, but only with approval of two senior managers. A configuration management tool that provides support for modeling these workflows can adapt itself to the habits and processes of the system administrators.

2.1.3.7 Access control

If an infrastructure is configured and managed based on a specification, control of this specification implies control of the full infrastructure. It might be necessary to restrict access to the configuration specification. This is a challenge, especially in large infrastructures where multiple system administrators with different responsibilities need to make changes to this specification. Large infrastructures are often federated infrastructures, so one specification can be managed from different administrative domains.

Authenticating and authorizing system administrators before they make changes to the system configuration can for example prevent a junior system administrator, who is only responsible for the logging infrastructure, to make changes to other critical software running on managed devices.

Many version control systems can enforce access control but the level on which the authorisation rules are expressed differs from the abstraction level of the specification itself. In most systems, this is based on the path of the file that contains the code or specification. But in most programming languages and configuration management tools, the relation between the name of the file and the contents of the file is very limited or even non-existing. For example an authorisation rule could express that users of the *logging* group should only set parameters of object from types in the logging namespace. With path-based access control this becomes: users of group logging should only access files in the */config/logging* directory. The latter assumes that every system administrator uses the correct files to store configuration specifications.

2.1.4 Tool support

The final properties in the framework relate to how a user of a tool is supported: available documentation, commercial support, is there an active community and how mature is the tool.

2.1.4.1 Available documentation

To quickly gain users, tools have to make their barriers to entry as low as possible. A ten minutes tutorial is often invaluable to achieve this. When users get more comfortable with the tool, they need extensive reference documentation that describes all aspects of the tool in detail alongside documentation that uses a more process-oriented approach covering the most frequent use cases.

Thus, documentation is an important factor in the adoption process of a tool.

2.1.4.2 Commercial support

Studies [60] show that the need for commercial support varies amongst users. Unix users do not call support lines as often as their Window-colleagues. The same holds true for training opportunities. In all cases, the fact that there is a company actively developing and supporting a tool helps to gain trust amongst system administrators and thus increases adoption.

2.1.4.3 Community

In our online society, community building is integral part of every product or service. Forums, wiki's and social networks can provide an invaluable source of information that complements the official documentation of a tool and introduces system administrators to other users of their preferred tool.

2.1.4.4 Maturity

Some organizations prefer new features above stability, and others value stability higher than new features Therefore, it is important to know what the maturity of the tool is.

2.2 Tool comparison

This section provide a summary of the evaluation of eleven tools. These tools consist of commercial and open-source tools. The set of commercial tools is based on market research reports [27, 53] and consists of BMC Bladelogic Server Automation Suite, Computer Associates Network and Systems Management, IBM Tivoli System Automation for Multiplatforms, Microsoft System Center Configuration Manager and HP Server Automation System. For the open-source tools set of tools that were most prominently present in discussions at the previous LISA edition and referenced in publications is selected: it consists of BCFG2, Cfengine3, Chef, Netomata, Puppet and LCFG.

Due to space constraints the results of the evaluation is a summary of the findings for each property, the full evaluation of each tool is available on the website. This chapter based the evaluation on the versions of each tool listed in Table 2.1.

Tool	Version
BCFG2	1.0.1
Cfengine 3	3.0.4
Opscode Chef	0.8.8
Puppet	0.25
LCFG	20100503
BMC Bladelogic Server Automation Suite	8
CA Network and Systems Management (NSM)	R11.x
IBM Tivoli System Automation for Multiplatforms	4.3.1
Microsoft Server Center Configuration Manager (SCCM)	2007 R2
HP Server Automation System	2010/08/12
Netomata Config Generator	0.9.1

Table 2.1: Version numbers of the set of evaluated tools.

2.2.1 Specification properties

2.2.1.1 Specification paradigm

Language type Cfengine, Puppet, Tivoli, Netomata and Bladelogic use a declarative DSL for their input specification. BCFG2 uses a declarative XML specification. Chef on the other hand uses an imperative ruby DSL. LCFG uses a DSL that instantiates components and set parameters on them. CA NSM, HP Server Automation and MS SCCM are like LCFG limited to setting parameters on their primitives.

User interface As with the language type, the tools can be grouped in open-source and commercial tools. The open-source tools focus on command-line interface while the commercial tools also provide a graphical interfaces. Tools such as Cfengine, Chef and Puppet provide a web-interface that allows to manage some aspects with a graphical interface. In the commercial tools all management is done through command-line and graphical interfaces.

2.2.1.2 Abstraction mechanisms

All tools can at least express configurations in function of configuration files. The tool generates these configuration files from templates. Chef provides a framework to program the state of resources that it can enforce the desired state from, in an imperative programming language (Ruby). In Chef generic object oriented design can be used as an abstraction mechanism. In CA NSM, LCFG and IBM Tivoli higher abstraction level resources can be programmed in the tool but this is not available in the specification language itself. In Cfengine, Puppet and Bladelogic the specification language provides features to create simple abstractions similar to functions in a generic programming language. Cfengine and CA NSM can also define abstractions of heterogeneity.

2.2.1.3 Modularization mechanisms

Type of grouping All tools provide a grouping mechanism for managed devices or resources. HP Server Automation, Tivoli and Netomata only provide static grouping. CA NSM and BCFG allow static grouping and hierarchies of groups. LCFG supports limited static, hierarchical and query based grouping through the C-preprocessor. Bladelogic supports static, hierarchical and query based groups. Cfengine and Puppet use the concept of classes to group configuration. Classes can include other classes to create hierarchies. Cfengine can assign classes statically or conditionally using expressions. Puppet can assign classes dynamically using external tools. Chef and MS SCCM can define static groups and groups based on queries.

Configuration modules BCFG, HP Server Automation, MS SCCM and Netomata have no support for configuration modules. Bladelogic can parametrise resources based on node characteristics to enable reuse. Tivoli includes sets of predefined policies that can be used to manage IBM products and SAP. LCFG can use third party components that offer a key-value interface to other policies, CA NSM provides a similar approach for third party agents that manage a device or subsystem. Cfengine uses bundles, Chef uses cookbooks and Puppet uses modules to distribute a reusable configuration specification for managing certain subsystems or devices.

2.2.1.4 Modeling of relations

BCFG, CA NSM, HP Server Automation and MS SCCM have no support for modeling relations in a configuration specification. Bladelogic can model one-to-one dependencies between scripts that need to be executed as a prerequisite, these are instance relations. Cfengine supports one-to-one, one-to-many and many-to-many relations between instances, parameters and between parameters and instances. On these relations generative constraints can be expressed. Chef can express many-to-many dependency relations between instances. Tivoli can also express relations of all arities between instances and parameters and just like Cfengine express generative constraints. LCFG can express one-to-one and many-to-many relations using spanning maps and references between instances and parameters. Netomata can model one-to-one network links and relations between devices. Finally Puppet can define one-to-many dependency relations between instances. The virtual resource functionality can also be used to define one-to-many relations between all instances.

2.2.2 Deployment properties

2.2.2.1 Scalability

The only method to evaluate how a tool scales is to test each tool in a deployment and scale the number of managed nodes. In this evaluation does not do this. To have an indication of the scalability cases of real-life deployments are used. The tools are in three groups based on the number of managed devices and a group of tools for which no deployment information was available.

less than 1000 BCFG2

between 1000 and 10k LCFG and Puppet

more than 10k Bladelogic and Cfengine,

unknown CA NSM, Chef, HP Server Automation, Tivoli, MS SCCM and Netomata,

2.2.2.2 Configuration update workflow

BMC Bladelogic and HP Server Automation integrate with an orchestration tool to support coordination of distributed changes. Cfengine and Tivoli can coordinate distributed changes as well. MS SCCM and CA NSM support maintenance windows. Distributed changes in Puppet can be sequenced by exporting and collecting resources between managed devices. BCFG2, LCFG, Chef and Netomata have no support for workflow.

2.2.2.3 Deployment architecture

Translation agent Cfengine uses a strongly distributed architecture where the emphasis is on the agents that run on each managed device. The central server is only used for coordination and for policy distribution. Bladelogic, CA NSM and MS SCCM use one or more central servers. BCFG2, Chef, HP Server Automation, Tivoli, Netomata and Puppet use a central server. Chef and Puppet can also work in a standalone mode without central server to deploy a local specification.

Tool	Platform support
BCFG2	*BSD, AIX, Linux, Mac OS X and Solaris
Cfengine 3	*BSD, AIX, HP-UX, Linux, Mac OS X, Solaris and Windows
Opscode Chef	*BSD, Linux, Mac OS X, Solaris and Windows
Puppet	*BSD, AIX, Linux, Mac OS X, Solaris
LCFG	Linux (Scientific Linux)
BMC Bladelogic Server Automation Suite	AIX, HP-UX, Linux, Network equipment, Solaris and Windows
CA Network and Systems Management (NSM)	AIX, HP-UX, Linux, Mac OS X, Network equipment, Solaris and Windows
IBM Tivoli System Automation for Multiplatforms	AIX, Linux, Solaris and Windows
Microsoft Server Center Configuration Manager (SCCM)	Windows
HP Server Automation System	AIX, HP-UX, Linux, Network equipment, Solaris and Windows
Netomata Config Generator	Network equipment

Table 2.2: Version information for the set of evaluated tools.

Distribution mechanism The deployment agent of BCFG2, Cfengine, Chef, LCFG, MS SCCM and Puppet pull their specification from the central server. Bladelogic, CA NSM, HP Server Automation and Tivoli push the specification to the deployment agents. The central servers of Chef, MS SCCM and Puppet can notify the deployment agents that a new specification can be pulled. Netomata relies on external tools for distribution.

2.2.2.4 Platform support

The platforms that each tool supports is listed in Table 2.2. Linux is the most supported platform: each tool supports it except for Netomata which focusses on network equipment only and of course Microsoft's solution for Windows. With Linux most tools also support commercial Unix such as AIX and Solaris. The tools that also have commercial support, tend to support Windows.

2.2.3 Specification management properties

2.2.3.1 Usability

Usability Usability is a very hard property to quantify. The tools are categorised in easy, medium and hard. This was determined by assessing how easy a new user would be able to use and learn a tool. The evaluation of this property was as objective as possible carried out, but this part of the evaluation can be subjective. We found Bladelogic, CA NSM, HP Server Automation, Tivoli and MSCCM easy to start using. The usability of Cfengine, LCFG and Puppet is medium, partially because of the custom syntax. Puppet also has a lot of confusing terminology but tools such as puppetdoc and puppetca make up for it so it was not classified as hard to use. BCFG2 is hard to use because of the XML input and the specification is distributed in a lot of different directories because of their plugin system. Chef is also hard to use because of its syntax and the use of a lot of custom terminology. Netomata is also hard to use because of its very concise syntax.

Support for testing specifications BCFG2, Cfengine, LCFG and Puppet have a dry run mode. Netomata is inherently dry-run because it has no deployment part. Chef and Puppet support multiple environments such as testing, staging and production.

Monitoring the infrastructure BCFG2, Bladelogic, HP Server Automation, CA NSM, Tivoli, LCFG, Puppet and MS SCCM have various degrees of support for reporting about the deployment and collecting metrics from the managed devices. The commercial tools have more extensive support for this. Chef, LCFG, Puppet and Netomata can automatically generate the configuration for monitoring systems such as Nagios.

2.2.3.2 Versioning support

BCFG2, Bladelogic, Cfengine, Chef, Tivoli, LCFG, Netomata and Puppet use a textual input to create their configuration specification. This textual input can be managed in an external repository such as subversion or git. CA NSM and MS SCCM have internal support for policy versions. The central Chef server also maintains cookbook version information. For HP Server Automation it is unclear what is supported.

2.2.3.3 Specification documentation

BCFG2, Bladelogic, Chef, HP Server Automation, Tivoli, LCFG, Netomata and Puppet specifications can include free form comments. Cfengine can include structured comments that are used to generate documentation. Because Chef uses a Ruby DSL, Rdoc can also be used to generated documentation from structured comments. Puppet can generate reference documentation for built-in types from the comments included in the source code. No documentation support is available in CA NSM and MS SCCM.

2.2.3.4 Integration with environment

BCFG2, Cfengine, Chef, Tivoli, LCFG, MS SCCM and Puppet can discover runtime characteristics of managed devices which can be used when the profiles of each device are generated. Bladelogic can interact with external data sources like Active Directory.

2.2.3.5 Conflict management

BCFG and Puppet can detect modality conflict such as a file managed twice in a specification. Cfengine3 also detects modality conflicts such as an instable configuration that does not converge. Bladelogic and CA NSM have no conflict management support. Puppet also supports modality conflicts by allowing certain parameters of resources to be unique within a device, for example the filename of file resources.

2.2.3.6 Workflow enforcement

None of the evaluated tools have integrated support for enforcing workflows on specification updates. Bladelogic can tie in a change management system that defines workflows.

2.2.3.7 Access control

The tool that support external version repositories can reuse the path based access control of that repository. BMC, CA NSM, HP Server Automation, Tivoli, MS SCCM and the commercial version of Chef allow fine grained access control on resources in the specification.

2.2.4 Tool support

2.2.4.1 Available documentation

Bladelogic, CA NSM and HP Server Automation provide no public documentation. IBM Tivoli provides extensive documentation in their evaluation download. BCFG2, Cfengine, Chef, LCFG, MS SCCM and Puppet all provide extensive reference documentation, tutorials and examples on their websites. Netomata provides limited examples and documentation on their website and Wiki.

2.2.4.2 Commercial support

Not very surprising the commercial tools all provide commercial support. But most open-source tools also have a company behind them that develops the tool and provides commercial support. LCFG and BCFG2 have both been developed in academic institutes and have no commercial support.

2.2.4.3 Community

Cfengine, Chef, Tivoli, MS SCCM and Puppet have large and active communities. BCFG2 has a small but active community. CA NSM has a community but it is very scattered. BMC, Netomata and LCFG have small and not very active communities. For HP Server Automation it was not possible to determine if a community exists.

2.2.4.4 Maturity

Some of the evaluated tools such as Tivoli and CA NSM are based on tools that exist for more than ten years, while other tools such as Chef and Netomata are as young as two years. However no relation between the feature set of a tool and their maturity seems to exist.

2.3 Areas for improvement

The evaluation in Section 2.2 identified six areas for improvement in the current generation of tools. Tools that can address these areas will significantly advance the current state-of-the-art. The areas are:

1. **Input specification** Very few tools support creating higher-level abstractions like those mentioned in Figure 2.1 on page 17. If they do, those capabilities are hidden deep in the tool's documentation and not used often. We believe this is a missed opportunity. Creating higher-level abstractions would enable reuse of configuration specifications and lower the TCO of a computer infrastructure. To realize this, the language needs to (a) support primitives that promote reuse of configuration specifications like parametrization and modularization primitives, (b) support constraints modeling and enforcement, (c) deal with conflicts in the configuration specification and (d) model and enforce relations.

   The commercial tools in the study all start from scripting functionality: the system administrator can create or reuse a set of scripts and the tool provides a script-management layer. Research and experience with many open-source tools has shown that declarative specifications are far more robust than the traditional paradigm of imperative scripting. Imperative scripts have to deal with all possible states to become robust which results in a lot of if-else statements and unmaintainable spaghetti-code.

2. **Support true integrated management**: A tool should provide a uniform interface to manage all types of devices that are present in a computer infrastructure: desktops, laptops, servers, smartphones and network equipment. One tool with a single input specification for all devices allows each system administrator to *speak* the same language and think in the same abstractions: whether they are responsible for the network equipment, the data center or your desktops. The tool can then also support the specification and enforcement of relationships that cross platform boundaries: the dependencies between your web server farm and your Cisco load balancer, dependencies between desktops and servers, dependencies between your firewall and your DMZ servers, ... The current generation of tools either focuses on a single platform (Windows or Unix), focuses on one type of devices (servers) or needs different products with different interfaces for your devices (one product for network equipment, one for servers and one for desktops).

3. **Adapt to the target audience's processes**: A tool that adapts to the processes for system administration that exist in an organization is much more intuitive to work with than a tool that imposes its own processes on a system administrators. A few examples of how tools could support the existing processes better:

   - *structured documentation and knowledge management*: Cfengine3 is the only tool in the study that supports structured documentation in the input specification and has a knowledge management system that uses this structured documentation. Yet, almost all system administrators document their configurations. Some do it in comments in the configuration specification, some do it in separate files or in a fully-fledged content management system. In all cases, documentation needs to be kept in sync with the specification. Structured documentation in the configuration specification allows the tool to generate the documentation automatically.

   - *integrate with version control systems*: A lot of system administrator teams use a version control system to manage their input specification. It allows them to quickly rollback a configuration and to see who made what changes. Yet, very few tools provide real integration with those version control systems.
   - *semantic access controls*: In a team of system administrators, every admin has his own expertise: some are expert in managing networking equipment, other know everything from the desktop environment the company supports, others from the web application platform, ... As a consequence, responsibilities are assigned based on expertise and this expertise does not always align with machine boundaries. The ability to specify and enforce these domains of responsibility will prevent that for example a system administrator responsible for the web application platform modifies the mail infrastructure setup.
   - *flexible workflow support*: Web content management systems like Drupal have support for customized workflows: If a junior editor submits an article, it needs to be reviewed by two senior editors, all articles need to be reviewed by one of the senior editors, ... The same type of workflows exist in computer infrastructures: junior system administrators need the approval from a senior to roll out a change, all changes in the DMZ need to be approved by one of the managers and a senior system administrator, ... Enforcing such workflows would lower the number of accidental errors that are introduced in the configuration and aligns the operation of the tool with the existing processes in the organization.

   Additionally a focus on existing business processes is required: Most open-source tools in the study have their origin in academia. As a result, they lag behind on the features that are on the CIO's checklists when deciding on a configuration management tool: easy to use (graphical) user interface, reporting, auditing, compliance, reporting capabilities in nice graphs and access control support.

4. **A system is software + configuration + data**: No tool has support for the data that is on the managed machines. For example web server software needs needs configuration files and serves data. Configuration management tools can manage the software and configuration but have no support for state transfer: if the web server is moved to another node, the data needs to be moved manually.

2.4 Conclusion

The evaluation framework in this chapter can help both research to evaluate existing configuration management tools and help system administrators make a more informed, and as a consequence, a better choice for a configuration management tool. The framework is not a mechanical tool: it does not provide a check list based list which results in the perfect tool for a given environment. It does provide a structured overview of key properties to evaluate different tools: it quickly gives a high-level overview of the features of each tool.

The areas of improvement provide researchers with valuable input to determine a research agenda and provide requirements for next generation tools that advance the state-of-the-art. The different properties and the areas of improvement are directly used in this dissertation as input to improve how large distributed systems can be managed more cost effectively with less errors.

Supporting true integrated management and raising the abstraction level are important areas of research concerned with how the input of a configuration management tool is specified. Both areas of improvement are intertwined: only when the description of a distributed system is described at its highest level of abstraction, can it be refined to all subsystems and services that are part of the execution environment of a distributed system. These areas of improvement directly relate to the problem statement in Section 1.3: (1) a description of the configuration at a high abstraction level, (2) an integrated configuration model that manages the entire configuration, (3) refine the description at high abstraction level to lower levels of abstraction. Additionally, to handle state (application data) and state transitions (migrating a service to another server) an integrated model is required that captures all runtime dependencies between managed resources.

Chapter 3

A framework for integrated configuration management

This chapter proposes the Integrated configuration Management Platform, named IMP. It is a configuration management framework for managing complex and large scale distributed system in an integrated manner. The framework described in this chapter is the result of an iterative process of design, prototype implementation, implementing case studies and evaluating the cases. This chapter describes the rationale behind the current version of IMP. This rationale is complemented with a language reference (Appendix A), a tutorial (Appendix B) and the availability of the current version of the prototype and configuration modules support by this version (listed in the Publications chapter).

First, this chapter introduces IMP in section 3.1, followed by an overview of the platform architecture in section 3.2 and a description of the design of its two subsystems. Section 3.3 introduces the modeling language that is used to create the input specification of IMP. Section 3.4 goes deeper into how an integrated configuration model is generated and how the refinement mechanism works. IMP generates configuration artifacts from the complete and refined configuration model, this mechanism and the configuration targets are explained in section 3.5. Section 3.6 explains how IMP deploys the changes required to change the state of the managed resources to the desired state described in the configuration model.

3.1 Introduction

The IMP framework provides developers and operators of a distributed system with the following features to support true integrated management:

1. A configuration language for modeling and refining configurations.
2. A configuration generation system, to generate implementation-level configuration artifacts (such a configuration files, network switch definitions, service state, etc.) from a high level configuration.
3. A configuration deployment system, to manage the orchestration of configuration changes in a gradual, controlled and robust way.

The functionality described above gives operators the power to create an integrated configuration model. This configuration model matches the abstraction level that is used to reason about distributed systems: connections between components of the application and the allocation of components to machines in the infrastructures. This model should be a composition of reusable configuration modules combined with application specific configuration modules (see section 1.2). This configuration model can be used to manage a single subsystem on a single machine to a fully integrated configuration model. Five scenarios represent this range of deployment scenarios (Figure 3.1):

1. Use IMP to manage a single subsystem on a single machine. This is the minimal usage scenario of IMP. IMP does not manage any other subsystems on this machine or any other machines. These machines and subsystems are managed with an other tool or manually.
2. Use IMP to manage all subsystems on a single machine. This is an integrated configuration model for this single machine. However, inter-machine configuration parameters need to be duplicated.
3. Use IMP to manage a single subsystem on all machines where this subsystem is deployed. This is an integrated configuration for that single subsystem. This scenario also requires configuration parameters from other subsystems to be duplicated.
4. Use IMP with a true integrated configuration model. IMP manages all machines and all subsystems in the entire distributed systems. No configuration parameters need to be duplicated in the configuration.
5. Use IMP to manage all machines and all subsystems with an integrated configuration model. Additionally, the machines itself are also provisioned from

the configuration model. The previous scenarios start from machines which exist, e.g. physical servers in a rack, which received network connectivity from the NOC. In this scenario the infrastructure is fully or partially programmable, e.g. virtual machines on an IaaS [12, 50], software defined networking [49, 62, 116], software define storage [43, 70, 82]. For example, the IaaS determines which IP address a virtual machine receives. IMP should first provision the virtual machine so the IP address gets allocated, before it can proceed to configure the subsystems allocated to the virtual machine. Therefore IMP needs multiple iterations of generating and deploying the desired state, before IMP can generate and deploy a *complete* desired state.

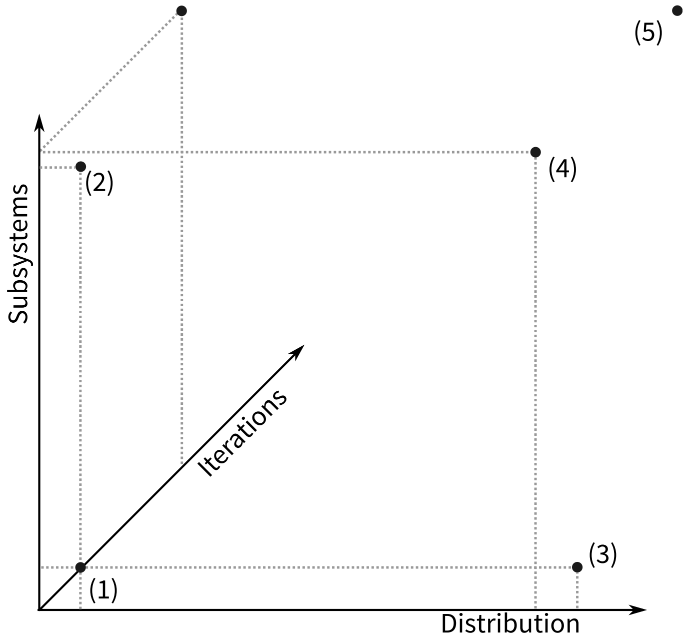


Figure 3.1: IMP usage scenarios illustrated in a three dimensional plane which represents the complexity of the configuration.

The scenarios described above lead to the following requirements for IMP:

- The input of IMP has to describe the desired state of the entire managed distributed system in single configuration model. This integrated model allows IMP to derive the actions required to bring the actual state of the distributed system to the desired state.

- IMP has to be able to enforce the desired state of the input on real world distributed systems deployed in a realistic execution environment. Additionally IMP has to support heterogeneous devices: servers, network equipment, programmable infrastructure,... from a single integrated configuration model.
- An integrated configuration model of a distributed system requires a substantial code base. Therefore, it should be created modular and allow for encapsulation of complexity to ensure reuse of configuration model code and make the development manageable. Additionally, the level of abstraction at which resources of the distributed system are managed can vary greatly and thus requires support to define all levels of abstraction in a single configuration model. For example, the binding between a hostname and an IP address. For a distributed system on a classic infrastructure with physical hardware, DNS server software is installed on a server and a zonefile with all hostname/IP address bindings needs to be generated. This same distributed system deployed on Amazon AWS, can use Amazon's Route 53 API [13] to directly configure this binding.
- The input has to capture all parameter relations at all level of abstraction: direct relations and relations that require a transformation of the parameter value. Additionally, such relations can include retrieving a parameter value from an external datasource, based on a parameter value in the configuration model. For example, retrieve the IP address of a virtual machine from an IaaS, based on the name of the virtual machine.
- IMP needs to support deployment dependencies that span machine boundaries: runtime dependencies of distributed systems and their services are not contained within machines. For example, server 1 depends on service 2 so start service 1 on machine A, only after server 2 on machine B has been started.
- Every managed infrastructure is unique and a one size fits all solution to configuration management is an illusion. IMP should reduce the need for custom scripts around and next to IMP to a minimum. A framework approach can provide the support and facilities to develop custom scripts that integrate with the configuration model. The definition of the model provides a stable interface for the scripts. Additionally, the desired state description of IMP ensures that operations are idempotent.

3.2 Platform architecture

The architecture of the IMP platform consists of a generation and an enforcement subsystem. The generation subsystem enables stepwise refinement of high-level configuration descriptions, and the automated generation of configuration artifacts. The enforcement is responsible for gradually deploying the generated configuration artifacts onto the managed distributed system or infrastructure. The design of the architecture (Figure 3.2) follows the principle of a software framework. Users of IMP can complement the functionality of both subsystems with a plug-in mechanism. This mechanism allows its users to integrate ad-hoc scripts in the well defined and reproducible configuration generation of IMP.

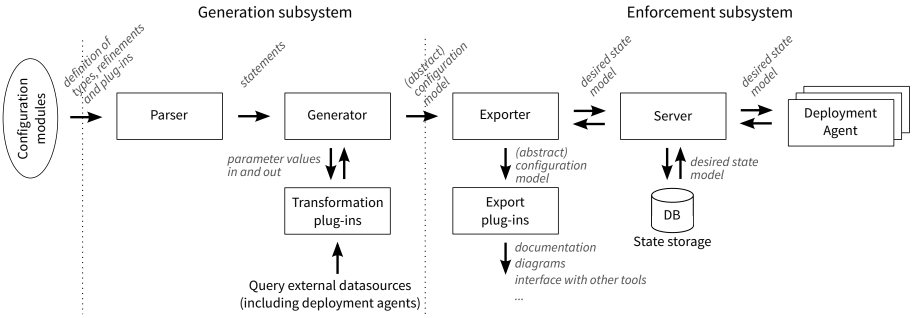


Figure 3.2: The architecture of IMP with its two main subsystems and the components in each of the subsystems.

The generation subsystem uses an input specification to create an integrated configuration model that expresses the desired state of the managed distributed system. This subsystem is responsible for the compilation process of IMP's modeling language and to execute the refinement process to generate an integrated configuration model (Figure 3.3). The input specification is contained in configuration modules that bundle reusable and modular configuration models. Each configuration module defines types and refinements for types. Modules do not instantiate entities (except in refinements), the initial model contains all instantiations.

A system administrator creates an initial configuration model that instantiates types defined in the configuration modules. IMP refines the initial configuration model to a complete configuration model (more on this process in section 3.4) that contains the desired state of all resources in the distributed system that IMP manages. IMP uses the refinements contained in the configuration modules. During the refinement process IMP can use transformation plug-ins to extend the refinement process.

Subsequent refinement iterations generate a layered configuration model that expresses instances (an instance has a type that is defined in a configuration module) in one layer in function of instances of the next layer. Therefore the initial model defines the first layer of the abstract configuration model. IMP refines the configuration model until it completes the entire model. However, to support the fifth scenario IMP needs to deploy configuration models that are not entirely completed because it is possible that a model requires multiple generate-deploy iterations before IMP has all configuration parameters to refine a complete configuration model.

The enforcement subsystem is responsible for enforcing the generated desired state upon the distributed system and its execution environment. This subsystem manages the state of the resources in the distributed system that are represented in the completed configuration model. Enforcing the desired state is either performed by IMP's built-in deployment agents and server or by 3rd party management tools. In the enforcement subsystem the completed configuration model is either exported by an export plug-ins to 3rd party tools or to a desired state model that only contains the state of the managed resources (Figure 3.3).

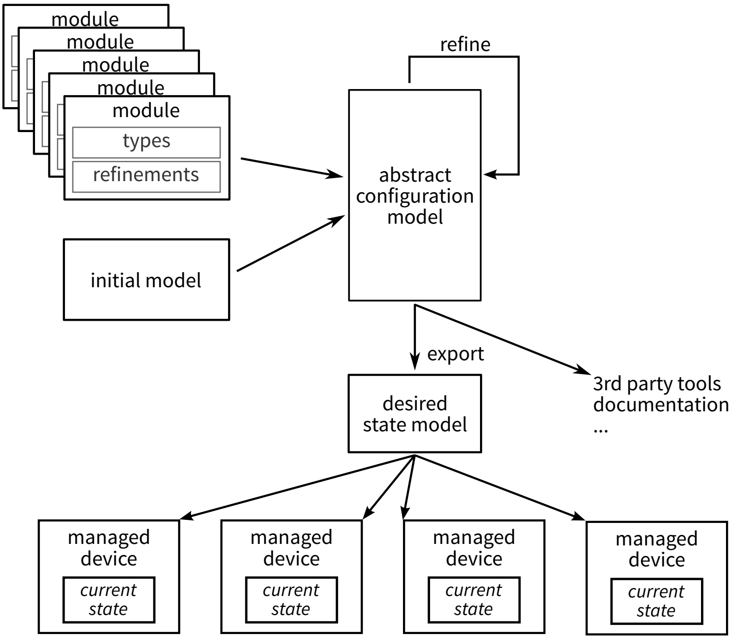


Figure 3.3: An overview and the relation between the different models used in IMP.

The deployment agents compare the desired state in the model, with the current state of the managed resources. From this comparison it derives the actions to advance the current state to the desired state. The deployment agent applies the concept of convergence [20] to ensure that failures during actions or missing runtime dependencies are resolved during the deployment process. This process also automatically detects and reverts external changes due to bugs, manual changes, etc.

3.3 Modelling language

IMP has its own modeling language (DSL) to specify the IMP configuration model. The completed configuration model is a declarative desired state model [20, 30], inspired by concepts from object-oriented software development [33]. IMP encourages to develop a modular configuration model. Each module defines types and refinements that can be used by other modules or to create an initial configuration model. The IMP DSL contains three important concepts to define types and refinements:

Entity An entity is a representation of a resource that is configured or is part of the (real world) configuration. The initial configuration model and the refinements instantiate instances of entities. Entities closely resemble classes in object-oriented programming languages, except that an entity only defines an interface and does not have an implementation. Entities have attributes with primitive types such as string, number or boolean or a one of the former types with additional constraints. For example, a string that only accepts valid IP addresses. Entities can inherit attributes from multiple parents. Line 5 in Figure 3.4 defines a new entity that models a webapplication that by default is hosted by a single webserver, but can have a higher replication factor if the load requires this.

Relation A relation defines an association between two entities. Each relation defines a multiplicity for both ends. Relations between instances of an entity are an important mechanism to allow operators to define parameter values only once, to ensure that the configuration is consistent. Line 12 of Figure 3.4 defines a relation between a WebHost and a Httpd server. On lines 24 and 40 the instantiated httpd server is added to the relation (assignment to a list results in an append).

Implementation Entities (with its attributes and relations) provide a well-defined interface for a managed object. However, this does not mean that it is possible to directly manage that resource. An implementation defines a refinement for entities. Each implementation refines an instance of an entity in function of instances of other entities. Lines 15 and 28 in Figure 3.4 define alternate implementations for the WebHost entity. Lines 46 and 47 connect the entity and the implementations based on a condition, whether the application is deployed on a single or multiple webservers. In the case of multiple webservers a loadbalancer is included.

The initial model instantiates types from one or more modules for a specific distributed system. IMP refines this initial model, based on the available refinements. A system administrator defines an initial model by including multiple configuration modules and creating one or more instances from the types in these modules. This initial model does not yet explicitly define the desired state of all managed resources: it still

```
# define hoststring as string with a constraint, it should match the given
    ↪ regular expression
typedef hoststring as string matching /^[A–Za–z0−9−]+(\.[A–Za–z0−9−]+)*\$/

# define a new entity (type) with three attributes with a primitive type
entity WebCluster:
    string application
    hoststring hostname
    number replication = 1
end

# a webcluster contains webservers (webservers can exists without being
    ↪ defined inside a webcluster)
WebCluster web_host [1:] — [0:] httpd::Server webservers

# refinement for a single webserver webcluster
implementation webServer for WebCluster:
    # define a virtual machine for the webserver and retrieve its IP from
        ↪ the IaaS that hosts the vm
    vm = vm::Host(name = ”web−1”, os = ”centos−6”)
    vm.ip = vm::get_vm_ip(vm)

    # define a new webserver on hte vm
    server = httpd::Server(host = vm, application = application, servername
        ↪ = hostname)

    # set the webserver − webcluster relation
    self.webservers = server
end

# refinement for a loadbalancer and 2 or more webservers
implementation webCluster for WebCluster:
    lb_vm = vm::Host(name = ”loadbalancer−1.{{␣hostname␣}}”, os = ”centos−6”)
    lb_vm.ip = vm::get_vm_ip(lb_vm)

    lb = proxy::LoadBalancer(host = lb_vm, servername = hostname)

    for i in sequence(replication):
        vm = vm::Host(name = ”web−{{␣i␣}}.{{␣hostname␣}}”, os = ”centos−6”)
        vm.ip = vm::get_vm_ip(vm)

        http_server = httpd::Server(host = vm, application = application)

        lb.backend_servers = http_server
        self.webservers = http_server
    end
end

# connect the two available refinements with the webcluster entity, choose
    ↪ the implementation based on the replication level
implement WebCluster using webServer when replication == 1
implement WebCluster using webCluster when replication > 1
```

Figure 3.4: An example of a module definition in the IMP DSL. It defines the types of the module and two possible refinements.

```
...

# create a web cluster that deploys the application large_app on 10 machines
large_cluster = web::WebCluster(hostname = "large", application = large_app,
    ↪ replication = 10)

# create a web host cluster that deploys the application small_app on single
    ↪ webserver
small_cluster = web::WebCluster(hostname = "small", application = small_app,
    ↪ replication = 1)
```

Figure 3.5: The initial configuration model that instantiates two web clusters using the types defined in Figure 3.4.

is an abstract configuration model. For example Figure 3.5 shows an instantiation of the types from the web module defined in Figure 3.4.

IMP refines the abstract configuration model of Figure 3.5 in the following refinement steps. These steps represent a simplified version of the refinement process, a complete version is discussed in section 3.4:

1. Instantiate two instances of web::WebCluster, as defined in Figure 3.5.
2. IMP refines the two applications according to the available refinements. In the case of web::WebCluster two refinements (implementation) are defined, each with a different conditions when they should be used. IMP refines the large application according to the `webCluster` implementation and the small application according to the `webServer` implementation. IMP selects the refinements based on the replication level.

   This example only follows the refinement of the large application. To refine a WebCluster instance with replication level 10. IMP instantiates 11 virtual machines, of which one is named `loadbalancer-1` and the other are named `web-1` to `web-10`. IMP also creates an instance of a loadbalancer that it deploys on the loadbalancer virtual machine and it creates 10 instances of an httpd server that is deploys on the 10 dedicated virtual machines. Additionally it links the loadbalancer and the webservers so the proxy configuration module can configure the loadbalancer.
3. In this iteration IMP refines the instances of `proxy::LoadBalancer` and `httpd::Server`. Instances of `vm::Host` have no refinement. However, they have a resource handler in the enforcement system that enforces the desired state of `vm::Host`. IMP refines each server instance (of loadbalancer and http server) to instances of the type:

   std::Package Instances of this type represent software packages that need to be installed on the target virtual machine. In this example (provided by existing configuration modules not mentioned in the example), IMP will install Apache HTTPD [10] for instances of `httpd::Server` and Varnish [107] for instances of `proxy::LoadBalancer`.

std::File IMP refines server instances to multiple configuration files. IMP uses templates to generate the contents of these configuration files. A template is a transformation plug-in (albeit a complex one) that queries the configuration model.

std::Service IMP refines servers to instances of a service. These represent operating service which on Unix systems are typically controlled by the init process. In a well designed configuration module, these services have requires relations to the packages and configuration files that determine how the service should function. This allows the IMP enforcement subsystem to schedule configuration updates in the correct order: only start a service after it has been installed and configured or restart a service when its configuration changes.

The refinement process in the example completes when all entities have been refined. In the example the refinement stops at the entities of the std configuration module. These entities represent managed resources that can be managed by IMP's enforcement subsystem. In the std module they have empty refinements. Section 4.1.2 further elaborates on these entities.

Figure 3.6 shows the generated configuration artifacts after refinement step 3. It generated 11 virtual machines, that each have their hostname and IP-address parameters configured (other modules not shown in the example, provide the refinements to generate the configuration files to set these two parameters). These two parameter are used within the vm itself and on other virtual machines. Additionally, configuration files on all virtual machines contain global parameters from the initial configuration model.

Many configuration parameters are either duplicate, derived from one or more other parameters or they are available in an external databases (as shown in Figure 3.5. IMP uses relations to model such parameter dependencies. However, not all relations between parameters can be expressed with a relation. Transformation plug-ins extend the IMP DSL to provide complex parameter transformations based on the modelled relations. A transformation plug-in provides functions written in an imperative programming language. Each function of a plug-in accepts parameters and returns a new configuration parameter.

Figure 3.4 uses the `get_vm_ip` transformation plug-in from the vm module to lookup a configuration parameter in an external database. This transformation plug-in gets a reference to the instance that represents the virtual machine. The plug-in uses the

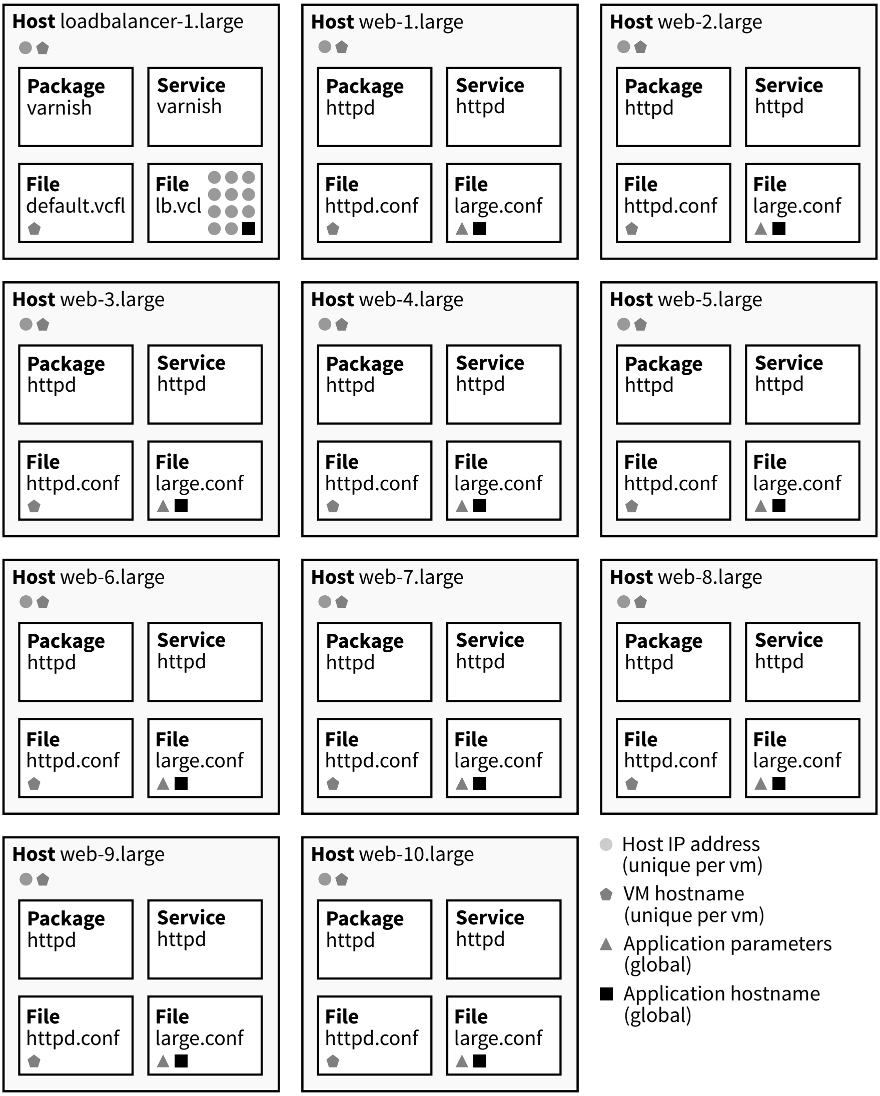


Figure 3.6: The generated configuration (low level) artifacts after the refinement of Figure 3.5. The symbols indicate the duplication of configuration parameters in the generated configuration artifacts.

name of the instance to lookup the IP address of the virtual machine, and stores that value in the ip attribute of the vm instance in the configuration model.

Figures 3.7 and 3.8 show an example of the use of the template transformation plug-in. The template plug-in is a very useful transformation plug-in because it generates configuration file content from the configuration model. Figure 3.7 shows a possible refinement of the proxy::LoadBalancer entity using Varnish. This example defines that the varnish package needs to be installed, that two configuration files need to be installed with their content based on a template and it ensures that the varnish system service is running and enabled at boot time. The template to generate the load-balancer configuration, based on the relations in the configuration model (Figure 3.4), is shown in Figure 3.8.

The template transformation plug-in takes the path of the template as argument. The template plug-in can also access variables in the scope where the template is called. The template (Figure 3.8) uses the webservers attribute to get a list of all backend webservers (line 2). From each backend webserver the template gets the name and the ip of the host the webserver is deployed on (lines 3 and 4).

```
implement proxy::LoadBalancer using varnishProxy

implementation varnishProxy for proxy::LoadBalancer:
    # install the varnish package
    p_varnish = std::Package(host = host, name = ”varnish”, state =
        ↪ ”installed”)

    # deploy its main configuration file with generic settings. This file
        ↪ may only be deployed when the varnish package is installed. All
        ↪ resources that depend on this file, should ”reload” whenever
        ↪ this file changes.
    f_main_config = std::File(host = host, path =
        ↪ ”/etc/varnish/default.vcl”, owner = ”root”, group = ”root”, mode
        ↪ = 644, content = template(”varnish/default.vcl”), requires =
        ↪ p_varnish, reload = true)

    # deploy the configuration file that configures varnish as loadbalancer
    f_lb_config = std::File(host = host, path =
        ↪ ”/etc/varnish/loadbalancer.vcl”, owner = ”root”, group = ”root”,
        ↪ mode = 0644, content =
        ↪ template(”varnish/loadbalancer.vcl.tmpl”), requires = p_varnish,
        ↪ reload = true)

    # ensure the varnish service is running and that it starts at boot time.
        ↪ This service also depends on its configuration files. When these
        ↪ change, the service will reload.
    std::Service(host = host, name = ”varnish”, state = ”running”, onboot =
        ↪ true, requires = [f_main_config, f_lb_config])
end
```

Figure 3.7: An example of the usage of the template transformation plug-in.

```
# Define the list of backends (web servers).
{% for backend in webservers %}
backend {{ backend.host.name }} {
    .host = ”{{ backend.host.ip}}”;
    .probe = {
        .url = ”/”;
        .interval = 5s;
        .timeout = 1s;
        .window = 5;
        .threshold = 3;
    }
}
{% endfor %}

# Define the director that determines how to distribute incoming requests.
director default_director round-robin {
{% for backend in webservers %}
    { .backend = {{ backend.host.name }} }
{% endfor %}
}

# Respond to incoming requests.
sub vcl_recv {
    # Set the director to cycle between web servers.
    set req.backend = default_director;
}
```

Figure 3.8: The template (varnish/loadbalancer.vcl.tmpl) that generates the contents of the configuration file for the Varnish proxy to load-balance its requests over multiple backend servers.

3.4 Model refinement

IMP refines the initial configuration model, which is a high level description of a distributed system, to the exact state of all managed resources of the distributed system. This section starts with a detailed explanation of how the configuration model is evaluated from the input to the actual configuration model. Next, it provides more details on how IMP handles configuration models for which not all parameters are known, until the model is partially deployed (scenario 5). Finally, it describes how IMP integrates the imperative code of the transformation plug-ins in the refinement process.

3.4.1 Refining the configuration model

The generation subsystem builds a complete configuration model by evaluating the modeling DSL. It executes the following steps to build the configuration model:

- Load all configuration modules in the search path of IMP and parse the input specification in the configuration modules to statements. Each statement represents an operation on the configuration model. For example, define a new entity with the following name and attributes, or assign a value to an attribute of an instance.

- Evaluate all statements that define the types and refinements of the configuration model: defining entities, relations, implementations, ...
- The refinement process is started to instantiating entities and then refine these entities until all entities are refined. A system administrator should instantiate at least one instance, defined in the initial model, for IMP to start the refinement process. Typically this initial model is defined in an external file (not in a configuration module) and explicitly passed to the framework.

The next paragraphs explain how IMP performs each step and provide the algorithms behind these steps.

Generation subsystem initialization The subsystem initializes itself by processing all configuration modules available in the search path (Figure 3.9). For each module, IMP will load the plug-in code and register it with the platform. Next, IMP compiles the DSL code to a list of statements. Each statement is a single operation on the model. For example: assign a value to a variable, instantiate an entity, etc. Finally, each statement is registered with the platform.

```
for module in compiler.get_configuration_modules():
    for plugin in module.plugins:
        runtime.register_plugin(plugin)

    statements = module.parse_dsl_code()
    runtime.queue_statements(statements)
```

Figure 3.9: A pseudo-code representation how IMP loads each configuration module

DSL model statements The statements from the DSL that operate on the configuration model transition through multiple stages before they are evaluated. IMP makes a distinction between two types of statements: statements that define types and statements that instantiate or refine the configuration model. Evaluating type statements is straightforward because types (and thus all types statements) are known before the evaluation starts. Statements that instantiate or refine are more complex because when an entity is refined by IMP, new statements from the selected implementation need to be evaluated. Each statement goes through the following stages:

1. `types` Collect the entity types this statements uses. For example, a constructor statement (it instantiates an entity) requires the entity that it instantiates.

2. `references` Collect references to instances or attributes the statement will perform actions upon. For example, all values that are assigned to attributes in the constructor statement.

3. `new_statements` Collect additional statements that this statement generates. This method is for example used by the constructor statement to set attributes that are defined in the constructor statement. The constructor statement will in this stage, emit SetAttribute statements.

4. `actions` Collect the actions the statements performs upon the references. Possible actions are: get the value of a variable, set a value of a variable and add a value to a list.

5. `can_evaluate` IMP requires that all dependencies of a statement are resolved and it has provided the actions it performs on its references, before it can be evaluated. The statement itself can require for additional conditions to be met to determine if the statement can be evaluated.

6. `evaluate` Evaluate the statement. The statement will perform its actions on the configuration model. At this point, statements can halt the execution and indicate that it will perform more actions than initially indicated. This mechanism is used by the statement that handles plug-in calls to indicate additional dependencies in the imperative plug-in code (more on this further on in this section)

Type statements only go through stages 1 and 6, stages 2 to 5 are only required for the statements that operate on the full configuration model.

Processing types First IMP loads the types and refinements from each of the loaded configuration modules (Figure 3.3) to initialise the type system of the compiler. The compiler starts with an empty model on each invocation. It first initializes the model by defining primitive types such as string, number and boolean and Entity (which is an implicit parent of all other entities). It then creates types by evaluating each type statement. These statements include defining new entities or relations between entities. During this phase, IMP verifies if all dependencies between configuration modules have been met.

This evaluation process is shown in pseudo-code in Figure 3.10. The first line defines a dependency graph to which all statements are added, the second line creates the empty configuration model. The dependency graph registers all statements and creates a graph based on the dependencies of each statement. This graph determines in which order IMP will evaluate statements. Lines 9 to 23 evaluate type statements iteratively

```
graph = DependencyGraph()
model = Model()
statements = runtime.get_statements()

# register statements in the dependency graph
for stmt in statements:
    graph.add(stmt, stmt.types())

# evaluate all statement until an iteration does not provide any progress anymore
old_len = len(statements) + 1
while old_len > len(statements):
    old_len = len(statements)
    for stmt in statements:
        if len(stmt.dependencies) == 0 and stmt.is_metastatement():
            meta_model.evaluate(stmt)
            for other in stmt.dependents():
                other.dependencies.remove(stmt)

            statements.remove(stmt)

if len(statements) > 0:
    report_unresolved_statements()
    exit()
```

Figure 3.10: Processing the types from the model code defined in the configuration modules. (Error-handling, detecting dependency loops or incomplete references are left out for brevity.)

until all statements are evaluated or until the two consecutive iteration do not provide any progress anymore. If the statement list after the iteration still contains statements, the user receives an error report with the cause. In this step, this is often caused by missing dependencies.

Refining the configuration model In this step the configuration model is created by refining the entities defined in the initial configuration model (Figure 3.11). This is, like the previous step, an iterative process. Each statement is added to the dependency graph (in the first iteration these are the statements that define the initial model). This graph uses the types, references to variables and the actions on the references from stage 1, 2 and 4 to determine if all dependencies of a statement have been resolved and have been evaluated. At this point all conditions are met for IMP to evaluate the statement. During each iteration, statements the refine instances of entities, generate new statements. The refinement iteration ends when all statements have been evaluated or the refinement process does not progress between to consecutive iterations.

Determining if all conditions for evaluation are met is done in iteratively in two phases: in each iteration of the algorithm the runtime queues all available statements, of which all references are resolved, into three queues: the evaluation queue, the plug-in queue and the list queue. When all references resolve, the runtime collects the actions that a statements performs on the configuration model and adds these to the dependency graph as well.

On each iteration the `evaluate_queue` routine tries to evaluate all statements currently in the evaluation queue. It evaluates these statements iteratively when the dependency graph and the statement itself indicate that it can be evaluated. The method keeps iterating until either the evaluation_queue is empty or there are no statements left of which their dependencies are available. Statements can generate new statements during their execution (for a example to refine the configuration model). These statements will be included in the next iteration of the main loop in the runtime when the `queue_statements` routine is called. The algorithm described above evaluates all statements except the statements that use lists and that execute transformation plug-ins. The algorithm delays these because both are difficult to schedule correct. Therefore the runtime only evaluates them when the execution of the other statements does not progress anymore or if there are no other statements left to evaluate.

- Statements that execute transformation plug-in code are delayed as long as possible because their dependencies are not (or only partially) known before they are evaluated. When an unknown dependency is not available during execution, the runtime will backtrack and add it as a new dependency. This is an expensive operation and for this reason the runtime waits as long as possible to evaluate them.

- Statements that use lists have to wait until the all other statements have been evaluated. Because the modeling language is declarative, attributes in the configuration model are immutable. This means that an attribute of an entity instance can only be set once. This definition is too strict for relations with a multiplicity higher than one. A list is initially created empty, and statements can append values to that list.

  The compiler evaluates statements that append first. Whenever a list is read, it is frozen to deny any new appends. In the algorithm (Figure 3.11) statements that use a list move from the list queue to the evaluation queue when all other statements have been evaluated. Statements in the list queue are grouped there per iteration. Only statements from the oldest iteration are placed on the evaluation queue when required to progress the evaluation. This ensures that statements have to wait as long as possible before execution. It is possible that a statement reads a list variable before all items have been added. This can happen with a configuration model that does not have horizontal layering and has a lot of very cross-cutting relations. Whenever this condition does occur, the operator will be notified with an error. In practice, we have never encountered a real configuration model that did not compile for this reason.

```
# the three queues the scheduler moves statements in
evaluation_queue = []
plugin_queue = []
list_queue = [] # a queue of queues per iteration

def evaluate_queue():
  # Process evaluation_queue until there is no longer progress
  ...

def queue_statements():
  # Give precedence to statements that refine the model instead
  # of completing it.
  new_list_queue = []
  for stmt in graph.statements():
    if stmt.is_evaluated() or stmt in evaluation_queue or stmt in plugin_queue or stmt in list_queue:
      pass

    elif stmt.is_resolved():
      graph.add_actions(stmt, stmt.actions())
      if graph.uses_list(stmt):
        new_list_queue.push(stmt)
      elif stmt.calls_plugin():
        plugin_queue.push(stmt)
      else:
        evaluation_queue.push(stmt)

  if len(new_list_queue) > 0:
    list_queue.push(new_list_queue)

i = 0
while i < MAX_ITERATIONS:
  queue_statements()

  if len(evaluation_queue) == 0 and len(plugin_queue) == 0 and len(list_queue) == 0:
    break

  i += 1

  result = False
  if len(evaluation_queue) > 0:
    result = evaluate_queue()

  # not everything was evaluated -> try all statements now
  if not result:
    # move statements from the plugin and list queue. move statements
    # that use a list when no other are available.
    length = len(evaluation_queue)

    if len(plugin_queue) > 0:
      evaluation_queue.extend(plugin_queue)
      plugin_queue = []

    elif len(list_queue) > 0:
      # pop the oldest queue from the list queue and append it to the evaluation queue
      evaluation_queue.extend(list_queue.pop())

    if len(evaluation_queue) > length:
      evaluate_queue()

exporter = Exporter()
exporter.run(model)
```

Figure 3.11: Evaluating all statements to build a complete configuration model. (Error-handling and detecting dependency loops is left out for brevity.)

3.4.2 Deploying incomplete configuration models

Transformation plug-ins give operators a mechanism to express complex parameter transformations within the IMP model. This mechanism can also serve to lookup parameters in external databases or systems. For example, retrieve the mac-address of a server from the configuration management database to configure its network interfaces. This is a parameter lookup for which a value will always be available (except for unexpected network or service failures). Virtualization and especially cloud computing complicates this mechanism because a parameter might only be available when a part of the configuration model is already deployed. For example, IaaS providers offer a fully *programmable* infrastructure. The configuration model can determine how many of which type of virtual machines are provisioned. This implies that configuration parameters such as MAC address or even IP addresses are only known when a virtual machine has been created. As a consequence the configuration model is incomplete until a part of it is actually deployed. This matches scenario 5 from the introduction of this chapter.

IMP allows operators to specify an incomplete configuration model with parameters for which its value is not known. The enforcement subsystem can deploy such an incomplete configuration model. It marks unavailable parameters as unknown. This is done whenever a transformation plug-in cannot determine the value. For example because the virtual machine has not been provisioned yet. The `vm::get_vm_ip` plug-in from the example in Figure 3.5 marks the IP attribute as unavailable when the virtual machine does not exist yet. The unknown tag is propagated to all parameters that are derived from it. This is also the case for plug-ins that use an unknown parameter. In the example this means that the content of the loadbalancer configuration file will be marked as unknown because it uses the ip parameter of the virtual machines.

Each export plug-in needs to determine how they handle unknown values. The export plug-in that sends the configuration model to the enforcement subsystem ignores any device for which a resource has an unknown parameter. This means that it prunes all managed resources of the device of which a parameter is unknown. As a consequence, when for example the MAC-address of the device to generate the networking configuration file is not yet known, none of the resources that the IMP agent on that device manages will be deployed. This mechanism also ensures that a *green field* deployment of a distributed system on an IaaS is correctly orchestrated to first create all virtual machines and only then try to configure them.

3.4.3 Integrating imperative code

Each transformation plug-in is developed in an imperative generic programming language. The platform calls the plug-in code during the refinement of the configuration model. This implies that it needs to know which instances and values from the configuration model it will reference during execution of the imperative code. Additionally IMP needs to track the use of unknown parameters. A transformation plug-in gets a list of arguments at invocation which is passed to the imperative code and it returns a value. IMP executes the plug-in whenever its arguments are available.

However, many plug-ins use their arguments to navigate through the configuration model or even access variables available in the scope where the plug-in was called (for example the template plug-in). All arguments and values passed to the imperative code are wrapped with a proxy to ensure that plug-in code *cannot* modify the configuration model and to detect queries to parts of the model that have not been refined yet. The proxy tracks if the accessed part of the configuration model is available, if not it will abort the execution of the plug-in code and add the accessed part of the configuration model as an implicit argument. Additionally the proxy also tracks unknown parameters. Whenever all arguments are available, IMP will try to execute the plug-in again.

3.5 Generating configuration artifacts

The refinement process generates a configuration model that contains the desired state of all resources IMP manages in the distributed system and its execution environment. In scenarios 1 to 4 this configuration model is a complete configuration model. For scenario 5, it is possible that the configuration model has not been fully refined yet. The export component of enforcement subsystem uses export plug-ins to transform this complete configuration model to artifacts that can be deployed: either by the enforcement subsystem itself or by 3rd party tools.

The latter is performed by export plug-ins (to interface with 3rd party tools). These plug-ins indicate the entity type of the resources the tool can manage. Additionally, export plug-in can be used to generate documentation such as network or deployment diagrams.

The former target is a desired state model that the IMP agents can deploy. Each configuration module can define resource plug-ins. A resource plug-in extracts attributes (e.g. the contents of a file) and deployment dependencies for managed resources from the refined configuration model. The result of the plug-ins is sent to the deployment agents. IMP executes all resource plug-ins on instances that represent managed resources. This creates a graph of the desired state where the nodes in the graph are the managed resources and the edges are the deployment dependencies between these resources. Such dependencies include: a parent directory needs to be deployed before a file is deployed and one service (e.g. of an application server) can only start after an other service is started (e.g. the database server). The desired state model is uploaded to the IMP server and this server will broadcast each resource to all deployment agent.

Each resource is uniquely identified by means of the type of the entity it was derived from, the name of the agent that manages that entity and an attribute and its value that uniquely define the instance. For example, the id of a file is `File[server1,path=/etc/hosts]`. In this example, server1 is the name of the deployment agent, which typically is the name of the devices it manages. Additionally each instance is tagged with an increasing version number to distinguish between two generation runs. The highest version number is always authoritative for the desired state. Per deployment agent, IMP verifies that no resources exist that has unknown configuration parameters. If a parameter is unknown, it will prune all resources from its deployment agent.

3.6 Deploying configuration changes

The IMP agent and server enforce the desired state in the configuration model onto the managed distributed system. The resource plug-ins extract the configuration parameters the deployment agent requires to enforce the desired state. IMP submits this desired state to the IMP server. The server stores the state of each instance and broadcasts to the deployment agents. Whenever a deployment agent is not available during the broadcast, it can later retrieve the latest version from the server. In essence, the main function of the server is connecting the generation subsystem with the deployment agents and adding persistence so the latest version of the configuration model is always available or queryable.

Module developers can define resource handlers, which the framework deploys to the IMP agents. A resource handler is a plug-in that can determine the current state of a managed resource and execute the required actions to bring the current state to the desired state in the configuration model. The export component not only uploads the desired state to the IMP server and agents, but also all available resource handlers. The agent selects a resource handler, from the available handlers, to deploy a resource. The selection is performed based on the criteria the handlers defines itself. For example, resource handlers to manage software packages vary based on the type of package management system used: rpm/yum for RedHat derivatives, deb/apt for Debian derivatives or MSI for Windows based systems.

The deployment agent is the component of the enforcement subsystem that actually enforces configuration changes. The agent manages a server or systems such as a switch. It receives the desired state of resources it manages. The agent queues the desired state together with all resource state updates and their deployment dependencies. When a resource has no unmet deployment dependencies it is passed to a resource handler. The handler checks what the current state of the resource is and what its desired state is. From this it derives the required actions to bring the resource into the desired state. When all handler actions are finished the agent broadcasts that the resource is up to date. This allows all agents to remove this resource from the deployment dependency list of other resources still in the queue. Whenever the deployment does not convergence because of a missing deployment dependency (which can be caused by an agent not running, a deployment error or a resource handler that is not available for the current operating system), the user is notified of this. This failed deployment can be solved by compiling an updated configuration model.

A second function of the server is collecting facts about the distributed system. Transformation plug-ins can query these facts on the server. Most facts are attributes of the distributed system that are only known at the time of the deployment, but are necessary to create a complete configuration model. Resource handlers extract these attributes from the resources they manage. For example, the MAC and IP address of a newly booted virtual machine.

3.7 Conclusion

This chapter introduced the IMP framework for integrated configuration management. It offers system administrators a tool to manage the configuration of a distributed system and its execution environment from an integrated configuration model. The configuration model can express a configuration at a high level of abstraction, and refine that model to the level of abstraction at which the configuration is enforced on real world infrastructures. The integrated model, the multiple levels of abstraction and the modeling of relations between configuration parameters, reduces configuration parameter duplication. Additionally, the transformation plug-ins and the plug-ins for deployment (export plug-ins, resource plug-ins and resource handlers) give system administrators a framework to port their ad-hoc scripts to.

Chapter 4

Evaluation

This chapter evaluates IMP and how it automates configuration tasks. The evaluation considers the following properties:

1. Usability of the framework with special attention to: (a) increasing the level of automation and thus reducing the time required to do the initial configuration and to make configuration changes and (b) reducing configuration errors by keeping interdependent configuration parameters up to date.
2. Maintainability of the configuration model.
3. Scalability of the configuration model, maximizing the size of the infrastructure that can be managed by using the framework within reasonable computation time.

Ideally, an empirical evaluation would be quantify the achievements in a systematic way. Unfortunately is such an evaluation in the context of this dissertation not feasible in terms of effort and cost. Skilled system administrators are a scarce good on the job market [48]. This chapter evaluates IMP in three case studies, each highlighting a property of IMP. The evaluation of each case study gives measurements to indicate how IMP compares with manual configuration on the properties listed above.

First, this chapter discusses the IMP prototype that can manage a distributed system and its entire execution environment by means of an integrated configuration model. The prototype implementation follows with three case studies that validate and evaluate a contribution of IMP:

Case 1 manages the entire infrastructure to host a web application on a physical infrastructure in a datacenter. The infrastructure has two internal networks: one for the webserver and an other for all servers that store state (file- and database servers). This case study evaluates IMPs capabilities to manage heterogeneous equipment (namely Linux servers and Cisco network equipment) and heterogeneous configuration (namely networking and application and middleware), from a single integrated configuration model. Moreover, because of to this integrated configuration model IMP can configure a firewall on *each* device (routers, firwalls and servers) from a single high level security policy.

Case 2 automatically allocates and manages IPv4, IPv6 or dual stack networks on Linux servers and network equipment such as Cisco routers. IMP cannot perform automatic configuration parameter allocation based on constraints in the configuration model. However, transformation plug-ins can be used to allocate specific configuration parameters automatically. This case study uses high level IP configuration as the initial configuration model (the entire subnet, the physical network layout, border routers,...) and automatically allocates IP ranges for IPv4 and IPv6.

Case 3 manages a scalable and distributed application on an IaaS platform. An IaaS platform is a fully programmable virtualized infrastructure and does not require physical setup of servers and network. IMP bootstraps the entire execution environment of the application on an IaaS platform and scales the application automatically.

This chapter concludes with an evaluation of IMP according to the framework in chapter 2, and performs a critical analysis of how IMP improves upon the areas of improvement listed in chapter 2.

4.1 Prototype implementation

This section describes the implementation of an IMP prototype based on the design outlined in chapter 3. It shows the feasibility of the framework. The prototype consists of the generation and the deployment subsystem and configuration modules that define types and refinements. This section provide details about important implementation and design decisions during the implementation.

4.1.1 Configuration model and transformation plug-ins

The IMP runtime is implemented in Python [108] and the parser for the DSL is generated with ANTLR [89]. Transformation plug-ins define configuration parameter transformations in an imperative programming language. In the prototype this imperative language is also Python. During the evaluation of the DSL input the runtime builds a configuration model representation in Python. All types in the configuration model get a representation in the Python type-system through meta-programming. When the DSL instantiates a new entity, the runtime creates a Python object of the corresponding meta-programmed Python type.

The choice to mirror the entire IMP typing in the Python type-system is a deliberate choice. It enables IMP to create an intuitive and easy to use transformation plug-in API. Operations by plug-in code on this representation translates directly to operations on the configuration model with the correct IMP semantics. These operations are read-only because the configuration model is declarative and values can only be assigned once. The Python API provides system operators with an as low as possible learning curve to convert their current ad-hoc scripts to scripts that work inside the IMP framework.

IMP is implemented in Python 3 to allow developers to annotate arguments and the return value of a plug-in function with types. This allows the IMP prototype to maintain its strong typing. Figure 4.1 shows the definition of a simple plug-in that accepts a string, reverses it and returns it.

```
from Imp.plugins.base import plugin

@plugin
def reverse(param : ”string”) —> ”string”:
    # Reverse the string in param and return it
    return param.reverse
```

Figure 4.1: A transformation plug-in that takes a parameter of the type string and return the string reversed.

A common parameter transformation is generating a configuration file from a template and parameters from the configuration model. The template transformation plug-in provides this functionality using the Jinja2 template engine [93]. Templates can access all parameters available in the scope where the template transformation function is used. Additionally, all transformation plug-ins available in IMP modules are also callable from the template engine. The template plug-in adds a template directory to a configuration module. The path argument passed to the template plug-in is a relative path: the first part is the name of the configuration module, the remainder is de relative path inside the template subdirectory of the configuration module. The file and source transformation plug-ins are similar, they directly copy a file from the files subdirectory in the configuration module. The first copies the file without pulling the file content in the configuration model, the second reads in the file content and includes it as a configuration parameter in the model.

4.1.2 Managing Linux with IMP: the std module

The design of IMP outlined in chapter 3 outlines a platform that is agnostic of the systems it manages. The prototype includes a module with basic types and handlers to enforce their desired state on Fedora, CentOS and Ubuntu. This module is named std and contains types that represent:

std::Host A representation of a managed device or server

std::File A file on the file system, a file has a path, content and is deployed on a host. Additionally it has attributes related to access control.

std::Directory Similar to a file, but does not have content

std::Service A system service, which on Linux is often a daemon process that init starts at boot time.

std::Package A software package that a package manager installs. (e.g. apt, yum,...)

The std module does not refine the entities it defines. It does provide resource plug-ins to extract the required configuration parameters from the configuration model, and resource handlers so the IMP agent can enforce the desired state on Linux systems (Fedora, CentOS and Ubuntu). Each handler for a resource needs to implement two methods to:

1. Determine the current state of the managed resource. IMP derives a list of configuration parameters that do not match the desired state in the configuration model
2. Derive and execute the actions required to bring the current state to the desired state.

A configuration module is implemented as a directory that contains two subdirectories. The first directory is `model` and contains files with the input specification in the IMP DSL. The namespace for the types defined in this module is derived from the name of the module. All types in this namespace should be defined in _init.cf. Files which are names can also define types, but will exist in a subnamespace. For example, httpd::Server is defined in the module httpd and the entity Server is defined in the file _init.cf. Module developers can add plug-ins (transformation, export, resource, handlers, helper functions, etc.) to a module in the `plugins` directory. This directory is loaded as a Python package (it should have a __init__.py file) and is also registered in the Python runtime as part of a Python module with the name of the IMP module. A tutorial in Appendix B provides a hands on guide to module development.

4.1.3 Modes of operation

The IMP prototype has three modes of operation that each allow for a more integrated adoption of IMP, each in the line of the scenarios defined in chapter 3:

single IMP refines the entire configuration model and then starts an embedded deployment agent in offline mode (without a server) to deploy the desired state model to either the local machine or on a remote machine using remote IO over an SSH connection. This mode requires no manual deployment or configuration of the IMP server and agent. It can also serve to manage a small number of machines such as in scenarios 1, 2. This mode does not allow for the IMP agent to autonomously converge to the desired state. If a deployment dependency is not modelled and it is violated, the system administrators needs to rerun the entire process again.

enforce Refine the entire configuration and submit the model to the management server. The management server broadcasts this to all management agents. The desired state is persisted at the server. Deployment agents will iteratively enforce the desired state, effectively allowing the convergence process to occur. This mode aligns with scenarios 3, 4 and 5. The single mode can be used to deploy the management server and agent and then move to this mode.

bootstrap In this mode IMP manages all (virtual) machines on one or more IaaS platforms. IMP first starts and configures an IMP management server, next it boots and configures all servers that are defined in bootstrap mode. After the bootstrap finishes, a new invocation in the *enforce mode* can deploy the final configuration. This mode converges in multiple iterations to a configuration model without unknown configuration parameters. This mode primarily exists to support scenario 5.

4.1.4 Communication

The compilation subsystem submits its results to the IMP server over a RESTful HTTP protocol. The IMP server and all IMP deployment agents communicate over an AMPQ bus using pub/sub. The server broadcasts the new desired state of managed resources to all deployment agents over the AMQP bus. Additionally agents broadcast to all other agents when a resource is updated so each agent can track dependencies between managed resources. For large resource parameters, such as the content of a configuration file, agents communicate directly with the IMP server over HTTP to limit the amount of data on the AMQP bus.

4.2 Case 1: integrated network management

A typical enterprise webapplication is deployed as a three tier application. The browser of the client is the first tier and presents the application to the user. The second tier hosts the application logic which is executed in an application server. The third tier stores the state of the application and typically contains database and file servers. Such applications often require a high available infrastructure to minimize downtime due to individual server failure or network device failure. Additionally, to minimize the impact of security vulnerabilities multiple firewalls are required between each machine (second or third tier). This infrastructure is complex to manage because it is a heterogenous infrastructure (network and server equipment) with many configuration parameters duplicated due to high-available services and a single security policy translated in firewall rules on multiple firewall. This results in many duplicated configuration parameters that need to be kept consistent.

This case study develops a configuration model to manage an infrastructure that hosts such a multi-tier application. This infrastructure consists of physical servers, switches, routers and firewalls. This application requires a high-available infrastructure and at least two firewalls between each subnet of the infrastructure. This leads to the following requirements for the configuration model:

1. The entire network has to be redundant: each server has two network interfaces that connect to diffirent switches of which one link is used to failover when the first link fails. All network equipment that provides routing or firewalling has to be high available. Additionally, all services on the
2. Each machine (servers, routers, firewalls) need to have a firewall configured that by default drops all incoming and outgoing packets and does not forward packets from devices it does not know.
3. All devices need to be managed from a single integrated configuration model, both networking equipment and servers.

Both requirements lead to a complex configuration that is very prone to errors (especially the security requirement because of the numerous firewall rules). All related configuration should be encapsulated in configuration modules and they should offer the types to configure the following services: a network with vlans and IP subnets, DHCP, DNS, monitoring, a loadbalanced webserver cluster, routing and firewalls in a redundant and fault tolerant setup. The refinements are implemented for UNIX based systems (CentOS Linux in particular) and Cisco switches and routers. The configuration model either generates Cisco configuration files or a desired state model for the IMP deployment agents that installs all packages and configuration files and starts the required services.

The next section ellaborates on the design of the configuration modules, followed by a description of the infrastructure on which the configuration modules have been evaluated and the results of the evaluation. This case studies concludes with an overview of specific related work of this case study and a conclusion.

4.2.1 Configuration modules

The configuration model for this case studies is split up in multiple configuration modules. Each module is responsible for the configuration of a subsystem in the infrastructure:

network module models the physical links between networked devices and switches, and the properties of these links such as the mac-address, active vlans, active/standby links or software bridges. This module also contains two additional export plugins that generate a layer-1 diagram of all physical network connections and a diagram of all Virtual LAN's (vlans).

IP module is conceived in such a way that each separate IP subnet is mapped to a distinct vlan network. This makes the network more manageable and requires no additional maintenance effort because the Ethernet network module configures all network devices automatically. This partial domain model defines IP subnets, assigns IP addresses to interfaces and provides hot standy ip addresses to implement fail-over services. It also generates forward and reverse DNS records for the DNS module. Client/server connections are also modelled and can be used to validate them against the firewall policy.

DNS module models DNS servers that are either master or slave for a DNS zone. Other modules use these types so DNS resource records are automatically added to the zone they belong to.

DHCP module configures a redundant DHCP server setup with static and dynamic address leases. The static leases are generated from the MAC-addresses defined in the types of the ethernet module and the IP-addresses assigned to the designated ethernet interfaces.

firewall modules defines a policy that expresses what IP traffic is allowed, all other traffic is rejected by default. The firewall module provides the types to model a firewall policy. It will check each client/server connections in the model against that policy. If a connection is not allowed the compiler issues an error.

The firewall refinements can use the firewall policy and the configuration model with the Ethernet and IP configuration to generate firewall rules for each device in the network. A server only receives the firewall rules that concern connections to and from that server. Gateways receives all firewall rules that concern packets that can pass through it. This model enables firewalls on all machines without any management overhead, thus tightening security because it applies defense in depth.

Figure 4.2 shows the types of the firewall module. It is inspired by the policy language of Firmato [15]. In figure 4.3 a policy for the network of this case is shown. An operators defines roles and policy rules between roles for certain services (in the network sense: 53/udp, 80/tcp, ...). An operator can assign these roles to network interfaces or servers, as shown in Figure 4.4 on line 24.

monitoring module provides types to monitor devices and the hosts they run on. For a example the DNS module creates an IP-server instance and connects a DNS-monitor client to it. Monitor servers can use that information in the model to generate monitoring configuration for the server. It will also generate IP-clients that connect to the services to ensure a correct firewall configuration that allows the monitor server to access the monitored server.

load-balancer and webserver modules model a cluster of webservers, redundant loadbalancers and assign virtual web hosts to a cluster. Each of the webservers and loadbalancers are configured to serve that virtual web host. This module also uses the firewall, DNS and monitor module to generate the required supporting configuration: hostnames, firewall rules and service monitoring.

The configuration modules provide refinements for their types for Cisco (network equipment) and CentOS [94] linux as operating system on all servers. Both have a different approach. The Cisco implementation supports switches and routers. The functionality of these devices is known and is configured with a single configuration file that contains all configuration parameters. On a general purpose operating system services and components can be added. For this evaluation only one implementation per service is provided for CentOS. For example, the DNS server is only refined to the BIND on CentOS, but other refinements such as PowerDNS on Solaris could be added.

More specific for this case study the following refinements for types of the configuration modules have been developed:

1. Cisco Catalyst 2950, 2960 and 3500XL network switches and Cisco 7200VXR routers.
2. the network configuration on CentOS
3. redundant DHCP servers using ISC DHCP [64] and BIND [63] based master and slave DNS servers that are automatically used by all hosts in the network.
4. linux iptables [83] based firewalls on each device and failover IP-addresses for a redundant firewall setup

```
typedef direction as string matching self == ”one”
        or self == ”both”

interface Role:
    string name
end

Role roles [0:] -- [0:] ip::Network networks
Role roles [0:] -- [0:] net::DefaultVlanInterface interfaces
Role roles [0:] -- [0:] std::Host hosts
Role roles [0:] -- [0:] ip::Ip ips
Role roles [0:] -- [0:] ip::services::VirtualSide virtual

interface Policy:
    string      name
    direction   direction
end

typedef DPolicy as Policy(direction = ”one”)

Policy policies             [0:] -- [1:] ip::Service services
Policy source_policies      [0:] -- [1:] Role source
Policy destination_policies [0:] -- [1:] Role destination
```

Figure 4.2: The types of the firewall module to define the security policy for the entire network.

5. monitoring system based on NAGIOS [45]
6. a webcluster using Linux Virtual Server [95] for routing based redundant loadbalancers and the Apache webserver [10].

This case study was performed before the IMP prototype had a deployment subsystem. The configuration model was deployed by generating configuration manifests for the Puppet configuration management tool.

4.2.2 Evaluation infrastructure

The infrastructure for this case is designed for hosting a multi-tier webapplication that can scale. All machines are physical and located in a datacenter with an uplink and power. Each device is connected to a least two of the three switches and VLANs separate network subnets. The network setup is shown in figure 4.5. A redundant firewall pair connects to the internet (uplink) and provides stateful packet filtering. They advertise their routes to the routers of the uplink provider for connectivity. These firewalls connect to a redundant router pair that handles routing between the internal, web and net subnet. Each network device connects with their management interface to a management network *net*. The webservers and web loadbalancers that execute the code and handle requests from the internet are isolated in a separate *web*

```
# these roles are linked to networks
r_intern_net       = fw::Role(name = ”the␣internal␣network”)
r_net_net          = fw::Role(name = ”the␣net␣network”)
r_fw_net           = fw::Role(name = ”the␣firewall␣network”)
r_web_net          = fw::Role(name = ”webserver␣network”)

# these roles are linked to services
r_internal_dns     = fw::Role(name = ”internal␣dns␣server”)
r_dns_client       = fw::Role(name = ”client␣role␣for␣dns”)
r_web_server       = fw::Role(name = ”web␣server”)
r_web_client       = fw::Role(name = ”web␣client”)
r_web_lb           = fw::Role(name = ”web␣load␣balancer”)

# allow access to the internal dns
p_internal_dns = fw::DPolicy(destination = r_internal_dns,
    name = ”networks␣to␣the␣internal␣dns␣servers”,
    services = ip::services::dns)
p_internal_dns.source = [r_intern_net, r_fw_net, r_mgmt_net,
    r_web_net, r_dns_client]

# allow internal dns slaves to contact the dns master for zone transfers
p_intern_zone_transfer = fw::DPolicy(source = r_internal_dns,
    destination = r_internal_dns, services = ip::services::dns_tcp,
    name = ”zone␣transfers␣between␣internal␣dns␣servers”)

# allow the lb to the webservers
p_web = fw::DPolicy(source = r_web_lb, destination = r_web_server,
    services = ip::services::http_all, name = ”access␣web␣servers”)

# allow clients to access the lb
p_lb = fw::DPolicy(source = r_web_client, destination = r_web_lb,
    services = ip::services::http_all, name = ”access␣web␣lb”)
```

Figure 4.3: A network security policy expressed with the firewall configuration module and enforced on Cisco devices with extended access lists and iptables on Linux based devices.

network. The internal network locates the servers that contain state, such as DHCP, DNS, monitoring, management servers, database and fileservers. These devices are not accessible from the internet and are separated by a router pair from the webservers. Each server also has a packet filter configured. Firewall rules are calculated and deployed on each each server, the routers and the firewalls. This ensures that each device has at least one firewall between every other device and three firewalls between each server and the internet.

4.2.3 Evaluation results

This case evaluates the usability of the framework and the maintainability of the configuration model. Usability is interpreted in this context, as the potential to achieve correct configurations with less effort. We have observed the case of an initial configuration, followed by the process of adding four web servers. We have compared

```
# eth0 and eth1 configuration
switch_port_eth0 = net::FastPort(port_num = 1, comment = hostname,
    switch = site::network::switch_1)
eth0 = net::NetworkCard(name = ”eth0”, mac = ”00:60:08:73:31:9d”,
    peer = switch_port_eth0)
switch_port_eth1 = net::FastPort(port_num = 1, comment = hostname,
    switch = site::network::switch_2)
eth1 = net::NetworkCard(name = ”eth1”, mac = ”00:b0:d0:3a:91:0d”,
    peer = switch_port_eth1)

bond0 = net::bond::BondedInterface(name = ”bond0”, mode = ”active—backup”)
bond0.slaves = [eth0, eth1]

# vlan interfaces
bond0_web = net::vlan::VlanInterface(raw_interface = bond0,
    vlan = site::network::web_vlan, default = true)

# ip configuration
bond0_web_ip = ip::Address(ipaddress = ”172.16.1.11”,
    netmask = ”255.255.255.0”, iface = bond0_web)

# include the configuration to configure this server as a webserver
include site::web::node
self.roles = site::firewall::r_web_server
```

Figure 4.4: An excerpt of the network configuration of a server with two interfaces. These two interfaces are bonded for L2 redundancy and vlan is defined to place the machine in the *web* network.

manual configuration with an IMP based scenario. Maintainability is interpreted in this contexts, as the potential to achieve (manage) large changes in the infrastructure by performing limited modifications of the configuration model. The improvement of using IMP over manual configuration is evaluated based on metrics of the number of lines of code. A comparison of the lines of code required to manage the infrastructure with IMP and the number of lines needed to manually configure the infrastructure provides an indication of how much configuration files need to be written to manually manage an infrastructure. The results are provided in table 4.1. The first column are the numbers for the initial configuration. The second column are the numbers for a change where four machines that already had a basic configuration where added to the webserver cluster. The generated configuration is representative for the configuration required to manage an infrastructure manually.

IMP requires less lines to be written and with the reusable IMP model, only half the lines are required. This is for the initial setup, the benefits are even more apparent for the number of lines that need to change in the IMP configuration model to add four additional webservers to the cluster and the amount of real configuration required to deploy this. These numbers also do not take in account that to maintain manually an infrastructure where 26 *different* firewalls need to be configured and kept consistent is very labour intensive and error prone.

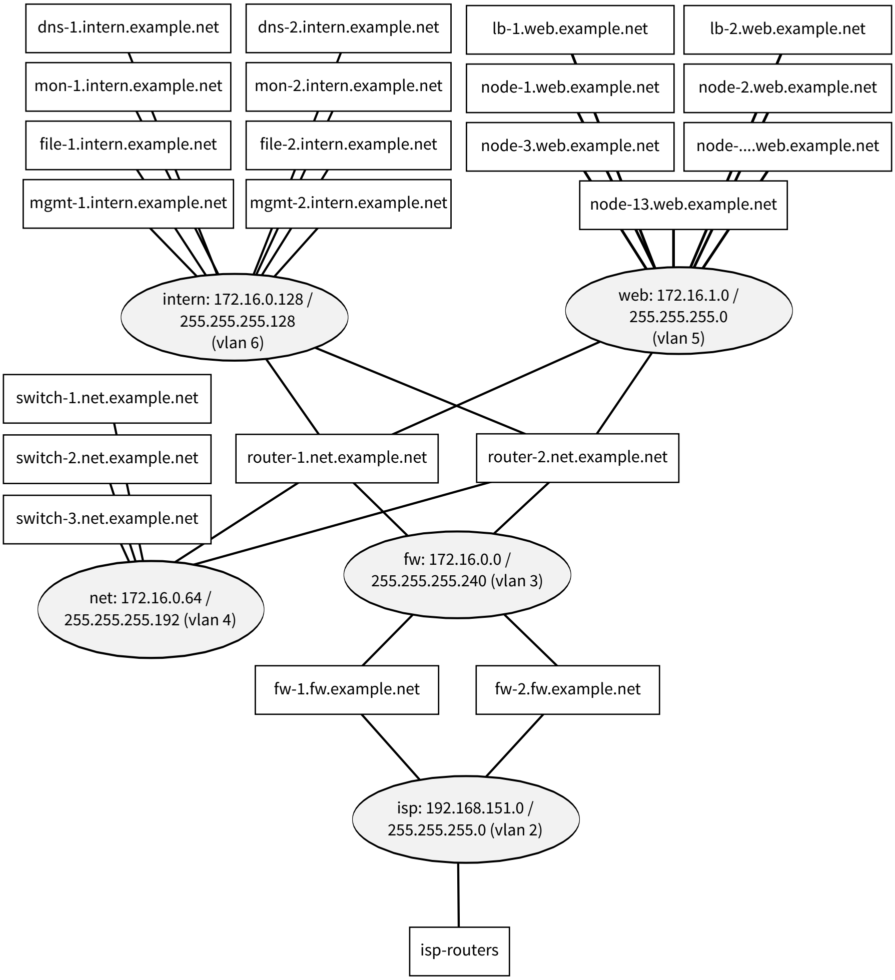


Figure 4.5: A diagram of the network setup used in case study 1

4.2.4 Related work

This section discusses related work specific to integrating network and server management and managing firewall policies. PRESTO [44] is a system configuration tool that focuses on greenfield configuration of very large scale ISP infrastructures. They never model the entire infrastructure but manage each customer separately. Relations over abstractions and over the entire ISP infrastructure cannot be modeled thus creating the risk of inconsistent parameters.

Scenario	initial	4 extra webservers
Written by user		
Site specific IMP model	666	670 (+4)
IMP		
model	1034	1034
templates	1425	1425
python code for plugins	842	842
Generated		
Unix config files	5291	5801 (+510)
Cisco config files	894	894
Puppet manifests	4170	4271 (+101)
total lines	10355	10966 (+611)

Table 4.1: Lines of code in each scenario. For the IMP model these are the number of lines in the input specification that defines the configuration model. For the configuration files this is the number of lines in each configuration file.

Firmato [15] models the firewall policy of a network and generates firewall configurations for routers in a network. It also provides a tool to visualize firewall rules. FIREMAN [118] also models a security policy and verifies single or distributed firewalls against it. This approach detects anomalies and misconfigurations in the rules of each individual firewall. They both are limited to firewalls and do not integrate with the configuration of the network or the entire infrastructure.

4.2.5 Conclusion

This case shows that IMP can manage heterogeneous devices, namely Linux servers and Cisco switches and routers, from a single integrated configuration model. Management tools for servers and networks are often different and in many companies even different departments (NOC vs Operations). This integrated configuration model removes the need to duplicate configuration parameters from one management tool to an other.

In this case a limited configuration of services on top of the network layer is automated: the configuration model includes all services, how they connect to each other and the layout and configuration of the network. This allows IMP to use a transformation plug-in that enumerates all network connections in the model and use this to enforce network security in two ways:

1. Only services and network connections that the security policy allows can be instantiated and refined in the configuration model. All other services result in compilation errors.
2. IMP enumerates on each host (server) and each router or firewall all connections that originate, terminate or pass through this node and add the appropriate packet filtering rules.

This results in a configuration where the configuration and deployment effort is equal for one firewall, or a firewall on each device.

4.3 Case 2: automated configuration parameter allocation

This case focusses on configuring and managing a dual stack network infrastructure. In the high level configuration model abstractions of the network components and relations between them are specified. Dependencies between low level parameters are resolved automatically, and a uniform interface is offered that encapsulates heterogeneity. The proposed model provides a uniform interface to the configuration of the network infrastructure, which increases the efficiency of configuration and management.

This evaluation validates two aspects of IMP by developing a dual stack IP allocation configuration module:

1. The first shows that a model-based approach to configuring a dual stack network infrastructure with IMP is feasable. Moreover it reduces the complexity associated with managing a dual stacked network.
2. The second aspect relates to automation. IMP does not provide generic automated constraint satisfaction like for example PoDIM [31, 33] (a predecessor of IMP) does. For example, in PoDIM operators could specify that two available server should be configured as a mailserver. This solution does not scale very well and has stability issues because a configuration should be stable and for example not move mailservers around because a slightly more optimal solution exists. In IMP a system administrator explicitly needs to define which machines are configured as mailserver. However, with the transformation plug-in mechanism IMP can be extended to provide similar but domain specific constraint satisfiction. This evaluation validates this by adding fully automated dual stack IP network configuration with a prefix allocation algorithm. This algorithm uses a single prefix specified in the high level model and splits this to allocate subnets in an optimal way.

4.3.1 Design of a dual stack IP allocation configuration module

A configuration module to configure dual and allocate dual stack networks has the following requirements:

- **Efficiency**: The main aim of the module is to improve the efficiency of configuration by decreasing complexity. Therefore configuration can happen with less effort and the likelihood of configuration errors is lower. It achieves this by using a high level configuration model to specify the configuration.
- **Support for dual stack IP networks**: The IPv4 address space is depleting, despite measures taken in the 90's like CIDR [52] and NAT [109]. The demand for IPv4 address space has increased the last few years, with the advent of internet enabled mobile devices like smartphones and many new internet users in developing countries. The only worldwide accepted solution for this problem is the migration to IPv6. A first step in this migration is running networks in a dual stack configuration. Special attention is paid to make sure dual stack networks are supported by the configuration module as well.
- **Support for supporting network services**: The aim of the module is to set-up basic network functionality in a network infrastructure, facilitating deployment of more advanced services like mail servers or remote file systems. A minimal set of services is selected which are necessary for a functional network:
    - Host configuration protocol: The configuration of DHCPv4 [37], DHCPv6 [38] or SLAAC [110] in all the subnets of the network.
    - DNS Server: The configuration of DNS servers in a master-slave configuration, including the generation of forward and reverse zones containing all hosts.
    - Routing: The configuration of both static and dynamic routing. The dynamic routing protocols supported are OSPFv2 [79] and RIP [73] for IPv4, and OSPFv3 [26] and RIPng [74] for IPv6.
    - Generic servers: A hosts in a subnet can be designated as a server, which will be automatically configured with a static IP and are included in the DNS records. This basic configuration can be used by other services in the infrastructure.

4.3.1.1 Types and refinements

The advantage of specifying the configuration in a high level configuration model is that the complexity of the configuration problem is reduced, consequently reducing the effort and time required to enact configurations. Dependencies between low level parameters can be derived from the model and are automatically kept consistent. Heterogeneity is encapsulated in high level abstractions, and the configuration model uses a uniform platform independent syntax. Because the configuration is centralized in the configuration model and deployment is automated, the reconfiguration of the network infrastructure can happen efficiently. The system administrators can reason on a higher level without being preoccupied by low level configuration parameters. An example of a configuration model that uses the meta-model of the dual stack IP module is shown in Figure 4.6.

```
# create a network and set its global configuration parameters
net1 = ip::Network(ipversion = ”ds”, border = ro1, routingv4 = ”rip”,
    ↪ routingv6 = ”ospf”):

ip::ipv6::Range(net = net1, range = ”2001:06a8:2900:3828::”, mask = 61)
ip::ipv6::DNSConf(net = net1, conf = ”dhcpv6”)
ip::ipv6::HostConf(net = net1, conf = ”radv”)
ip::ipv6::DNS(net = net1, addr = ”2001:06a8:2900:3820::1”)
ip::ipv4::Range(net = net1, range=”172.16.0.0”, mask=16)
ip::ipv4::DNS(net = net1, addr=”192.168.150.254”)
ip::DNSDomain(net = net1, domain=”netwglab.cs.kuleuven.be”)

# define each of the IP subnets in the network
subnet1 = ip::Subnet(net = net1, name = ”subnet1”, size = 200)
subnet2 = ip::Subnet(net = net1, name = ”subnet2”, size = 200)
subnet3 = ip::Subnet(net = net1, name = ”subnet3”, size = 200)

# define and configure router−1 (Cisco)
ro1 = ip::Router(name = ”router−1”, net = net1, os = ”ios”):

ip::RouterInterface(host = ro1, mac = ”00:b0:8e:0c:84:1c”, subnet = subnet1,
    ↪ name = ”FastEthernet1/0”)
ip::RouterInterface(host = ro1, mac = ”00:b0:8e:0c:84:1d”, subnet = subnet2,
    ↪ name = ”FastEthernet1/1”)

# define and configure router−2 (Linux machine) and make it the primary DNS
    ↪ server of the network
ro2 = ip::Router(name = ”router−2”, net = net1, os = ”ubuntu−11.10”)

ip::RouterInterface(host = ro2, mac = ”00:0a:5e:3c:6b:e6”, subnet = subnet1,
    ↪ name = ”eth0”)
ip::RouterInterface(host = ro2, mac = ”00:c0:4f:01:9a:06”, subnet = subnet3,
    ↪ name = ”eth1”)
ip::DNSServer(host = ro2, dnstype = ”primary”)
```

Figure 4.6: An example of the level of abstraction in which a dual stack network can be configured with the IMP configuration model.

4.3.1.2 Prefix allocation algorithm

To further increase the efficiency and achieve a higher level of abstraction, the allocation of prefixes to subnets is automated. In the configuration model, a prefix is specified for the entire network. When refening the configuration model to low level configuration artifacts such asartifacts such as files, this prefix has to be divided over the subnets of the network. This has to be an optimal allocation, without wasting address space and possibly allowing aggregated routing. The subnet prefixes are allocated based on the network topology and the input network prefix. The network topology consists of a number of interconnected routers and subnets, with one designated border router connecting the network to the Internet. Prefix allocation happens differently for IPv4 and IPv6. The prefix allocation algorithms proposed are inspired on an algorithm to allocate an IPv6 prefix discussed in [16]. The algorithm proposed in [16] is not fit for the prefix allocation required, mainly due to the strict need of a tree topology. Another problem is the assumption that routers should be connected via point-to-point links, and subnets with hosts are only allowed as leaves in the tree topology. Due to these inflexibilities, some of the basic ideas like the use of pools and metrics are reused to design new algorithms for both IPv4 and IPv6 prefix allocation.

IPv4 In the case of IPv4, address space is limited. The given network prefix has to be divided as efficient as possible over the subnets. Therefore, no special care is taken to allow aggregated routing aggregated along the topology. The IPv4 allocation algorithm is displayed in Figure 4.7. The allocation algorithm uses a global pool which contains all the prefixes of varying length available. All the subnets in the network are assigned a prefix derived from this pool. The needed prefix length of each subnet is determined by their size, which is specified in the input configuration model. This size should always be a power of 2. When the pool does not contain a prefix matching the required size, an attempt is made to split a prefix of a smaller length until the desired length is reached. The residual prefixes are stored in the pool for reuse. For example, when splitting a /8 to get a /11 prefix, a /9, /10 and a /11 is left over. These prefixes are put back into the pool. This ensures an efficient allocation which minimizes the waste of the address space.

Used variables and functions:

- `P`: Pool of all available prefixes of varying length.
- $\mathtt{M}_\mathtt{s}$: Desired subnet size for subnet `s`, rounded up to a power of 2.
- `prefix.split()`: Splits a prefix of length `l` in 2 prefixes of length `l-1`.
- `neededPrefixLength(size)`: `32 - ` $\mathtt{log}_2$`(size)`. Calculates the minimum prefix length needed by a subnet based on its size.

```
//Entry point
void allocateSubnets() {
    P.add(Network.getIPv4Prefix());
    for(s in Subnets) {
        length = neededPrefixLength(M_s);
        s.prefix = getPrefix(length);
    }
    return;
}

//Returns a prefix of the given length, splitting prefixes in the pool when necessary
Prefix getPrefix(length) {
    pref = getAvailablePrefix(length);
    while(pref.length < length) {
        (pref,pref2) = pref.split();
        P.add(pref2);
    }
    return pref
}

//Returns a prefix from the pool smaller or equal to the given length
Prefix getAvailablePrefix(length) {
    if(length < 0 || length > 32) {
        printError(”No␣available␣prefixes!”);
    } else if(!P.containsLength(length)) {
        return getAvailablePrefix(length − 1);
    } else {
        pref = P.getPrefixWithLength(length);
        P.remove(pref);
        return pref;
    }
}
```

Figure 4.7: Pseudocode prefix allocation IPv4 subnets

IPv6 Contrary to the IPv4 prefix allocation, the IPv6 allocation algorithm does not have to be as efficient in terms of address use. Additionally, the prefix length of an IPv6 subnet is always /64 per specification [58]. Therefore, the prefix length does not have to be calculated based on the size of the subnet. The proposed IPv6 allocation algorithm tries to allocate subnet prefixes to allow aggregated routing. The algorithm operates in 3 steps:

- **Step 1 - Dijkstra** In the first step the network graph is reduced to a tree with the border router as root. Because deeper trees will cause a less ideal prefix allocation, a shortest path tree is preferred. Dijkstra's algorithm is used to generate a shortest path tree from the network graph, minimizing the distance between each router and the border router. The output of this step will consist of the connected child routers of each router per interface.
- **Step 2 - Metric propagation** In the second step each router and each interface is assigned a metric denoting the amount of /64 subnets that are below the

```
//Entry point
void propagateMetric() {
    anounceMetric(border);
}

//Recursive function, calculates the metric for the given router
int anounceMetric(Router r) {
    M_r = 0;
    for(i in r.Interfaces) {
        M_i = 0;
        for childRouter in C_i {
            M_i = M_i + announceMetric(childRouter);
        }
        subnet = i.connectedSubnet;
        if(!subnet.hasGateway() && (!C_i.empty() ||
            ↪ !subnet.isTransitSubnet()) {
            M_i = M_i + 1;
            G.add(i);
            subnet.addGateway(i);
        }
        M_i = metricCeil(M_i);
        M_r = M_r + M_i;
    }
    M_r = metricCeil(M_r);
    return M_r;
}
```

Figure 4.8: Pseudocode metric propagation IPv6

router or interface in the tree. The metric propagation algorithm is displayed in Figure 4.8. The algorithm is written recursively and the network tree is traversed depth-first. The output of this step are metrics of each interface and router and also a set of of router interfaces that are the default gateway interface on their connected subnet.

Used variables and functions:

- M_r: Metric of router `r`.
- M_i: Metric of interface `i`.
- C_i: Output of the Dijkstra step. For each interface `i`, this is a set of connected child routers in the reduced tree.
- `G`: This set contains all the router interfaces that are the default gateway interface on their subnet.
- `metricCeil(int metr)`: This function rounds its input up to a power of 2.
- `subnet.isTransitSubnet()`: This function checks whether a subnet is a transit subnet of a router. A subnet is a transit subnet if any router in the topology has child routers in the reduced tree via this subnet.

- **Step 3 - Prefix allocation** In the final step the prefixes of the reduced tree are allocated top to bottom to the subnets top.The algorithm is shown in Figure 4.9. The same pool system is used as in the IPv4 allocation algorithm, except that there is a pool for each router and interface. Each pool contains the prefixes that can be allocated to the subnets below the interface or router it belongs to. Initially, the network prefix is added to the border router pool. Subsequently, in the *assignPrefixRec()* function, the router interfaces of the border router are allocated a prefix from this pool based on their metric. The allocated prefixes are added to the corresponding interface prefix pools. From these pools, the connected subnets are allocated a /64 prefix when the connecting interface is a gateway interface. Also the child routers connected through these interfaces are allocated prefixes and the *assignPrefixRec()* function is called recursively for each child router. This continues until the leaf subnets are reached and all the subnets have a /64 prefix assigned.

  Used variables and functions:

  - C_i: Output from step 1.
  - M_r: Output from step 2.
  - M_i: Output from step 2.
  - `G`: Output from step 2.
  - P_r: Prefix pool of router `r`. Contains all prefixes available for this router.
  - P_i: Prefix pool of interface `i`. Contains all prefixes available for this interface.
  - `neededPrefixLength(metric): 64 -` $\log_2$`(metric).` Calculates the minimum prefix length required according to the given metric.
  - `prefix.split()`: Splits a prefix of length `l` in 2 prefixes of length `l-1`.

4.3.2 Evaluation of the dual stack IMP module

The primary requirement is to increase the efficiency of the configuration. In this evaluation an attempt is made to quantify this increase of efficiency. This increase in efficiency again provides an indication of the usability of the framework and the maintainability of the configuration model. The tool is evaluated by comparing an automated configuration with a manual configuration. The configuration is carried out for an existing network topology currently in use at the Department of Computer Science of the KU Leuven shown in Figure 4.10.

A good measure for the increased efficiency is the comparison of the amount of effort the network administrators have to perform in both cases. This effort translates to the

```
//Entry point
void assignPrefix() {
    P_border.add(Network.getIPv6Prefix());
    assignPrefixRec(border);
}

//For a given router, assign prefixes to connected subnets and child routers
    ↪ and recurse over children
void assignPrefixRec(Router r) {
    for(i in r.Interfaces) {
        length = neededPrefixLength(M_i);
        prefix = getPrefix(P_r,length);
        P_i.add(prefix);
        if(G.contains(i)) {
            i.connectedSubnet.prefix = getPrefix(P_i,64);
        }
        for(c in C_i) {
            length = neededPrefixLength(M_c);
            prefix = getPrefix(P_i,length);
            P_c.add(prefix);
            assignPrefixRec(c);
        }
    }
}

//Returns a prefix from the given pool of the given length, splitting
    ↪ prefixes when necessary
Prefix getPrefix(P_x,length) {
    pref = getAvailablePrefix(P_x,length);
    while(pref.length < length) {
        (pref,pref2) = pref.split();
        P_x.add(pref2);
    }
    return pref
}

//Returns a prefix from the given pool smaller or equal to the given length
Prefix getAvailablePrefix(P_x,length) {
    if(length < 0 || length > 64) {
        printError(”No␣available␣prefixes!”);
    } else if(!P_x.containsLength(length)) {
        return getAvailablePrefix(length - 1);
    } else {
        pref = P_x.getPrefixWithLength(length);
        P_x.remove(pref);
        return pref;
    }
}
```

Figure 4.9: Pseudocode prefix allocation IPv6

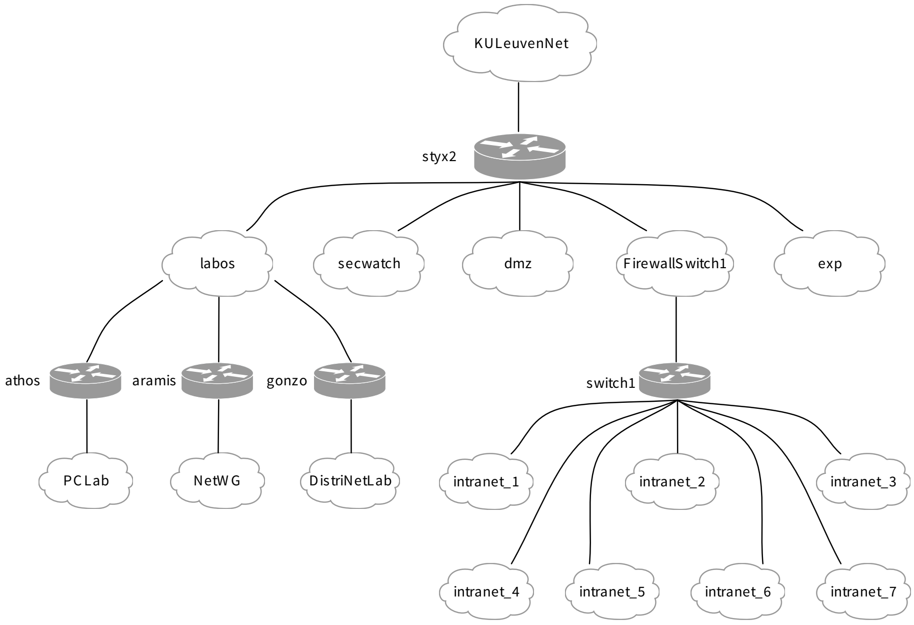


Figure 4.10: The network topology of the Department of Computer Science KU Leuven used in the evaluation.

amount of lines of configuration that have to be written. The amount of configuration lines contained in the input configuration model is compared to the number of lines in the various output configuration files.

The lines of codes are categorized based on their complexity. The complexity of a change is derived from the required effort to make a change or how error prone the change is, e.g. very cross-cutting configuration parameters. This categorisation provides a more nuanced comparison of the manual and IMP automated effort, compared to case 1. An overview of the categories:

- **Addresses**: All lines with an IP address or a netmask. These parameters are very error prone to configure manually. Often dependencies exist between addresses, increasing the complexity.
- **Regular parameters**: This is a residual group containing all the lines that still contribute to the configuration but do not fall under the **Addresses** category. Frequently these lines are switches activating or deactivating a function.
- **Configuration metadata**: This category contains all the lines which do not contribute any useful configuration, but still are required to understand

the structure of the configuration file. This category consists of brackets (`{`,`}`,`(`,`)`,`[`,`]`) and keywords.

- **Comments and whitespace**: These are the lines in a configuration file that are completely meaningless configuration wise. The effect of the configuration would be identical without these lines.

The result of the evaluation is shown in Table 4.2. The input configuration model is compared to the output configuration files. When using the configuration model, network administrators have to produce approximately 6.7 times less lines of configuration. This ratio further increases when only the most complex category **Addresses** is compared. In this category, the ratio is roughly 12.8. This shows the positive effects of using a high level configuration model. The amount of complex parameters that have to be produced by the administrators decreases. This is due to the abstractions of the various network elements and their mutual relations in the configuration model. Dependencies between low level parameters are derived from the configuration model, avoiding repetition of parameters. This way less configuration parameters are needed when expressing the configuration in a high level model.

The compilation time the tool needs to translate the high level model is negligible. Our largest configuration model with 70 managed hosts requires less than 5 seconds to compile.

	Configuration model	Configuration files
Addresses	46	590
Parameters	75	328
Meta	29	81
Comments	37	538
Total	187	1537
Total without comments	150	999

Table 4.2: Lines of code for each category of configuration

4.3.3 Related work

The related work specific for this case can be divided into automatic configuration of network infrastructures and prefix allocation algorithms.

Narain et al. [80] propose a solution for the network configuration problem by using an object oriented configuration model based on first order logic. A solver is used to generate low level parameters using model finding. This solution differs from our approach by specifying configuration in first order logic. This complicates configuration specification; system administrators have to be familiar with first order logic concepts like inference, quantification and implication. Furthermore, when predicates are written in an inefficient but correct way, a solution may not be found.

Elbadawi et al. [42] present a framework that translates a high level configuration specification to low level parameters of each network component. The tool builds on top of NETCONF, an IETF standardized XML based RPC protocol for network management. The tool coordinates the configuration of interdependent network devices it manages via NETCONF by supplying a global NETCONF interface itself.

Atallah et al. [11] propose an algorithm for IPv4 prefix allocation and prove the optimality of the algorithm mathematically. The key difference with the proposed algorithms is the lack of support for aggregated routing.

Bhatia et al. [17] propose a distributed algorithm allocating IPv6 prefixes to MANET (Mobile Ad-hoc Network) nodes using ICMPv6. While a lot of effort is done to recuperate prefixes from dropped out or leaving nodes, no special care is taken to allocate prefixes in an aggregated way. Also, our allocation algorithm guarantees prefixes and routing happens along the shortest path tree with the border router as root. Whereas, in this approach interconnections between nodes are made randomly and a shortest path route to the border router is not guaranteed.

A more advanced dynamic prefix allocation algorithm for MANETs is proposed by Jelger et al. [65]. Introducing the concept of *prefix continuity*, they can guarantee prefixes are allocated in an aggregated way. In addition, path length is taken into account.

Kong et al. [68] developed a dynamic IPv6 prefix allocation algorithm inspired by memory management techniques using a paging mechanism. Bookkeeping guarantees subnet prefixes are split optimally and registers when subnets are added or removed. Route aggregation is not taken into account.

4.3.4 Conclusion

This case demonstrates IMP can raise the abstraction level of the configuration of a specific problem domain with many parameters at a low abstraction level such as network management. Additionally, it shows that the transformation plug-ins can express complex configuration parameter relations. This case fully automates the allocation of IPv4 and IPv6 network management. IMP requires all configuration parameter to be specified explicitly and does not have generative constraints (as opposed to for example PoDIM [32, 33]). However, transformation plug-in can automate domain specific parameter allocations. This case represents scenario 3 from chapter 3 where IMP manages a limited number of subsystems (everything related to networking) on all systems.

4.4 Case 3: integrated management of a cloud application

In this section we discuss our largest and most complete case (in terms of integrated configuration management): a configuration model to manage a distributed batch job application in a cloud setting, based on an industry case. The middleware for the batch job application is a generic middleware to enable workload distribution in a cloud computing environment. This service will be called the *Workload Management Middleware* from this point onwards. This Workload Management Middleware (WMM) has been developed and validated by applying it in the execution environment of a real world application case [66].

This case study uses IMP as a configuration management platform to automate the deployment and management of the workload management middleware. The platform fully automates the deployment and reconfiguration of an application container and a scalable database. Additionally the platform also deploys monitoring, logging and backup for delivering a production grade service. This section first discusses the architecture of the WMM and second describes the design of the IMP configuration model that automates the management and deployment of the middleware. Next this section presents an evaluation of the configuration model. Finally this section finished with an overview of related work for this case and a conclusion.

4.4.1 Case architecture

The application case is a distributed software system for document processing and document management. It supports the creation, the generation of layout, the business specific processing and the storage of business documents for various organisations. The application is a multi-tenant application where each tenant defines the specific document processing workflow. A workflow defines how the input data of a tenant needs to be processed into documents. This application handles batches of documents and therefore is confronted with large variations in work load. The strong variation is also caused by the type of documents which are often recurring, for example invoices or paychecks which are commonly generated at the end of the month. Due to these types of variations this document processing application is hosted in a cloud setting. Peak loads, such as the spikes at the end of a month are handled by allocating additional capacity in a public cloud. The latter is the responsibility of the WMM.

The application and its underpinning run-time environment has a layered architecture (Figure 4.11) consisting of three layers: the execution environment, the middleware layer and the application layer. The entire system, including these three layers, typically runs on the premises of multiple IaaS providers. This infrastructure layer is an additional lower-level layer in the overall architecture.

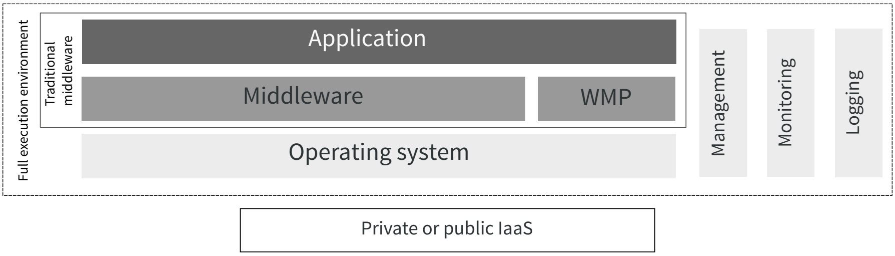


Figure 4.11: The three layered architecture with the execution platform, middleware platform and the application logic

Layer 1 includes virtual machines with an operating system and the system services required to operate the distributed application successfully. The most important subset includes: service monitoring, log collection, metric collection, backup, time synchronisation and naming. The virtual machines in the execution platform are allocated and provisioned on an IaaS platform.

Layer 2 provides middleware services that manages jobs, starts and stops worker threads, provides persistence of tasks and their results, communication between middleware instances and job and task scheduling. These services are realised in the form of a process on each of the virtual machines that execute tasks. Next to the middleware there is also a storage subsystem that provides scalable distributed storage in a reliable manner through replication.

Layer 3 is the application layer that hosts the application components itself. The components in this case are *workers* that are deployed on the middlware and that handle a specific step in a document processing workflow. Different worker components exhibit a different load on the underlying infrastructure and thus not all workers are deployed in the same numbers and same configuration. For example, rendering a PDF takes longer than splitting up a file in records.

In a traditional middleware setting, the deployment support remains limited to the composition of artifacts in layer 3 with artifacts of layer 2, i.e. the services that are embedded in the middleware platform. When addressing the challenge of integration development and operations, one aims for configuring the artifacts of layer 3 and layer 2 with a broad range of artifacts from layer 1, while shielding the low level details of layer 1 from the developers and operators. In other words, the goal is to create a completely configured system while hiding and managing the complexity of the configuration details.

The management of an application on the Cloud introduces additional complexity due dynamic scaling of the application, which may result in many changes in the configuration, changes that may occur at a fairly high frequency. The complexity of managing such a distributed environment is characterised by the following metrics, that correspond to the setting that we have used in our evaluation (Section 4.4.4):

- the number of virtual machines; for example 82 virtual machines
- the number of managed system services; for example 733 managed system services
- the number of installed software packages; for example 734 software package installed
- the number of configuration files; for example 1485 configuration files

Based on an analysis of deployment requirements from our real world application case study, we have identified a set of configuration changes that must be supported by our configuration management platform in order to deal with the complexity introduced above. We will use these configuration changes to drive the development and the evaluation of the IMP configuration model. These configuration changes are summarized below:

1. Deploy the application and the entire run-time environment on an IaaS platform. This setting defines a minimal deployment context, which is sufficient to deploy the application with minimal complexity.
2. Add additional workers: a worker is defined as a virtual machine that contains an instance of the application logic and the database, configured to execute one or more tasks of the application. Each new worker virtual machines is integrated with the layer services such as naming, monitoring and backup.
3. Remove workers
4. Decommission the application: i.e. remove the application and its run-time environment from the IaaS.

4.4.2 Design of the configuration model

The idea behind the design of the configuration model is straightforward: it is a layered configuration model in which each layer of the design consits configuration modules.

Each module increases the level of abstraction by encapsulating configuration details behind a clear interface that other modules can use. Conversely, configuration modules in layer N refine the configuration model in function of concepts (entities) of configuration modules of layer N-1.

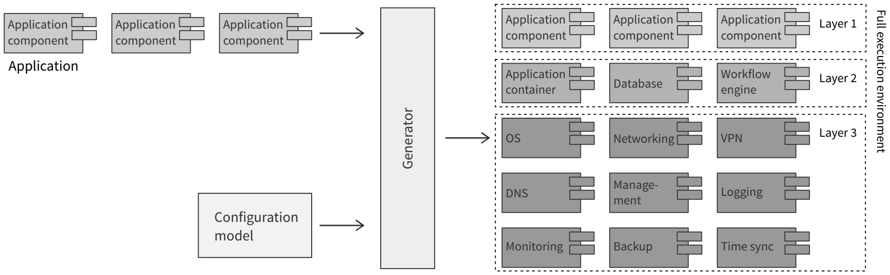


Figure 4.12: Our middleware extension generates a deployment and configuration of the full execution environment (including operational services) based on the application and the configuration model.

4.4.2.1 A baseline configuration at the infrastructure level (layer 1)

At the infrastructure level, an initial configuration is required to shield the variety of infrastructures (typically IaaS offerings) that the application may be deployed on. The configuration model therefore configures the IaaS environment to obtain similar behaviour on each IaaS technology that must be supported. This creates a baseline configuration for all the possible deployment situations that need to be addressed.

Each IaaS defines flavors of virtual machines which determine the amount of available memory, cpu and other resources. The configuration model makes abstraction of this heterogeneity by providing its own flavors that map to IaaS specific flavors. For example, the platform defines the type normal which on a private cloud provides 4GB of memory and 3.75GB on the Amazon EC2 IaaS.

Each virtual machine also requires a baseline configuration that the configuration model defines. The baseline configuration ensures that a virtual machine is integrated in the platform by means of: configuration of a management agent, configuration of networking and routing, a DNS record for the host, configuration of nameservers, centralized logging, time synchronisation, central collection of system and application metrics and the setup of a backup agent.

Additionally the configuration model also includes operational services at the infrastructure level, such as: monitoring, backup, time synchronisation, VPN to include the cloud network in the enterprise network,… They complete the baseline configuration to provide an operational execution platform in layer 1. The management server in this layer also interfaces with the IaaS platforms to provision additional virtual machines.

4.4.2.2 Configuration of the distributed middleware layer (layer 2)

IMP deploys a the middleware layer that provides a distributed workflow middleware and a clustered database on the baseline configuration of layer 1. The deployment and configuration management of this layer is fully automated. The layered architecture (Figure 4.11) ensures that layer 2 can make assumptions about many functional aspects of the execution platform such as consistent naming, integration in the enterprise network through a VPN and the availability of monitoring and log management and most important the necessary services in place to fully automate the configuration of layer 2 and the application in layer 3. The implementation of layer 2 refines the types provided in layer 2 in function of types defined in layer 1.

4.4.2.3 Fully automated application configuration (layer 3)

The configuration model exposes the middleware platform to the application layer through the concept of *Worker nodes*. The configuration model describes the configuration of the middleware layer on each worker node and allows the application layer to further define how a worker node is implemented. In this application layer the operator defines each of the worker node types which map to a type of virtual machine and a set of worker components is deployed. The WMM uses worker nodes as unit of scale. Workers components are scaled by starting or stopping workers nodes.

4.4.3 Implementation overview

The implementation consists of generic modules that configure a service or subsystem except for the *drm* and *demo* module (Figure 4.13). The drm module provides types to model the configuration of the case architecture and the necessary refinements in function of the generic modules. The demo module instantiates the architecture on an IaaS platform with the configuration of the workers as specified in this section. The demo module exposes this instantiated configuration model parametrized with respect to the number of workers of each type.

The drm module provides all the configuration concepts to build a complete distributed system on a public or private cloud. For the case described above, a composition of the concepts in the drm module needs to be instantiated to define the initial configuration

Module	Layer	Description
apache	1	Implementation of the httpd types for the Apache webserver
backup	1, 2	Configure files and directories to include in backups
bind	1	Implementation of the DNS
cassandra	2	The distributed storage system used in case
collectd	1	Daemon that collects system sensor data on each server such as cpu usage or disk load
demo	3	Instantiates entities of the drm module that are environment and application specific and configures the workers
dns	1	Types for the DNS subsystem
drm	1, 2, 3	This module implements the architecture of the system presented in this case
duplicity	1	A backup system that implements the backup types
elasticsearch	1	A document store and indexer used by the flens and kibana modules to collect logs
flens	1	A monitoring systems for sensor and log data
graphite	1	A monitoring system that stores and graphs data
hbase	1	The column oriented store of the Hadoop project, configured in single node configuration
httpd	1	Types for configuring http servers
imp	1	Configures IMP server and agents on machines
ip	1, 2	Types to represent network related configuration
jackrabbit	2	A webdav server used by the archive worker to archive documents produced by the document processing workflow
java	2	Installs the java virtual machine
kibana	1	A frontend to search logs in elasticsearch
logging	1	Types for configuring logging
monitoring	1	Types for configuring monitoring
net	1	Types for network related configuration
ntp	1	Configuration for the network time protocol
opentsdb	1	A time-series database that stores all data and does not resample monitoring data like graphite
rabbitmq	1	A message queue used by IMP
redhat	1	Configuration for redhat based systems
ssh	1	Configures ssh keys and server
std	1	Types that represent resources managed on servers and the handlers required for the agent to deploy changes to these resources
taskworker	2	Configures the taskworker middleware
vm	1 (0)	Configures virtual machines and contains the handlers to manage these virtual machines on an Openstack IaaS platform
yum	1	A module to configure yum software package repositories

Figure 4.13: IMP modules used in this case and the layer they are used in.

model (Figure 4.15). However this code can be furthered parametrized in function of the scale of the deployed application. In (Figure 4.14) the actual model code is shown that is used to manage the entire distributed system and which we will use in the evaluation section. This effectively reduced the entire configuration of the distributed system to a single line with 4 parameters.

```
 entity Cfg:
     number preprocess = 0
     number render = 0
     number postprocess = 0
     number master = 0
end
implement Cfg using infrastructure

demo::Cfg(preprocess = 1, render = 1, postprocess = 1, master = 1)
```

Figure 4.14: The configuration statement that defines the entire configuration in function of 4 parameters that determine the scaling of the managed distributed system.

4.4.4 Evaluation

The evaluation scales the configuration model from the previous section to evaluate IMP performance. The previous two cases evaluated the usability properties of the framework and maintainability of the configuration model. This evaluation focuses on the scalability of the framework. It scales the model from the base infrastructure (5 VM's) to an infrastructure with 92 virtual machines. The evaluation shows how IMP performs in a realistic infrastructure both in terms of duration and the amount of automation it achieves.

4.4.4.1 Evaluation scenarios and metrics

The configuration model is compiled by IMP in 12 different configurations. For each of the configurations it compiles the configuration model and exports the configuration model but does not submit it to the IMP server to actually deploy it. From the exported configuration model metrics about the number of managed resources and the use of configuration parameters is extracted.

Figure 4.16 shows the number of virtual machines that the configuration model generates and configures in each configuration of the evaluation. Additionally it shows the input parameters of the configuration model that determine how the model scales.

```
implementation infrastructure for Config:
# create a new dynamic resource management infrastructure
infr = drm::Infrastructure(domainname = ”dreamaas.cs.kuleuven.be”)

# add cloud providers
home = vm::IaaS(name = ”dnetcloud”, homecloud = infr)
home.config = vm::IaaSConfig(connection_url =
    ↪ ”http://dnetcloud.cs.kuleuven.be:5000/v2.0”, username =
    ↪ drm::credentials(”home_username”), password =
    ↪ drm::credentials(”home_password”), tenant =
    ↪ drm::credentials(”home_tenant”), type = ”openstack”)

# add time servers
infr.ntp_servers = ntp::ExternalServer(name = ”134.58.255.1”)

# create node types and mappings
small = drm::NodeType(name = ”small”)
normal = drm::NodeType(name = ”normal”)
large = drm::NodeType(name = ”large”)
storage = drm::NodeType(name = ”storage”)

image = drm::Image(image_id = ”ed9ae349—9b80—4d13—b123—f4d6ffbf752d”, os =
    ↪ ”fedora—18”)
admin_key = ssh::Key(name = ”admin”, public_key = source(”demo/admin.pub”))

infr.authorized_keys = [admin_key]

# map the type to vm flavors in the home cloud
drm::TypeMapping(node_type = small, iaas = home, vm_type = ”small”, image =
    ↪ image, ssh_key = bartvb_key)
drm::TypeMapping(node_type = normal, iaas = home, vm_type = ”normal”, image
    ↪ = image, ssh_key = bartvb_key)
drm::TypeMapping(node_type = large, iaas = home, vm_type = ”large”, image =
    ↪ image, ssh_key = bartvb_key)
drm::TypeMapping(node_type = storage, iaas = home, vm_type = ”large.plus”,
    ↪ image = image, ssh_key = bartvb_key)

# add base services
svc_naming = drm::services::Naming(infrastructure = infr, node_type = small,
    ↪ hostmaster = ”bartvb@cs.kuleuven.be”)
svc_mq = drm::services::MQService(infrastructure = infr, node_type = small)
svc_mgmt = drm::services::Management(infrastructure = infr, node_type =
    ↪ small, mq = svc_mq, public_key = bartvb_key)
svc_mon = drm::services::Monitoring(infrastructure = infr, storage_node_type
    ↪ = storage)
svc_vpn = drm::services::VPN(infrastructure = infr, node_type = small)

# add workers
preprocess_workers = drm::capacity::WorkerType(worker_type = ”preprocess”,
    ↪ infrastructure = infr, spillover = false, instances =
    ↪ self.preprocess, node_type = normal)
render_workers_home = drm::capacity::WorkerType(worker_type = ”render”,
    ↪ infrastructure = infr, spillover = false, instances = self.render,
    ↪ node_type = large)
join_worker = drm::capacity::WorkerType(worker_type = ”postprocess”,
    ↪ infrastructure = infr, spillover = false, instances =
    ↪ self.postprocess, node_type = large)
master_worker = drm::capacity::WorkerType(worker_type = ”master”,
    ↪ infrastructure = infr, spillover = false, instances = self.master,
    ↪ node_type = normal)
end
```

Figure 4.15: An instantiation of the configuration model, specifically for the case described in this section. This model is parametrized with respect to scaling. This model code is contained in the *demo* configuration module.

configuration	master	render	preprocess	postprocess	total
1	0	0	0	0	5
2	1	1	1	1	9
3	1	3	1	1	11
4	1	6	1	2	15
5	1	10	2	4	22
6	1	15	3	5	29
7	1	20	4	7	37
8	2	25	6	10	48
9	2	30	7	12	56
10	2	35	10	15	67
11	2	40	12	18	77
12	2	50	15	20	92

Figure 4.16: The size (in number of managed virtual machines) in each of the configurations.

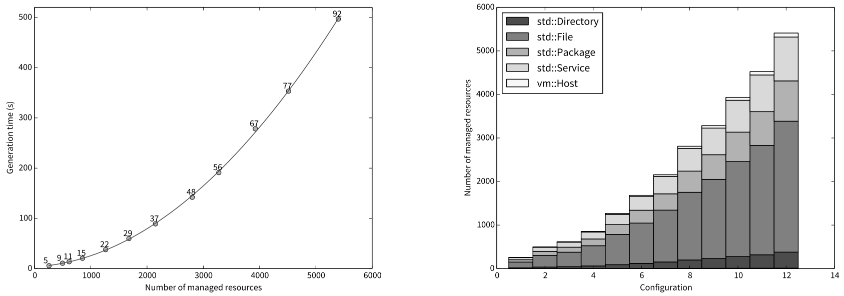
After each configuration in the evaluation scenario the number of managed resources (virtual machines, files, directories, software packages and system services) is reported. Additionally the number of parameters substituted in the templates that generate the configuration files is collected. The number of resources that are managed gives an indication of the amount of work required to make a change, while the number of configured parameters gives an indication of how complex a configuration is. Moreover, the number of parameters also indicates how error prone the configuration is when changes are made [86]. We can categorize configuration parameters in seven categories related to their effect on the consistency of the configuration of a distributed system:

- **file** A parameter that is configuration file specific and that has no meaning outside the configuration file.
- **network references** A parameter that references to another resource in the distributed system itself (over the network). The best known example is a hostname or ipaddress. These parameters are the most error prone because many instances of a single parameter are scattered over multiple configuration files and thus need to be kept consistent. If not, this can affect the entire distributed system.
- **network** A configuration parameter of the network. For example the domainname of all hosts in the distributed system.

- **host** A configuration parameter that is only meaningful within a single host.
- **host reference** A reference within a host, for example the filename where the private key of a service is stored.
- **global** A configuration parameter that is global for the distributed system.
- **external reference** A reference to something that is external to the distributed system. For example a repository with additional software packages or the hostname of a time service.

4.4.4.2 Measurements

The measurement is the generation time required for each of the configurations of the model. The results of this benchmark (Figure 4.17) show a quadratic relation between the number of resources that IMP manages and the evaluation time, albeit with a very small quadratic parameter and a large linear contribution to the generation time. The resource that IMP manages the most is clearly files which are mostly configuration files.



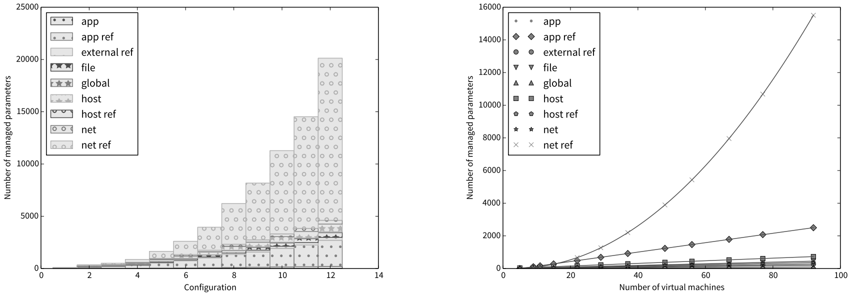
(a)

(b)

Figure 4.17: (a) The generation time for all managed resources in function of the number of managed resources. The data points are labelled with the number of virtual machines that are managed in the corresponding configuration model. (b) A breakdown of the types of resources that are managed in each configuration.

IMP generates configuration files by substituting parameters from the configuration model in a template (Figure 4.18). These results show that network references and application references have a quadratic contribution to the number of parameters substituted in a template. The second graph (Figure 4.18) plots a second order polynomial regression of each parameter in function of the number of virtual machines.

This plot clearly shows that references to resources over the network, which is very common in distributed systems, grow quadratic with the number of virtual machines. While all other parameters grow linear with the number of virtual machines.



(a)

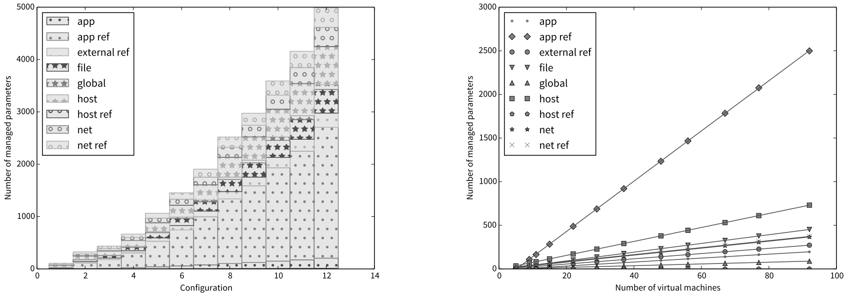
(b)

Figure 4.18: (a) Each bar represents a configuration of the distributed system and the number of parameters that have been used to generate the configuration files. (b) The number of managed parameter of each type in function of the number of virtual machines that IMP manages. The curves through the point are the result of a second order polynomial regression.

However, this quadratic growth may be attributed solely to the use of a snitch file for Cassandra. This file contains a list of all nodes in the Cassandra's cluster and their location. Cassandra uses this file to determine the distance and placement of replicas. In figure 4.19, a plot of the same curves and the total compilation time is shown for a configuration which does not generate the snitch file but uses Cassandra's built-in discovery mechanism. These results show that the number of references to resources on the network is still large but also grows linearly with the number of managed machines. The effect on the compilation time is however negligible. Now the largest number of managed parameters comes from the application, because IMP is also used to generate the deployment descriptor for the application components.

Figure 4.20 shows a breakdown of the number of managed files in four categories:

- files that are new in this configuration (based on host and pathname)
- files that were deployed in the previous configuration and the contents has not changed
- files that were deployed in the previous configuration but their contents has changed



(a) (b)

Figure 4.19: The generation time (a) and the number of managed parameters (b) for each of the configurations when the built-in Cassandra peer detection is used.

- files that were deployed in the previous configuration, but are deleted in the new configuration (This category does not occur here because we only scale up)

These results show that adding virtual machines (or other devices) to an infrastructure also has an effect on existing configuration and not only adds new files. This means that adding new virtual machines is a cross-cutting operation in the configuration.

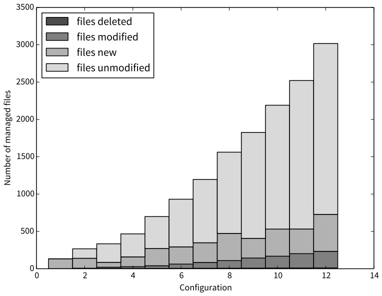


Figure 4.20: An breakdown of the number of managed files in each configuration.

4.4.4.3 Discussion and future work

The case in the evaluation and the results of the evaluation show that it is possible to fully manage a distributed application with an integrated configuration management tool. Even for large sizes the generation time of the configuration model is reasonable. The time to deploy changes is very similar to other configuration management solutions, as the deployment time is limited by the time the package manager takes to download and install software packages.

The steep increase in number of managed resources and especially the increase in the use of network reference parameters shows that automated configuration management is a must. Even a relatively small infrastructure with almost 100 virtual machines results in thousands of managed files, and each additional virtual machine requires existing configuration files to be updated. This also indicates that adding (and conversely removing virtual machines) is an error prone operation that is very cross-cutting in the deployed configuration.

The evaluation shows that discovery mechanism such as used by Cassandra can be avoided when IMP is used to manage a distributed system. A fixed configuration which is consistent over all machines can provide the most reliable configuration. But discovery mechanisms are often used instead because the number of configuration parameters quickly explodes to unmanageable proportions with manual configuration in peer-to-peer distributed systems. The effect on the generation time of IMP is negligible, although an order of magnitude more parameters need to be managed. This offers a choice to operators to choose for manual configuration instead of using discovery mechanisms which are not always reliable on all environments: for example, the lack of broadcast on many IaaS platforms. Additionally, these dynamic mechanisms are often difficult to debug.

The compile time grows more than linear which can become an issue once IMP would be used to manage hundreds of systems in an integrated configuration model. However, this evaluation is based on a research prototype that is single threaded and has not been optimized. We do think that for future work there is much opportunity for optimizing the generation process.

4.4.5 Related work

The systematic integration of operational aspects in a modern development and deployment environment (which is the role of many contemporary middleware platforms) is not an over-crowded research area. We have addressed this challenge, in fact inspired by recent literature: our middleware extension addresses many of the configuration management concerns raised in the paper of Cook et al. [29]. Our middleware extension provides the tool for developers and operators to publish, provision and deploy applications onto cloud platforms.

Related work can be found in the actual operational support that comes with cutting-edge middleware platforms, especially those that deal with very complex and dynamic environments. This brings us in the domain of PaaS. Indeed, some integration PaaS platforms [5, 55, 76, 92, 99, 117] also deploy and entire execution environment based on the application artifacts. Yet these are relatively closed environments: users of these platforms are limited to the specific middleware that the PaaS offers. For example, only a few platforms such as OpenStack or Cloud Foundry [92, 99] support private clouds. In adidition, such solutions do not hande the full execution environment down to the infrastructure level: they only fully automate the 'traditional' middleware stack (layer 2 in this paper) and not the additional system level services such as monitoring.

Yet we have to mention a few important pieces of research, that operate in the same space, while focusing on other priorities.

For example, the work of Dumitras et al. [39] is complementary to ours in the sense that it focusses mainly on the *migration* of the application state and on the verification of correct deployment of configuration changes in running applications. Our work mainly focusses on the description of the execution platform and executing the changes starting from a greenfield setting.

Finally, Claudia [100] is also a very interesting result because it builds an abstraction layer above IaaS platforms to manage applications. However the focus of this work is on the *automatic scaling* of cloud services. Our work in contrast focusses on the management of the entire execution environment (including operational services) starting from the actual execution of the configuration changes.

4.4.6 Conclusion

This case shows that IMP can deploy incomplete configuration models in a setting with an integrated configuration model. IMP controls the provisioning, deployment and configuration of the entire distributed system and its execution environment. This validates that IMP can support scenarios 1 to 5 with an emphasis in this case on scenario 5. Moreover it does this with a limited generation time which allows fast scaling of the infrastructure.

IMP supports configuration module reuse. Many configuration modules in this case were further developed from case 1. It became clear that the design patterns used in the modules of case 1 worked against this re-usability: modules in case 1 had many implicit configuration parameters in its interface. For example, the relation of files and packages to the device they are managed on were derived from the context during the evaluation with a special language construct. This sped up initial development and allowed for the reuse of development best practices from tools such as Puppet where the managed device is always implicit. However, it was counter productive for module reuse: the actual interface of types only becomes clear during the refinement process and resulted in trial and error development. This shows that concepts used in software engineering such as well-defined interfaces, also applies to the development of configuration models. The reused modules from case 1 that were reused in case 3 have been re-factored to explicitly define all configuration parameters.

4.5 Analysis of IMP as a tool

In Chapter 2 we proposed an evaluation framework for configuration management tools. In this section we evaluate IMP based on this framework by analysing how IMP performes on each of the properties of the framework. We also take a look at our gap analysis from Chapter 2 and evaluate if and how IMP address the identified shortcomings.

4.5.1 Specification properties

The main input to define the configuration model is the IMP modeling DSL. In this evaluation we consider this as the input language and only consider other specifications such as the Python plug-ins and templates when they are relevant.

Specification paradigm The IMP domain specific language is a *declarative* language which specifies a desired state model of the managed infrastructure. The input specification is entirely by means of source code, therefore the user interface is *text based.*

Abstraction mechanisms Refinement with types and refinements in IMP provide an abstraction mechanism for heterogeneity and complexity. To make abstraction of heterogeneity refinement are selected at runtime based on properties of the environment to make abstraction of heterogeneity. In our framework, IMP only expresses instance configurations because IMP does not have generic support to determine configuration parameters based on constraints.

However, case 2 and 3 demonstrate that it is possible to express end-to-end requirements such as in Figure 2.1. IMP can create a configuration model that is parametrized to the replication level of the mail cluster, additionally IMP can deploy a monitoring system to monitor the response time of the cluster. A transformation plug-in (such as used in case 2) can determine the required replication level of the cluster to achieve the requested latency requirement.

Modularization mechanisms An IMP configuration model (as used in the three cases) is built from many reusable configuration modules and one or more environment specific modules that create an integrated model from the reusable configuration modules. Relations and object oriented design can be used to achieve grouping for the assignment of roles to devices.

Modeling of relations A key focus of IMP is modeling all relations between configuration parameters. Combined with the abstraction mechanism of IMP, it is feasible to capture all relevant relations in a single integrated configuration model to ensure that there are *no* duplicated configuration parameters. Relations are typed and have a multiplicity. Additionally, a library of transformation plug-ins can be used to define assertion over relations to validate them. The most comprehensive plug-ins that achieves this, is the transformation plug-in to generate firewall rules. This plug-in issues compilation warnings whenever a client-server relation is modelled that is not allowed by the security policy.

4.5.2 Deployment properties

IMP has two subsystems, generation and enforcement, that affect the deployment properties, we will discuss their impact separate in each of the properties.

Scalability case 3 shows that IMPs generation subsystem can scale to at least 100 virtual machines in a complex integrated configuration model. As this is a research prototype that does not focus on performance, there is room for improvement. The generation subsystem currently relies on duck typing, however during further development of the modules of case 1 to create case 3 it was apparent that this makes the modules very brittle and hard to reuse. Implementations in configuration modules of case 3 currently only bind variables that are defined explicitly in the interface (entity). This opens up possibilities to remove the heuristic based evaluation because variable bindings can be exact for all DSL statements and a faster execution.

The enforcement subsystem is pub/sub based and is only limited to the message rates the AMQP server supports. If one message broker cannot handle all deployment agents anymore, a cluster of brokers could scale the number of deployment agents further. Additionally there is much room for speedups in the current resource handlers.

Workflow IMP can deploy changes with dependencies over multiple hosts. For the deployment agent there is no difference between intra and inter-host dependencies. IMP has no state transfer support. The current configuration modules that exist, only model deployment dependencies between resources such as files, packages and services within one machine. Extending this to dependencies between services that connect over the network is currently subject of further research.

Deployment architecture The generation subsystem of IMP is centralised, the enforcement subsystem of IMP is weakly distributed with a hybrid push/pull model.

Platform support Resource handlers exist for managing files, directories, symlinks, packages and system services on Fedora, Ubuntu and CentOS. IMP can also manage virtual machines on Openstack. However most configuration modules in case 3 only support Fedora.

4.5.3 Specification management properties

These properties focus on the support to develop IMP configuration models.

Usability The modeling language of IMP should be fairly easy to learn for operators that have experience with object oriented modeling and programming. This statement is also supported by the fact during this PhD thesis, five master thesis student independently learned to used IMP with limited documentation at their hand.

There is no explicit support to test configuration modules, except by defining assertion in-line in the model code. Support for monitoring the infrastructure itself is not available in the generation subsystem or the deployment agents, apart from the capability to determine the current state of a resource.

Versioning support All artifacts that are part of a configuration module (model code, templates, plugins, etc.) are text based. This implies that source code management systems, such as git or subversion, can be used to track the development of an integrated configuration model or configuration module.

Specification documentation IMP has syntax support to add in-line documentation in the model code but currently lacks a tool to distill configuration from it. Export plug-ins are intended to generate such documentation. The *graph* module is an example of a documentation export plug-in. It allows an operator to generate a diagram from a configuration model. Figure 4.5 is an example of such a generated graph (the layout was cleaned up for publication purposes).

Integration with environment IMP's transformation plug-ins provide an easy extension mechanism to integrate with existing databases. For example, based on the hostname and the name of the network interface lookup its MAC address.

Conflict management if conflicts are encoded in the meta-model as type constraints or added as assertion, IMP will generate compilation errors for both application specific conflicts and for modality conflicts.

Workflow enforcement IMP has no update workflow enforcement support.

Access control IMP has no access control support.

4.5.4 Support

The properties related to support are mainly important for companies that adopt a configuration management tool. IMP is a research prototype therefore support is limited.

Available documentation Limited to a language reference and a tutorial that guides the reader through the setup of a two tier web-application on two virtual machines.

Commercial support No

Community No

Maturity Research prototype

4.5.5 Gap analysis

In Section 2.3 we identified six areas for improvement for existing configuration management tools. In this section we compare the contribution of IMP with these areas for improvement.

1. **Create better abstractions** IMP provides the required abstraction capabilities for defining the configuration of a complex distributed system (case 3) at a high abstraction level and refine that configuration to low level resources such as configuration files.
2. **Adapt to the target audience's processes** IMP's framework approach allows system administrators to embed their scripts inside IMP. However, it still requires a radical approach of managing an infrastructure.
3. **Support true integrated management** The case studies show that an integrated configuration model for an entire infrastructure is feasible with IMP.
4. **Become more declarative** IMP is fully declarative similar to existing tools such as Puppet en Cfengine.

5. **Focus on existing business processes** IMP's prototype has not focussed on this. The suggestions from the areas of improvement can be added but need to be development from scratch as there is nothing there yet.
6. **A system is software + configuration + data** The enforcement subsystem has no support for state but the integrated configuration model of IMP does provide the required relations to reason about this during deployment. In fact, a master thesis student is currently taking the first steps in this direction.

4.6 Conclusion

The three cases and their evaluation show that the IMP design and prototype can:

1. Manage an entire infrastructure (applications, servers and network) from a single integrated configuration model.
2. Fully automate the allocation and assignment of configuration parameters with domain specific algorithms.
3. Bootstrap and deploy and entire execution environment on a cloud infrastructure and automate the scaling of the platform.

The development of the modular configuration models of the three case also indicate that many software engineering concepts can also apply to the development of a configuration model:

- Modelling techniques used in object orientation: represent concepts and things from the real world in the configuration model.
- Encapsulation (refinement) allows complexity to be hidden and raise the abstraction level.
- Modular development of configuration modules.
- Well-defined interfaces that make all parameters and dependencies explicit.

The IMP prototype and many configuration modules from case 3 are publicly available on GitHub[1] under the Apache License.

---

[1] https://github.com/bartv/imp and https://github.com/bartv/imp-* for the modules

Chapter 5

Related work

The related work that comes with this dissertation is versatile. Most related work is concentrated in Chapter 2 and Chapter 4. Chapter 2 gives a generic overview and analysis of related work in the space of configuration management tools in industry and research that can be used to manage real infrastructures. A second body of related work is presented in Chapter 4. This related work is specific to the domains of each of the case studies. More specifically in sections 4.2.4, 4.3.3 and 4.4.5. The role of this chapter is only to complement the core of related work that has been presented before.

IMP's contribution is in the improvement of how configuration management tools specify the input of the tool to raise the abstraction level in which configurations are expressed and generate deployable configuration artifacts from this input. Section 5.1 discusses related configuration management tools and research. It addresses tools that offer a generic configuration management tool or use techniques that can apply to generic configuration management. It does not include solutions for domain specific configuration management problems as these focus on parameter value allocation or deployment of changes, for example from the domain of QoS in networks, scaling of distributed applications or distributed firewall policy configuration. Section 5.2 provides an overview of related work on modeling and representing a configuration. Section 5.3 touches upon the challenge of making configuration management tools useful for and appreciated by the human end users (typically system administrators). We present a brief intro to related work, as it has been a source of inspiration for this dissertation. Yet it should remain clear that usability aspects do not belong to the scope and to the main contributions of the dissertation.

5.1 Configuration management tools

IMP has important related work in system configuration tools from industry: Cfengine3 [22] and Puppet [96] both work with a desired state model of the configuration. IMP shares this architecture with these tools, but IMP improves upon this by providing capabilities to define abstractions over machine boundaries. This is represented in the fact that the agent of both cfengine and puppet trigger a compile of a deployable configuration model for the host the agent manages. IMP generates a single integrated configuration model that contains all dependencies. In practice users of these tools extract parameters that span over host boundaries to an external database [1] to ensure that there is only one authoritative source for a parameter. However this is actually history repeating itself: tools are once again wrapped with deployment specific ad-hoc scripts. IMP makes most parameters implicit by including a relation between parameters in the model. This lack of abstraction mechanisms also forces operators to resort to the execution of custom imperative scripts on machines instead of only relying on the declarative desired state model.

IMP provides a framework to operators to manage their systems. Chef [24] also provides a framework to operators. Operators specify in an internal Ruby DSL how the configuration of the distributed system should be configured. Chef provides APIs to manage resources on a machine in a declarative fashion. However it is still a script which is executed in the order in which statements are specified and without abstraction capabilities for configuration management. Additionally this approach often does not result in a deterministic behaviour [61].

SmartFrog [54] provides a framework for configuration management of distributed systems. It differs from IMP in the fact that SmartFrog does not offer any mechanisms to build abstraction and only focusses on the level of abstraction of the managed resources the framework supports, such as an Apache webserver. It does have the notion of direct references to configuration parameters, in contrast with IMP's transformation mechanism.

Burgess and Couch [21] introduce concepts that a next generation configuration management tool should contain. They focus on modularisation but the main problem with their requirements is that they introduce new and mostly ambiguous terminology, instead of relating to existing fields such as software engineering. What they call promises is very related to a stable and well defined interface, exactly what IMP provides with its entities and relations. Most requirements relate to the interfaces between modules in a configuration model and less on raising the abstraction level and trying to model all relations in the infrastructure. The authors used the concepts in this paper in Cfengine version 3.

Pattern-based Composite Application Deployment [41] describes the deployment of a desired state configuration model using existing workflow oriented imperative scripts or management interface. They extend the model driven approach from Eilam et. al [40] to transform the desired state model into an ordered sequence of script or workflow invocations. For the deployment of an IMP configuration model there are two options. First, with the deployment subsystem which for each deployable resource requires a handler to transform a desired state to a set of imperative actions. Second, with a custom export plug-in which interfaces with an existing management interface.

A formal model driven approach [40] for management tools employs a GUI to create the models and either the GUI or an API for refinement transformations. Their main requirement is separation of concerns for the different stakeholders in deploying and managing applications in a distributed system. This differs from our work in the following aspects: Our approach is more related to object oriented development and agile methodologies, while their approach is in the tradition of model driven development. This also manifests in a second difference. The architecture of IMP enables scripting within the framework in the form of transformation plug-ins. These scripts can be used to automate specific parameter transformations. The work of Eilam et al. needs domain specific search heuristics for automatic model transformations which are non-trivial to develop according to the authors and not reusable. Additionally IMP uses text based input which operators prefer over GUI based tools to create the models [14].

A mechanism to do dependable upgrades is proposed by Dumitras et al. [39]. The upgrade process is performed by switching to a mirror of the infrastructure that has been upgraded. An upgrade is split into phases: bootstrapping the mirror infrastructure, datatransfer, termination, testing, and switchover. This work mainly focusses on the state transfer while the bootstrapping phase is not covered in the paper. For a repeatable bootstrap and upgrade process an integrated configuration model is required.

iManage [69] is policy-driven approach for self-management of enterprise-scale systems. The system is able to manage several parameters of an distributed application to adapt it to shifting workloads. If these parameters are changed, the middleware platform needs to reconfigure. The parameters can have repercussions to the execution environments that is not part of the middleware platform. If a parameter is changed in a configuration model of the full infrastructure, the entire infrastructure can be reconfigured and adapted to the shifted workload. A framework such as iManage could be part of IMP to automatically allocate specific configuration parameters.

The white paper “An architectural blueprint for autonomic computing” [28] from IBM introduces a blueprint for an autonomic system that manages technology with technology. It introduces the MAPE-K cycle (Monitor-Analyse-Plan-Execute based on Knowledge). This is how the deployment agent of the framework functions for managing resources based on their desired state. It can however, not compose other resources in function of resources in the MAPE-K cycle. The generation subsystem provides this composition for manual managers. System operators can use the transformation plug-ins (as shown in case 2) to close the MAPE-K loop for specific management problems. In the light of ITSM, the framework focuses on the control scope axes of the autonomic computing adoption model. It provides the ability to model a complete business system. For the functionality axis it moves to the "instrument and monitor level" (by automatically configuring monitoring systems), but for specific problems it can move to a closed loop system. The configuration model contributes to the knowledge in MAPE-K.

PoDIM [32, 33] introduces new concepts in configuration models: use object oriented domain model so an operator can model *the real world*, much like in object oriented software development. IMP also uses object oriented modeling. Additionally, PoDIM is uses an SQL like language to instantiate the model. The management engine can instantiate the entities in the model based on two language constructs. First by directly instantiating entities in the domain model. The second language construct can define constraints such as: every network needs to have two DHCP servers. The management engine will create instances of the entities in the domain model to satisfy the constraints. IMP does not provide these creation constraints because it is very difficult to reach a stable state with no reconfigurations. Additionally, it requires the operator to formally encode all constraints that he takes into consideration to make decisions. This results in a configuration that is optimal for a constraint solver but not for a human operator. However, IMP can fully automate specific well understood configuration parameter allocations such as shown in case 2.

Narian et al. [81] propose a method based on constraint satisfaction and model finding. Their method provides a higher level input language for a configuration model and combines model finding with other methods to speed up resolving the model. The authors validated the tool in specific configuration tasks such as IP address assignment. The tool does not have facilities to model abstractions or extending the tool with scripting. A future addition to IMP could use model finding for the allocation of a specific set of parameters in a configuration model. This method requires to much formalised configuration knowledge to replace many of the decisions an operator would make while defining an integrated configuration model for an entire distributed system.

Next to Delaet et al. with PoDIM and Narian et al. many other systems exist that use constraint satisfaction for configuration management [47, 59, 80, 97, 102–105]. The main issue with automatic constraint satisfaction is the need to formally specify information, including experience of system administrators. Even if all information required to decide values of configuration parameters is available to a tool, for large infrastructure the computation time would be large, as indicated by the evaluation of these tools (order of magnitude of hours or even days). The need to for many reconfigurations per day or even per hour makes generic constraint satisfaction not feasible. IMP's transformation plug-ins allow to use constraint satisfaction for the allocation of specific configuration parameter sets, based on the parameters available in the model (removing the need to formally model information in for example first order logic).

5.2 Modelling of configurations

IMP now contains its own types to manage servers. Existing models could serve as input for this. The CIM [36] standard from the DMTF provides a domain model for management tools which can be implemented in IMP. IMP can also use a different deployment subsystem based on for example WS-MAN [3] or OpenLMI [2] which provides a CIM implementation and deployment agent for Linux based systems.

Rodosek proposes a methodology [101] for modeling services instead of devices. She also proposes to use an object oriented approach combined with UML. His work is complementary to our work as she proposes a methodology that is compatible with how operators can develop an IMP configuration model. Case 3 models services in the initial configuration model and does not contain any visible virtual machines.

Agrawal et al. [4] describe the design of a CIM compatible policy language. This language is also declarative in the sense that it is event-condition based. Our approach is a state based approach, where we model the desired state of the infrastructure.

The OASIS TOSCA [84] provides support for cases similar to what Case 3 illustrates. However, it does not provide the capabilities to build up abstractions (refinement) and thus reuse configuration effort. In that sense it is very similar to what Ubuntu Juju [71] provides.

5.3 Humans and automation

The main contribution of this dissertation is automating the configuration management of a large distributed system. There is a body of work related to automating technical tasks in which humans are involved. The main focus of that research is operator jobs in plants or pilots in a cockpit, but many conclusions apply to operators of IT infrastructures as well. Additionally in the previous decade research was carried out that investigates operators of IT infrastructures as a very specific group of computer users.

Hrebec et al. provide a detailed survey of the demographics of operators and their mental models [60]. Sheridan gives an overview of the state-of-the-art in Human Factors research in automation [106]. Their work that focusses on automating tasks without taking away control of the operator is relevant for automating system administration tasks. Desai et al. [35] show that the social aspect of configuration management automation is a challenge as well.

The body of work exists that gives detailed and specific guidelines for automating system administration tasks [14, 18, 57, 114]. The authors base these guidelines on observations of the working operators and surveys. They focus on current imperative or task based tools and not on configuration management tools that use a desired state configuration model to configure and manage an infrastructure. Nonetheless this work inspired the design and implementation of the framework and its prototype.

Chapter 6

Conclusion

Many applications and software services that we depend on in our day to day lives, for both consumers and enterprises, are developed and deployed as distributed systems. Developing and operating a distributed system is more complex than a typical single machine application. Software engineering has progressed over the past decades to make it possible to design and develop large (even globally distributed) and complex distributed systems. Operating a distributed systems is as complex as developing it. Unfortunately, it has not received as much attention both from research and industry. Moreover, cloud computing allows us to increase the size of distributed even more and appears to have raised attention for the operational side of software, both from practitioners and research. This is illustrated by: movements such as DevOps, new monitoring and configuration management tools that appear and research into cloud scaling, monitoring and configuration management.

6.1 Contributions

In this dissertation we proposed IMP, an integrated configuration management framework. The contributions of IMP and this dissertation are:

1. **Integrated configuration management** IMP manages an entire infrastructure from an integrated, yet modular and reusable, configuration model. The model represents the desired state of the configuration of the distributed system and its execution environment. IMP can deploy configuration changes to enforce this desired state, including deployment dependencies between managed resources that cross machine boundaries.

2. **Model refinement** An integrated configuration model should describe a configuration as a function of high level artifacts, which often are the artifacts in which the architecture of an application is expressed. The configuration of these same artifacts is often enforced as a low abstraction level configuration artifact, such as configuration files and system level services. IMP offers the refinement technology to gradually refine a configuration description from a high abstraction level to the level of abstraction of the managed resources.

3. **Real world deployment** IMP can already manage realistic and real world distributed systems. Additionally its framework approach allows system administrators to port their existing scripts to the framework.

4. **Principled approach** System administrators need to remove any duplicate configuration parameters from their configuration model. This requires the capture of all relevant relations between configuration artifacts and concepts in the distributed system. IMP offers the technology for a principled approach to configuration management that should result in a configuration that is always consistent, even at high rates of configuration updates.

6.2 Lessons learned and limitations

The case studies show that IMP can manage the entire configuration of an infrastructure with heterogeneous devices, at any level of abstraction from a single integrated configuration model. As a consequence, the integrated model enables the configuration of supporting services such as monitoring, firewall or DNS servers, at virtually no additional cost. Because they require many configuration parameters that have to be fully consist with the services they support. For example, a connection between a client and a server cam be used by the firewall to generate its rules, or the configuration of a service that can also be used to derive the configuration to monitor the service.

My experience in system administration, both manual and with configuration management tools, was both valuable and counterproductive. From the initial prototype IMP has been used in many case studies and configuration management problems, of which a subset in presented in this dissertation. This resulted in early feedback. However, the experience with existing tools resulted in a configuration model which was designed to be reusable but in hindsight was not due to many implicit interfaces. Explicit interfaces, such as the host the configuration belongs to, is common in existing tools. The support for these implicit interfaces was part of the language and has been partially removed in the current prototype. This is an indication software engineering techniques might be useful to develop large configuration models. In the future work section we further elaborate on how to solve some of the limitations of IMP.

The initial focus of my PhD was on the deployment of configuration updates (the work concerning authorisation and workflow was conducted in this light [111]). It became clear that to improve the deployment of configuration updates on a complete infrastructure of a distributed system (handle removed resources, ensure a consistent state of the infrastructure, handle application data and transactional deployment), also requires dependencies between resources and not only their desired state. The integrated configuration model of IMP improves upon the state-of-the-art by including relations between managed resources and including all layers of abstraction in a single integrated configuration model. However, it does not use this additional information in the deployment process. How this can be done is subject of ongoing and future work.

Finally, IMP describes the desired state in a configuration model that needs to be developed. The development of large code bases is often aided by integrated development environments that provide support to the developer. IMP's large integrated configuration could also benefit from features such as auto-completion of types and suggestions for configuration parameters based on the available types and refinements. Current monitoring information can also function as a source for auto-completion. This dissertation does not focus on monitoring the current state, but integration of monitoring and configuration management could in many cases close the control loop to automate many scaling and allocation problems.

6.3 Future work

This dissertation focussed on creating a complete and integrated configuration model of complex distributed systems. A complete system configuration tool requires more than only a configuration model. In this section we give an overview of future work and research directions that can further build upon this work.

6.3.1 Deployment

Our research focussed on creating an integrated configuration model and a deployable configuration from this. There are many challenges left for deploying configuration updates:

Enhancing support for interdependencies between resources Configuration management tools can only handle deployment dependencies between resources within one managed system (e.g. server, switch, etc.). In distributed system there are also many dependencies between resources over machine boundaries. The current state-of-the-art relies on the process of convergence, eventually the correct desired state will be achieved. On each deployment the current state advances towards the desired state. This process works relatively well but requires the rate of configuration changes to be low enough. The rise of cloud computing and their pay per use model (billing per minute of used computation resources) increases the rate of change significantly. An integrated configuration model should contain relations that result in deployment dependencies. These dependencies should be used during deployment and will result in a deployment workflow that could for example serve as input for the deployment process of Eilam et. al [41].

Specifically for IMP, it is not yet clear how IMP can translate dependencies (relations) at a high abstraction level to deployment dependencies. For example, an application that connects to a database cluster implies that IMP should deploy changes to the database cluster before changes to the application. At deployment time this results in a dependency between the system service that starts the application and the system service that start the database cluster. IMP's integrated configuration model allows the deployment subsystem to reason about complex dependencies between managed resources. More research is required to translate all configuration model relations to deployment dependencies and use these dependencies to generate a total ordering on all configuration updates. This should result in a deployment subsystem that can deploy any complex configuration update in a single run without having to rely on the process of convergence. At the time of writing a master student is working in the context of his master thesis on a heuristics based approach that translates relations between instances to deployment dependencies.

Adding support for handling unmanaged resources Configuration management tools can deploy new managed resource or change the current state of a managed resource to the desired state. A desired state model can indicate that a resource should no longer exist, however they do not automatically remove resources that are no longer defined in the desired state model. A straightforward solution to determine which resource should be removed is to calculate the difference between the latest desired state model and the previous model and mark the removed resources as removed. Actually deploying a model that removes unmanaged resources is not as straightforward and poses many research challenges:

- What to do with dependencies between services?
- What to do with resources that contain other services? For example, virtual machines that are managed by a tool. The tool needs to ensure that nothing depends on resources on this virtual machine, before it is shut off.

- What to do with state that a resource has accumulated during its deployment? Can this state be discarded or does it need to be migrated?

Adding support for application state Desired state models assume that it can move the infrastructure from one state to another state. For example, storage systems are a class of applications that have state that needs to be accounted for. Many distributed storage systems can handle adding and removing nodes dynamically but the correct procedures needs to be carried out. For example, Cassandra in case 3 of Section 4.4 can remove nodes from its storage ring. Before the Cassandra service is stopped, it needs to be marked as unavailable so it can redistribute its data to other peers. Once this process it finished, the service can be removed safely without loosing any data. These procedures are very application specific and should be investigated to determine how this can be included in IMP's configuration model and deployment proces and deployment process.

6.3.2 Configuration model development

A complete and integrated configuration model requires a large code base that needs to be maintained and evolved over time. The current input DSL of IMP can create complex configuration models but has many quirks due to its organic growth over time:

- It allows implicit interfaces, which are counter-productive in stimulating configuration module reuse. These became apparent when case 3 needed to reuse modules from case 1. A solution to this problem is to make the language stricter and only allow binding variables that are part of the interface of an entity.
- The language is limited in specifying deployment dependencies in the configuration model. Although this is mainly a deployment issue we suspect that further deployment research will result in additional language constructs to specify how relations are handled during refinement.
- An instance of an entity has to be uniquely defined at one place. This complicates the creation of modular configuration models, because multiple modules can require a specific package or a service to be available on machine.
- The generation time should be improved to manage very large infrastructures. A possible improvement could result from studying IMP's input language and its compilation from a programming language perspective. For example, making the language more strict could enable the compiler to evaluate statements more efficiently.

IMP raises the abstraction level in which a configuration model can be expressed. Additionally IMP users develop configuration models in a modular fashion and IMP allows other stakeholders than system administrators to collaborate on the development of a configuration model. In section 1.2 we gave our vision on how all stakeholders in the configuration and deployment process interact. This is a first attempt at defining such a process which has not been validated. More research is required to see how the use of an integrated configuration model can integrate in the application life-cycle.

When the development of an application and the use of an integrated configuration model to manage an entire distributed system is fully integrated in the application life-cycle, access to this model and authorising updates to this model becomes a point of attention. We already laid the groundwork to add fine-grained access control to configuration models [111–113] with a source based input. However, further research is required to apply this to a more expressive and modular modeling language such as the one used by IMP.

The framework approach of IMP provides the first steps for a principled approach to configuration management. A system administrator (and other stakeholders in the configuration process) can structure the configuration in an integrated configuration model and port existing ad-hoc scripts to execute within the control of the IMP framework. For a complete principled approach to configuration management more research is required in the development process of such an integrated configuration model.

6.4 Concluding thoughts

This dissertation follows up on a tradition of research into the operational side of the software life cycle within our research group, of which configuration management side started as a research track almost 10 years ago, long before Cloud computing created the current sense of urgency. This dissertation shows that an integrated approach to configuration management is required to increase the efficiency of configuration management and reduce the amount of failures caused be configuration errors. It also shows that high levels of automation in configuration management can be achieved without the use of constraint satisfaction techniques. Additionally, the integrated model of IMP can include runtime dependencies between services. This lays the groundwork for the enforcement of configuration updates to take into account application state and state transitions during deployment. A requirement for fully automating systems management.

Appendix A

Language reference

This chapter is a reference for the IMP DSL. The IMP language is a declarative language to model the configuration of an infrastructure. The evaluation order of statements in the IMP modeling language is determined by their dependencies on other statements and not based on the lexical order. The correct evaluation order is determined by the language runtime.

A.1 Literal values and variables

This section first introduces the basics of the DSL: literal values. The values can be of the type `string`, `number` or `bool`. IMP also provides lists of values. When a value is assigned to a variable, this variable becomes read-only. Because variables can only be assigned once, it is not necessary to declare the type of the variable. The runtime is dynamically typed.

Listing A.1: Assigning literal values to variables

```
var1 = 1 # assign an integer, var1 contains now a number
var2 = 3.14 # assign a float, var2 also contains a number
var3 = ”This␣is␣a␣string” # var3 contains a string
# var 4 and 5 are both booleans
var4 = true
var5 = false
# var6 is a list of values
var6 = [”fedora”, ”ubuntu”, ”rhel”]
# var 7 is a label for the same value as var 2
var7 = var2
# next assignment will return an error because var1 is read—only after it was
# assigned the value 1
var1 = ”test”
```

A.2 Constraining literal types

Literal values are often values of configuration parameters that end up directly in configuration files or after transformations such as templates. These parameters often have particular formats or only a small range of valid values. Examples of such values are tcp port numbers or a MAC address of an Ethernet interface.

A typedef statement creates a new literal type which is based on one of the basic types with an additional constraint. A typedef statement starts with the `typedef` keyword, followed by a name that identifies the type. This name should start with a lowercase character and is followed by uppercase and lowercase characters, numbers, a dash and an underscore. After name an expression follows which is started by the `matching` keyword. The expression is either an IMP expression or a regular expression. A regular expression is demarcated with slashes.

IMP expressions can use logical operators such as greater than, smaller than, equality and inclusions together with logical operators. The keyword `self` refers to the value that is assigned to a variable of the constrained type.

Listing A.2: Constraining types as validation constraints

```
typedef tcp_port as number matching self > 0 and self < 65565
typedef mac_addr as string matching /([0−9a—fA—F]{2})(:[0−9a—fA—F]{2}){5}$/
```

A.3 Enumerations

Enumerations are a special case of constrained literal types. An enumeration provides a specific list of value values for a type. It also organizes the possible values in a tree where the parent-child relation indicates an is-a relationship. This relationship can be tested in expression using the `is` operator.

The enumeration is defined in the module where the root element of the value tree is defined. Other values can be added to the tree from other modules. A statement that adds values under an other value starts with the `enum` keyword followed by the name of the enumeration type. Next are the keywords `with parent` to define the value to add valid values under. In case of the root node the parent value is the keyword `root`. After the parent value the keyword `as` is added followed by the list of valid values. Values are literal values such as strings.

Listing A.3: An enumeration to define an operating system taxonomy

```
# define an enumeration of operating systems
enum os with parent root as ”unix”, ”windows”
enum os with parent ”unix” as ”linux”, ”solaris”, ”freebsd”, ”openbsd”,
     ↪ ”macos”
```

A.4 Transformations: string interpolation, templates and plug-ins

At the lowest level of abstraction the configuration of an infrastructure often consists of configuration files or attributes that are set to certain values. These configuration files and attribute values are a transformation of one or more parameters that are available in the configuration model. In IMP there are three mechanism available to perform such transformation: string interpolation, templates and plugins. In the next subsection each of these mechanisms are explained.

A.4.1 String interpolation

The easiest transformation but also the least powerful is string interpolation. It enables the developer to include variables as parameters inside a string. The included variables are dynamically looked up at the location where the string they are included in is instantiated. This is important to note for later in this chapter when the language constructs are introduced that provide encapsulation.

Listing A.4: Interpolating strings

```
hostname = ”wwwserv1.example.org”
motd = ”””Welcome␣to␣{{{hostname␣}}}\n”””
```

A.4.2 Templates

IMP has a built-in template engine that has been tightly integrated into the platform. IMP integrated the Jinja2 template engine [93]. A template is evaluated in the location and scope where the `template` function is called. This function accepts as an argument the location of the template. A template is identified with a path: the first item of the path is the module that contains the template and the remainder of the path is the path within the template directory of the module.

The integrated Jinja2 engine is limited to the entire Jinja feature set, except for subtemplates which are not supported. Additionally a small change to the Jinja2 syntax has been made to support the double colon to seperate namespaces (`::`) in Jinja templates so fully qualified IMP variables names can be used inside Jinja. During execution Jinja2 has access to all variables and plug-ins that are available in the scope where the template is evaluated. The result of the template is returned by the template function.

Listing A.5: Using a template to transform variables to a configuration file

```
hostname = ”wwwserv1.example.com”
admin = ”joe@example.com”
motd_content = template(”motd/message.tmpl”)
```

Listing A.6: The template used in the previous listing

```
Welcome to {{ hostname }}
This machine is maintainted by {{ admin }}
```

A.4.3 Transformation plug-ins

Transformation plug-ins provide an interface to define a transformation in Python. Plugins are exposed in the IMP language as function calls, such as the template function call. A template accepts parameters and returns a value that it computed out of the variables.

IMP has a list of built-in plug-ins that are accessible without a namespace. Each module that is included can also provide plug-ins. These plug-ins are accessible within the namespace of the module. Each of the IMP native plug-ins and the plug-ins provided by modules are also registered as filters in the Jinja2 template engine. Additionally plug-ins can also be called from within expressions such as those used for constraining literal types. The validation expression will in that case be reduced to a transformation of the value that needs to be validated to a boolean value.

A.5 Entities

The types that a system administrator uses to model concepts from the configuration are entities. Entities are defined with the keyword `entity` followed by a name that starts with an uppercase character. The other characters of the name may contains upper and lower case characters, numbers, a dash and an underscore. With a colon the body of the definition of an entity is started. In this body the attributes of the entity are defined. The body ends with the keyword `end`.

Entity attributes are used to add properties to an entity that are represented by literal values. Properties of entities that represent a relation to an instance of an entity should be represented using relations which are explained further on. On each line of the body of an entity definition a literal attribute can be defined. The definition consists of the literal type, which is either `string` , `number` or `bool` and the name of the attribute. Optionally a default value can be added.

Entities can inherit from multiple other entities, thus multiple inheritance. Inheritance implies that an entity inherits attributes and relations from parent entities. Inheritance also introduces a is-a relationship. It is however not possible to override or rename attributes. Entities that do not explicitly inherit from an other entity inherit from `std::Entity`

Instances of an entity are created with a constructor statement. A constructor statement consists of the name of the entity followed by parenthesis. Optionally between these parenthesis attributes can be set. Attributes can also be set in separate statements. Once an attribute is set, it becomes read-only.

In a configuration often default values for parameters are used because only in specific case an other values is required. Attributes are read-only once they are set, so in the definition of an entity default values for attributes can be provided. In the cases where multiple default values are used a default constructor can be defined using the `typedef` keyword, followed by the name of the constructor and the keyword `as`, again followed by the constructor with the default values set. Both mechanisms have the same semantics. The default value is used for an attribute when an instance of an entity is created and no value is provided in the constructor for the attributes with default values.

Listing A.7: Defining entities in a configuration model

```
entity File:
    string path
    string content
    number mode = 640
end

motd_file = File(path = "/etc/motd")
motd_file.content = "Hello␣world\n"

entity ConfigFile extends File:

end

typedef PublicFile as File(mode = 0644)
```

A.6 Relations

IMP makes from the relations between entities a first class language construct. Literal value properties are modeled as attributes, properties that have an other entity as type are modeled as a relation between those entities. Relations are defined by specifying each end of the relation together with the multiplicity of each relation end. Each end of the relation is named and is maintained as a double binding by the IMP runtime.

Listing A.8 shows the definition of a relation. Relations do not start with a specific keyword such as most other statements. Each side of a relation is defined an each side of the – keyword. Each side is the definition of the property of the entity on the other side. Such a definition consists of the name of the entity, the name of the property and a multiplicity which is listed between square brackets. This multiplicity is either a single integer value or a range which is separated by a colon. If the upper bound is infinite the value is left out. Relation multiplicities are enforced by the runtime. If they are violated a compilation error is issued.

Relations also add properties to entities. Relation can be set in the constructor or using a specific set statement. Properties of a relations with a multiplicity higher than one, can hold multiple values. These properties are implemented as a list. When a value is assigned to a property that is a list, this value is added to the list. When this value is also a list the items in the list are added to the property. This behavior is caused by the fact that variables and properties are read-only and in the case of a list, append only.

Listing A.8: Defining relations between entities in the domain model

```
# Each config file belongs to one service.
# Each service can have one or more config files
ConfigFile configfile [1:] -- [1] Service service

cf = ConfigFile()
service = Service()

cf.service = service
```

A.7 Refinements

Entities define a domain model that is used to express a configuration in. For each entity one or more refinements can be defined. When an instance of an entity is constructed, the runtime searches for refinements. Refinements are defined within the body of an `implementation` statement. After the implementation keyword the name of the refinement follows. The name should start with a lowercase character. A refinement is closed with the `end` keyword.

In the body of an refinement statements are defined. This can be all statements except for statements that define types and refinements such as entities, refinements and relations.

An implement statement connects refinements with entities. As such the entity is used as an interface to one or more refinements that encapsulate implementation details. An refine statement starts with the `implements` keyword followed by the name of the entity that it defines a refinement for. Next the keyword `using` follows after which refinements are listed, separated by commas. Such a statement defines refinements for instances of an entity when no more specific refinements have been defined. In an implement statement after the refinements list the `when` keyword is followed by an expression that defines when this refinement needs to be chosen.

In some cases each instance of an entity requires an other refinement. For these cases anonymous refinements are available. Directly after the constructor that instantiates an entity, a refinement body follows that defines the refinements for this specific instance of an entity. This construction does not provide the ability to provide multiple refinements like the implement statement does. Instead it is possible to use the `include` keyword followed by the name of the refinement that needs to be included.

Listing A.9: Refinements for an entity

```
# Defining refinements and connecting them to entities
implementation file1 for File:
end

implement File using file1

host_a = std::Host(name = "hosta.example.com"):
    file_a = std::File(path = "/etc/motd", content =
        ↪ template("hosts/motd.tmpl"))
end
```

A.8 Indexes and queries

One of the key features of IMP is modeling relations in a configuration. To help maintaining these relations the language provides a query function to lookup the other end of relations. This query function can be used to lookup instances of an entity. A query is always expressed in function of the properties of an entity. The properties that can be used in a query have to have an index defined over them.

An index is defined with a statement that starts with the `index` keyword, followed by the entity thats to be indexed. Next, between parenthesis a list of properties that belong to that index is listed. Every combination of properties in an index should always be unique.

A query on a type is performed by specifying the entity type and between square brackets the query on an index. A query should always specify values for all properties in an index, so only one value will be returned.

Listing A.10: Define an index over attributes

```
entity File:
    string path
    string content
end

index File(path)

# search for a file
file_1 = File[path = ”/etc/motd”]
```

Appendix B

IMP tutorial

This appendix gives a short introduction to IMP. It provides the basics to get started with IMP:

- Install IMP
- Create an IMP project
- Use existing configuration modules
- Create a configuration model to deploy a LAMP (Linux, Apache, Mysql and PHP) stack
- Deploy the configuration
- Deploy a configuration to multiple hosts
- Define a new configuration module
- Add new types (entities and refinements) to a module

The IMP framework exists of several components:

- A stateless compiler that builds the configuration model
- The central IMP server that stores states
- IMP agents on each managed system that deploy configuration changes.

In this appendix we install the framework on a single machine and use that machine as the management server that configures itself and other machines.

Note: *Currently only Fedora is supported, both in the deployment agent and the configuration modules. Because IMP is a tool to manage systems and their configuration, we recommend to start with IMP on a virtual machine to ensure that your current machine keeps working as expected. This guide has been tested on a Fedora 18 virtual machine.*

Warning: *DO NOT run this guide on your own machine, or it will be reconfigured. Use a fedora 18 VM, with hostname vm1 to be fully compatible with this guide.*

B.1 Installing IMP

The source of IMP is available on GitHub at https://github.com/bartv/imp On Fedora run the following commands to install dependencies, the python3 runtime and install IMP from its source repository.

```
yum install -y python3-setuptools python3-amqplib python3-tornado \
        python3-dateutil python3-plyvel python3-execnet git
git clone https://github.com/bartv/imp
cd imp
python3 setup.py install
```

B.2 Create an IMP project

An IMP project bundles modules that contain configuration information. A project is nothing more than a directory with an .imp file, which contains parameters such as the location to search for modules and where to find the management server.

We create a directory `quickstart` with a basic configuration file:

```
mkdir quickstart
cd quickstart
cat .imp <<EOF
[config]
lib-dir = libs
export =
EOF
mkdir libs
touch main.cf
```

The configuration file defines that re-usable modules are stored in `libs`. The empty export options indicate that only the built-in deployment subsystem should be used. The IMP compiler looks for a file called `main.cf` to start the compilation from. The last line, creates an empty main.cf. In the next section we will re-use existing modules to deploy our LAMP stack.

B.3 Re-use existing modules

On GitHub many modules are already hosted that provide a types that represent configuration concepts used in real configurations and often refinements for one or more operating systems. Our modules are available in the https://github.com/bartv/imp-* repositories.

In the previous section we configured the project to use the imp-agent for deployment. This agent knows how to deploy and configure File, Directories, Services and Packages. The `std` module defines these concepts and therefore it is the first package we need.

```
cd libs
git clone https://github.com/bartv/imp—std.git std
```

For the LAMP stack we need modules that provide basic functionality such as networking (`net` and `ip`) and support for redhat based operating systems (`redhat`). Addionally also the modules that configure the Apache webserver and a Mysql server (`apache`, `httpd`, and `mysql`). And finally a module to configure PHP and Drupal (`php` and `drupal`).

```
for mod in net ip redhat httpd mysql apache php drupal; do
    git clone https://github.com/bartv/imp—$mod.git $mod
done
```

We now have all configuration modules required to deploy Drupal on a LAMP stack.

B.4 The configuration model

In this section we will use the configuration concepts defined in the existing modules to create new composition that defines the initial configuration model. In this guide we assume that drupal will be installed on a server called `vm1`.

B.4.1 Compose a configuration model

The modules we installed in the previous section encapsulate the configuration required for certain services or subsystems. In this section we make a composition of the configuration modules to deploy and configure a Drupal website. This composition is the initial configuration model and is stored in the main.cf file.

```
# define the machine we want to deploy Drupal on
vm1 = ip::Host(name = ”vm1”, os = ”fedora—18”, ip = ”172.16.1.3”)

# add a mysql and apache http server
web_server = httpd::Server(host = vm1)
mysql_server = mysql::Server(host = vm1)
```

```
7
8   # define a new virtual host to deploy drupal in
9   vhost_name = httpd::VhostName(name = ”localhost”)
10  vhost = httpd::Vhost(webserver = web_server, name = vhost_name,
11      document_root = ”/var/www/html/drupal_test”)
12
13  # deploy drupal in that virtual host
14  drupal::Common(host = vm1)
15  db = mysql::Database(server = mysql_server, name = ”drupal_test”,
16      user = ”drupal_test”, password = ”Str0ng—P433w0rd”)
17  drupal::Site(vhost = vhost, database = db)
```

On line 2 we define the server on which we want to deploy Drupal. The name is the hostname of the machine, which is later used to determine what configuration needs to be deployed on which machine. The os attribute defines what operating system this server runs. This attribute can be used to create configuration modules that handle the heterogienity of different operating systems. The ip attribute is the ipaddress of this host. In this introduction we define the IP attribute manually, however IMP can manage this automatically, as seen in case 3 in Section 4.4.

Lines 5 and 6 deploy an httpd server and mysql server on our server.

Lines 9 to 11 define a virtual host in which we want to deploy our Drupal website.

Line 14 deploys common Drupal configuration on our server and line 17 creates a Drupal site on the virtual host we defined previously.

Line 16 defines a database for our Drupal website.

B.4.2 Deploy the configuration model

The normal mode of operation of IMP is in a setting where each managed host runs a configuration agent that is receives configuration updates from a central server. This setup is quite elaborate and in this introduction we will use the single shot *deploy* command. This command compiles, exports and enforces the configuration of the server it is executed on.

The configuration mode we made in the previous section can be deployed by executing the deploy command in the IMP project.

```
imp deploy
```

IMP will compile the configuration model, generate configuration artifacts and enforce their desired state on the current machine. IN this case the virtual machine vm1.

B.4.3 Making it work

In a default fedora SELinux and the firewall are configured. This may cause problems because managing these services is not covered here. We recommend that you either set SELinux to permissive mode and disable the firewall with:

```
setenforce 0
sed —i ”s/SELINUX=enforcing/SELINUX=permissive/g” /etc/sysconfig/selinux
systemctl stop firewalld
```

Or allow apache to connect to the network and open up port 80 in the firewall.

```
setsebool httpd_can_network_connect true
firewall–cmd ——permanent ——zone=public ——add—service=http
```

B.4.4 Accessing your new Drupal install

Use ssh port-forwarding to forward port 80 on vm1 to your local machine, to port 2080 for example (`ssh -L 2080:localhost:80 ec2-user@172.16.1.3`). This allows you to surf to http://localhost:2080/

Warning: Using localhost in the url is essential because the configuration model generates a named based virtual host that matches the name *localhost*.

On the first access the database will not have been initialised. Surf to http://localhost:2080/install.php to start the initialisation process. The database connections has already been configured and Drupal should skip to the point where you can configure details such as the admin user.

Note: Windows users can use putty for ssh access to their servers. Putty also allows port forwarding[1].

B.5 Managing multiple machines

The real power of IMP appears when you want to manage more than one machine. In this section we will move the mysql server from vm1 to a second virtual machine called vm2. We will still manage this additional machine in single shot mode using a remote deploy.

---

[1] http://the.earth.li/ sgtatham/putty/0.63/htmldoc/Chapter3.html#using-port-forwarding

B.5.1 SSH setup

The remote deploy feature uses an ssh connection to the target host to deploy changes to that host. Additionally the remote deploy command requires the name used in the configuration model and not the ip which you would use to connect to the host (in the absence of a configured DNS server).

```
1  ssh−keygen −t rsa
2  cat /root/.ssh/id−rsa.pub
3  ssh−rsa AAAAB3NzaC1yc... wtIyJa1BFd9t8wYlT3/J+uSzAfifN+sjPL root@vm1
```

First as root generate a new ssh key on vm1 with ssh-keygen (on line 1). This command will ask you where to store the key and whether you want to set a passphrase on this key. Use the default location and do not set a password (For this simple demo in throw away virtual machines this increased security is not required). Output this key to the terminal with the command on line 2 and copy its output. The output will be a base64 encoded string that starts with ssh-rsa and ends with something like what is shown on line 3.

```
1  echo ”ssh−rsa␣AAAAB3NzaC1yc... yJa1BFd9t8wYlT3/J+uSzAfifN+sjPL␣root@vm1” >>
       ↪ /root/.ssh/authorized_keys
```

Append the public key to /root/.ssh/authorized_keys on vm2. You can achieve this by pasting the copied key content between quotes and appending it to that file, as shown on line 1.

B.5.2 vm2 preparation

On vm2 we also need to open up the firewall for the services that it will host. On this virtual machine this is port 3306/tcp for mysql.

```
firewall−cmd −−permanent −−zone=public −−add−port=3306/tcp
```

B.5.3 Update the configuration model

A second virtual machine is easily added to the system by adding the definition of the virtual machine to the configuration model and assigning the mysql server to the new virtual machine.

```
# define the machine we want to deploy Drupal on
vm1 = ip::Host(name = ”vm1”, os = ”fedora−18”, ip = ”172.16.1.3”)
vm2 = ip::Host(name = ”vm2”, os = ”fedora−18”, ip = ”172.16.1.4”)

# add a mysql and apache http server
web_server = httpd::Server(host = vm1)
mysql_server = mysql::Server(host = vm2)
```

```
8
9  # define a new virtual host to deploy drupal in
10 vhost_name = httpd::VhostName(name = ”localhost”)
11 vhost = httpd::Vhost(webserver = web_server, name = vhost_name,
12         document_root = ”/var/www/html/drupal_test”)
13
14 # deploy drupal in that virtual host
15 drupal::Common(host = vm1)
16 db = mysql::Database(server = mysql_server, name = ”drupal_test”,
17         user = ”drupal_test”, password = ”Str0ng—P433w0rd”)
18 drupal::Site(vhost = vhost, database = db)
```

On line 3 the definition of the new virtual machine is added. On line 7 the mysql server is assigned to vm2. No other changes are required. IMP uses the relation between the drupal site and the mysql server to configuration the database connection.

B.5.4 Deploy the configuration model

Deploy the new configuration model by invoking a local deploy on vm1 and a remote deploy on vm2. Because the vm2 name that is used in the configuration model does not resolve to an IP address we provide this address directly with the -i parameter.

```
imp deploy
imp deploy —r vm2 —i 172.16.1.4
```

B.6 Create your own modules

IMP enables developers of a configuration model to make it modular and reusable. In this section we create a configuration module that defines how to deploy a LAMP stack with a Drupal site in a two or three tiered deployment.

B.6.1 Module layout

A configuration module requires a specific layout (Figure B.1):

- The name of the module is determined by the top-level directory. In this directory the only required directory is the `model` directory with a file called _init.cf.
- What is defined in the _init.cf file is available in the namespace linked with the name of the module. Other files in the model directory create subnamespaces.
- The files directory contains files that are deployed verbatim to managed machines

- The templates directory contains templates that use parameters from the configuration model to generate configuration files.
- Python files in the plugins directory are loaded by the framework and can extend it using the IMP API.

```
module
|__ files
|    |__ file1.txt
|
|__ model
|    |__ _init.cf
|    |__ services.cf
|
|__ plugins
|    |__ functions.py
|
|__ templates
     |__ conf_file.conf.tmpl
```

Figure B.1: The layout of an IMP configuration module.

We will create our custom module in the `libs` directory of the quickstart project. Our new module will call `lamp` and only the _init.cf file is really required. The following commands create all directories to develop a full-featured module.

```
cd /root/quickstart/libs
mkdir {lamp,lamp/model}
touch lamp/model/_init.cf

mkdir {lamp/files,lamp/templates}
mkdir lamp/plugins
```

B.6.2 Configuration model

In lamp/model/_init.cf we define the configuration model that defines the lamp configuration module.

```
entity DrupalStack:
    string stack_id
    string vhostname
end

index DrupalStack(stack_id)

ip::Host webserver [1] -- [0:1] DrupalStack drupal_stack_webserver
ip::Host mysqlserver [1] -- [0:1] DrupalStack drupal_stack_mysqlserver

implementation drupalStackImplementation:
    # add a mysql and apache http server
```

```
13      web = httpd::Server(host = webserver)
14      mysql = mysql::Server(host = mysqlserver)
15
16      # define a new virtual host to deploy drupal in
17      vhost_name = httpd::VhostName(name = vhostname)
18      vhost = httpd::Vhost(webserver = web, name = vhost_name,
19              document_root = ”/var/www/html/{{␣stack_id␣}}”)
20
21      # deploy drupal in that virtual host
22      drupal::Common(host = webserver)
23      db = mysql::Database(server = mysql, name = stack_id,
24              user = stack_id, password = ”Str0ng—P433w0rd”)
25      drupal::Site(vhost = vhost, database = db)
26  end
27
28  implement DrupalStack using drupalStackImplementation
```

On line 1 to 4 we define an entity which is the definition of a concept in the configuration model. Entities behave as an interface to a refinement of the configuration model that encapsulates parts of the configuration, in this case how to configure a LAMP stack. On line 2 and 3 typed attributes are defined which we can later on use in the implementation of an entity instance.

Line 6 defines that stack_id is an identifying attribute for instances of the DrupalStack entity. This also means that all instances of DrupalStack need to have a unique stack_id attribute.

On lines 8 and 9 we define a relation between a Host and our DrupalStack entity. This relation represents a double binding between these instances and it has a multiplicity. The first relations reads as following:

- Each DrupalStack instance has exactly one ip::Host instance that is stored in the webserver attribute.
- Each ip::Host has zero or one DrupalStack instances that use the host as a webserver. The DrupalStack instance is stored in the drupal_stack_webserver attribute.

Warning: *On line 8 and 9 we explicity give the DrupalStack side of the relation a multiplicity that starts from zero. Setting this to one would break the ip module because each Host would require an instance of DrupalStack.*

On line 11 to 26 an implementation is defined that provides a refinement of the DrupalStack entity. It encapsulates the configuration of a LAMP stack behind the interface of the entity by defining DrupalStack in function of other entities, which on their turn do the same. The refinement process is evaluated by the compiler and continues until all instances are refined into instances of entities that IMP knows how to deploy.

Inside the implementation the attributes and relations of the entity are available as variables. They can be hidden by new variable definitions, but are also accessible through the `self` variable (not used in this example). On line 19 an attribute is used in an inline template with the `{{ }}` syntax.

And finally on line 28 we link the implementation to the entity itself.

B.6.3 The composition

With our new LAMP module we can reduce the amount of required configuration code in the main.cf file by using more `reusable` configure code. Only three lines of site specific configuration code are left in the initial configuration model:

```
# define the machine we want to deploy Drupal on
vm1 = ip::Host(name = ”vm1”, os = ”fedora—18”, ip = ”172.16.1.3”)
vm2 = ip::Host(name = ”vm2”, os = ”fedora—18”, ip = ”172.16.1.4”)

lamp::DrupalStack(webserver = vm1, mysqlserver = vm2,
    stack_id = ”drupal_test”, vhostname = ”localhost”)
```

B.6.4 Deploy the changes

Deploy the changes as before and nothing should change because it generates exactly the same configuration.

```
imp deploy
imp deploy —r vm2 —i 172.16.1.4”
```

B.6.5 Deploy a file

Until know we only used high level concepts in the new configuration module. In this section we will add an additional implementation (that is also always selected) and installs a customized message of the day file on both virtual machines.

```
implementation stackMotd:
    std::File(host = webserver, path = ”/etc/motd”, owner = ”root”,
            group = ”root”, group = ”root”, mode = 644,
            content = template(”lamp/motd.tmpl”))

    std::File(host = mysqlserver, path = ”/etc/motd”, owner = ”root”,
            group = ”root”, group = ”root”, mode = 644,
            content = template(”lamp/motd.tmpl”))
end

implement DrupalStack using stackMotd
```

B.7 Conclusion

This tutorial gave an introduction to managing a services on multiple servers. It demonstrated how the use of encapsulation and relations between instances can create very powerful abstractions. These abstraction ensure that the complexity of managing a distributed system is reduced in the same way that middleware platforms have done for building distributed applications.

Appendix C

Overview of configuration management tool properties

This appendix provides a concise overview of the properties of each of the tools evaluated in chapter 2. The properties are listed in eight tables: two tables for each of the four categories of properties and for each of the 11 tools.

Property	Cfengine3	Puppet	Bladelogic	Chef	Netomata
Specification paradigm					
Language type	Declarative	Declarative, Imperative	Imperative	Imperative	Declarative, Imperative
User interface	Command line	GUI, Command line	GUI, Command line	GUI, Command line	Command line
Abstraction mechanisms	Configuration files, Implementation dependent instances, Instance configurations	Configuration files, Implementation dependent instances	Configuration files, Implementation dependent instances	Imple- mentation dependent instances	Configuration files
Modularization mechanisms					
Type of grouping	Static grouping, Query based groups, Hierarchical groups	Static grouping, Query based groups, Hierarchical groups	Static grouping, Query based groups, Hierarchical groups	Static grouping, Query based groups	Static grouping
Configuration modules	yes	yes	yes	yes	no
Modelling of relations					
Arity	many-to-many, one-to-many, one-to-one	one-to-many, one-to-one	one-to-one	many-to-many	one-to-one
Constraints	generative constraints				
Granularity	Instance relations, Parameter - instance relations, Parameter relations	Instance relations	Instance relations	Instance relations	Instance relations

Table C.1: Specification properties

Property	MS SCCM	CA NSM	Bcfg2	LCFG	HP SA	IBM Tivoli
Specification paradigm						
Language type	Imperative	Declarative, Imperative	Declarative	Declarative	Imperative	Declarative
User interface	GUI	GUI	Command line	Command line	GUI	GUI, Command line
Abstraction mechanisms	Configuration files	Configuration files, Implementation dependent instances, Instance configurations	Configuration files	Implementation dependent instances	Configuration files	Implementation dependent instance
Modularization mechanisms						
Type of grouping	Static grouping, Query based groups	Hierarchical groups	Static grouping, Hierarchical groups	Static grouping, Query based groups	Static grouping	Static grouping
Configuration modules	no	yes	no	yes	no	yes
Modelling of relations						
Arity				many-to-many, one-to-one		many-to-many, one-to-many, one-to-one
Constraints						generative constraints
Granularity				Instance relations, Parameter relations		Instance relations, Parameter-instance relations, Parameter relations

Table C.2: Specification properties (continued)

Property	Cfengine3	Puppet	Bladelogic	Chef	Netomata
Scalability	> 10000	1000 - 10000	> 10000	unknown	unknown
Configuration update workflow	Coordination of distributed changes	Coordination of distributed changes	Coordination of distributed changes		
Deployment architecture					
Translation agent	Strongly distributed management	Centralized management	Weakly distributed management	Centralized management	Centralized management
Distribution mechanism	Pull, Push	Pull, Push	Push	Pull, Push	
Platform support	*BSD, AIX, HP-UX, Linux, Mac OS X, Solaris, Windows	*BSD, AIX, HP-UX, Linux, Mac OS X, Solaris	AIX, HP-UX, Linux, Network equipment, Solaris, Windows	*BSD, Linux, Mac OS X, Solaris, Windows	Network equipment

Table C.3: Deployment properties

Property	MS SCCM	CA NSM	Bcfg2	LCFG	HP SA	IBM Tivoli
Scalability	unknown	unknown	< 1000	1000 - 10000	unknown	unknown
Configuration update workflow	Support for organisational policies	Support for organisational policies			Coordination of distributed changes	Coordination of distributed changes
Deployment architecture						
Translation agent	Weakly distributed management	Weakly distributed management	Centralized management	Centralized management	Centralized management	Centralized management
Distribution mechanism	Pull, Push	Push	Pull	Pull	Push	Push
Platform support	Windows	AIX, HP-UX, Linux, Mac OS X, Network equipment, Solaris, Windows	*BSD, AIX, Linux, Mac OS X, Solaris	Linux	AIX, HP-UX, Linux, Network equipment, Solaris, Windows	AIX, Linux, Solaris, Windows

Table C.4: Deployment properties (continued)

Property	Cfengine3	Puppet	Bladelogic	Chef	Netomata
Usability					
Ease of use	medium	medium	easy	hard	hard
Support for testing specifications	dry run mode	dry run mode, integration of staging and testing		integration of staging and testing	
Monitoring the infrastructure	integrated monitoring	interaction with external monitoring	integrated monitoring		interaction with external monitoring
Versioning support	external repository	external repository	external repository	external repository, integrated repository	external repository
Specification documentation	structured documentation	free form documentation, structured documentation	free form documentation	free form documentation, structured documentation	free form documentation
Integration with environment	runtime characteristics	external databases, runtime characteristics	external databases	external databases, runtime characteristics	
Conflict management	Modality conflicts	Modality conflicts			
Workflow enforcement	no	no	yes	no	no
Access control	Path based	Path based	Hierarchical	Path based	

Table C.5: Specification management properties

Property	MS SCCM	CA NSM	Bcfg2	LCFG	HP SA	IBM Tivoli
Usability						
Ease of use	easy	easy	hard	medium	easy	easy
Support for testing specifications			dry run mode	dry run mode		
Monitoring the infrastructure	integrated monitoring	integrated monitoring	integrated monitoring	integrated monitoring, interaction with external monitoring	integrated monitoring	integrated monitoring
Versioning support	integrated repository	integrated repository	external repository	external repository		external repository
Specification documentation			free form documentation	free form documentation	free form documentation	free form documentation
Integration with environment	external databases, runtime characteristics	runtime characteristics	runtime characteristics	runtime characteristics		runtime characteristics
Conflict management			Modality conflicts			
Workflow enforcement	no	no		no	no	no
Access control	Hierarchical	Hierarchical	Path based	Path based	Hierarchical	Hierarchical

Table C.6: Specification management properties (continued)

Property	Cfengine3	Puppet	Bladelogic	Chef	Netomata
Available documentation	Process documentation, Reference, Tutorial	Process documentation, Reference, Tutorial	Process documentation, Reference, Tutorial	Reference, Tutorial	Reference, Tutorial
Commercial support	Training, Company backed development, Support lines	Training, Company backed development, Support lines	Training, Company backed development, Support lines	Training, Company backed development	Company backed development
Community	5	5	1	4	1
Maturity	5	3	5	2	1

Table C.7: User support properties

Property	MS SCCM	CA NSM	Bcfg2	LCFG	HP SA	IBM Tivoli
Available documentation	Process documentation, Reference, Tutorial		Process documentation, Reference, Tutorial	Process documentation, Reference, Tutorial		Process documentation, Reference, Tutorial
Commercial support	Training, Company backed development, Support lines	Training, Company backed development, Support lines			Training, Company backed development, Support lines	Training, Company backed development, Support lines
Community	4	3	2	1		5
Maturity	3	5	4	5	5	4

Table C.8: User support properties (continued)

Bibliography

[1] Hiera 1: Overview. `http://docs.puppetlabs.com/hiera/1/`, 2014.

[2] OpenLMI: Configure, Manage and Monitor Linux Systems. `http://www.openlmi.org`, 2014.

[3] Web Services Management. `http://www.dmtf.org/standards/wsman`, 2014.

[4] Dakshi Agrawal, Seraphin Calo, Kang-won Lee, and Jorge Lobo. Issues in designing a policy language for distributed management of it infrastructures. *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, 2007.

[5] Amazon Inc. Aws elastic beanstalk. `http://aws.amazon.com/elasticbeanstalk`, 2013.

[6] Paul Anderson. *Short Topics in System Administration 14: System Configuration*. USENIX Sage, Berkeley, CA, 2006.

[7] Paul Anderson and Alva Couch. What is this thing called "system configuration"? LISA Invited Talk, November 2004.

[8] Paul Anderson and Alastair Scobie. Large scale linux configuration with lcfg. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, Georgia, USA, 10 2000.

[9] Paul Anderson and Edmund Smith. Configuration tools: Working together. In *Proceedings of the Large Installations Systems Administration (LISA) Conference*, Berkeley, CA, December 2005. Usenix Association.

[10] Apache Software Foundation. Apache http server project. http://httpd.apache.org, 2014.

[11] Mikhail J. Atallah and Douglas E. Comer. Algorithms for variable length subnet address assignment. *IEEE Transactions on Computers*, 47(6), 06 1998.

[12] Amazon AWS. Amazon elastic compute cloud: Api reference. `http://docs.aws.amazon.com/AWSEC2/latest/APIReference/Welcome.html`, 2014.

[13] Amazon AWS. Amazon route 53. `http://docs.aws.amazon.com/Route53/latest/APIReference/Welcome.html`, 2014.

[14] Rob Barrett, Paul P. Maglio, Eser Kandogan, and John Bailey. Usable autonomic computing systems: The system administrators' perspective. *Advanced Engineering Informatics*, 19(3), 2005. Autonomic Computing.

[15] Yair Bartal, Alain Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. *ACM Trans. Comput. Syst.*, 22(4), 2004.

[16] Frédéric Beck, Olivier Festor, Isabelle Chrisment, and Ralph E. Droms. Automated and secure IPv6 configuration in enterprise networks. In *International Conference on Network and Service Management (CNSM '10)*, 2010.

[17] Ashutosh Bhatia and Shubhranshu Singh. A Distributed Prefix Allocation Scheme for Subordinate MANET. In *IEEE Asia-Pacific Services Computing Conference (APSCC '08)*, 12 2008.

[18] Lance Bloom and Nancy Clark. It-management software deployment: field findings and design guidelines. In *CHiMiT '08: Proceedings of the 2nd ACM Symposium on Computer Human Interaction for Management of Information Technology*, New York, NY, USA, 2008. ACM, ACM.

[19] Mark Burgess. Cfengine: a site configuration engine. *USENIX Computing Systems*, 8(3), 1995.

[20] Mark Burgess. Computer Immunology. In *LISA '98: Proceedings of the 12th USENIX conference on System administration*, Berkeley, CA, USA, 1998. USENIX Association.

[21] Mark Burgess and Alva Couch. Modeling next generation configuration management tools. In *Proceedings of the 20th Large Installation System Administration (LISA) Conference*, Berkeley, CA, USA, 2006.

[22] Cfengine Inc. Configuration Management Software. `http://cfengine.com`, 2014.

[23] Marinos Charalambides, Paris Flegkas, George Pavlou, Arosha K. Bandara, Emil C. Lupu, Alessandra Russo, Naranker Dulay, Morris Sloman, and Javier Rubio-Loyola. Policy conflict analysis for quality of service management. In *POLICY '05: Proceedings of the Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05)*, Washington, DC, USA, 2005. IEEE Computer Society.

[24] Chef Inc. Chef. `http://www.getchef.com/chef/`, 2014.

[25] S. Childs, M. E. Poleggi, C. Loomis, L. F. M. Mejías, M. Jouvin, R. Starink, S. De Weirdt, and G. C. Meliá. Devolved Management of Distributed Infrastructures With Quattor. In *Proceedings of the 22nd Large Installation System Administration (LISA) Conference*, San Diego, California, 2008. USENIX Association.

[26] R. Coltun, D. Ferguson, J. Moy, and A. Lindem. OSPF for IPv6. RFC 5340 (Proposed Standard), July 2008.

[27] Ronni J. Colville and Donna Scott. Vendor Landscape: Server Provisioning and Configuration Management. Gartner Research, May 2008.

[28] Autonomic Computing et al. An architectural blueprint for autonomic computing. *IBM White Paper*, 2006.

[29] Nigel Cook, Dejan Milojicic, and Vanish Talwar. Cloud management. *Journal of Internet Services and Applications*, 3(1), 2012.

[30] Alva Couch, John Hart, G. Elizabeth Idhaw, and Dominic Kallas. Seeking closure in an open world: A behavioral agent approach to configuration management. In *Proceedings of the 17th Large Installations Systems Administration (LISA) conference*, Baltimore, MD, USA, 10/2003 2003. Usenix Association, Usenix Association.

[31] Thomas Delaet. *Improving Software and Data Management in Contemporary IT Systems (Het verbeteren van software- en gegevens-beheer in hedendaagse IT systemen)*. PhD thesis, IWT, December 2009.

[32] Thomas Delaet, Paul Anderson, and Wouter Joosen. Managing real-world system configurations with constraints. In *Seventh International Conference on Networking,*, April 2008.

[33] Thomas Delaet and Wouter Joosen. Podim: a language for high-level configuration management. In *Proceedings of the 21st Large Installation System Administration Conference (USENIX LISA 2007)*. Usenix Association, November 2007.

[34] Narayan Desai. Bcfg2: A pay as you go approach to configuration complexity. In *Australian Unix Users Group (AUUG2005)*, Sydney, Australia, 2005, 10/2005 2005.

[35] Narayan Desai, Rick Bradshaw, Scott Matott, Sandra Bittner, Susan Coghlan, Rémy Evard, Cory Lueninghoener, Ti Leggett, John-Paul Navarro, Gene Rackow, Craig Stacey, and Tisha Stacey. A case study in configuration management tool

deployment. In *Proceedings of the 19th Large Installation System Administration (LISA) Conference*, Berkeley, CA, USA, 2005. USENIX Association.

[36] DMTF. Common Information Model (CIM) Standards. http://www.dmtf.org/standards/cim/, 05 2010.

[37] R. Droms. Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard), March 1997. Updated by RFCs 3396, 4361, 5494.

[38] R. Droms, J. Bound, B. Volz, T. Lemon, C. Perkins, and M. Carney. Dynamic Host Configuration Protocol for IPv6 (DHCPv6). RFC 3315 (Proposed Standard), July 2003. Updated by RFCs 4361, 5494.

[39] Tudor Dumitraş and Priya Narasimhan. Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise system. In *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, New York, NY, USA, 2009. Springer-Verlag New York, Inc., Springer-Verlag New York, Inc.

[40] T. Eilam, M.H. Kalantar, A.V. Konstantinou, G. Pacifici, J. Pershing, and A. Agrawal. Managing the configuration complexity of distributed applications in Internet data centers. *IEEE Communications Magazine*, 44(3), March 2006.

[41] Tamar Eilam, Michael Elder, Alexander V Konstantinou, and Ed Snible. Pattern-based composite application deployment. *12th IFIPIEEE International Symposium on Integrated Network Management IM 2011 and Workshops*, 2011.

[42] Khalid Elbadawi and James Yu. Improving Network Services Configuration Management. In *Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN '11)*, 07 2011.

[43] EMC. Vipr software-defined storage. `http://www.emc.com/data-center-management/vipr/index.htm`, 2014.

[44] William Enck, Patrick McDaniel, Subhabrata Sen, Panagiotis Sebos, Sylke Spoerel, Albert Greenberg, Sanjay Rao, and William Aiello. Configuration management at massive scale: system design and experience. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2007. USENIX Association, USENIX Association.

[45] Nagios Enterprises. Nagios it monitoring. `http://www.nagios.org`, 2014.

[46] Remy Evard. An Analysis of UNIX System Configuration. In *LISA '97: Proceedings of the 11th USENIX conference on System administration*, Berkeley, CA, USA, 1997. USENIX Association.

[47] Gerhard Fleischanderl. Constraints Applied to Configurations. In *KI '01: Proceedings of the Joint German/Austrian Conference on AI*, London, UK, 2001. Springer-Verlag.

[48] USENIX Special Interest Group for Sysadmins. Lisa 2011 salary survey. 2011.

[49] Open Networking Foundation. Software-defined networking (sdn) definition. `https://www.opennetworking.org/sdn-resources/sdn-definition`, 2014.

[50] Openstack Foundation. Openstack api complete reference. `http://api.openstack.org/api-ref.html`, 2014.

[51] Zhi (Judy) Fu and S. Felix Wu. Automatic generation of ipsec/vpn security policies in an intra-domain environment. In *In Proceedings of the 12th International Workshop on Distributed System Operation & Management (DSOM)*, 2001.

[52] V. Fuller and T. Li. Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan. RFC 4632 (Best Current Practice), August 2006.

[53] Jean-Pierre Garbani and Peter O'Neill. The IT Management Software Megavendors. Forrester, August 2009.

[54] 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), January 2009.

[55] Google Inc. Google app engine. `https://developers.google.com/appengine`, 2013.

[56] Google Inc. Ushering in the next generation of computing at google i/o. `http://googlecloudplatform.blogspot.be/2013/05/ushering-in-next-generation-of.html`, 05 2013.

[57] Eben M. Haber and John Bailey. Design guidelines for system administration tools developed through ethnographic field studies. In *CHIMIT '07: Proceedings of the 2007 symposium on Computer human interaction for the management of information technology*, New York, NY, USA, 2007. ACM, ACM.

[58] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. RFC 4291 (Draft Standard), February 2006.

[59] T. Hinrich, N. Love, C. Petrie, L. Ramshaw, A. Sahai, and S. Singhal. Using Object-Oriented Constraint Satisfaction for Automated Configuration Generation. *LECTURE NOTES IN COMPUTER SCIENCE*, 2004.

[60] Dennis G. Hrebec and Michael Stiber. A survey of system administrator mental models and situation awareness. In *SIGCPR '01: Proceedings of the 2001 ACM SIGCPR conference on Computer personnel research*, New York, NY, USA, 2001. ACM, ACM.

[61] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. Testing idempotence for infrastructure as code. In *Middleware '13: Proceedings of the 14th ACM/IFIP/USENIX International Conference on Middleware*, Bejing, China, 2013. Springer-Verlag New York, Inc., Springer-Verlag New York, Inc.

[62] Cisco Inc. Cisco open network environment. `http://www.cisco.com/web/solutions/trends/open_network_environment/index.html`, 2014.

[63] The internet system consortium. Bind. `https://www.isc.org/downloads/bind/`, 2014.

[64] The internet system consortium. Isc dchp. `https://www.isc.org/downloads/dhcp/`, 2014.

[65] Christophe Jelger and Thomas Noel. Prefix Continuity and Global Address Autoconfiguration in IPv6 Ad Hoc Networks. In *Challenges in Ad Hoc Networking, OCP Science*, volume 2. 2006.

[66] Wouter Joosen, Bert Lagaisse, Eddy Truyen, and Koen Handekyn. Towards application driven security dashboards in future middleware. *Journal of Internet Services and Applications*, 3(1), 2012.

[67] Zeus Kerravala. As the value of enterprise networks escalates, so does the need for configuration management. *The Yankee Group*, January 2004.

[68] Ruoshan Kong, Jing Feng, and Huaibei Zhou. A Conservative Prefix Delegation Policy for Nested Mobile Networks Based on Paging Mechanism. In *International Conference on Computational Intelligence and Software Engineering (CISE '10)*, 12 2010.

[69] Vibhore Kumar, Brian F. Cooper, Greg Eisenhauer, and Karsten Schwan. imanage: policy-driven self-management for enterprise-scale systems. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, New York, NY, USA, 2007.

[70] Puppet Labs. Razor: Next-generation provisioning from puppet labs and emc. `http://puppetlabs.com/solutions/next-generation-provisioning`, 2014.

[71] Canonical Ltd. Juju: Automate your cloud infrastructure. `https://juju.ubuntu.com`, 2014.

[72] E. Lupu and M. Sloman. Conflict analysis for management policies. In *Proceedings of the Vth International Symposium on Integrated Network Management IM'97*. Chapman & Hall, May 1997.

[73] G. Malkin. RIP Version 2. RFC 2453 (Standard), November 1998. Updated by RFC 4822.

[74] G. Malkin and R. Minnear. RIPng for IPv6. RFC 2080 (Proposed Standard), January 1997.

[75] Jean-Philippe Martin-Flatin, Simon Znaty, and Jean-Pierre Hubaux. A survey of distributed enterprise network and systems management paradigms. *Journal of Network and Systems Management*, 7(1), 1999.

[76] Microsoft. Windows Azure. `http://www.windowsazure.com`, 2013.

[77] Microsoft. Windows azure: Announcing new dev/test offering, biztalk services, ssl support with web sites, ad improvements, per minute billing. http://weblogs.asp.net/scottgu/archive/2013/06/03/ windows-azure-announcing-new-dev-test-offering-biztalk-services-ssl-support-with-web-sites-ad-improvements-per-minute-billing.aspx, 06 2013.

[78] Jonathan D. Moffett. Requirements and policies. In *Proceedings of the Policy Workshop*, November 1999.

[79] J. Moy. OSPF Version 2. RFC 2328 (Standard), April 1998.

[80] Sanjai Narain. Network configuration management via model finding. In *Proceedings of the 19th conference on Large Installation System Administration Conference - Volume 19*, LISA '05, Berkeley, CA, USA, 2005. USENIX Association, USENIX Association.

[81] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative Infrastructure Configuration Synthesis and Debugging. *Journal of Network and Systems Management*, 16(3), 2008.

[82] NetApp. Netapp software-defined storage. `http://www.netapp.com/us/technology/software-defined-storage/`, 2014.

[83] Netfilter.org. The netfilter.org "iptables" project. `http://www.netfilter.org/projects/iptables/`, 2014.

[84] OASIS. Oasis topology and orchestration specification for cloud applications (tosca) tc. `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca`, 2014.

[85] D. Oppenheimer. The importance of understanding distributed system configuration. In *Proceedings of the 2003 Conference on Human Factors in Computer Systems workshop*, April 2003.

[86] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, Berkeley, CA, USA, 2003. USENIX Association, USENIX Association.

[87] David Oppenheimer and David A. Patterson. Studying and using failure data from large-scale internet services. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop*, New York, NY, USA, 2002. ACM, ACM.

[88] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1972.

[89] Terence Parr. Another tool for language recognition. `http://www.antlr.org`, 2014.

[90] Patrick Debois. Infrastructure as code - a comprehensive overview. `http://www.jedi.be/blog/2013/05/24/InfrastructureasCode/`, 2012.

[91] David A. Patterson. A simple way to estimate the cost of downtime. In *Proceedings of the 16th USENIX conference on System administration*, Berkeley, CA, USA, 11/2002 2002. USENIX Association, USENIX Association.

[92] Pivotal. Cloud Foundry. `http://www.cloudfoundry.com`, 2013.

[93] Pocoo. Jinja2 template engine. http://jinja.pocoo.org/docs/, 2014.

[94] The CentOS project. Centos. `http://www.centos.org/`, 2014.

[95] The Linux Virtual Server project. Linux virtual server. `http://www.linuxvirtualserver.org`, 2014.

[96] Puppet Labs Inc. Puppet Labs: IT Automation Software for System Administrators. `http://www.puppetlabs.com`, 2014.

[97] Lyle Ramshaw, Akhil Sahai, Jim Saxe, and Sharad Singhal. Cauldron: A Policy-Based Design Tool. In *POLICY '06: Proceedings of the Seventh IEEE International Workshop on Policies for Distributed Systems and Networks*, Washington, DC, USA, 2006. IEEE Computer Society.

[98] D. Raymer, J. Strassner, E. Lehtihet, and S van der Meer. End-to-end model driven policy based network management. In *Policies for Distributed Systems and Networks, 2006. Policy 2006. Seventh IEEE International Workshop*, 2006.

[99] Red Hat. OpenShift by Red Hat. `https://www.openshift.com`, 2013.

[100] Luis Rodero-Merino, Luis M. Vaquero, Victor Gil, Fermín Galán, Javier Fontán, Rubén S. Montero, and Ignacio M. Llorente. From infrastructure delivery to service management in clouds. *Future Gener. Comput. Syst.*, 26(8), October 2010.

[101] G.D. D Rodosek. A generic model for IT services and service management. *IFIP/IEEE Eighth International Symposium on Integrated Network Management, 2003.*, 2003.

[102] D. Sabin and E. C. Freuder. Configuration as composite constraint satisfaction. In *Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop*. AAAI Press, 1996.

[103] A. Sahai, S. Singhal, V. Machiraju, and R. Joshi. Automated policy-based resource construction in utility computing environments. *Network Operations and Management Symposium, 2004. NOMS 2004. IEEE/IFIP*, 1, April 2004.

[104] Akhil Sahai, Sharad Singhal, and Vijay Machiraju. Automated Generation of Resource Configurations through Policies. In *POLICY '04: Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, Washington, DC, USA, 2004. IEEE Computer Society.

[105] C. A. Santos, A. Sahai, X. Zhu, D. Beyer, V. Machiraju, and S. Singhal. Policy-Based Resource Assignment in Utility Computing Environments. *Lecture notes in computer science*, 2004.

[106] Thomas B. Sheridan. *Humans and Automation: System Design and Research Issues*, volume 3 of *HFES Issues in Human Factors and Ergonomics Series*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[107] Varnish Software. Varnish cache. `https://www.varnish-cache.org`, 2014.

[108] Python software foundation. Python. `https://www.python.org`, 2014.

[109] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational), January 2001.

[110] S. Thomson, T. Narten, and T. Jinmei. IPv6 Stateless Address Autoconfiguration. RFC 4862 (Draft Standard), September 2007.

[111] Bart Vanbrabant, Thomas Delaet, and Wouter Joosen. Federated access control and workflow enforcement in systems configuration. In *Proceedings of the 23rd Large Installations Systems Administration (LISA) conference*, Baltimore, MD, USA, November 2009.

[112] Bart Vanbrabant, Thomas Delaet, and Wouter Joosen. Authorising and directing configuration updates in contemporary it infrastructures. In *SafeConfig workshop, CCS 2010 Conference and co-located workshops*, Ghent, Belgium, October 2010.

[113] Bart Vanbrabant, Joris Peeraer, and Wouter Joosen. Fine-grained access control for the puppet configuration language. In *Proceedings of the 25th Large Installations Systems Administration (LISA) conference*, Boston, MA, USA, December 2011.

[114] Nicole F. Velasquez and Suzanne P. Weisband. Work practices of system administrators: implications for tool design. In *CHiMiT '08: Proceedings of the 2nd ACM Symposium on Computer Human Interaction for Management of Information Technology*, New York, NY, USA, 2008. ACM, ACM.

[115] D.C. Verma. Simplifying network administration using policy-based management. *IEEE Network*, 16(2), Mar/Apr 2002.

[116] Open vSwitch. Open vswitch: An open virtual switch. `http://openvswitch.org`, 2014.

[117] WS-O2 Inc. WS-O2. `http://wso2.com`, 2013.

[118] Lihua Yuan, Jianning Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah, and Prasant Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2006. IEEE Computer Society, IEEE Computer Society.

List of publications

International conference, and workshop paper

2013 B. Vanbrabant and W. Joosen. A framework for integrated configuration management tools. In *Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, Ghent, Belgium, May 2013.

2013 W. Daniels, B. Vanbrabant, D. Hughes, and W. Joosen. Automated allocation and configuration of dual stack ip networks. In *Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, Ghent, Belgium, May 2013.

2011 B. Vanbrabant, J. Peeraer, and W. Joosen. Fine-grained access control for the puppet configuration language. In *Proceedings of the 25th Large Installations Systems Administration (LISA) conference*, Boston, MA, USA, December 2011

2011 B. Vanbrabant and W. Joosen. Integrated management of network and security devices in it infrastructures. In *International conference on Network and Service Management (CNSM)*, Paris, France, October 2011.

2010 T. Delaet, W. Joosen, and B. Vanbrabant. A survey of system configuration tools. In *Proceedings of the 24th Large Installations Systems Administration (LISA) conference*, San Jose, CA, USA, November 2010.

2010 B. Vanbrabant, T. Delaet, and W. Joosen. Authorizing and directing configuration updates in contemporary it infrastructures. In *SafeConfig workshop, CCS 2010 Conference and co-located workshops.* Chicago, IL, USA, October 2010.

2009 B. Vanbrabant, T. Delaet, and W. Joosen. Federated access control and workflow enforcement in systems configuration. In *Proceedings of the 23rd Large Installations Systems Administration (LISA) conference*, Baltimore, MD, USA, November 2009. *Awarded Best Student Paper!*

Other publications

2010 T. Delaet, W. Joosen, and B. Vanbrabant. A survey of system configuration tools: The full evaluation of each tool. `https://distrinet.cs.kuleuven.be/software/sysconfigtools/`

Prototypes

The prototype of the IMP configuration management framework is available at `https://github.com/bartv/imp` The following list of configuration modules are available for this prototype of IMP:

1. **demo** Initial configuration model used in case 3 of the evaluation and in the technology demonstrator of the DREAMaaS research project. `https://github.com/bartv/dreamaas-demo`
2. **apache** Refinements for the httpd configuration module, implemented with the Apache webserver. `https://github.com/bartv/imp-apache`
3. **apt** Types and refinements to configure apt repositories. `https://github.com/bartv/imp-apt`
4. **aws** Resource handlers to deploy vm::Host types, as defined in the vm module. `https://github.com/bartv/imp-aws`
5. **backup** Types to configure backup on hosts. `https://github.com/bartv/imp-backup`
6. **bind** Refinements for the DNS configuration module, implemented with ISC bind. `https://github.com/bartv/imp-bind`
7. **bucky** Types and refinements to configure the bucky monitoring metric aggregation service. `https://github.com/bartv/imp-bucky`
8. **cassandra** Types and refinements to configure a Cassandra cluster. `https://github.com/bartv/imp-cassandra`
9. **ceph** Types and refinements to configure a Ceph distributed storage cluster. `https://github.com/bartv/imp-ceph`
10. **cinder** Types and refinements to configure the OpenStack block storage. `https://github.com/bartv/imp-cinder`

11. **collectd** Refinements for the monitoring types to collect system metrics on a host. `https://github.com/bartv/imp-collectd`
12. **dns** Types to configure DNS servers and DNS resource records. `https://github.com/bartv/imp-dns`
13. **drm** Module that implements the architecture of the DREAMaaS research project. `https://github.com/bartv/imp-drm`
14. **drupal** Types and refinements to configure Drupal (module used for the tutorial of Appendix B. `https://github.com/bartv/imp-drupal`
15. **duplicity** Refinements for the backup module, implemented with duplicity. `https://github.com/bartv/imp-duplicity`
16. **elasticsearch** Types and refinements to configure an elasticsearch instance. `https://github.com/bartv/imp-elasticsearch`
17. **exec** Type, resource and resource handler to execute arbitrary commands during the deployment, with conditions to make commands idempotent. `https://github.com/bartv/imp-exec`
18. **flens** Types and refinements to deploy the prototype of the MonArch monitoring system. `https://github.com/bartv/imp-flens`
19. **glance** Types and refinements to configure the OpenStack image delivery service. `https://github.com/bartv/imp-glance`
20. **graph** Types and export plug-ins to generate graphs from the configuration model, e.g. deployment diagrams. `https://github.com/bartv/imp-graph`
21. **graphite** Refinements for types of the monitoring module to configure a metric storage system. `https://github.com/bartv/imp-graphite`
22. **hbase** Types and refinements to deploy HBase on a single node. `https://github.com/bartv/imp-hbase`
23. **hosts** Types and refinements to manage the /etc/hosts file as a lightweight DNS. `https://github.com/bartv/imp-hosts`
24. **httpd** Types to configure webservers. `https://github.com/bartv/imp-httpd`
25. **imp** Types and refinements to configure the IMP server and agent. `https://github.com/bartv/imp-imp`
26. **influx** Types and refinements to configure the InfluxDB metric storage system. `https://github.com/bartv/imp-influx`

27. **ip** Types to model IP network related configuration. `https://github.com/bartv/imp-ip`
28. **jackrabbit** Types and refinements to configure the Jackrabbit WebDAV server. `https://github.com/bartv/imp-jackrabbit`
29. **java** Types and refinements to install Java, enable JMX and model Java services. `https://github.com/bartv/imp-java`
30. **jboss** Types and refinements to configure the JBoss Application server (clustered). `https://github.com/bartv/imp-jboss`
31. **keystone** Types and refinements for the OpenStack identity service. `https://github.com/bartv/imp-keystone`
32. **kibana** Types and refinements to configure Kibana, a dashboard to show and search logs. `https://github.com/bartv/imp-kibana`
33. **logging** Types to configure log aggregation. `https://github.com/bartv/imp-logging`
34. **logstash** Refinements to configure logstash for log processing and aggregation. `https://github.com/bartv/imp-logstash`
35. **monitoring** Types to configure monitoring. `https://github.com/bartv/imp-monitoring`
36. **mysql** Types and refinements to configure the Mysql database server. `https://github.com/bartv/imp-mysql`
37. **net** Types to model network (layer 2) configuration. `https://github.com/bartv/imp-net`
38. **neutron** Types and refinements to configure the OpenStack network controller. `https://github.com/bartv/imp-neutron`
39. **nfs** Types and refinements to configure the NFS filesystem. `https://github.com/bartv/imp-nfs`
40. **ntp** Types and refinements to configure network time synchronisation. `https://github.com/bartv/imp-ntp`
41. **openstack** Types to model the OpenStack architecture and deployment. `https://github.com/bartv/imp-openstack`
42. **opentsdb** Types and refinements to configure OpenTSDB as metric storage. `https://github.com/bartv/imp-opentsdb`

43. **openvpn** Types and refinements to configure OpenVPN, specifically to connect the network of two infrastructures over the internet (e.g. hybrid cloud). `https://github.com/bartv/imp-openvpn`
44. **php** Types and refinements to configure PHP in a webserver. `https://github.com/bartv/imp-php`
45. **pki** Types and transformation plug-ins to setup a CA to use in an infrastructure. `https://github.com/bartv/imp-pki`
46. **rabbitmq** Types and refinements to configure RabbitMQ as an AMPQ message broker. `https://github.com/bartv/imp-rabbitmq`
47. **redhat** Types and refinements specifically for RedHat based operating systems (RHEL, CentOS and Fedora). `https://github.com/bartv/imp-redhat`
48. **ssh** Types and refinements to configure an SSH server and authorized keys per user. `https://github.com/bartv/imp-ssh`
49. **std** Types, resource plug-ins and resource handlers for managed resources such as files, directories, software packages and system services. `https://github.com/bartv/imp-std`
50. **supervisord** Types and refinements to configure Supervisord to supervise processes. `https://github.com/bartv/imp-supervisord`
51. **taskworker** Types and refinements to configure the middleware of the DREAMaaS technology demonstrator. `https://github.com/bartv/imp-taskworker`
52. **tomcat** Types and refinements to deploy webapplication in a Tomcat server. `https://github.com/bartv/imp-tomcat`
53. **ubuntu** Refinements and resource handlers to support Ubuntu. `https://github.com/bartv/imp-ubuntu`
54. **vim** Types and refinements to configure ViM. `https://github.com/bartv/imp-vim`
55. **vm** Types to model virtual machines and resource plug-ins and resource handlers to deploy them on OpenStack. `https://github.com/bartv/imp-vm`
56. **yum** Types and refinements to configure YUM repositories. `https://github.com/bartv/imp-yum`

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMINDS-DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Heverlee
bart.vanbrabant@cs.kuleuven.be
http://www.cs.kuleuven.be