

AGH UNIVERSITY OF SCIENCE  
AND TECHNOLOGY IN KRAKÓW, POLAND

Faculty of Electrical Engineering, Automatics,  
Computer Science and Electronics

Department of Computer Science

# Installation of complex e-Science applications on heterogeneous cloud infrastructures

Bartosz Wilk

Master of Science Thesis  
in Computer Science

Supervisor: Dr Marian Bubak

Consultancy:  
Marek Kasztelnik (ACC Cyfronet AGH, Kraków),  
Dr Adam Beloum (Informatics Institute, University of Amsterdam)

Kraków, August 2012

Aware of criminal liability for making untrue statements I declare that the following thesis was written personally by myself and that I did not use any sources but the ones mentioned in the dissertation itself.

.....

AKADEMIA GÓRNICZO-HUTNICZA  
IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Elektrotechniki, Automatyki,  
Informatyki i Elektroniki

Katedra Informatyki

# Instalacja złożonych aplikacji e-Science na zasobach chmur obliczeniowych

Bartosz Wilk

Praca magisterska  
Kierunek studiów: Informatyka

Promotor: dr inż. Marian Bubak

Konsultacja:  
mgr inż. Marek Kasztelnik (ACK Cyfronet AGH, Kraków),  
Dr Adam Beloum (Informatics Institute, University of Amsterdam)

Kraków, Sierpień 2012

Oświadczam, świadomy odpowiedzialności karnej za poświadczanie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.

.....

---

# Abstract

---

Nowadays virtually any serious scientific research has to be supported by computing. As the time it takes to lead computations from the phase of design to the end of execution is crucial for the experimentation process, productivity of the scientists' work depend on issues inherently associated with the process of computational environment preparation. Moreover, following the modern trends of computer systems virtualization, more and more today's scientists decide to deploy their experiments in a relatively cheap, robust and efficient distributed infrastructure based on cloud services. In this thesis a solution that responds to a need for means of automatic e-Science application deployment in a heterogeneous cloud infrastructure is presented.

The scientific objective of this work is to evaluate the possibility of applying the methodology of Software Product Line to the experiment environment preparation process. The created tool proves the feasibility of this concept by adapting Feature Modeling to the domain of e-Science and allowing for deployment of application comprising a selection of features. The implementation takes advantage of State-of-the-art of large scale software installation automation methods, and uses a popular provisioning tool (Chef) to support deployment in heterogeneous cloud infrastructure. The system was built on the basis of requirements originating from VPH-Share, and allows scientists involved in the project for quick and intuitive creation of experiment environment, by using interface accessible through a Web Browser. The application was deployed in the production environment of the project and evaluated in several categories including usability, security and scalability.

Furthermore, there was elaborated a generic architectural concept of extensible Software Product Line, inspired by the experience gained in process of tool design and implementation. The concept applies to generation of production line architecture directly from the Feature Model. Presented reasoning can constitute a base for a framework to creation of declaratively managed, plug-in-based Software Product Line, extensible in terms of feature installation methods.

After an introduction (Chapter 1), there is presented a comparison of provisioning tools (Chapter 2) evaluated in the phase of research. The

## ABSTRACT

---

definition of Software Product Line, a concept of its application and a review of technologies suitable for implementation of the tool is introduced in Chapter 3. The following two chapters present the tool design (Chapter 4) and details of its implementation (Chapter 4). After a chapter devoted to the system review (Chapter 6), further reasoning on Software Product Line inspired by the research is described (Chapter 7). Chapter 8 summarizes the thesis and describes plans for future work.

**KEYWORDS:** e-Science, Cloud Computing, Software Product Line, Feature Model, Provisioning Tools, VPH-Share

---

# Acknowledgments

---

I dedicate this thesis to my parents for their love and continuous support in everything I do. I would like to thank my supervisor Dr Marian Bubak for sharing his experience and guidance throughout the process of writing this thesis. I would also like to express my appreciation to Marek Kasztelnik for his advisory in designing and implementation of the tool developed in scope of this work. Eryk Ciepiela and Jan Meizner are thanked for their helpful insights and suggestions.

This thesis was realized partially in the framework of the following projects:



Virtual Physiological Human: Sharing for Healthcare (VPH-Share) - partially funded by the European Commission under the Information Communication Technologies Programme (contract number 269978).



**KAPITAŁ LUDZKI**  
NARODOWA STRATEGIA SPÓJNOŚCI



**UNIA EUROPEJSKA**  
EUROPEJSKI  
FUNDUSZ SPOŁECZNY



Project UDA-POKL.04.01-01-00-367/08-00 "Improvement of didactic potential of computer science specialization at AGH", at the Department of Computer Science, AGH University of Science and Technology, Al. A. Mickiewicza 30, 30-059 Krakow





---

# Contents

---

<b>Abstract</b>	<b>i</b>
<b>Acknowledgments</b>	<b>iii</b>
<b>Contents</b>	<b>v</b>
<b>List of Figures</b>	<b>vii</b>
<b>Abbreviations and Acronyms</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Survey of automatic software installation methods . . . . .	3
1.3 Objectives of the thesis . . . . .	7
<b>2 Provisioning in cloud infrastructure</b>	<b>9</b>
2.1 Comparison of provisioning tools . . . . .	9
2.2 Provisioning with Chef . . . . .	12
2.3 Summary . . . . .	15
<b>3 Software Product Line as a generic approach to software creation</b>	<b>17</b>
3.1 The concept of using Software Product Line methodology . . . . .	17
3.2 Overview of Feature Modeling . . . . .	18
3.3 Families of Feature Model reasoning algorithms . . . . .	21
3.4 Solvers based on Boolean Satisfiability Problem and Binary Decision Diagrams . . . . .	24
3.5 Tools for Feature Model reasoning . . . . .	26
3.6 Summary . . . . .	32
<b>4 Experiment environment preparation tool overview</b>	<b>33</b>
4.1 Cloudberry general description . . . . .	33
4.2 Specification of requirements . . . . .	34
4.3 High-level system design . . . . .	36
4.4 Summary . . . . .	39

## CONTENTS

---

<b>5</b>	<b>Implementation of the tool</b>	<b>41</b>
5.1	Proof of Concept . . . . .	41
5.2	Cloudberries - the tool . . . . .	45
5.3	User interface . . . . .	48
5.4	Summary . . . . .	52
<b>6</b>	<b>Validation of tool</b>	<b>55</b>
6.1	Case study . . . . .	56
6.2	System usability evaluation . . . . .	57
6.3	System security . . . . .	58
6.4	Limitations . . . . .	59
6.5	Summary . . . . .	60
<b>7</b>	<b>A concept of Feature Model - based automatic SPL generation</b>	<b>63</b>
7.1	Feature Model adaptation . . . . .	63
7.2	A concept of Feature Model - driven Software Product Line architecture . . . . .	72
7.3	Further thoughts . . . . .	78
7.4	Summary . . . . .	81
<b>8</b>	<b>Summary and Future Work</b>	<b>83</b>
	<b>Appendices</b>	<b>85</b>
<b>A</b>	<b>Glossary</b>	<b>87</b>
<b>B</b>	<b>Installation Guide</b>	<b>91</b>
B.1	Installing prerequisites . . . . .	91
B.2	Cloudberries portlet installation . . . . .	93
	<b>Bibliography</b>	<b>99</b>

---

# List of Figures

---

1.1	Schematic presenting components of the tool being developed in scope of this thesis. Two main architectural components (Configuration and Deployment modules) are interconnected and built on top of the cloud infrastructure. . . . .	7
2.1	The architecture of Chef [3]. The administrator uses his Workstation to connect to the server, upload installation packages (cookbooks), and deploy a selection of environment components on the node machine. In order to perform the configuration remotely Chef uses its own client application. . . . .	13
3.1	Graphical representation of feature relationship types. The image is based on a picture presented in [8]. . . . .	19
3.2	Sample feature model describing a product line from telecommunications industry [8]. The model presents hierarchical structure of mobile phone features. In order to include some additional constraints in the model, cross-tree constraints may be provided. . . .	20
3.3	The rules for translating Feature Model relations into Boolean formulas [8], in order to represent the model in a form as a Boolean Satisfiability Problem. The third column is a mapping of relationships presented as an example in the Figure 3.2. . . . .	21
3.4	Sample Binary Decision Diagram [2]. Round nodes represent variables (in this case they are product features). Squares external nodes denote values of the formula. Solid edge represents 1, dotted - 0. A path from the root to an external node is a representation of a variable values. . . . .	22
3.5	FaMa modular architecture [22]. As one can see there are four reasoners that can be used to operate on feature models. FaMa is built as a Software Product Line, so the library not only supports the methodology, but also is an example of its implementation. . .	28

3.6	The architecture of SPLOT [49]. The application is a web based tool allowing to perform various operations on feature models. Under a web user interface there is an application using SPLAR library implementing the logic of operations. As one can see in the diagram, SPLOT is based on JavaEE and uses Servlets to handle HTTP requests. . . . .	30
3.7	A diagram of the core SPLOT classes. A single servlet uses multiple handlers. The source code of handlers can be reused quite easily. . . . .	31
4.1	The design of Cloudberryes experiment developer interface. Experiment Developer can manipulate three types of entities. <i>Configuration</i> is a selection of environment components that can be deployed in the experiment environment after providing necessary attributes. <i>Deployment Template</i> allows to store a composition of default values for the attributes. <i>Deployment Task</i> is a representation of running deployment process. The user can monitor an installation by viewing details of the corresponding <i>Deployment Task</i> . . . . .	37
4.2	The design of Cloudberryes administration interface. The branch on the left side is an interface accessed via Web Browser, and the one on the right requires usage of linux shell. . . . .	38
4.3	The architecture of Cloudberryes. There are two basic components of the system - an instance of Chef server and the main Cloudberryes application running in web application container. Both of these components need an access to the cloud infrastructure. The scientific user accesses Cloudberryes using Internet Browser. Administrative tasks are performed using linux shell. . . . .	39
5.1	Example of a feature model in the extended SXFM notation that is used by Cloudberryes. There are all types of the feature relationship used in this model. The structure is hierarchical and very intuitive to understand. After a colon there is a character which determines type of the relationship with the parent feature. Blank character means that the feature is a leaf in the tree. Feature identifiers are written in brackets. Cross tree constraints has to be formulated in Conjunctive Normal Form and placed in the <i>constraints</i> tag. The features marked with exclamation mark are treated as installable and will be considered during validation of the installation package repository. The use of exclamation marks for distinguishing installable features differs format used by Cloudberryes from the original SXFM. . . . .	43

5.2	Sample configuration in the format that is accepted by the Cloudberries prototype. Configuration was prepared using SPLOT. The constraints in the model (See Figure 5.1) allow SPLOT to automatically guess all of the above decisions. . . . .	44
5.3	The entity Relationship Diagram of the Cloudberries database. This diagram should be read starting from FEATURE_MODEL_FILE table and finishing on DEPLOYMENT_TASK. The names of tables are quite intuitive and match corresponding items in the user interface. . . . .	47
5.4	Creation of an experiment environment configuration. On the left side of the page one can see a hierarchical representation of the feature model loaded by Cloudberries from a file saved under a location specified in settings. Features represent elements of the environment. Each configuration step, the underlying application layer updates configuration changes and excludes all of the useless features from the configuration space. On the right side there is placed a table of configuration steps. Configuration process can be automatically completed by selecting either Less Features (selects as little as possible) or More Features (selects as much as possible). . . . .	49
5.5	In this screenshot one can see the list of configurations that were previously created and saved by the users. Each configuration is assigned to a feature model which was used to create it. Before any further usage, the configuration is validated with respect to the model, in order to avoid errors in the installation process. . .	50
5.6	Creation of a deployment template. A user can entitle the template and provide some information notes. The box on the left allows to select attributes for the configuration elements. The one on the right allows to specify values for the attributes. This is very similar to the process of <i>Deployment Task</i> creation. . . . .	51
5.7	Monitoring of a <i>Deployment Task</i> . After selecting the task from the list in the <i>Deployment Tasks</i> page, the user can see a page similar to the above. The table on the top contains information about installation steps. By selecting a step one can see the corresponding log. . . . .	52
7.1	Software Product Line engineering schematics [36]. There are three spaces that has to be connected in order to create production line. The most challenging process in the lifecycle of the production line is a mapping from the space of product features (that are relevant to stakeholder) to the space of artifacts (that are relevant to the developers). . . . .	64

LIST OF FIGURES

---

7.2	A schematic of production line using several production sites to produce single feature. Each production site is capable for its own production procedures that can be applied to realize partial production of given feature. . . . .	65
7.3	A cycle in the process of production. Requirement should be treated as a dependency of production stages. Production stages of the given features cannot be put in any order that guarantees to meet all of the requirements. . . . .	66
7.4	A schematic of production line using single production site to provide a single feature. The process of feature installation is atomic and cannot be split between production sites. So that, there is no need for management of dependencies of intermediate production stages. . . . .	68
7.5	Sample feature model enriched with installation ordering relationship. The installation ordering relationship extends the original Feature Model relationships in order to provide additional information that is needed to perform scheduling. . . . .	70
7.7	Graphical illustration of the ordering algorithm presented before. The Feature Model was replaced by the directed acyclic graph with edges determined by the ordering relationship. The graph is sorted topologically by visiting adjacent nodes starting from the Root. . . . .	71
7.6	Simple recursive ordering algorithm. The nodes in the graph represent features of the feature model. The edges are built out of the installation ordering relationship. . . . .	71
7.8	The problem of ambiguous installation ordering. Both feature 3 and feature 2 should be installed before installation of the feature 1 (ordering relationship marked with orange). Presented algorithm does not specify the installation order of indirectly connected feature 3 and feature 2. . . . .	72
7.9	The internal construction of the main production line component. The main inner component is the <i>Workflow Manager</i> that controls the production process. <i>Production Site</i> plug-ins, <i>Feature Models</i> and <i>Feature Descriptors</i> are registered in appropriate registries. . .	73
7.10	The concept of Feature Descriptor structure. This structure can be for example mapped to XML format. . . . .	74
7.11	ProductionSite interface to be implemented in order to provide means of feature installation (see procedure <code>installFeature</code> ). As a Production Site is controlled from the outside the function <code>isFeatureSupported</code> has to be implemented to declare support for production of feature with a given identifier. . . . .	75
7.12	Sample contract of interfaces. Installation procedure of feature 2 depends on 1 and provides feature 1 <i>Production Site</i> with its own input. Feature 1 will be installed before feature 2. . . . .	77

7.13	Sample contract of interfaces. Feature 1 depends on 2. Feature 1 will be installed before feature 2 and provides feature 1 with output emerged from its own installation. . . . .	77
7.14	An illustration of the problem of scheduling the installation performed by <i>Group Production Site</i> . The <i>Group Production Site</i> is a <i>Production Site</i> that does not allow for separate installation of supported features. The process of subordinated features installation has to be performed as an atomic, indivisible group. . . . .	79
7.15	The figure presents exactly the same situation as in the previous picture, but all of the features installed by a <i>Group Production Site</i> are treated as a single feature (the green rectangle). This approach allows to check if there are any cycles which causes scheduling unfeasible. In this figure we can see a cycle of the installation ordering relationship graph, so the model that contain features arranged in this way, should be considered invalid. . . . .	80
7.16	A symbolic representation of a feature model part with features assigned to a <i>Group Production Site</i> in a way that is correct in terms of installation scheduling. Scheduling is feasible when features connected to a group with installation ordering relationships can be split into two independent groups. The first contains only features that are installed before the features in green rectangle (installed by <i>Group Production Site</i> ). Second group contains only features installed afterwords. . . . .	80





---

# Abbreviations and Acronyms

---

AHEAD	Algebraic Hierarchical Equations for Application Design
API	Application Programming Interface
ATS	AHEAD Tool Suite
BDD	Binary Decision Diagram
CNF	Conjunctive Normal Form
COSL	Commercial Open Source License
CP	Constraint Programming
CSP	Constraint Satisfaction Problem
DAG	Directed Acyclic Graph
DSL	Domain Specific Language
FM	Feature Model
FOP	Feature Oriented Programming
FORM	Feature Oriented Reuse Method
GNU	GNU's Not Unix
GPL	General Public License
GPS	Global Positioning System
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
ID	Identifier
IP	Internet Protocol
JEE	Java Enterprise Edition
JSON	JavaScript Object Notation
JSR	Java Specification Request
JVM	Java Virtual Machine

## ABBREVIATIONS AND ACRONYMS

---

MAC	Media Access Control
MVC	Model View Controller
OS	Operating System
PaaS	Platform as a Service
PXE	Preboot Execution Environment
REST	Representational State Transfer
RHEL	RedHat Enterprise Linux
SaaS	Software as a Service
SAT	Boolean satisfiability problem
SPL	Software Product Line
SPLAR	Software Product Lines Automated Reasoning
SPLOT	Software Product Line Online Tools
SQL	Structured Query Language
SSH	Secure Shell
SSL	Secure Socket Layer
SXFM	Simple XML Feature Model
TFTP	Trivial File Transfer Protocol
VM	Virtual Machine
VPH	Virtual Physiological Human
XML	eXtesnible Markup Language

# Chapter 1

---

## Introduction

---

*This chapter presents the background and general objectives of the thesis. The motivations for the project are described in the first section. Next section describes different approaches to solve the problem addressed by the thesis. The last one presents some high-level design considerations for the tool developed in scope of the thesis that summarizes the preceding discussion.*

### 1.1 Background

In the 21st century we are living in the world of massive progress in culture, technology and science. In the domain of information technology we already reached a level in which computerization affected virtually every aspect of human life. Revolution of computer technology made us belong to an information society in which knowledge, communication, entertainment and many other areas become increasingly subordinated to management of information. Science is no different in this respect from other fields. Since the capabilities of computers can give people a promise to finish complex computations in a reasonable time, simulation have become a modern paradigm of scientific research and its popularity is continually growing. Therefore, nowadays experiments *in silico* become an important and powerful approach in scientific research that brings lots of challenges every day. What is more, as increasing number of scientists decide to use the power of computer systems to bring the productivity of their research to the next level, the complexity of tasks delegated to the responsibility of computer systems is continuously growing. In this scope a particularly interesting field is computationally intensive science, carried out in highly distributed network environments. Traditionally, e-Science denotes a field of scientific research that is oriented towards grid computing [30], which provides users with broad access to the means of high performance computation. Although computing on the grid is a paradigm that was designed to satisfy specific needs of the scientific community, the use of the infrastructure is quite bulky and inconvenient. Never-

theless, the computational grid is continuously evolving and still particularly valuable for e-Science.

On the other hand, a great expansion of virtualization of computer systems has become a fact. It can be observed that a massive growth of popularity regarding services of virtualized computer infrastructure has a significant influence on the service market. Today, there are quite a lot of solutions that provide access to commercial cloud infrastructure, offering virtualization of computer systems at different level - Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Several encouraging factors such as low costs, high availability, decent performance regarding usage in scientific applications, causes that cloud computing attracts a growing number of researchers. This particular case of scientific research is the subject of this work. The objective of this thesis is to investigate some issues regarding deployment of e-Science applications on a cloud infrastructure, point out the problems, and provide a solution that will help to avoid them.

There is a basic concept of e-Science application which has to be described before the further reasoning can be started. e-Science applications are built in context of scientific research and should be treated as implementations of scientific experiments. As an application in terms of e-Science usually consist of components originating from several domains including various types of data, software applications, libraries and others, the concept of e-Science application is very generic and may be defined as a composite of the mentioned components. Moreover, the components of a single application may be distributed between virtual or physical machines in virtually unlimited way. So that, the task of e-Science application installation, addressed by the author of this thesis, can be in fact described as a process of deployment of a complex (and possibly heterogeneous) environment distributed among virtual machines operating in a cloud infrastructure. Furthermore, as this work originates from VPH-Share project [58] founded by European Commission it concerns applications from various areas of science related to medicine. Scientists that take part in the project, represent wide range of interests concerning simulations of human heart (euHeart [21]), neural system (@neurIST [42]), and others.

Following the definition of e-Science application presented before, now we can describe the process of its deployment. In fact, after preparing a distribution plan of e-Science application components, the installation of an application in cloud infrastructure is a process that consists of two major steps - instantiation of virtual machine and installation of selected components on a running VM instance. As the second task is the most time-consuming and troublesome, it is particularly interesting from the perspective of this work. As the person executing experiments is generally a scientist and not a system administrator, they do not necessarily have to be familiar with methods of installation and configuration of environment components on a particular

operating system. This circumstance is the main reason for the work.

Therefore, one of this thesis' objectives is to design and provide implementation of a tool making the process of experiment environment preparation as easy as it is possible. Another interesting aspect of the given task is the cost of VM image storage. Because of the virtual machine nature, some of the components may be preinstalled on the saved VM image. In order to bother the scientist as little as possible, the best solution of the problem of the configuration complexity, would be storage of previously configured VM images that are specifically suitable for the particular experiment. There is an obvious need for configuration of the image before it can be saved in the form corresponding to requirements of particular experiment, but during every following execution a scientist can use the environment prepared before. There is no doubt that in this case any further executions of the same experiment, will be free from unnecessary configuration. The most significant problem regarding this approach the cost of the storage that will affect every saved image. Alternatively, just the configuration of environment can be saved and redeployed on demand. Sacrificing a little time to perform the deployment again, the cost of storage will be greatly reduced.

Summing up the above considerations, we can clarify the objective of this thesis, describing it as creation of tool that will allow a scientist to perform deployment and reconstruction of experiment environment in an easy and intuitive way. Deployment should apply to installation of various software products, data, and configuration of the operating system. Although the main application of the system is the use in terms of VPH-Share project [58], it might be used as a general purpose tool. In the next sections there are presented some high-level requirements regarding the tool, possible approaches to meet these requirements and vision of the system implementation that can satisfy them.

## 1.2 Survey of automatic software installation methods

For a scientist, preparation of an environment for experiment execution is a process that in some aspects can be compared to everyday activities of a person administering a distributed computer infrastructure. Although there is a slight difference between tasks performed by these two types of specialists, some requirements affect both groups. What should be outlined, the discussed case is focused on administration of environments based on virtualized infrastructure. In design of a system that has to satisfy needs of scientific users, several aspects should be taken into account:

- **A need for simultaneous configuration of a number of virtual machines.** When the experiment makes use of a distributed infrastructure to replicate computation. Each node of the infrastructure is configured in the same way.

- **A need for reusing a configuration to deploy it again and rebuild infrastructure.** Reusing a configuration of a single computation node can also be taken into account. This aspect matters when the experiment is executed again.
- **A need for reusing an existing virtual machine image.** In order to deploy a configuration it may be reasonable to reuse a stored image providing base elements of the environment.

Despite some commonalities between the two groups, the same task may for a scientist mean a different problem to be addressed. During the software installation, they can face some issues they are not familiar with. Problems may come from a broad domain comprising such issues as the need for operating system administration skills, familiarity with infrastructure, having appropriate privileges and others. So that, in order to meet the requirements of usability, the level of environment preparation complexity in the designed system should be as low as possible.

There are several methods known to satisfy the above needs. In this section three classes of them will be presented in order to outline different approaches to the problem.

### 1.2.1 Distributed Shell

Distributed shells is a class of software products that allow to configure multiple instances of similar operating systems. There are plenty of applications belonging to this class of software. Some of the examples are clusterssh [16], pssh [44], clusterit [15], omnitty [43], taktuk [53], dish [19], dsh [20].

Distributed shell allows to simultaneously log into a number of remote consoles (all of the above solutions are based on SSH) and replicate commands among them. The key assumption of the tools, which belong to this class of software, is a wish that all of the operating systems that are configured at once, are virtually identical and react in the same way. Of course applications from this group will not satisfy the need for easing the installation process. It is also hard to imagine automation based on this kind of software. The clear advantage over configuration without any additional tools is a possibility to save some time that is wasted on repetition of identical configuration steps.

### 1.2.2 Unattended Installation

Unattended installation is a notion that describes a class of software products, which allow to install a specific configuration of on operating system without any (or just a little) interaction with user. As the automation is performed at the installation phase, the starting point of this solution is a situation when we have a machine (in particular a virtual machine) without

any operating system. The requirements specific for this thesis imply that the cloud provider will support such situation. However, as it was examined that factor should not raise any problems as long as we think of local cloud administration. The method can be useful in case of using Preboot Execution Environment (PXE). PXE allows to boot an installer of operating system from the network, download a *kickstart* configuration file (the *kickstart* name is used here to represent of a class of solutions rather than a concrete product) from TFTP server, and perform unattended installation. This class of solutions is addressed to administrators who want to ease the process of system installation, especially when the target infrastructure is composed of a considerable number of nodes.

On the market there are a number of solutions specific to an operating system that support Unattended Installation paradigm - RedHat/Fedora KickStart [1], Ubuntu KickStart [4], Solaris JumpStart [50], as well as Unattended [57] and Fog Project [25] (cloning image only) for Windows.

This approach should satisfy most of the requirements as the process of installation is fully automatic. A configuration of single node can be easily replicated and there is an easy way to reuse a configuration that has been created before, so this solution seems to be promising in terms of the requirements mentioned before. A disadvantage of this approach is the fact that there is no way to make use of previously saved virtual machine image. So that each configuration has to be deployed from scratch including the installation of an operating system. It seems that this is a limitation that may considerably slow down the process of deployment.

### 1.2.3 Provisioning Tools

The last class of software to automatically deploy a distributed environment, covered by this review is a group of provisioning tools. These tools are usually built in a client /server architecture, and allow to manage operating systems by installing a lightweight client on a target machine. Client configures the operating system and performs installation of software on behalf of the central management system. Most of them allow to monitor some system attributes as the architecture, type of operating system, network addresses (MAC, IP), memory etc. The software of that type is oriented towards operation during the entire operating system lifecycle so there are no limitations as those listed in the previous paragraph. There are a number of provisioning tools on the market and some of the most popular are Bcfg2 [6], CFEngine [9], Chef [11], Puppet [45].

Although there are some similarities of the software products belonging to this group, each of them implements its own concept of deployment mechanism. Differences start from the form of installation packages, concern scripting and configuration representation languages, operating system support and available API. Most of them allow to use SSH in order to bootstrap

(install) a client on a remote machine (in case of Unix/Linux). Some of the tools provide cloud support in terms of virtual machine instantiation, so it should be also taken into account during the comparison.

This group of software seems to meet all of the requirements specific to the thesis. Tools in the group of software provisioning allow to automate deployment process without additional limitations such as the application in a specific phase of operating system lifecycle. They allow to easily reuse configurations and virtual machine images. They can also satisfy a need for multiple installation of the same selection of environment components.

### 1.2.4 Evaluation of software installation tools

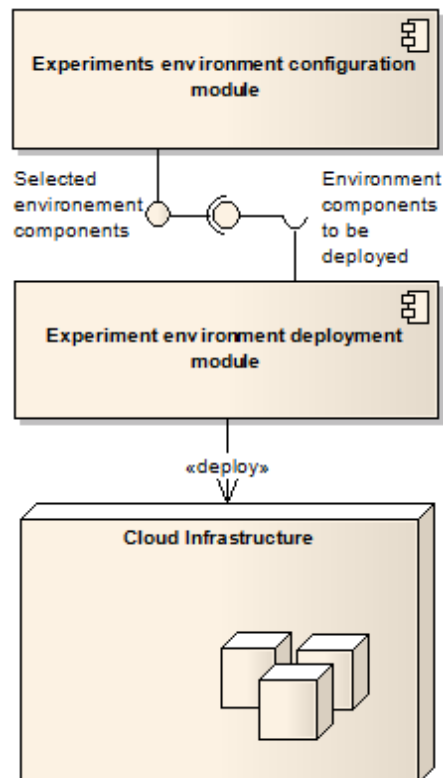
In this chapter there were presented three classes of software products that can help a scientist in a process of experiment environment preparation. The tasks which has to be performed by a person deploying experiment environment, were compared to activities of distributed (and possibly heterogeneous) computer environment administrator. In respect to that, the selection of the presented tools were based on analysis of software products commonly used by administrators. At the beginning of the chapter there were presented some requirements of the deployment process and specific circumstances of the application in the target use case. Based on the specific conditions of use the presented groups of application can be judged and compared in terms of usefulness. *Distributed shells* are tools that help the user to save time wasted on repetition of configuration. In fact they are not pure automation tools, and will not ease the process of configuration in terms of providing any simplification of manual installation. The tools of *Unattended installation* help an administrator to automate the process of single installation of an operating system. After providing an appropriate configuration the user is discharged from the supervision of the installation process. This solution is functionally similar to using *Provisioning tools* but offers less flexibility. Inability to use saved virtual machine image again, makes it much less useful and that is why wthere was made a decision of using provisioning tool to automate the process of deployment. In this work there is a separate chapter dedicated to provisioning tools, so more detailed description and comparison of specific tools will be presented there.

As the use of provisioning system can still be a difficult task for a scientist, the application being developed in the scope of this project should provide a layer that will hide unnecessary complexity and make environment configuration as easy as possible. In the next section there is presented a vision of a system to experiment environment preparation based on provisioning application that will focus on minimizing complexity of a configuration process.



### 1.3 Objectives of the thesis

As it was mentioned in the previous section, in scope of this thesis there is a need for creation of a tool that will extend capabilities of a provisioning system, in order to enable e-Science researchers to use it. The extension can be understood as providing an additional layer in order to facilitate the use. In this section there is presented a conceptual vision of the system. At this point of description, the architecture can be presented as three high-level components visible in the figure 1.1.



**Figure 1.1.** Schematic presenting components of the tool being developed in scope of this thesis. Two main architectural components (Configuration and Deployment modules) are interconnected and built on top of the cloud infrastructure.

The architectural element responsible for installation of environment components, which was presented in the previous section, is located in the middle of the Figure 1.1. An additional layer built on top of the deployment layer, has to provide a scientific user with an interface for selecting components of experiment environment. In terms of project design there are two main challenges regarding development process. First is a task of deploy-

ment of a provisioning system, and implementation of its management. The other one is a task of implementation of an environment configuration user interface.

During the research that was made in order to find a suitable approach for modeling of environment configuration, there emerged an idea of analogy of e-Science application to Software Product Line operation. SPL refers to software engineering techniques for creation of similar software systems from a shared set of software assets. If we would think for a moment of installation of e-Science application as a process of software composition from specific prerequisites, it would appear that the case may have much in common with a task of software product creation. The similarities are manifested in several aspects. The most significant characteristic, which applies to both cases is orientation on features of the target product. Talking about Software Product Line it cannot be failed to mention that there are number of feature-oriented techniques and tools of modeling product configuration used in the context of SPL. Feature Modeling which is a common subject of interest of those involved in Software Product Lines, can be easily adapted to modeling environment experiment configuration. As research has shown, the paradigm is worth taking a closer look, and considering in the project design.

In the description of high level tool architecture it should be noted that the design is based on the idea of implementing Software Product Line methodology. So that, the target system can be treated as a product line with a module responsible for automatic installation on a cloud, which is based on provisioning system, and a user interface allowing for configuration of experiment environment based on theoretical foundations of Feature Modeling. Components of the target environment are represented as its features. A resulting e-Science application is an analogy of software product in the nomenclature of Software Product Line. In order to clarify this concept, a wider description of SPL adaptation will be introduced later, in a separate chapter (Chapter 3).

This thesis presents reasoning on selected aspects regarding the system design. First, there is described a comparison of various provisioning tools in order to present the circumstances of selecting one that is most suitable for implementation. The selected provisioning system is described in more detail. Then, more complex description of Software Product Line as well as algorithms and tools supporting Feature Modeling are introduced. A few interesting software products based on Feature Modeling are described more broadly, in order to present solutions that can be used in development of the tool. After describing issues connected with research on State-of-the-art, there is presented precise specification of requirements and selected aspects of the tool implementation. Finally there is presented reasoning on Software Product Line and Feature Modeling, inspired by the experience gained during the research in scope of this thesis. The reasoning leads us to the concept of a framework to automatic Feature Model - based generation of production line architecture presented in the Chapter 7.

## Chapter 2

---

# Provisioning in cloud infrastructure

---

*In the section 1.2 a comparison of several approaches to administration of distributed computer environment was presented. As using provisioning tools was assessed to be the best solution in the context of presented requirements, this paradigm will be described broader. In this chapter the capabilities of the previously mentioned provisioning systems is presented and the solution, which was chosen to be used in the implementation, is described more broadly.*

### 2.1 Comparison of provisioning tools

Cloud provisioning tools are framework applications, built to bring the benefits of automatic configuration management to cloud infrastructure. On the market there are many provisioning solutions providing slightly different flavors of automatic deployment. In order to choose one, which will be particularly valuable for the project, following comparison was made. As it is mentioned below some of the frameworks are delivered in both free and commercial product versions. For the project this is a limitation that substantially narrows a research. In the review presented below following products are compared: Bcfg2 [6], CFEngine [9], Chef [11], Puppet [45].

To choose a suitable provisioning application a few aspects has to be taken into account. Application has to be run on Ubuntu Linux and client should support as many operating systems as possible (Windows support is important). Java or REST API is very convenient in terms of invocation from the code of the tool. License policy is another crucial limitation - free use is required. Because the information presented in this section has been collected from various sources, it is very hard to impose consistency in the tables below. So that, the versions of the operating systems supported by the following provisioning systems are listed in the form they were presented in the literature.

### 2.1.1 Bcfg2

Bcfg2	
Language	Declarative management via XML
API	no API
License	BSD
Supported OS	AIX, FreeBSD, OpenBSD, Mac OS X, OpenSolaris, Solaris, ArchLinux Blag, CentOS, Debian, Fedora, Gentoo, gNewSense, Mandriva, openSUSE, Red Hat/RHEL, SuSE/SLES, Trisquel, Ubuntu, Windows not directly supported [7]
Community	no community

**Table 2.1.** Summary of Bcfg2 features.

Bcfg2 [6] is a tool to configure a large number of computers, developed in Python by members of the Mathematics and Computer Science Division of Argonne National Laboratory. It is based on a client-server architecture and the client is responsible for interpreting the configuration provided by the server. Client translates a declarative configuration specification into a set of configuration operations which will attempt to change its state (if the process of configuration fails, the operation can be rolled-back). So that, the declarative specification for environment components are separated from the imperative operations implementing configuration changes. After completion of the configuration process, client application uploads statistics to the server [18]. Generators enable code or template based generation of configuration files from a central data repository [17]. The Bcfg2 client internally supports the administrative tools available on different architectures. Following table presents facts about Bcfg2:

### 2.1.2 CFEngine

CFEngine	
Language	Declarative management via XML
API	REST (commercial version only)
License	Commercial/Open Source(limited functionality)
Supported OS	Linux, Unix, Solaris, AIX, FreeBSD, Macintosh, Windows (CygWin is required)
Community	no community

**Table 2.2.** Summary of CFEngine features.

CFEngine [9][17] is an extensible framework for management of either individual or networked computers developed in C. It has existed as a software suite since 1993 and then the third version published under the GNU Public License (GPL v3) and a Commercial Open Source License (COSL). The engine is different from most automation tools that runs a process of configuration and stops when installation is finished. Every configured environment is also continuously verified and maintained. After installation of a lightweight agent it continues to run during the environment lifecycle. Any agent state which is different from the policy description is reverted to the desired state.

### 2.1.3 Chef

Chef	
Language	Ruby DSL
API	REST Server API, JClouds-Chef third party Java API
License	Apache License
Supported OS	Ubuntu (10.04, 10.10, 11.04, 11.10), Debian (5.0, 6.0), RHEL, CentOS (5.x, 6.x), Fedora 10, Mac OS X (10.4, 10.5, 10.6), Windows 7, Windows Server 2003 R2, 2008 R2, Ubuntu (6.06, 8.04-9.10)*, Gentoo (11.1, 11.2)*, FreeBSD (7.1)*, OpenBSD (4.4)*, OpenSolaris (2008.11)*, Solaris 5.10 (u6)*, Windows XP, Vista*
Community	Script repository for user scripts, hosted by Opscode

**Table 2.3.** Summary of Chef features. (\*) - As Chef documentation claims "Additionally, chef-client is known to run on the following platforms"

Chef [11][10][17] is a library, configuration management system, system integration platform and API written in Ruby that uses a Ruby DSL for writing configuration "recipes" [17]. These recipes are basically bundles of installation steps (or scripts) to be executed. There is quite a big community of users who share their recipes via repository managed by Opscode company. Chef can be used in one of possible two modes - either client-server or solo [10]. There is also a possibility to use commercially hosted Chef for free - up to 5 nodes of provisioned infrastructure.

### 2.1.4 Puppet

Puppet	
Language	Own language/Ruby DSL
API	REST
License	Apache License/Commercial
Supported OS	RHEL (4 - agent only, 5, 6), Ubuntu 10.04 LTS, Debian (5, 6), CentOS (4 - agent only, 5, 6), Scientific Linux (5, 6), Oracle Linux (5, 6), SLES 11, RHEL 4 (agent only), Solaris 10 (agent only), Windows (commercial version only)
Community	Package repository

**Table 2.4.** Summary of Puppet features.

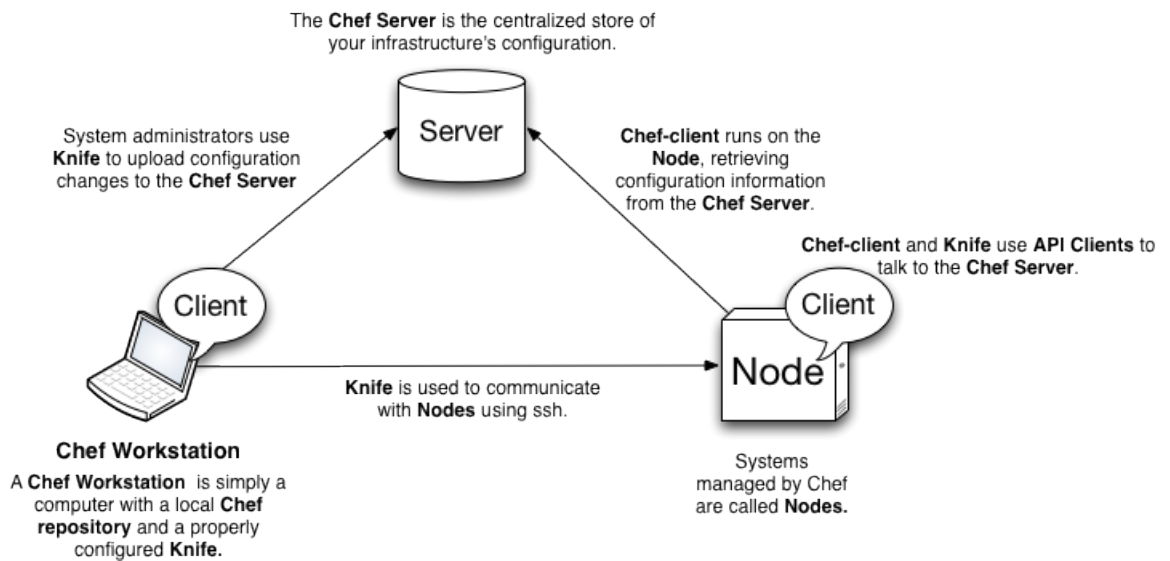
Puppet [45] is an open source configuration management tool allowing to manage Unix-like and Microsoft Windows systems declaratively. Puppet can be used by writing a configuration either in Puppet own declarative language or in a Ruby DSL. Puppet, like the rest of the presented provisioning tools is a client-server solution.

### 2.1.5 Result of provisioning tools comparison

As all of the previously mentioned provisioning tools are suitable for the given task, it is possible to show some stronger and weaker points of each, but it's hard to be fair while doing comparison, because all of them are slightly different. Anyway it seems that Chef is the most promising among free solutions above, because of full Windows support, ease of use of the third party Java API, user community providing ready-to-use scripts. So that, Chef wins the competition and it will be the solution used in the implementation. That is the reason why the next section covers a wider description of Chef.

## 2.2 Provisioning with Chef

The purpose of this section is to describe Chef [11] architecture in a nutshell and introduce vocabulary strictly connected with Chef. For more detailed description of Chef refer to the Chef documentation.



**Figure 2.1.** The architecture of Chef [3]. The administrator uses his Workstation to connect to the server, upload installation packages (cookbooks), and deploy a selection of environment components on the node machine. In order to perform the configuration remotely Chef uses its own client application.

### 2.2.1 Chef architecture

Chef architecture[3] in its simplest form is presented in the following Figure 2.1. As you can see there are three core elements - workstation, server and node.

Chef documentation introduces specific vocabulary, one have to get used to while using Chef. Below there are explained some elements that are crucial to understanding the rest of the thesis, in order to skip these details in the tool architecture overview.

- **Server** - A Chef server is a centralized store of infrastructure configuration. It manages users, nodes, cookbooks (provides access to the central cookbook repository), attributes, roles etc. Server is a passive element of the architecture. Client communicates it whenever it has to obtain any needed information.
- **Workstation** - Workstation is a computer station of a system administrator. In order to communicate with Chef Server administrator is using a command line tool called *Knife*. A Workstation is also a local repository of cookbooks which will be uploaded to the server.
- **Node** - A node is a host that runs the Chef client. The primary features of a node from Chef's point of view are its attributes and its run list.

- **Run List** - A run list is a list of the recipes that a client will run. Assuming the cookbook metadata is correct, you can put just the recipes you want to run in the run list, and dependent recipes will be run automatically if needed. Ordering is important: the order in which recipes are listed in the run list is exactly the same order in which chef will run them.
- **Cookbooks** - A cookbook is a collection of recipes, resource definition, attribute, libraries, cookbook files and template files that chef uses to configure a system, plus metadata. Cookbooks are typically grouped around configuring a single package or service. The MySQL cookbook for example contains recipes for both client and server, plus an attributes file to set defaults for tunable values. Cookbooks are the unit of distribution and sharing in Chef. Most of the time you are using Chef, you are writing cookbooks.
- **Recipes** - recipes are bundles of installation steps (or scripts) to be executed. They are files where you write resources and arbitrary ruby code (Ruby DSL). Although writing recipes is quite easy, it is needed to understand a little bit about how Chef runs in order to write them.

### 2.2.2 Chef in operation

In order to prepare Chef to be used for software installation, an administrator has to provide cookbooks that can be used later. To create a cookbook, administrator prepares installation scripts in a form of recipes, provides needed tarballs of software, fills in the metadata, and performs tests of the package using local repository. Then they can upload a cookbook to the central repository using *Knife*.

The process of environment preparation is performed on the node by the client application. To describe the idea of a simple Chef run in a nutshell, the process of deployment is presented omitting unnecessary details:

- **Client installation on the node.** The Chef client application can be installed on the node either manually or automatically using *Knife* application. Chef supports automatic installation (bootstrap) of a client on a number of popular operating systems. In addition to that it is possible to provide your own installation script, and extend functionality of Chef. After the step of client installation, a node become visible for the server.
- **Updating nodes' Run List and populating node attributes on the Chef server.** In order to do that, you may use Knife command or Chef server web user interface/REST API. Run List and attributes are stored on server as an additional node parameters.



- **Execution of the installed Chef client.** The client can be run either manually by executing shell command on a node machine (eg. via ssh) or using Knife.
- **Client downloads the Run List and attributes from server.**
- **Client performs the installation.** Now the client invokes all of the scripts provided by the server.
- **Client executes handlers** in order to return statuses and logs of the installation process.

### 2.2.3 Chef based tools

Chef provides cloud provisioning automation functionalities, and several methods to achieve them. Chef server REST API [48] can be used to obtain cookbooks, recipes and node information, to check repository content and perform similar tasks. To manage cookbooks stored in the central cookbook repository, bootstrap and start client, receive installation logs and so on, a command line tool called *Knife* is provided. The main disadvantage of Chef regarding the project design is the fact that it does not provide any API regarding software installation.

It should be mentioned that there is a third-party library called *jclouds-chef* [33] that provides convenient Java and Clojure API for Chef. It is neither a part of Chef project nor JClouds (which is pretty popular provider agnostic library used to access cloud) and it lacks a decent documentation. Anyway it seems that *jclouds-chef* API can cover most of the tasks carried out by Chef REST API and Knife. Experimentation with the library will be presented later as a part of prototype description.

## 2.3 Summary

In this chapter a number of provisioning tools were presented. As all of them provide different flavors of similar functionalities it is hard to compare them without experience in usage. Moreover a process of making comparison may be really time consuming. As many administrators managing their infrastructure using provisioning software usually recommend the solution they are familiar with, it is even hard to find a reasonable comparison of tools. What is more, most of the articles [55] [41] [46] that present some comparison are highlighting several features of a particular system, and neglect the others, so it is really hard to find a common domain of features of provisioning tools that allow to fairly compare the software products. Therefore, in order to compare selected solutions there was chosen some factors in terms of which the systems can be analyzed and compared. The deciding factors in the comparison were - the license, the ease of possible integration of a

## 2. PROVISIONING IN CLOUD INFRASTRUCTURE

---

product with a code that is written in Java and support for Windows operating system. These three factors favor Chef among the others, because of Apache License, third party Java API, and support for several systems from the Microsoft family. So that, although probably all of the presented solutions would be applicable for the given task, Chef was chosen to be used. In this chapter was also introduced some vocabulary that is specific to Chef, and there was presented Chef architecture in a nutshell. In the further part of this work it was assumed that the reader is familiar with the content of this chapter.

## Chapter 3

---

# Software Product Line as a generic approach to software creation

---

*Software Product Line is a paradigm of software creation, defining software production as a process based on an analogy to production line. In this chapter the concept of this analogy is presented in order to apply it to the process of e-Science application production. As the design of the system built in scope of the thesis follows some of the Software Product Line principles, some of the basic SPL concepts of the methodology are presented in this chapter. This chapter introduces a language for production line configuration modeling, data structures to represent the model and algorithms for automating product configuration. Then, the review of approaches to Feature Modeling adaptation in software product line is presented. The chapter ends with the conclusion on the selection of tools that is used in the process of the tool implementation.*

### **3.1 The concept of using Software Product Line methodology**

Software product line (SPL) engineering is a paradigm for systematic reuse [56]. Product line engineers define common assets, from which different programs of a domain can be assembled. Programs are distinguished by features, which are domain abstractions relevant to stakeholders and are typically increments in program functionality [56]. Each program is defined by a unique composition of features, and no two programs have the same combination of features. In order to define domain of software products the notion of feature model was introduced. Model based on features allow to present an abstraction of software product components, their hierarchy and dependencies. As Feature Model is a generic concept and there is no limitation of the model semantics, it can be found an appropriate representation of product domains for various product lines. When we take a look at Software

### 3. SOFTWARE PRODUCT LINE AS A GENERIC APPROACH TO SOFTWARE CREATION

---

Product Line Online Tools web page [51] there we can find examples of feature models defining TREK's Bikes Catalog and DELL's Laptops Catalog. So as the nature of Feature Model is very generic, its flexibility was a main reason to think about adaptation of Software Product Line concept in the process of automatic creation of e-Science application environment on a cloud. The second reason was the quite obvious fact of inclusion of the e-Science applications domain in the domain of software products. As the notion of e-Science application can be understood as a software product built from a number of reusable software components, it seems to be another aspect bringing it closer to the concept of Software Product Line.

The circumstances mentioned before lead to the idea of thinking of experiment environment deployment in terms of creating a product in a production line. The concept presented in this thesis is based on the assumption, that we can treat the components of experiments environments as features of environment. As it is presented in the chapter regarding target system design and implementation, in context of using Chef as a basis of the system implementation, the term of cookbook (refer to Chef documentation) is conceptually not that far from the idea of feature. Actually a cookbook represent means of installation of environment components connected to a single entity uniquely identified by name. Therefore, in the further reading the notion of feature can be regarded as a entity that can be mapped to cookbook in the final system implementation. Nevertheless, before any concept of implementation will be introduced there is a need to provide a description of several aspects connected with Feature Modeling.

In the following chapters there will be presented the Feature Model notation, data structures of model representation, basic algorithms of automatic reasoning on feature models and tools that will allow to make use of feature modeling in the application of e-Science application deployment.

## 3.2 Overview of Feature Modeling

### 3.2.1 Feature Model notation

A "feature" is defined as a "prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system" [35]. Features express the variabilities and commonalities among programs in a software product line. Feature model represents the information of all possible products in a product line in terms of features and relationships among them. A feature model is represented as a hierarchically arranged set of features which is similar to a tree. Relationships between features can be expressed in two forms [8]:

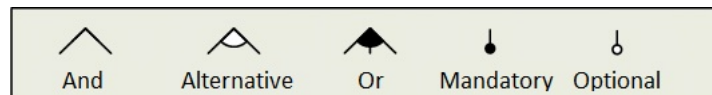
- As relationships between a parent (or compound) feature and its child features (or subfeatures). Relationships form a tree-like structure of a model.

- As Cross-tree (or cross-hierarchy) constraints that are typically inclusion or exclusion statements in the form: if feature F is included, then features A and B must also be included (or excluded).

Feature Models Relationships between a parent (or compound) feature and its child features (or subfeatures) are categorized as:

- **And** - all mandatory subfeatures must be selected. In the example in Figure 3.2, a mobile phone consists of 4 main features - Calls, GPS, Screen and Media.
- **Alternative** - only one subfeature can be selected. In the example in Figure 3.2, mobile phones may include support for a basic, color or high resolution screen but only one of them.
- **Or** - one or more can be selected. When cell phone supports Media, Camera or Mp3 must be chosen.
- **Mandatory** - features that are required. For instance, every mobile phone system must provide support for calls.
- **Optional** - features that are optional. In the example, software for mobile phones may optionally include support for GPS.

**Or** relationships can be extended with additional information concerning cardinalities: n:m - a minimum of n features and at most m features can be selected.



**Figure 3.1.** Graphical representation of feature relationship types. The image is based on a picture presented in [8].

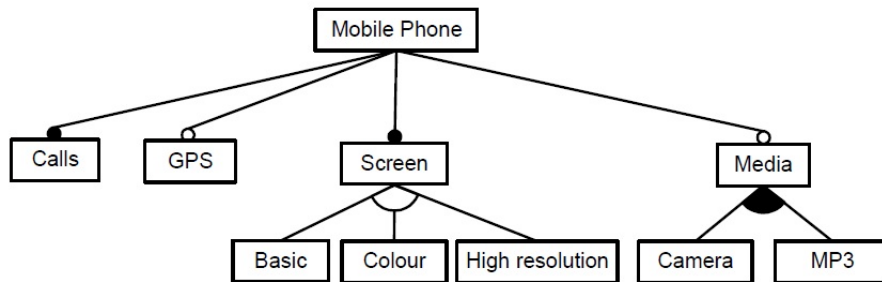
Cross-tree constraints were introduced to represent require and exclude restrictions. In our example Camera feature may require Screen with High resolution and GPS can exclude Basic Screen. In order to represent cross-tree constraint boolean expressions are used (eg. clause in form of Conjunctive Normal Form, where each feature corresponds to a different literal).

### 3.2.2 Proposed operations on feature models

In order to bring feature based software to life it is necessary to properly process feature model. If you imagine that a feature model is large it becomes quite a complex problem to define product domain or find out if product configuration is valid for the model. However, quite a lot of research on how

### 3. SOFTWARE PRODUCT LINE AS A GENERIC APPROACH TO SOFTWARE CREATION

---



**Figure 3.2.** Sample feature model describing a product line from telecommunications industry [8]. The model presents hierarchical structure of mobile phone features. In order to include some additional constraints in the model, cross-tree constraints may be provided.

to deal with similar issues has been already made. Based on the work [8], several operations meant to be performed by different feature-model-based programmatic tools can be mentioned. This summary is presented to point out the range of issues connected with feature model processing.

- **Void feature model** - check if feature model describes any valid configuration.
- **Valid product** - check if configuration of software product is valid for a given feature model.
- **Valid partial configuration** - check if partial configuration is valid and allows for further selection of features.
- **All products** - count products.
- **Filter** - limit the domain of configuration by providing constraints.
- **Anomalies detection** - Dead features, Conditionally dead features, False optional features, Wrong cardinalities, Redundancies.
- **Explanations** - find a reason for model/configuration validation or other failure.
- **Feature model relations** - comparison between models (Refactoring, Generalization, Specialization, Arbitrary edit).
- **Optimization** - guidance on feature model edits for optimization of operations.
- **Core features** - features that are selected in every configuration.

### 3.3. Families of Feature Model reasoning algorithms



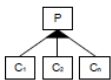
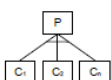
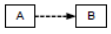

- **Other** - for further information refer to [8].

Regarding objectives of the project some of the operations are useful, some of them turn out to be needles. Decisions on how to adapt feature model concept to e-science application component connection configuration will be presented later.

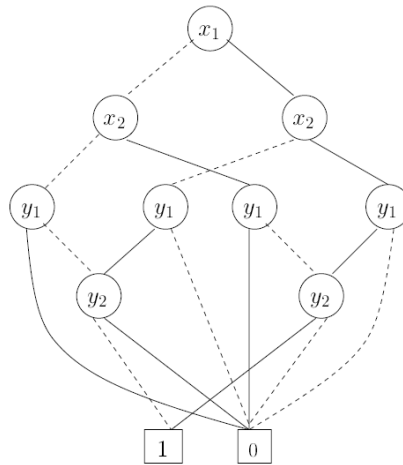
### 3.3 Families of Feature Model reasoning algorithms

In order to perform automatic reasoning on a feature model, there is a need to provide its formal representation. According to the research that was made in this area it seems that there are two main approaches to feature model representation and two corresponding algorithm families of efficient reasoning on that model. This section presents a problem of boolean satisfiability (SAT and Binary Decision Diagrams (BDD) as these two families are mainly used in this scope.

#### 3.3.1 Constraint Satisfaction Problem

	Relationship	PL Mapping	Mobile Phone Example
MANDATORY		$P \leftrightarrow C$	MobilePhone $\leftrightarrow$ Calls MobilePhone $\leftrightarrow$ Screen
OPTIONAL		$C \rightarrow P$	GPS $\rightarrow$ MobilePhone Media $\rightarrow$ MobilePhone
OR		$P \leftrightarrow (C_1 \vee C_2 \vee \dots \vee C_n)$	Media $\leftrightarrow$ (Camera $\vee$ MP3)
ALTERNATIVE		$(C_1 \leftrightarrow (\neg C_2 \wedge \dots \wedge \neg C_n \wedge P)) \wedge$ $(C_2 \leftrightarrow (\neg C_1 \wedge \dots \wedge \neg C_n \wedge P)) \wedge$ $(C_n \leftrightarrow (\neg C_1 \wedge \neg C_2 \wedge \dots \wedge \neg C_{n-1} \wedge P))$	(Basic $\leftrightarrow$ ( $\neg$ Color $\wedge$ $\neg$ Highresolution $\wedge$ Screen)) $\wedge$ (Color $\leftrightarrow$ ( $\neg$ Basic $\wedge$ $\neg$ Highresolution $\wedge$ Screen)) $\wedge$ (Highresolution $\leftrightarrow$ ( $\neg$ Basic $\wedge$ $\neg$ Color $\wedge$ Screen))
IMPLIES		$A \rightarrow B$	Camera $\rightarrow$ Highresolution
EXCLUDES		$\neg(A \wedge B)$	$\neg$ (GPS $\wedge$ Basic)

**Figure 3.3.** The rules for translating Feature Model relations into Boolean formulas [8], in order to represent the model in a form as a Boolean Satisfiability Problem. The third column is a mapping of relationships presented as an example in the Figure 3.2.



**Figure 3.4.** Sample Binary Decision Diagram [2]. Round nodes represent variables (in this case they are product features). Squares external nodes denote values of the formula. Solid edge represents 1, dotted - 0. A path from the root to an external node is a representation of a variable values.

Constraint satisfaction (CSP) [37] is a mathematical problem defined as a set of variables, and a set of constraints the variables must satisfy. A solution to the problem is a vector of variables that satisfies all the constraints. Constraint satisfaction can represent numerous practical combinatorial problems including scheduling, planning and configuration. Moreover, recently CSP is being applied to other domains such as natural language processing (for construction of parsers), computer graphics (for visual image interpretation), biology (for DNA sequencing), business applications (trading) and others .

Boolean Satisfiability (commonly known as SAT) is a case of constraint satisfaction problems in which all the variables are boolean. To adapt boolean satisfiability problem to automatic reasoning on feature models, we treat each feature as a variable (see the Figure 3.3). A value of a variable will be 1 when the corresponding feature is selected by user in the configuration process, 0 will be assigned otherwise.

In order to find a solution to a constraint satisfaction problems, constraint solvers are used. Many different algorithmic techniques are applied by modern constraint solvers such as backtracking search, local search, and dynamic programming. In this review I will not focus on the specific type of algorithm as this is not the main point of interests regarding the thesis specificity.

### 3.3.2 Binary Decision Diagrams

Binary decision diagrams [2] are compact encodings for Boolean formulas that provide numerous efficient reasoning algorithms. BDDs have been



widely explored in many research areas such as model checking, formal verification, optimizing, etc.

In terms of data structure BDDs are directed acyclic graphs (DAGs) having exactly two external nodes representing constant functions 0 and 1, and multiple internal nodes labeled by variables. Each variable node has two outgoing edges representing a decision based on the variable value assignment. Solid edge represents 1, and dotted denote assignment to 0. A path from the root to an external node represents a vector of variable values. So a "configuration" of variable states is valid when the corresponding path exists in a decision diagram.

Example of Binary decision diagram is presented in the picture 3.4.

#### 3.3.3 SAT vs BDD

The advantage of BDDs over SAT solvers is a great performance of some BDD algorithms once the BDD structure is built. For example, while for a SAT solver it will take some noticeable time to count the number of possible solutions for a given problem, BDD can perform this operation very efficiently. Moreover, a single check of a Boolean formula is linear with the formula size using BDDs, while this is a NP-hard problem for SAT solvers. Finally what is especially important in interactive configuration, where system updates the available options while the user makes configuration decisions, there exists efficient BDD algorithms for calculating valid domains [28].

According the description above, BDD solvers seem to be perfect for the interactive configuration of a product. Unfortunately the structure of BDD have a significant drawback. The graph represents whole combinatorial space, which in comparison to SAT may cause huge growth in memory utilization of the solver. This may result in a situation, in which the representation of the model is exponentially larger than the number of variables, which may be unacceptable.

#### 3.3.4 The areas of application

Marcilio Mendonca in his work [39] presents following comparison of some of the mentioned operations being performed on the feature model. Based on literature (listed in the work) and his own experience he divides them into several groups and presents which feature model reasoning algorithm best fits each operation. Summary of Feature Model Reasoning Activities and Operations [39] with the suitable data structure/algorithm family is presented below:

1. Debugging :
  - Checking satisfiability of models (SAT),
  - Detecting if a given feature is "dead" (SAT),

### 3. SOFTWARE PRODUCT LINE AS A GENERIC APPROACH TO SOFTWARE CREATION

---

- Detecting "dead" features (SAT, BDD).
2. Refactoring:
    - Checking equivalence of feature models (SAT, BDD),
    - Checking extension of feature models (SAT, BDD).
  3. Configuring:
    - Checking specialization of feature models (SAT, BDD),
    - Validating partial or full configuration (SAT),
    - Calculating valid domains (BDD),
    - Enumerating one or more valid configurations (SAT, BDD),
    - Resolving decision conflicts in collaborative configuration (SAT, BDD).
  4. Measuring:
    - Counting valid configurations (BDD),
    - Computing variability factor (BDD),
    - Computing commonality of a feature (BDD).

#### 3.3.5 Comparison summary

Summing up the above considerations, it seems that BDD solver should be used whenever it is possible in context of feature model-based product configuration. The only limitation of this approach is memory utilization factor and a time of DAG model representation creation computational overhead. If the model representation will be generated only once and stored in the memory, a BDD solver will be more suitable for most of the operations performed on a feature model, than a solver based on SAT. However, if there is a need for multiple model object storage, serialization should be considered instead of model creation on demand. Although the BDD solvers may deliver better performance, if there is a need for ad-hoc model object creation, using SAT solver may be also reasonable.

### 3.4 Solvers based on Boolean Satisfiability Problem and Binary Decision Diagrams

There are several popular SAT and BDD solvers that may be suitable to perform reasoning on feature models that enable user to easily cope with software product configuration. This section presents three different Java-based reasoners in order to outline their capabilities. Each of the solvers uses a different paradigm, so they cannot be equally compared. The purpose of this

review is a presentation of part of the work that was done to analyze the possibility of using following libraries in the development process. As the next chapter shows, there exist some reasoning tools, which are dedicated to feature models and are more suitable to be used in target application development.

#### 3.4.1 SAT4J

Sat4j [47] is an open source library of SAT solvers providing cross-platform SAT-based solvers written in Java [38]. It is used by various Java-based software, in the area of software engineering, bioinformatics, or formal verification [38].

As a documentation claim a SAT solver in Java is about 3.25 times slower than its counterpart in C++, but the library is not focused on performance but to be fully featured, robust, user friendly and extensible [47].

SAT4J has been successfully applied to analysis of large feature models by Marcilio Mendonca [40].

#### 3.4.2 JavaBDD

JavaBDD [32] is a Java library for manipulating BDDs (Binary Decision Diagrams). The JavaBDD is providing Java API based on that of the popular package written in C language - the BuDDy. JavaBDD includes a pure Java BDD implementation and it can also be used as an uniform interface to access other libraries. JavaBDD provides an additional layer allowing to seamlessly switch between JDD Java-based library, and BuDDy, CUDD, and CAL libraries, which are written in C [32].

JavaBDD is designed for high performance applications, so it also exposes many of the lower level options of the BDD library, like cache sizes and advanced variable reordering [32].

#### 3.4.3 Choco reasoner

Choco is a Java library for constraint satisfaction problems (CSP) and constraint programming (CP). It is built on an event-based propagation mechanism with backtrackable structures [12].

It is an open, user-oriented solver which provides a separation between model and solver. It paves the way to provide a general problem solving library not necessarily dedicated to constraint programming [13]. Choco can be used for [12]:

- teaching (a user-oriented constraint solver with open-source code),
- research (state-of-the-art algorithms and techniques, user-defined constraints, domains and variables),

### 3. SOFTWARE PRODUCT LINE AS A GENERIC APPROACH TO SOFTWARE CREATION

---

- real-life applications (many application now embed CHOCO).

#### 3.4.4 Evaluation of the solvers

As it is shown in the table 3.1, all of the above solvers can be used to implement majority of the operations taken into account. As the FaMa team [24] implemented all of the presented operations using Choco solver it may be considered the most flexible in terms of automatic reasoning on feature models. However, the research in a domain of various solvers shows that there are some tools that are more suitable for the given task. A comparison of the tools designed specifically to manipulate feature models is presented in the next section.

Operation	SAT4J	JavaBDD	Choco
Validation	✓	✓	✓
Products	✓	✓	✓
Number of products	✓	✓	✓
Commonality	✓	✓	✓
Variability	✓	✓	✓
Valid product	✓	✓	✓
Valid configuration	✓	✓	✓
Detect errors	✓	✗	✓
Explain errors	✓	✗	✓
Product explanation	✗	✗	✓
Core features	✗	✗	✓

**Table 3.1.** A summary of operations on Feature Model that was implemented in [24] using SAT4J, JavaBDD and Choco reasoner. The table shows that Choco solver is suitable for all of the listed operations.

### 3.5 Tools for Feature Model reasoning

In the previous section there were presented three libraries that allow to analyze a feature model after translating the operations on a model into an appropriate combinatorial problem. The tools introduced in the current section can help to perform similar tasks, providing feature model centric interfaces. What it means, using them there is no longer needed to provide a special conversion of operations referring to a feature model. This approach results in more "blackbox feel" of a library, gives the user less control, but on the other hand abstracts the operation and significantly eases the use. Although in this section description of three popular tools will be provided, the last one will be emphasized because of its special suitability for implementation of the tool.

### 3.5.1 AHEAD tool suite

AHEAD [54] (Algebraic Hierarchical Equations for Application Design) model was presented by Don Batory, Jacob Neal Sarvela and Axel Rauschmayer in the work [5]. It is an architectural model for feature oriented programming (FOP) and a basis for large-scale compositional programming [54].

The AHEAD Tool Suite is a set of tools that support Feature Oriented Programming. ATS can be used to build a product line for software products reusing modularized units implementing features. The suite requires the use of extended Java languages and provides tools for [54]:

- step-wise refinement and compositional programming,
- metaprogramming,
- extending programming languages (e.g. Java).

Although AHEAD Tool Suite is basically regarding "Feature Oriented Programming" (in the sense of mixin code) some of the utilities are suitable for feature model edition and reasoning. As an example `guidsl` tool can be mentioned. Features of AHEAD Tool Suite are listed below:

- Model Debugging,
- Model Checker,
- Variable Table and Propositional Formulas,
- Variable Value Explanation,
- Saving Equations and Configurations.

A significant drawback of AHEAD is the fact that its components are accessed as a shell command. It implies an obvious inconvenience of using it as a part of another application (input/output parsing, process lifecycle management, etc.).

### 3.5.2 FaMa tools

FaMa [23] (FeAture Model Analyser) is a tool to analyse feature models. FaMa accepts models written either in XML or plain text Feature Model Format.

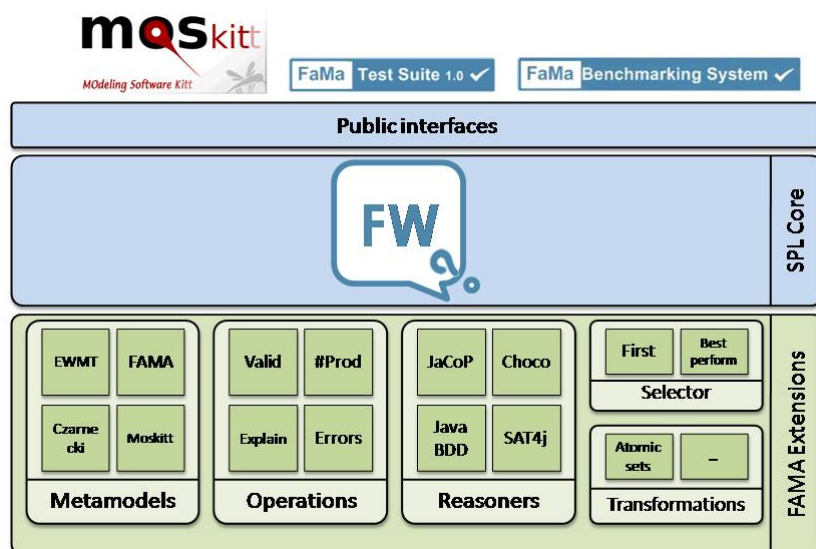
FaMa allows to perform following operations on feature model [24]:

- Validation - checks if a model is not empty.
- Products - calculates all valid products of a feature model.

### 3. SOFTWARE PRODUCT LINE AS A GENERIC APPROACH TO SOFTWARE CREATION

---

- Number of products - calculates the number of products.
- Commonality - calculates appearances of a given feature into the list of products.
- Variability - calculates variability degree of a feature model.
- Valid product - determines if a product is valid for a given model.
- Valid configuration - analyses if configuration that has not been completed yet is valid.
- Error detection.
- Error explanations - looks for relationship of conflicting features.
- Invalid product explanation - provides options to repair a invalid product for a given model.
- Core features - calculates features that are present on every product.
- Variant features - calculates features that are not present on every product.



**Figure 3.5.** FaMa modular architecture [22]. As one can see there are four reasoners that can be used to operate on feature models. FaMa is built as a Software Product Line, so the library not only supports the methodology, but also is an example of its implementation.

As one of the main advantages of FaMa the versatility of usage should be mentioned. FaMa tools can be accessed in four ways [24]: FaMa shell, FaMa Web Service, FaMa OSGi, FaMa standalone.

Moreover, FaMa design is modular and the tool suite is extendable by implementing reasoners, metamodels or transformations, and integration through OSGi. For now FaMa allows to use four reasoners: `choco`, `sat4j`, `jacop` and `javabdd`. Each of the operations on feature model has a number of implementations using different reasoners.

### 3.5.3 Software Product Lines Automated Reasoning (SPLAR) library

SPLAR [49] uses both SAT and BDD models to cope with memory/time intractability problems regarding reasoning. The library is particularly valuable for the project in terms of comparing both models. SPLAR was built as part of Marcílio Mendonca's PhD thesis [39] at the University of Waterloo and offers state-of-the-art BDD variable order heuristics designed to reduce BDD structure as much as possible. The library includes also SAT-based algorithms. SPLAR includes implementation of the operations such as valid domain computation, conflict detection, configuration auto-completion and others [49]. The library is written in Java and provides Java API, so it is convenient for the usage inside the Java code.

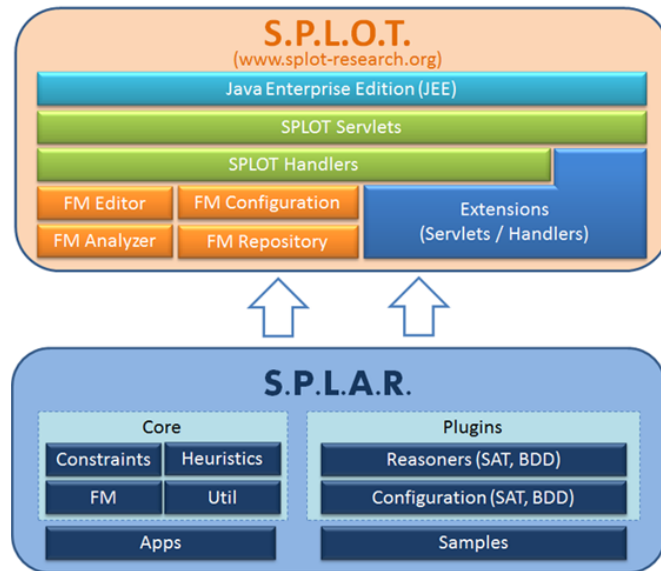
SPLAR was used to create a quite popular, open source, web application, which can be used to create, configure and analyze Feature Models - SPLOT (Software Product Line Online Tools) [51]. Moreover, on November 2010 SPLAR went open source so it is another favoring factor in context of the project needs.

What has to be outlined SPLAR can be particularly valuable for the thesis because of several aspects connected with its connection with SPLOT. First of all SPLOT is ready-to-use application that can be used for testing the prototype of the developed tool. It allows to generate feature models using an intuitive web interface and export them to SXFM format (which tends to be more and more popular). Once a model is created it can be saved in the online repository and tested. In order to test the feature model the product configuration web interface can be used by a user for selection of certain features. Each configuration can be exported to XML and CSV format. As all of the source code is freely available, some parts of the application can be reused in early phases of the implementation of the prototype. For example while using SPLAR to perform operations on feature model, SPLOT can be used to generate, export and re-read sample models or configurations. This can allow to focus on the development of deployment mechanism for the application being created.

Another promising factor of using SPLOT is the fact that it is implemented using Freemarker [26] (a template engine) and Java servlets [31] fol-

### 3. SOFTWARE PRODUCT LINE AS A GENERIC APPROACH TO SOFTWARE CREATION

---



**Figure 3.6.** The architecture of SPLOT [49]. The application is a web based tool allowing to perform various operations on feature models. Under a web user interface there is an application using SPLAR library implementing the logic of operations. As one can see in the diagram, SPLOT is based on JavaEE and uses Servlets to handle HTTP requests.

lowing the MVC pattern, so the code of View and Controller are well separated. This allows to easily reuse some interesting parts of the code.

What is more, as SPLAR library implements both SAT and BDD solver engines, the source code of the developed prototype can be easily modified to adapt a reasoner to specific memory utilization / efficiency requirements. As SPLAR is also open source, it opens possibilities of modifying the model representation which is very promising for the future work.

#### 3.5.4 Summary of the reasoners comparison

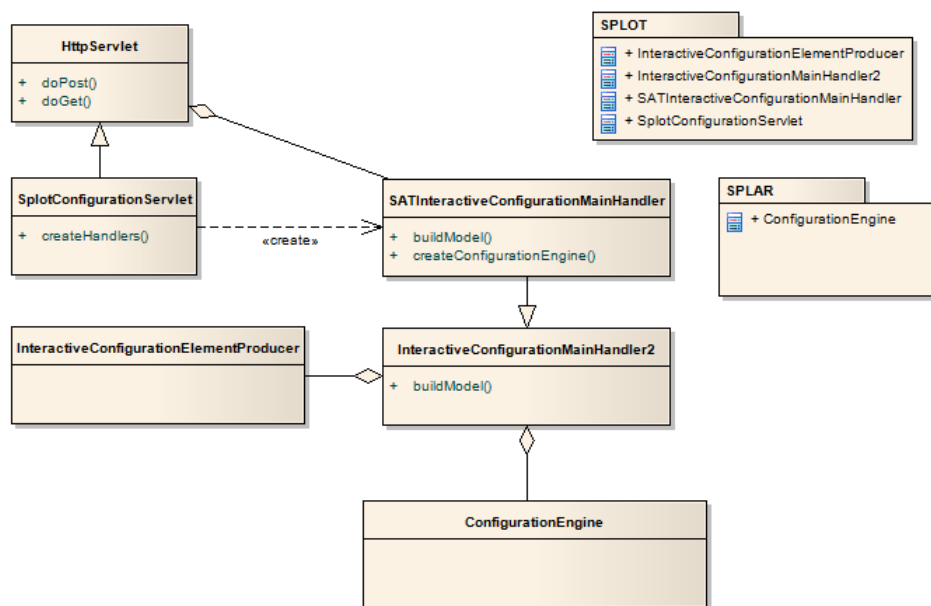
As all of the presented Feature Model reasoning libraries provide different flavors of similar operations on feature models, the comparison was focused on the adaptation profits of the particular tool. Taking the considerations presented in the section 3.5.3 into account, there was made a decision of testing SPLOT and SPLAR in the phase prototype development. As it was decided to use these tools the next section presents several aspects of SPLOT design in a nutshell.



### 3.5.5 Evaluation of Software Product Line Online Tools

As it was mentioned before, SPLOT [51] is an online tool that allows to perform several operations on feature model starting from its creation, ending on analysis of the structure of feature interconnections. From the perspective of the software product created in the scope of this thesis, the most interesting are the operations of creation and saving a feature model, selecting features in product configuration and finally exporting final configuration of a software product.

Once a feature model is created in SPLOT it can be exported to the SXFM format [52] (see the example below). SPLOT provides a parser, which allows to read and write the feature model to a file. As it was mentioned before it was decided to use SPLOT and SPLAR in order to build the target system prototype. It was also decided that the prototype (see the section regarding the Proof of Concept - 5.1) will be based on both - the SXFM model format and the XML configuration format.



**Figure 3.7.** A diagram of the core SPLOT classes. A single servlet uses multiple handlers. The source code of handlers can be reused quite easily.

In order to find out whether it is possible to make use of any of SPLOT web user interfaces, the source code was analyzed in terms of reusability, and the following architecture description was made. As it can be noticed, the most convenient approach would be using whole configuration interface of SPLOT in order to focus on implementation of experiment environment installation. In order to adapt the user interface of product configuration,

### 3. SOFTWARE PRODUCT LINE AS A GENERIC APPROACH TO SOFTWARE CREATION

---

there was a need to extract an appropriate module. SPLOT is built based on Java Servlets and as it is presented in the Figure 3.7, the product configurator has only one entry point - `SplotConfigurationServlet`. The servlet adapts its behavior according to the "action" parameter of HTTP GET method, and runs an appropriate handler. It means that even if it is necessary to write from scratch some code to handle HTTP requests, there is still a possibility to reuse the source code of the handlers.

Some handlers use SPLAR ConfigurationEngine (as `InteractiveConfigurationMainHandler2`) which is instantiated each time a new model is loaded. SPLOT uses SAT-based model reasoning algorithms for models created ad-hoc. At this point it should be mentioned that if a model is large enough to cause efficiency problems of SAT reasoner, using a Binary Decision Diagram generated and kept in memory should be considered.

As a summary of a study on SPLOT and SPLAR library, it can be said that both tools appear to suit well the tool design. SPLOT source code appear to be promising in terms of reusing user interfaces that can be extracted quite easily. SPLAR is used by the SPLOT for model creation and product configuration, so it proves that SPLAR provides sufficient operations to perform tasks that will be implemented by the tool developed in scope of the thesis. Moreover, SPLAR gives a programmer an opportunity to switch between two reasoning modes - using BDD or SAT-based reasoner - which allow to adjust the implementation to specific requirements for time of operation and memory utilization.

### 3.6 Summary

This chapter presents application of Software Product Line in the domain of e-Science. The concept is based on modeling the configuration of e-Science application components as a set of features that form a Feature Model. In order to introduce means of production for the production line, the idea of mapping from the space of e-Science application features to the space of environment component installation packages was described. Next, there were presented a few low-level approaches to perform operations on feature models that may be useful from the perspective of product configuration. As during the research it was discovered, there exist some Feature Model specific solutions that was found to be more useful for the implementation of the tool. As the SPLAR library was evaluated to be particularly valuable, there was made a decision to use it in the implementation. Chapter 4 presents how the SPLAR and SPLOT are adopted by the tool built in scope of this thesis.

## Chapter 4

---

# Experiment environment preparation tool overview

---

*In this chapter the detailed vision of the application created in scope of this thesis is presented. Starting from a general system description (section 4.1), the reader will go through the specification of requirements (section 4.2), to the description of the system design (section 4.3). The chapter should give the reader a high-level look at the system, before the actual description of implementation will be introduced.*

### 4.1 Cloudberries general description

In order to introduce the concept of the tool design, it is needed to describe what it is designed for. As it was mentioned before the system has to cope with the complexity of e-Science application installation. What this really means is the need to provide a tool that will allow a scientist to select, configure, and install components of the experiment environment in an easy and intuitive way. It also means that the set of environment components that can be installed using the system, will be easily extensible.

Cloudberries is the code name of a tool created in the scope of this thesis and this name will be used in the further reading. The Cloudberries is an tool that allows the end user to see a set of reusable components (organized in a certain way, and managed by the administrator), select a few of them, specify attributes (such as installation path, etc.), and deploy the corresponding experiment environment on the cloud infrastructure.

Looking at the tool in terms of requirements, it should be mentioned that the configuration selected by the user has always to be valid. So there is a need for environment component dependency management to cope with that problem. That's why feature modeling was engaged to express connections between elements of configuration. In the context of the reasoning that was presented before, it can be said that a developed system should

be treated as a Software Product Line, managed by an administrator. In the nomenclature of the SPL a single experiment environment can be regarded as a product, with its components as features. The scientist will see a graphical representation of a Feature Model and be able to decide which components he would like to include in his experiment environment.

Behind the user interface there will be working a provisioning system as a framework for automation of software installation. As it was mentioned before, after a comparison of several tools, Chef was chosen to be used.

In the following sections a detailed specification of requirements and the design of the system will be presented.

## 4.2 Specification of requirements

In this section a specification of requirements concerning the application is presented. Although the requirements come mainly from general description of application of the tool, as it was meant to be deployed in the VPH-Share, there are some additional limitations originating from the project.

### 4.2.1 Functional requirements

Base requirements:

- The application has to provide a function of loading model that specifies dependencies between features (environment components).
- The application has to determine whether the model is syntactically valid or not.
- The application has to provide an ability to supply a deployment script for a feature.
- The application has to determine whether the feature model is valid in terms of deployment capability. For each installable feature an installation script must be provided.
- The application should allow to view a feature model represented as a tree with additional requirement/contradiction integrity constraints.
- The application has to allow to create an environment configuration. Configuration process should be based on graphical representation of feature model.
- The application has to allow to save created configuration.
- The application has to determine whether configuration is valid or not. Validation should allow to avoid a situations when two selected features conflict in context of a feature model.

- The application has to allow to load a saved configuration.
- The application has to be able to deploy a configuration of environment components on the Virtual Machine in a cloud.

Additional requirement (not critical):

- The application may allow to edit feature model through the graphical interface.

### 4.2.2 Non-functional requirements

- Accessibility - multiple user, non-exclusive access.
- Configuration - Feature Models (eg. SXFM [52] notation) for feature relationship constraints. Plug-able installation scripts. Additional tool functionality may cover the need for feature model edition.
- Dependencies - future integration with the VPH-Share project user interface.[58].
- License - open license software use only.
- Platform - multiple cloud platform support. Possibility of extending for next vendors. Unix/Linux as an operating system, Java language (JEE). JetSpeed [34] portal integration may be considered.
- Security - user interface security can be provided by the JestSpeed portal. The system should be protected against interference from the outside without permission.
- Usability - user interface should be as easy and intuitive as possible. The less time spent on configuration of an experiment environment the better is the frontend.
- Documentation - Javadoc and in-code documentation (English language), user's manual (English), MSc thesis documentation (English) has to include architectural design and description of implementation.

### 4.3 High-level system design

In previous sections there was introduced a general description of the tool and requirements that had a great influence on the project planning phase. At this point, the design of the final version of Cloudberries should be presented. In this section there are described the user roles, their capabilities and main elements of the system architecture. This paragraph can be treated as a starting point to familiarize with Cloudberries architecture.

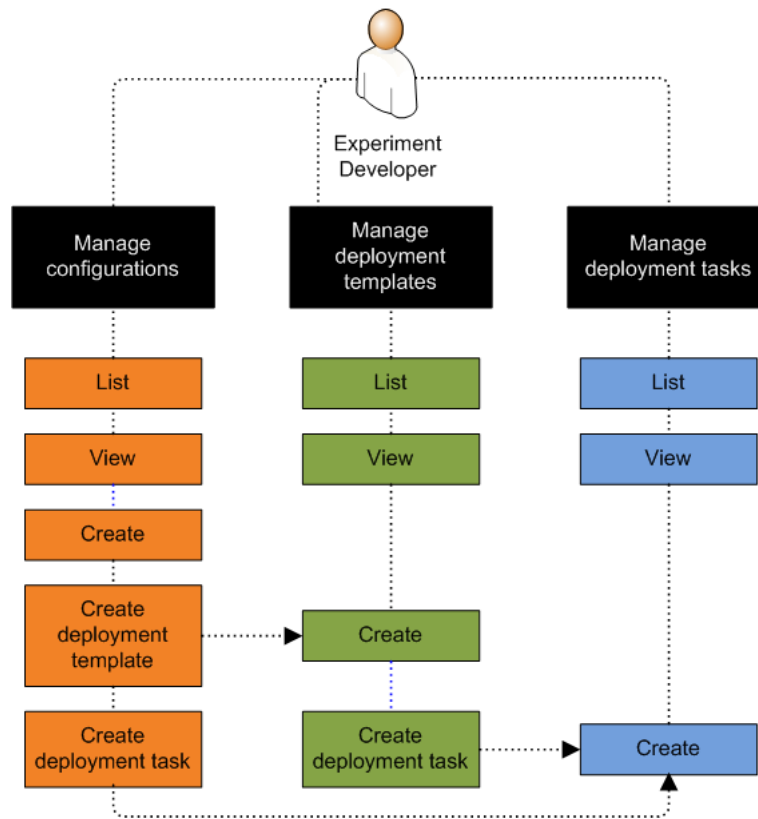
Cloudberries is a system designed to be used by two types of users:

- **Experiment Developer** - this user represents a scientist. In terms of the system design Experiment Developer is the end user of the tool. His needs can be described as a preparation of experiment environment in an easy and intuitive way.
- **System Administrator** - the administrator of the system that takes care of the means of environment preparation. He provides components (in the form of installation scripts) and component dependency structure (represented by Feature Model).

The end user (Experiment Developer) is provided with a graphical interface accessed via web browser (Figure 4.1). Once the user is logged in, he can manage three types of Cloudberries specific entities:

- **Configuration** - a selection of environment components that will be installed on the target Virtual Machine.
- **Deployment template** - a configuration of components, enhanced with attributes. For each installation components there are some additional parameters that can be specified by the user (eg. installation path, log folder, initial credentials). By specifying deployment template, default component attributes are overridden by the user.
- **Deployment task** - when the user selects a configuration of components or a deployment template, he has to provide Virtual Machine IP address (for now the user cannot request virtual machine instance creation), adjust the parameters, and the installation process can be started. Once the user starts the installation, the user can monitor the progress by viewing corresponding deployment task.

Defining configurations and deployment templates allows a user to simplify experiment preparation process at different levels. Persistent configuration stores information about coarse grained decisions of environment components. Each time the user faces the necessity of installing similar environment he can open saved configuration, provide required parameters and run the



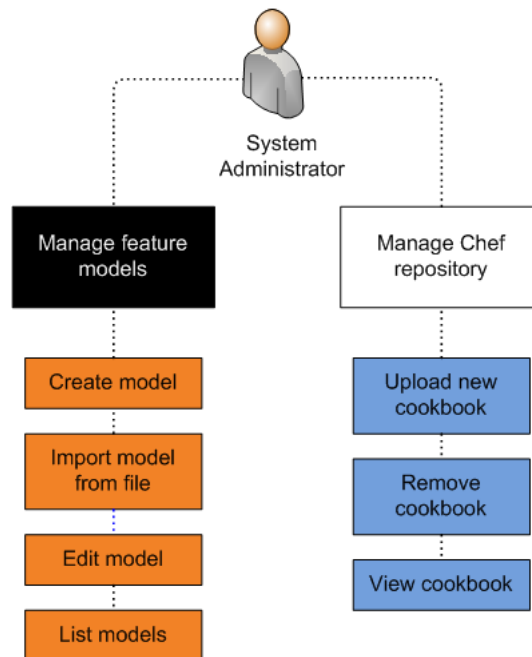
**Figure 4.1.** The design of Cloudberry experiment developer interface. Experiment Developer can manipulate three types of entities. *Configuration* is a selection of environment components that can be deployed in the experiment environment after providing necessary attributes. *Deployment Template* allows to store a composition of default values for the attributes. *Deployment Task* is a representation of running deployment process. The user can monitor an installation by viewing details of the corresponding *Deployment Task*.

installation process. If a user would like to predefine values for a selection of attributes and override attribute defaults (if such are specified in the cookbook metadata), they should take advantage of deployment template which will keep the defaults.

Each configuration, deployment template and deployment task must be named and may be described by the user. All the information are retained in the database and can be accessed again by selecting from adequate list. Once the components and assigned attributes are selected (either by using new/existing configuration, or retained deployment template), user can run the installation. This process starts from creation of a deployment task entity in the database. Deployment task can be observed through an Internet

#### 4. EXPERIMENT ENVIRONMENT PREPARATION TOOL OVERVIEW

---



**Figure 4.2.** The design of Cloudberries administration interface. The branch on the left side is an interface accessed via Web Browser, and the one on the right requires usage of linux shell.

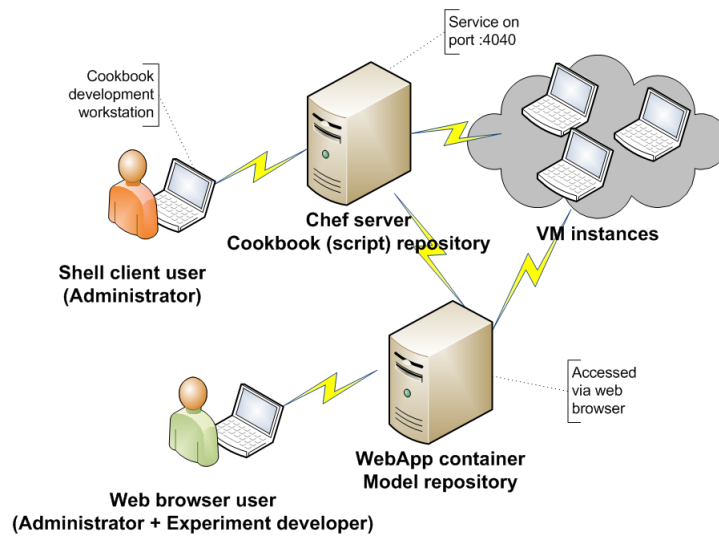
browser. The web interface provides information such as selection of components, parameter values, and state of deployment.

Before a user can configure their environment, the system administrator has to provide a hierarchical feature model of component dependencies (the format of feature Model is presented in the chapter concerning prototype development), and corresponding installation packages. The process of environment configuration is subordinate to Chef, so the installation packages have to be provided in form of a Chef cookbooks (to get more information refer to Opscode Chef documentation and Cloudberries user manual).

There are two types of user interfaces in terms of administering Cloudberries (Figure 4.2). The web browser interface (black in the picture below) was meant to allow management of feature models. As the implementation of this part of user interface was postponed a feature model loaded from a file is used instead. The second interface (white element) can be accessed by *Knife* command line tool. Each feature is bound by its identifier to a corresponding cookbook, which provides installation scripts, a definition of attributes, and some other stuff - refer to the Chef documentation [10].

Cookbooks are stored in the central cookbook repository managed by Chef server instance. Before the administrator can upload a cookbook to the





**Figure 4.3.** The architecture of Cloudberries. There are two basic components of the system - an instance of Chef server and the main Cloudberries application running in web application container. Both of these components need an access to the cloud infrastructure. The scientific user accesses Cloudberries using Internet Browser. Administrative tasks are performed using linux shell.

repository, he has to create and test it at his local workstation (have a look at the Chef architecture description - Section 2.2). Next, the cookbook needs to be added to his local cookbook repository and then uploaded using Knife (refer to Chef documentation [10]).

The architecture of Cloudberries in its simplest form can be presented as in the picture 4.3. The main component of the Cloudberries is a web application written in Java, running in a web application container (eg. Apache Tomcat). The application is in fact a portlet deployed in Jetspeed portal. All of the information collected by the system is stored in a relational database. Of course the Chef server and the web application container can be deployed on the same physical machine.

## 4.4 Summary

In this chapter there was introduced a vision of a tool to easily deploy an environment of e-Science application in the cloud infrastructure. Presented system should meet the requirements originating from the field of its application, the chosen approach to automation of the software installation (using Chef from a family of provisioning tools) and methodology used in the system implementation (Software Product Line). The system presented in this chapter can be described as a web based tool of software installation, which

#### 4. EXPERIMENT ENVIRONMENT PREPARATION TOOL OVERVIEW

---

is guided by the user, selecting features from the domain specified by a feature model. The implementation of the tool should be focused on easing the process of configuration, in order to satisfy the needs of non-technical users.

## Chapter 5

---

# Implementation of the tool

---

*This chapter presents selected aspects of implementation of the system that was built in scope of this thesis. First section describes details of the Proof of Concept implementation that was built in the phase of feasibility study. Presentation of the prototype familiarizes the reader with the course of the project, beginning from the assumptions that were made in the early phase of implementation. Next section presents the choice of technology used to create the tool, details of implementation and its operational rules. To confront ideas with the real appearance of the system some demonstrational screenshots are presented at the end of the chapter.*

### 5.1 Proof of Concept

The Cloudberry project realization took about eight months and besides State of the Art research the work may be split into two main phases - implementation of Proof of Concept prototype and actual implementation of the tool. The prototype was built as a number of Java classes, mainly to investigate capabilities of SPLAR library and Chef provisioning tool. It allowed to test some of the core functionalities of the tool. There are some elements of prototype that were inherited by the final version of Cloudberry. So that, this section presents several aspects of prototype realization before introducing description of a final version.

#### 5.1.1 Installation packages

Deployment of environment is performed using Chef. The features that are specified in a feature model represent components that can be included in a configuration. Each feature has a corresponding cookbook that has to be saved in the central cookbook repository of Chef server. Feature identifiers are mapped to the names of cookbooks stored in the repository.

### 5.1.2 Loading feature model

The prototype is able to load a feature model from a file in a form of extended SXFM format (to get more information about SXFM refer to the SPLOT documentation [52]). Extension affects the SXFM format only a little - exclamation mark is added when the feature should be treated as installable. Only installable features are analyzed while searching for a corresponding cookbook in the Chef repository. In order to read feature models the parser provided by SPLAR library was extended. The loaded model is kept by the SPLAR as a Java object providing the programmer with several operations. The object representation of the model is used in the final implementation of the tool in a process of configuration. Sample feature model in extended SXFM format is presented in the Figure 5.1.

### 5.1.3 Loading configuration

The prototype is able to load a configuration of features saved in XML generated by SPLOT (refer to SPLOT website). Each configuration corresponds to a certain feature model so it has to be validated in terms of compatibility with the model. A sample configuration is presented in the Figure 5.2

As one can see in the Figure 5.2, some of the features are selected by the user, and some of them are inferred automatically by SPLOT (*auto-completion, propagated*). The prototype implementation treats all of the features equally and matches them by the identifier with features in the model. The validation concerns name, type and correctness of the decision value in the context of model. When two configuration decisions conflict, the configuration is treated as invalid and deployment of environment fails.

The configuration is subjected to further validation in order to check if for each of the installable features a corresponding cookbook exists in the Chef repository. In order to do that jclouds-chef [33] library was used to communicate with Chef REST API.

### 5.1.4 Using provisioning API

As the design assumes Java is the main programming language used to develop the system. So that, there was a need for a solution that will allow to manage cookbooks, clients and nodes in a context of JVM. The obvious solution is system level java invocation of Knife shell command. In this case required Knife invocation parameters, such as Run List and cookbook attributes can be generated from java code and provided as a JSON file. The main drawback of this solution is obtaining responses from remote Chef-client. Using system command means that its output has to be read and processed (parsing command line output can be a challenge because of variety of it's possible forms).

```

1 <feature_model name="minimal">
2   <meta>
3   </meta>
4   <feature_tree>
5     :r minimal(_r)
6       :m! mandatory(_r_1)
7       :o! optional(_r_2)
8       :g (_r_3) [1,*]
9         :! or1(_r_3_4)
10        :! or2(_r_3_7)
11       :g (_r_5) [1,1]
12         :! xor1(_r_5_6)
13         :! xor2(_r_5_8)
14   </feature_tree>
15   <constraints>
16     constraint_1:~_r_5_6 or _r_3_7
17   </constraints>
18 </feature_model>

```

**Figure 5.1.** Example of a feature model in the extended SXFM notation that is used by Cloudberry. There are all types of the feature relationship used in this model. The structure is hierarchical and very intuitive to understand. After a colon there is a character which determines type of the relationship with the parent feature. Blank character means that the feature is a leaf in the tree. Feature identifiers are written in brackets. Cross tree constraints has to be formulated in Conjunctive Normal Form and placed in the *constraints* tag. The features marked with exclamation mark are treated as installable and will be considered during validation of the installation package repository. The use of exclamation marks for distinguishing installable features differs format used by Cloudberry from the original SXFM.

Another approach is to make use of `jclouds-chef` [33] plugin. The plugin is a `.jar` library based on `Jclouds` - commonly used java library that abstracts usage of several cloud providers and allows to unify infrastructure management. `Jclouds-chef` provides Java and Clojure API to perform both server- and client-side tasks while using Chef. It appears to be the right solution of the mentioned problem. Therefore using `jclouds-chef` was a first attempt during prototype development. Unfortunately, implementation of the system prototype proved that `jclouds-chef` plug-in is very underdeveloped. The most significant issue out of the many weak points of this library is the fact that it completely lacks documentation. Therefore, it becomes hard to guess how to appropriately use the complicated API. Moreover after several black box

```
1 <configuration model="minimal">
2   ...
3   <feature id="_r_1">
4     <name>mandatory</name>
5     <type>mandatory</type>
6     <value>1</value>
7     <decisionType>propagated</decisionType>
8     <decisionStep>1</decisionStep>
9   </feature>
10  <feature id="_r_2">
11    <name>optional</name>
12    <type>optional</type>
13    <value>0</value>
14    <decisionType>auto-completion</decisionType>
15    <decisionStep>2</decisionStep>
16  </feature>
17  <feature id="_r_3_4">
18    <name>or1</name>
19    <type>grouped</type>
20    <value>1</value>
21    <decisionType>auto-completion</decisionType>
22    <decisionStep>2</decisionStep>
23  </feature>
24  <feature id="_r_3_7">
25    <name>or2</name>
26    <type>grouped</type>
27    <value>0</value>
28    <decisionType>auto-completion</decisionType>
29    <decisionStep>2</decisionStep>
30  </feature>
31  <feature id="_r_5_6">
32    <name>xor1</name>
33    <type>grouped</type>
34    <value>0</value>
35    <decisionType>auto-completion</decisionType>
36    <decisionStep>2</decisionStep>
37  </feature>
38  ...
39 </configuration>
```

**Figure 5.2.** Sample configuration in the format that is accepted by the Cloudberries prototype. Configuration was prepared using SPLOT. The constraints in the model (See Figure 5.1) allow SPLOT to automatically guess all of the above decisions.

tests I find the code is very unstable (a few bugs was found) - the reason may be jclouds-chef 'beta' development stage. As another drawback I consider core jclouds developers' assumption, which imply that API communicates with the cloud provider directly. In regard to project security premise, system must use intermediary service to communicate with the cloud.

Having abandoned the idea of using jclouds-chef it was a turn to give the first approach, a second chance. The main reason for which the first solution was deemed insufficient was the fact that a shell command output is too complicated to be processed. Fortunately, as it was shown by second review of the Chef's documentation, there is a way to extend Chef client, in the way that will add custom error and report handlers. Handlers are simple Ruby scripts which are loaded and run during client process execution. The variety of error handling methods and output formats is virtually unlimited.

As a further jclouds-chef review showed, the library can be used pretty well to communicate with Chef server REST API. So when we forget about bootstrapping on a node and running a Chef client directly from jclouds-chef it can still be used for other purposes.

### 5.1.5 Software installation

The prototype is able to perform simple installation of selected components on the target virtual machine after providing SSH credentials. As jclouds-library turned out to be insufficient to perform installation, so the installation process was based on Knife (shell command) invocation. Reusability of this part of prototype code was very poor, so the installation process had to be implemented again in the final version of the system.

## 5.2 Cloudberries - the tool

To summarize previous mentions about the application architecture and the choice of technology this section provides a complete description of the implementation. Cloudberries is organized in form of Java-based web application, implementing "Model, View, Controller" design pattern.

### 5.2.1 The choice of technology

To generate HTML web pages it was decided to use Free Marker [26]. It is a generic tool to generate text output (anything from HTML to autogenerated source code) based on templates. As an input FreeMarker takes a static template and a data dynamically generated in Java code. As an output FreeMarker engine returns to the programmer a generated text. To use FreeMarker in context of a web application there is a need for a mechanism that will cope with HTTP requests/responses. In order to easily integrate

Cloudberryes with VPH-Share infrastructure, Java Portlet API (JSR-168, JSR-286) [29] and Jetspeed [34] portal was chosen. The user interface is written in HTML, JavaScript and Dojo toolkit using Ajax. A part of user interface regarding selection of environment components was based on implementation of open source Software Product Line Online Tools. So that the look-and-feel of SPLOT configuration panel was preserved.

In order to perform reasoning on feature model the design followed the same approach as it was taken in SPLOT realization. So that, SPLAR library is used. SPLAR provides an implementation for both presented families of reasoning engines: SAT-based and BDD. Cloudberryes uses the first one.

For installation of environment components Chef is used. The part of the source code that is responsible for provisioning using Chef was mainly implemented in the phase of the Proof of Concept prototype development. The design and implementation considerations concerning Chef are presented in the section 5.1

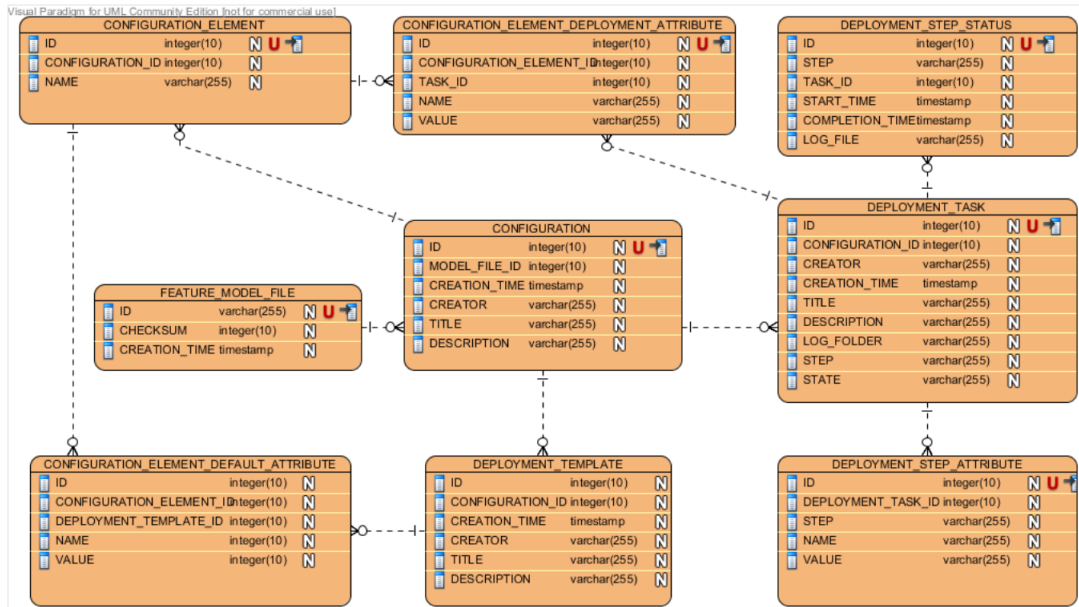
The application is a portlet, deployed in a Jetspeed portal, running on a Tomcat container and uses MySQL relational database management system. To communicate with database from the Java code I decided to use Java Persistence API and Hibernate.

### 5.2.2 The database schema

Entity Relationship Diagram in the picture 5.3 presents a schema of the Cloudberryes database.

This diagram should be read starting from `FEATURE_MODEL_FILE` and finishing on `DEPLOYMENT_TASK`. The first one represents a feature model file stored in the feature model repository (directory on the machine of deployment) and contains its name (ID) and checksum in order to easily check integrity of the model during the system operation. Each `CONFIGURATION` is built in a context of certain feature model so there is a relationship n:1 between them. Each configuration consists of a number of configuration elements. `CONFIGURATION_ELEMENTS` can have default attributes stored in a `CONFIGURATION_ELEMENT_DEFAULT_ATTRIBUTE` organized in `DEPLOYMENT_TEMPLATES`. So deployment template is in fact a set of default attributes for configuration elements. Another form of configuration attribute is related to deployment task, and is stored in `CONFIGURATION_ELEMENT_DEPLOYMENT_ATTRIBUTE` table. `DEPLOYMENT_TASK` represents a process of installation and holds its `STATE` (`NEW`, `RUNNING`, `COMPLETED`) and current deployment `STEP` (will be described later). Each of these properties is enumerated in application layer. Each deployment step is separately reported in the table `DEPLOYMENT_STEP_STATUS`. `DEPLOYMENT_STEP_ATTRIBUTE` table is used to save some output from deployment step which is other than log.





**Figure 5.3.** The entity Relationship Diagram of the Cloudberries database. This diagram should be read starting from `FEATURE_MODEL_FILE` table and finishing on `DEPLOYMENT_TASK`. The names of tables are quite intuitive and match corresponding items in the user interface.

### 5.2.3 The process of environment configuration

When the user opens a page regarding environment configuration, a feature model is loaded and validated. Then a feature model object is created in the memory, and it can be configured. Configuration process can be treated as providing the feature model instance with decisions about selection of features. Validation of the configuration is performed on the fly, so the user is guided through configuration process. So that, each configuration created using the user interface is correct in the context of the feature model at the time of configuration saving. It should be mentioned that only installable features are saved to the database.

When the user opens configuration stored in the database in order to create deployment task, he has to provide attributes for features. Attributes are stored in a cookbook (cookbook is in fact a directory) as a file named `metadata.rb` (refer to Chef documentation [10]). When the user creates either deployment template or deployment task, attributes of features are loaded from a central cookbook repository. Each time Cloudberries needs to access the Chef repository it uses `jclouds-chef` library.

When the configuration is supplemented with attributes the deployment process may be run. Even though the stored configuration is considered

valid, it is validated again before starting the installation. The process itself is similar to that, implemented as a part of the prototype and can be represented as a sequence of steps described below.

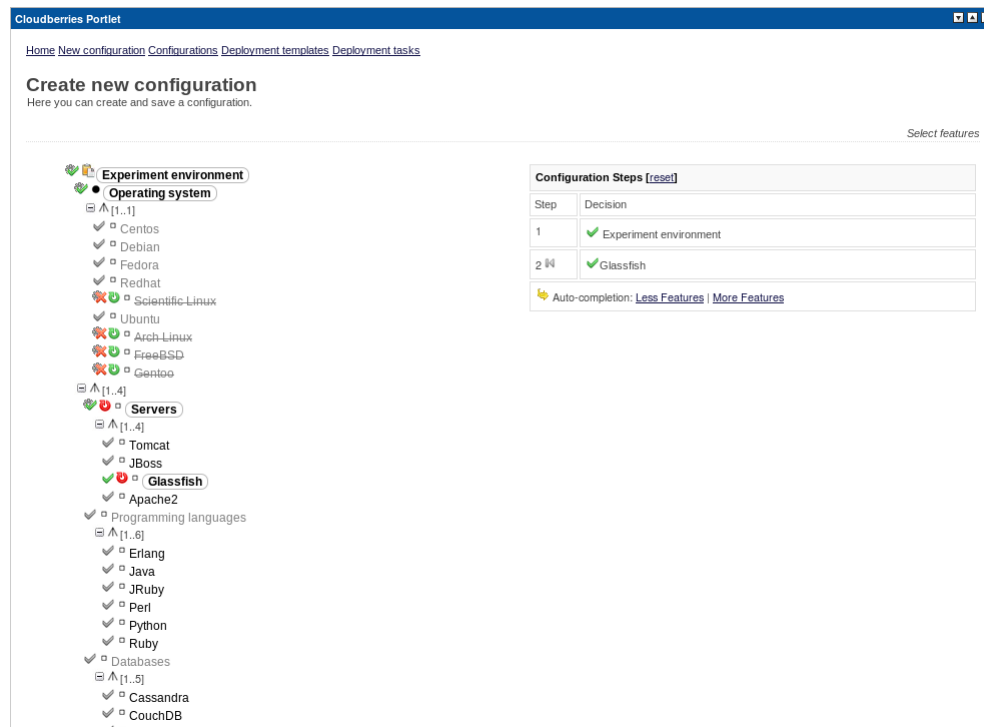
1. **VALIDATION** - similar to the process presented in the section concerning prototype development. This step of deployment takes a configuration loaded from the database as an input. **VALIDATION** process consists of instantiating the model (and checking its syntactical correctness), checking if for each feature there exists corresponding cookbook in the central Chef repository, validating the configuration (feature must exist in the model), automatic inference of feature installation order and creating Run List (refer to the Chef documentation). Installation order is based on a graph constructed using dependency relationship. If a dependency graph contains cycles configuration is discarded.
2. **BOOTSTRAP** - a process of installing Chef client on the Node. To do that, Knife shell command is used. When the client is installed, Cloudberryes apply the Run List, and user-defined attributes of cookbooks on the node. The jclouds-chef library is used in order to perform these tasks.
3. **DEPLOYMENT** - the installation is performed by the Chef client. In order to execute Chef client on the Node, Knife is used. Monitoring of the client state is based on the log output of Knife. As Knife is a command line application the log has a form of standard output from the command and it is not so easy to programmatically customize it. The logging can be improved by engaging Chef client's mechanism of handlers. It is a mechanism of plug-ins written in Ruby that allows to handle logging from the perspective of Chef client. When the client perform installation it invokes handler, providing them with information regarding installation process. Because logging is not so crucial, the plan of improvement is left to the implementation in the future.

### 5.3 User interface

In this chapter Cloudberryes user interface will be presented in a nutshell.

#### 5.3.1 Configuration

As it was mentioned before, end user is allowed to choose elements of environment and save the configuration in the database. After entering *New configuration* page (Figure 5.4), the user will see a visual representation of feature model similar to the one presented in the figure 5.4.



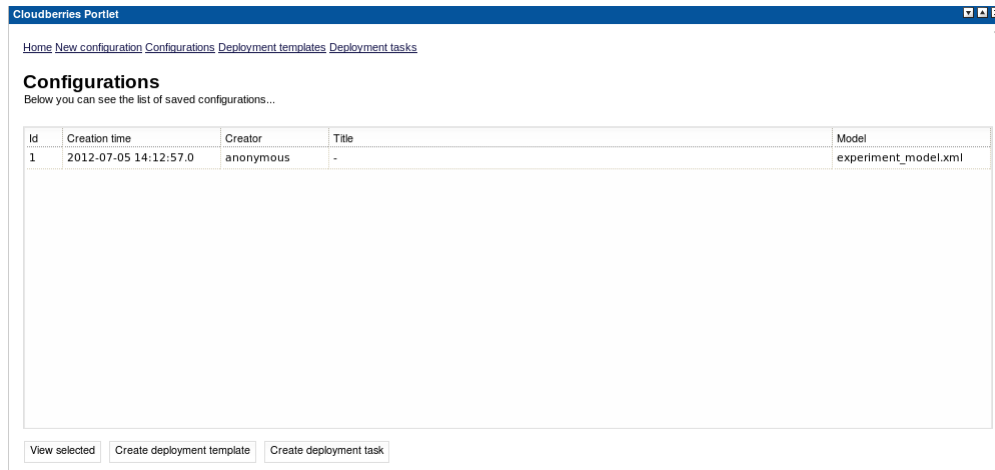
**Figure 5.4.** Creation of an experiment environment configuration. On the left side of the page one can see a hierarchical representation of the feature model loaded by Cloudberry from a file saved under a location specified in settings. Features represent elements of the environment. Each configuration step, the underlying application layer updates configuration changes and excludes all of the useless features from the configuration space. On the right side there is placed a table of configuration steps. Configuration process can be automatically completed by selecting either Less Features (selects as little as possible) or More Features (selects as much as possible).

As it was mentioned before, a single feature represents a component of experiment environment. Features are either installable or not, so that the installable ones are darker than the rest of features (black color). A feature can be selected by performing a single click at the corresponding checkmark. When a feature is selected, the user is being informed about possible conflicting decisions. Decisions that can be inferred are automatically made for the user. After selecting a single feature, the decision appears in the table on the right side. Each decision step can be undone, by clicking on the rewind mark. When the user intends to do so, they can automatically complete the configuration process by selecting either Less Features or More Features option. The first will result in selecting as little features as possible. The second, just the opposite.

## 5. IMPLEMENTATION OF THE TOOL

---

When a user sees a message *Done* which means the configuration process is finished, they can save the configuration and view it (Figure 5.5) by clicking *Configurations* link on top of the portlet.



**Figure 5.5.** In this screenshot one can see the list of configurations that were previously created and saved by the users. Each configuration is assigned to a feature model which was used to create it. Before any further usage, the configuration is validated with respect to the model, in order to avoid errors in the installation process.

Having a single configuration the user can create deployment template or deployment task.

### 5.3.2 Deployment template

In order to create deployment template, the scientist has to select a configuration on the *Configurations* page, and click *Create deployment template*. Then they will be redirected to template creation page, which looks similarly to the one in the Figure 5.6.

The screenshot shows the 'Create deployment template' interface in the Cloudberries Portlet. The page title is 'Create deployment template' with a subtitle 'Choose attributes subset and provide values you want to save...'. There are navigation links: Home, New configuration, Configurations, Deployment templates, and Deployment tasks. A 'Describe your template' link is on the right. The form includes a 'Template title' field with the value 'New template' and a 'Description' field. Below this is a 'Configure attributes' section with two side-by-side tables.

**Left Table (Attribute Selection):**

Attribute	Value
<input type="checkbox"/> mongodb/logpath	/var/log/mongodb
<input checked="" type="checkbox"/> mongodb/sharded_collections	
<input checked="" type="checkbox"/> mongodb/client_roles	
<input checked="" type="checkbox"/> mongodb/dbpath	/var/lib/mongodb
<input checked="" type="checkbox"/> mongodb/port	27017
<input type="checkbox"/> mongodb/cluster_name	
<input type="checkbox"/> mongodb/shard_name	default

**Right Table (Attribute Configuration):**

Attribute	Value
mongodb/sharded_collections	
mongodb/client_roles	
mongodb/dbpath	/var/lib/mongodb
mongodb/port	27017

**Figure 5.6.** Creation of a deployment template. A user can entitle the template and provide some information notes. The box on the left allows to select attributes for the configuration elements. The one on the right allows to specify values for the attributes. This is very similar to the process of *Deployment Task* creation.

After specifying a title and description of the template, the user has to select feature attributes which will be saved in the template, just as presented in the picture above. Left box shows default values of attributes and allows to select attributes that will be modified. Right box is used to specify attribute values.

Saved templates can be listed and viewed in a way similar to that presented in the configuration description.

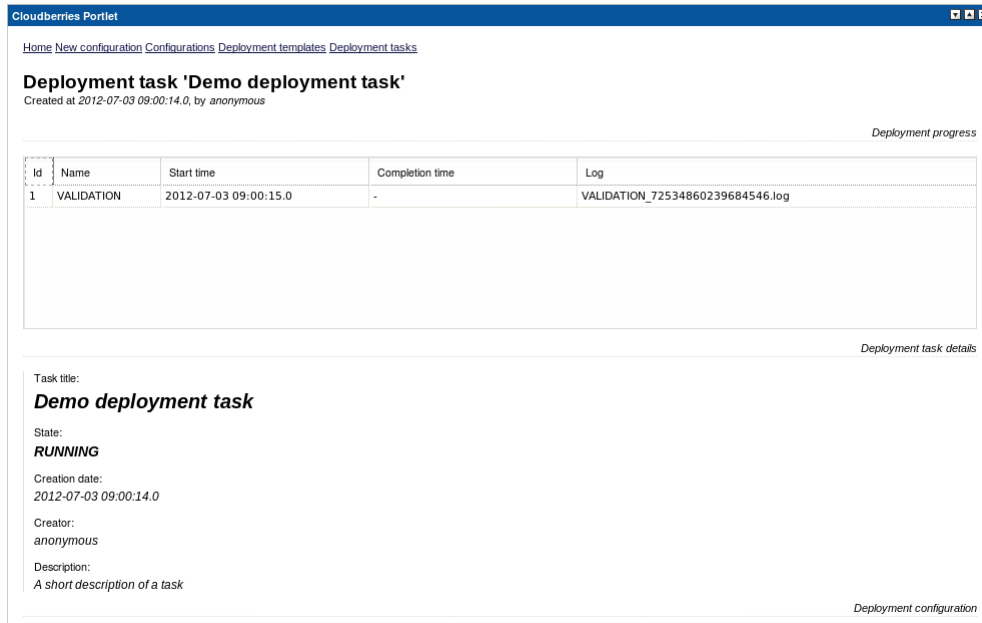
### 5.3.3 Deployment task

Deployment task can be created either by using configuration or deployment template. Regardless of which of the way is chosen, the user has to select an entity from a list, and click a button *Create deployment task*. The user interface for creating deployment task is very similar to that, creating deployment template. The difference lays in the need for providing IP address, user name and password in order to access target Virtual Machine via SSH.

Once the deployment template is created, the corresponding deployment process can be monitored by selecting a given task from the *Deployment tasks*

## 5. IMPLEMENTATION OF THE TOOL

page. The interface will be similar to the one presented in the picture 5.7.



The screenshot shows the 'Cloudberries Portlet' interface. At the top, there are navigation links: Home, New configuration, Configurations, Deployment templates, and Deployment tasks. The main heading is 'Deployment task 'Demo deployment task'', with a subtext 'Created at 2012-07-03 09:00:14.0, by anonymous'. Below this is a table titled 'Deployment progress' with the following data:

Id	Name	Start time	Completion time	Log
1	VALIDATION	2012-07-03 09:00:15.0	-	VALIDATION_72534860239684546.log

Below the table is a section titled 'Deployment task details' containing the following information:

- Task title: **Demo deployment task**
- State: **RUNNING**
- Creation date: 2012-07-03 09:00:14.0
- Creator: anonymous
- Description: A short description of a task

At the bottom right of the details section, there is a link for 'Deployment configuration'.

**Figure 5.7.** Monitoring of a *Deployment Task*. After selecting the task from the list in the *Deployment Tasks* page, the user can see a page similar to the above. The table on the top contains information about installation steps. By selecting a step one can see the corresponding log.

Deployment task page informs the user about the current state of the task and a set of deployment steps which are visible in the table on top of the page. Deployment steps are described in the section regarding technical aspects of the final version of the system (5.2).

Right click at a deployment step shows a corresponding log. The log will appear in the popup window at the same page.

## 5.4 Summary

This chapter presented design considerations of Cloudberries - the application that was built in scope of the thesis. In order to familiarize the reader with the course of the project there were presented details of the particular implementation phases. The division of this description is also justified by the fact that some assumptions of the project realization were made in the early phase of prototyping and can be clearly separated. Sections 5.1 and 5.2 allow to explore the choice of technology, low-level design considerations and principles of the system operation. At the end of the chapter there are

presented some screenshots that help the reader to visualize described application.





## Chapter 6

---

### Validation of tool

---

*The tool developed in scope of this thesis was installed in the production infrastructure of VPH-project. This chapter presents several aspect of deployment and evaluation. The objective of the tool creation was minimizing the complexity of experiment deployment by providing additional layer on top of provisioning system, so efficiency of the tool strictly depends on performance of provisioning. As it is very hard to introduce reasonable metrics to assess efficiency of this process, the tool was evaluated in terms of the usability, security and its limitations in the current state of implementation. This chapter presents also some lessons learned in the process of implementation and areas of possible improvement.*

In order to evaluate Cloudberryes, the tool was deployed in the production environment of VPH-Share project and runs in the project's private cloud infrastructure. Cloud Execution Environment of VPH-Share is based on OpenStack middleware suite responsible for providing VM lifecycle management and storage for a private cloud installation. So far the infrastructure consists of 7 physical nodes with identical hardware specifications (HP ProLiant BL2x220c G5, 2 x Intel Xeon L5420, 16GB RAM, 120 GB internal HDD). One of them acts as the Cloud Controller (CC) and the rest are used to run VMs. All nodes have access to approximately 3 TB of external shared storage space (NFS on iSCSI volume) backed by a disk array with fast (15000 RPM) SAS hard drives. This shared space is used to store VM templates and images that can be used to run experiments. Cloudberryes is installed in the Jetspeed portal running in the application server, deployed on a dedicated VM. The portal at the time of writing this thesis can be accessed via a Web browser at <http://vph.cyfronet.pl/puff>. As the tool allows for software installation, in order to gain the access to Cloudberryes an account in the portal is needed.

The instance of Cloudberryes is for now configured in a way that allows to handle installation of software used by GridSpace Experiment Workbench [14]. GridSpace is a virtual laboratory for running experiments that combines usage of various languages and interpreters (Bash, Mathematica, Matlab,

Python, Perl, Ruby, Gnuplot) as well as a variety of execution methods (SSH, QosCosGrid, GLite, Globus). The instance of Cloudberry is provided with a feature model and corresponding cookbooks that allow for installation of the prerequisites that are used by GridSpace execution environment (Erlang, Java, JRuby, Perl, Python, Ruby, etc.) and some other useful applications (eg. database management systems - Cassandra, CouchDB, MongoDB, MySQL; utilities - Cron, Maven, Nfs, Screen, Apt).

### 6.1 Case study

As a case study for Cloudberry evaluation euHeart e-Science application was chosen to be installed. As it appeared, its installation prescription gives a great opportunity for automation. The process of euHeart installation is presented below:

1. Instantiate a x64 Virtual Machine log into it.
2. Install required software. Although, the installation commands of these packages are basically one-liners, some operating system distributions may require installation from source. So that, there is a opportunity of automation and installation methods reuse. Following packages are needed for euHeart installation:
  - python,
  - python-pip,
  - xvfb,
  - wine,
  - libxp6,
  - openjdk-6-jdk,
  - python-dev,
  - libxslt1-dev.
3. Download instalation archive *heartgen.zip*.
4. Extract *heartgen.zip*.
5. Execute `easy_install Flask`.
6. Execute `./MCRInstaller.bin -console` from the extracted directory. Installation process is prompted by the user and requires some input. Interaction with user can be automated using *expect* tool.
7. Execute `easy_install soaplib suds`.

8. Create user *localuser* with home directory. Usually it means entering a command similar to `sudo useradd -d /home/localuser -m localuser`.
9. Set password for *localuser*.
10. Move files extracted from *heartgen.zip* to */home/localuser*.
11. Prepare service initialization file */etc/init/heartgen.conf*. The configuration can be included in installation package - for now it is not.
12. Start *heartgen* service.

After completing all of these activities the application may be tested using the script provided in the installation package. The process of installation consists of 12 steps. Although most of the the steps are not very complicated, they may require some attention (providing input, copying files, waiting for results) and operating system administration knowledge (using different flavors of commands, granting privileges). It means, that automation may save some valuable time - amount of time savings depends on individual abilities.

Most of the above installation steps can be easily mapped to cookbooks, as well as euHeart application can be represented as a single feature dependent on its prerequisites.

For installation of python, python-pip, xvfb, wine, libxp6, openjdk-6-jdk, python-dev, libxslt1-dev individual cookbooks were prepared, so that they constitute individual feature in the model. The cookbooks may be reused to deploy other applications. The procedures specific to installation of euHeart (moving files, unzipping, python packages installation, user creation) were wrapped together in a separate cookbook. Installation of *MATLAB* distribution (*MCRInstaller*) was automated using the tool called *expect*.

Using Cloudberryes the entire installation process can be scheduled by selection of the euHeart feature. So that, the productivity of scientist deploying euHeart is increased.

## 6.2 System usability evaluation

In order to review the tool usability it was evaluated in two different aspects. First aspect regards the usage of the user interface. As the user interface was planned to be as easy as possible it was mainly judged in terms of complexity. The second applies to the ease of installation.

### 6.2.1 User interfaces

From the user perspective, the usability of the system is connected with the level of environment configuration complexity. So that, Cloudberryes was

designed to minimize impact of this factor. As the use of the web user interface is really simple, the scientist has only provide the list of needed software components and some attributes which could not be prescribed by the administrator (eg. IP address). In addition to that, the user can save the configuration at two levels - using default or predefined attributes of environment components (configuration or deployment template).

Another aspect of the interface implementation that should be evaluated is the ease of searching for a certain configuration or template. In order to allow a large number of users to manage their configurations in a convenient way, there may be introduced some kind of filtering mechanism. In order to cope with possibly large number of database entities, users should have a possibility of selecting only elements created themselves, or a specific group of elements. There can be also some performance issues of the user interface that can affect usability, when Cloudberryes is used by a large number of users. This factor should be taken into consideration when the tool will be developed.

However Cloudberryes is pretty easy to use for the end-user (a scientist), the system was not planned to be focused on the usability in terms of administration. In the phase of the tool design there was considered a user interface module that allows the administration user to create, edit, and save feature models. In the current state of the implementation such operations are not allowed. Instead of this, the application allows to use models saved in the filesystem of the machine where the web application is deployed on. The appropriate model can be selected by specifying a filepath in the Cloudberryes configuration file.

### 6.2.2 Tool installation

Installation of the tool is easy and does not demand anything but copying and pasting into appropriate catalogue of Jetspeed portal (refer to the guide presented in Appendix B). The integration with other parts of portal is really simple and does not require any knowledge but deployment of regular Jetspeed portlet. The system architecture is loosely coupled so individual elements can either be distributed among dedicated machines or deployed together on a single server.

## 6.3 System security

Security of the user interface is provided by the Jetspeed Portal, and is based on login-password user verification. A communication between Cloudberryes backend and Chef server is secured by SSL, so there is very little chance of using Chef without permission. Authentication is based on a single *.pem* file, so currently there is no mapping between users of Cloudberryes to Chef server users.

Knife (which is used, inter alia, to install Chef client) uses SSH protocol so it can be also considered to be safe. Authentication via SSH is based on verification of login and password.

## 6.4 Limitations

In order to describe limitations of the current state of implementation, and guidelines to the future work, several aspects of the system capabilities are presented. Because in a number of cases the limitations are not strictly connected with implementation of Cloudberry and in some other they cannot be measured without manual beta testing (especially the usability of user interface), following description will be probably the best way to present these limitation issues. The limitations of the system will be described in terms of:

- System load - as a number of users that the system can handle.
- Domain of configurations - a number of environment components that can be maintained by the system.
- Scalability of deployment - a number of environment configurations that can be deployed simultaneously by a single user.

### 6.4.1 System load

*Number of users that the system can handle.* There is no explicit limitation of the number of users. As Cloudberry is a typical web application it scales just like web applications scale. In order to handle operation of a larger number of users that are simultaneously logged in, the population of users may be clustered into groups, and each group can be served by a different server, communicating the same database. In order to plan scaling the database, there would be a need to provide replication mechanism, so it may cause a need to rethink a choice of database engine (currently MySQL is used, which allows for some kind of replication, but there are some more mature solutions on the market like PostgreSQL, OracleDB). Fortunately as Cloudberry uses Java Persistence API it is suitable for database provider changes. Another aspect of scalability in terms of number of users is simultaneous installation. Because of Chef's architecture the Chef server load is only slightly dependent on a number of environments being deployed at the same time. As installation is performed by client application on the target node, the complexity of the process is split between nodes. The actual load of a Chef server depends on the size of cookbooks downloaded from the central repository. If cookbooks contain installation package tarballs, the server will suffer from large number of concurrent download processes. That is the reason for chef documentation not to present any performance evaluation. Nevertheless the load

of the Chef server can be managed by replication of Chef servers on separate machines and clustering the users between them.

### 6.4.2 Domain of configurations

*Number of environment components that can be maintained by the system.* The number of installation packages that can be managed by Cloudberries is subordinated to the capabilities of the system administrator. In order to manage dependencies and exclusions they have to cope with the complexity of components interaction. Complexity of feature model can obviously slow down the process of computation of a valid domain of environment components (during the configuration process), but this factor will not heavily affect performance of user interface.

### 6.4.3 Scalability of deployment

*Number of environment configurations that can be deployed simultaneously by a single user.* At the current state of the system implementation there is no way to run multiple installation processes at the same time, other than creating a deployment template and then running a few deployment tasks one after another. That procedure is not demanding and in fact requires only to click at the template the user wants to use more than once and then provide User/Password/IP address triple. To simplify this process even more, the part of user interface to create deployment tasks, could be equipped with a form allowing to input multiple triples presented before and start a group of deployment tasks. This would ease the deployment process of a large environments even more.

## 6.5 Summary

This chapter presents validation of the tool and some conclusions that were drawn in terms of its usability and security. The tool was evaluated in the production infrastructure of VPH-Share project and used for installation of ehHeart application. The evaluation proved that the system meets its requirements and can be successfully used in the main areas of application. There are also a few system limitation aspects that are addressed in this section. Apart from that, the process of implementation brought some specific type of results, which are presented in the next chapter. There was created a concept of Software Product Line architecture that could be applied in a design of application created for similar purposes as Cloudberries. The solution is very generic and extensible, in the way that it could allow to easily add additional types of features (e.g. instance of VM) and new means of component installation (e.g. a service of VM instantiation corresponding with the sample additional feature type). Because of the above, the Chapter 7 should

be treated as a further description of the lessons learned during the thesis realization.





## Chapter 7

---

# A concept of Feature Model - based automatic SPL generation

---

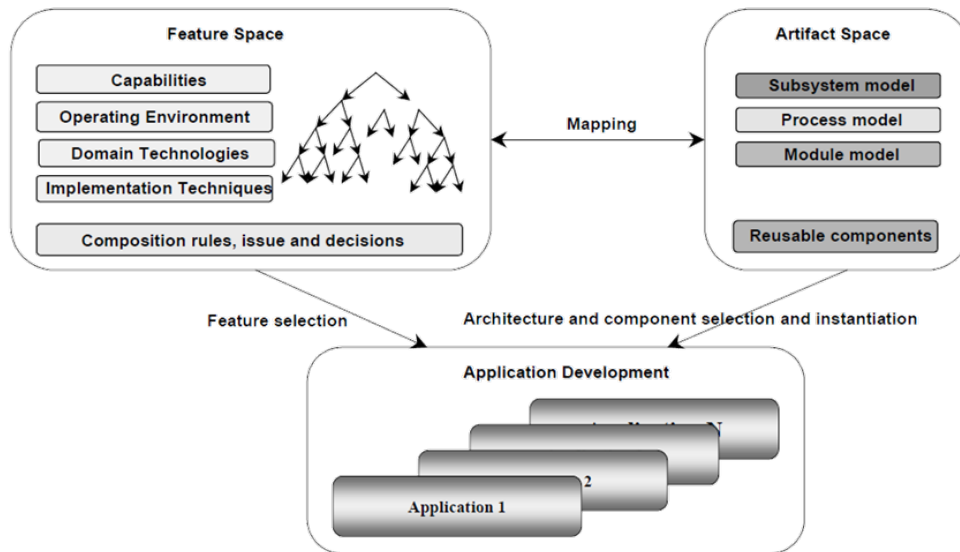
*this chapter presents further reasoning inspired by experiences, gained during the study of State-of-the-Art, design of the tool and its implementation. During the process of implementation it was discovered that as the tool architecture is based on Software Product Line methodology and usable for wide domain of purposes, it might be designed in more generic fashion. In this scope it has arisen a concept of an easily evolving software production line architecture. The architecture is oriented on the use of Feature Model for software products domain representation and based on a highly extensible mechanism of software installation. Production line operational rules are automatically derived from a model. The first part of this chapter presents the idea of Feature Model adaptation (7.1), and the second (7.2) describes the architectural concept that was based on this approach.*

### 7.1 Feature Model adaptation

This section presents the concept of Feature Model adaptation in order to introduce the idea of automatic derivation of product line architecture from a feature model. As the authors of the paper [27] claim, "Software product line engineering (SPL) involves developing the requirements, architecture, and component implementations for a family of systems, from which products (family members) are derived and configured."

Usually in order to build a functional product line, there is a need to go through a complex engineering process starting from a domain analysis.

## 7. A CONCEPT OF FEATURE MODEL - BASED AUTOMATIC SPL GENERATION

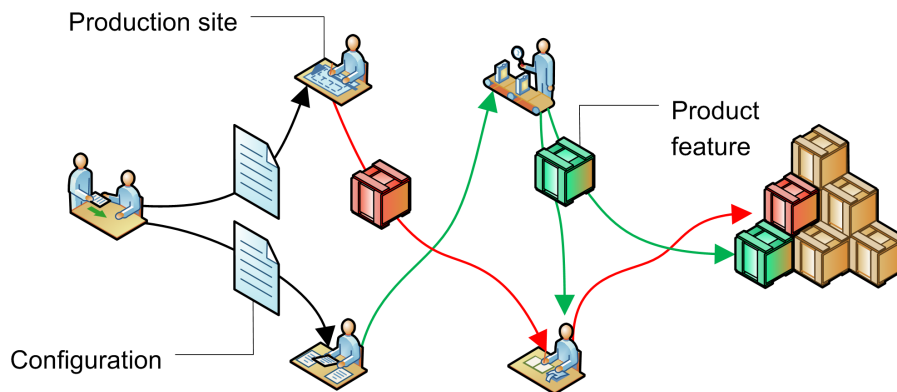


**Figure 7.1.** Software Product Line engineering schematics [36]. There are three spaces that has to be connected in order to create production line. The most challenging process in the lifecycle of the production line is a mapping from the space of product features (that are relevant to stakeholder) to the space of artifacts (that are relevant to the developers).

The key task of Software Product Line engineering is to distinguish features by detecting product commonalities and variabilities. In this way we can think of our product as of a set of certain characteristics. Treating our product in terms of features gives us a high-level view of our product family. High level of abstraction helps us to realize dependencies, so given a single product configuration even complex interactions among its features become visible. Moreover commonalities manifested in the model indicate clearly where reuse opportunities are.

Once we have our model defined, there comes a time to shift from the Feature Space to the Artifact Space (See figure 7.1). What it means is to create the architecture specification (defining reusable components and the way they are used) and to develop production line application. In other words we have to prepare our *means of production*.

Following the terminology of production line, software product manufacturing process can be seen as a succession of interrelated production stages. Specialized *production sites* cooperate with each other to assemble product parts according to a production plan. Production site should be understood as a part of production line system providing the means of production for specific stage of product creation.



**Figure 7.2.** A schematic of production line using several production sites to produce single feature. Each production site is capable for its own production procedures that can be applied to realize partial production of given feature.

The plan itself describes the rules of operation for the whole production line. It can be understood as a specification similar to that, presented in the table 7.1:

Feature	Stage	Action	Prerequisites
Feature X	1	Site 2 procedure 1	-
	2	Site 3 procedure 1	-
	3	Site 5 procedure 1	Feature Y after Stage 1
Feature Y	1	Site 3 procedure 1	-
	2	Site 2 procedure 2	Data input A

**Table 7.1.** A sample production plan for a production line. Production process of a single feature is split between several production stages. Each stage is realized by a different production site and has different requirements. Each site may offer several procedures that provide means of production specific to the feature.

Production plan matches stages of production for each feature with the corresponding production sites, prerequisites and actions to be performed on appropriate site. Introducing the notion of production site, we may regard it as an application module, providing some production capabilities that are split into procedures. In terms of production plan each feature must be described in order to provide a mapping from the feature space to the space of prerequisites and means of production (stages and installation procedures). On the basis of a production plan, in order to create a product, each production site should be provided with some input relevant to an installation of a given feature:

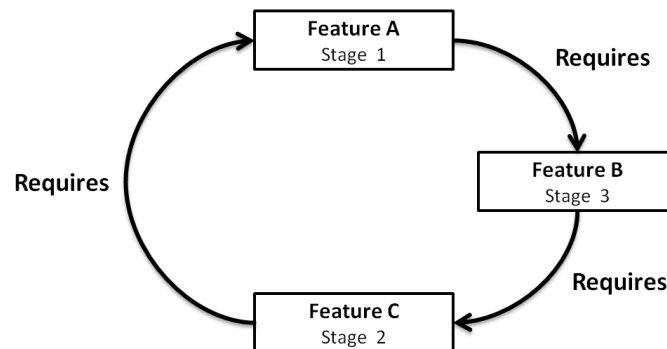
## 7. A CONCEPT OF FEATURE MODEL - BASED AUTOMATIC SPL GENERATION

---

- production schedule (inferred from the production plan),
- installation prerequisites (required reusable data, intermediate products of other sites),
- guidance on which action should be performed (if we think of sites, that are able to serve several production procedures).

What is worth mentioning, in order to prepare production schedule we need to look at the production plan and set production stages of the selected features in the correct order. As one can imagine even if we take a single configuration of product into account, the scheduling process is intellectually demanding and should be performed automatically.

At this stage, the issue I would like to point out, is the situation presented in the picture 7.3. The problem presented in the figure is pretty dangerous for the product line operation and what is more it may occur quite unexpectedly, especially when no product line validation is being performed. In the picture 7.3 we can see a graphical representation of feature installation requirements corresponding to three feature production stages. For example, a "requirement" denotes that production Stage 3 of Feature B should be completed before Feature A production Stage 1 starts. When we treat the rest of the "requirements" analogously, we can see a conflict, which causes the scheduling is not feasible (the chronological loop that is visible in the picture 7.3).



**Figure 7.3.** A cycle in the process of production. Requirement should be treated as a dependency of production stages. Production stages of the given features cannot be put in any order that guarantees to meet all of the requirements.

The occurrence of this conflict shows that either the configuration of these three features should not be permitted or production plan fails. In addition to that, when the production plan is growing bigger, we can see the need for automatic production line conflict resolution that is rather obvious. Further-

more, every single configuration has to be tested in terms of production plan feasibility, before the Feature Model can be accepted.

Therefore, the closer we look at the mapping of a model to the production line architecture, the more complicated the mapping process turns out. So if we manage to automatically derive the architecture of our production line from a feature model, several aspects have to be taken into account:

- In domain of Feature Modeling
  - selection of features - how products vary from themselves? What should be treated as a single feature,
  - feature model definition - what are dependencies between features,
  - feature model validation - testing whether the definition of feature model is semantically sensible.
- In definition of the means of production
  - defining reusable artifacts,
  - determination of the feature production stages - a procedures of asset installation in the product,
  - determination of the feature production stages interconnections,
  - mapping features to a succession of production stages.
- In product line validation and model refinement
  - production feasibility test for each valid product configuration,
  - exclusion of invalid products from model.

Some of the above tasks can be performed automatically, and some of them are a part of the analysis, which has to be done manually by the product line engineer. The challenge is to bring about a situation, in which the number of tasks that are the responsibility of the user would be minimal.

As the author of this thesis suppose it is very likely that at least feature selection and feature model definition has to be performed by the user. Assignment of a code to the specific features probably also needs to be done manually. The question is - where is a space for the automation?

Obviously the level of automation of software product line derivation process may be closely related to the specificity of software family which is being modeled. As it turns out during experimentation with SPL in application of experiment environment preparation, there was realized that there are some assumptions that can be made on the Feature Model, which can allow us to perform more of these tasks automatically.

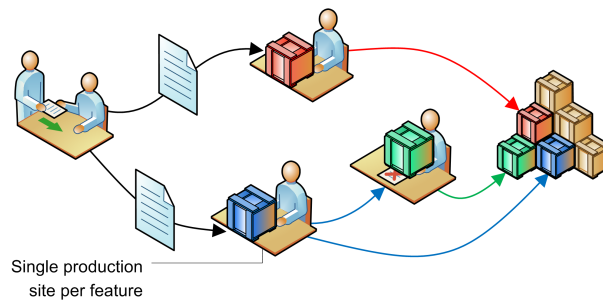
Feature models are built based on feature dependencies that are in principle very general (parent-child relationships, and cross-tree dependency). As

## 7. A CONCEPT OF FEATURE MODEL - BASED AUTOMATIC SPL GENERATION

we have seen in the example, in which we considered an attempt to installation of three features related in the sense of production line dependency, there exists an influence of production scheduling on the feature model. Moreover, the role of the dependency relationship can be crucial for a model correctness. When we take the dependency of installation processes into account, it can prove that the model includes a configuration of features that causes conflict in terms of production schedule.

The author realized that maybe there is a need to extend the model, in order to provide all the information needed to validate its domain of products in terms of production scheduling feasibility. For the author, the most intuitive way to do that would be expanding the relationships defined by the Feature Model, in the way that they would present the order of installation. Each of the original relationship between two features would be either directed one way or another, to represent the succession of feature installation.

As a natural consequence of this approach, the following assumption has to be made: Each feature must be produced at once (single stage), and production of feature must be performed by a single production site.



**Figure 7.4.** A schematic of production line using single production site to provide a single feature. The process of feature installation is atomic and cannot be split between production sites. So that, there is no need for management of dependencies of intermediate production stages.

Of course the assumption may be considered very strict and even inappropriate (from the perspective of Feature Modeling purpose), because it does not allow to split feature production between several reusable product line application components (production sites). On the other hand, it should be noted that each reusable code component which has had to be used before introducing the assumption (Single Stage + Single Production Site), now may be represented as a separate feature and used in its usual way. So we just shift the specification of production plan into Feature Model and gain an ability to verify if the product family is consistent, without looking deeper than the Model. The actual method of verification will be presented in the next section.

Further consequence of the assumption is the fact that Feature Model

becomes more detailed and consequently its level of abstraction is lower. In this way some irrelevant product details may be unintentionally presented to the end user. To avoid this effect we may introduce a concept of layered Feature Model similar to that presented in FORM [36]. The idea of layered model is also known in other approaches of domain engineering (eg. Model Driven Architecture). Layered model allows to keep all the valuable information and hide unnecessary details in the same time.

A quick summary of the above considerations:

- We assumed that each feature of the extended model is produced by a single production site of the production line.
- We assumed that feature production process is atomic and cannot be split or interrupted by another.
- We introduced the notion of a relationship that describes feature installation order.

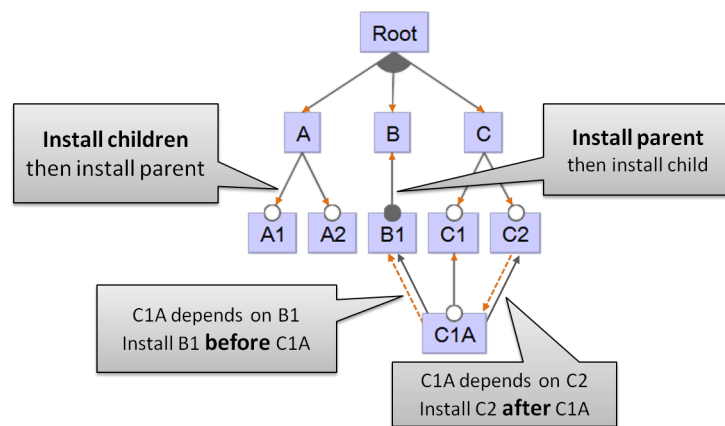
### **Ordering relationship**

To describe the installation ordering relationship a bit more precisely let's look at its characteristics:

- It is an element of Feature Model.
- It extends basic feature relationships:
  - parent-child relationships,
  - dependency relationships.
- For a pair of related features it determines the order of installation.
- The relationship can be graphically represented by an edge of a directed graph composed of feature as graph vertices.
- A graph created out of relationship edges is a dependency graph that can be analyzed in order to answer the questions:
  - whether every configuration represented by the model is valid in terms of production scheduling (graph is acyclic),
  - how to schedule production in order to create a product corresponding to a configuration?
- For a parent-child relationship the direction of the edge is identical for all of the children of a single parent vertex.

## 7. A CONCEPT OF FEATURE MODEL - BASED AUTOMATIC SPL GENERATION

The example in the Figure 7.5 shows the sense of the relationship more suggestively. In this picture you can see the feature installation order relationship marked in orange.

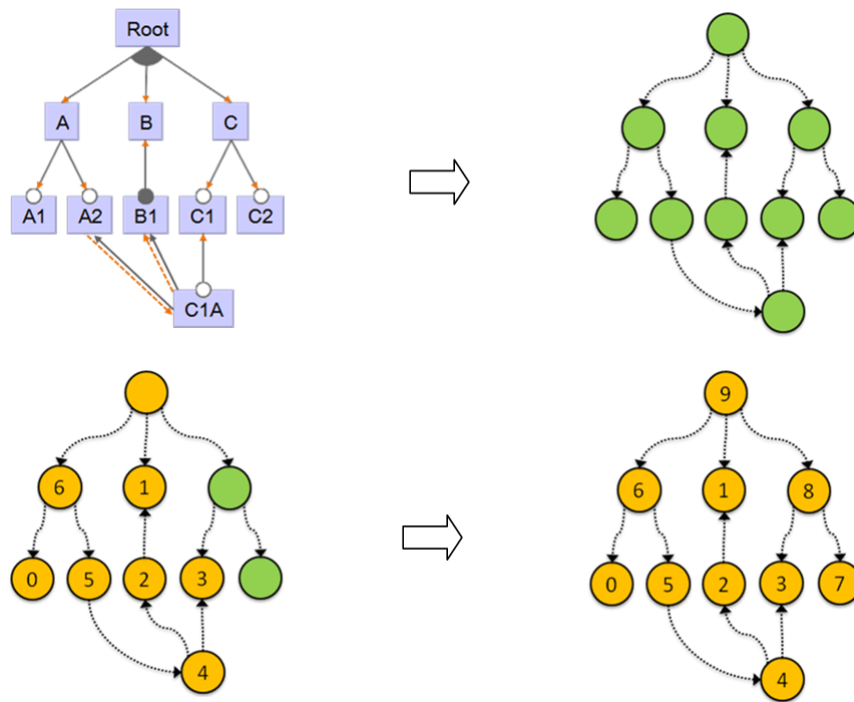


**Figure 7.5.** Sample feature model enriched with installation ordering relationship. The installation ordering relationship extends the original Feature Model relationships in order to provide additional information that is needed to perform scheduling.

Another example presents simple scheduling algorithm based on feature installation order relationship. In fact scheduling will be based on topological sorting of directed acyclic graph. In order to find a proper schedule we have to construct a graph out of the features represented by vertices and edges of installation order relationship. Now, we apply simple recursive topological sort algorithm presented in the Figure 7.6.

A sample use of the algorithm is presented in the Figure 7.7. The procedure is very simple and convenient, but unfortunately there exist yet another problem connected with dependency relationship.





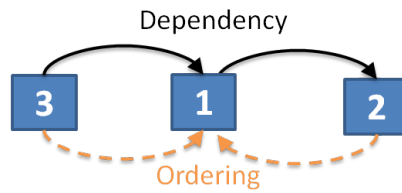
**Figure 7.7.** Graphical illustration of the ordering algorithm presented before. The Feature Model was replaced by the directed acyclic graph with edges determined by the ordering relationship. The graph is sorted topologically by visiting adjacent nodes starting from the Root.

```

1 L ← Empty list that will contain the sorted nodes
2 S ← Set of all nodes with no outgoing edges
3 for each node n in S do
4     visit(n)
5 function visit(node n)
6     if n has not been visited yet then
7         mark n as visited
8         for each node m with an edge from m to n do
9             visit(m)
10    add n to L

```

**Figure 7.6.** Simple recursive ordering algorithm. The nodes in the graph represent features of the feature model. The edges are built out of the installation ordering relationship.



**Figure 7.8.** The problem of ambiguous installation ordering. Both feature 3 and feature 2 should be installed before installation of the feature 1 (ordering relationship marked with orange). Presented algorithm does not specify the installation order of indirectly connected feature 3 and feature 2.

When we take into account a simple example, concerning three features dependent as presented in the Figure 7.8, we expect their installation processes to be ordered in a way represented by the numbers in the picture. While 3 depends on 1 and requires it to be installed before its own installation, we can expect that even if 1 depends on anything else (like the feature 2, installed after feature 1), all of the feature 1 dependencies will be installed before the feature 3 installation process begins.

It means that even though there is no explicit relationship between 2 and 3, an ordering dependency exists indirectly. So that, the algorithm should be improved to handle similar situations. The simple topological sorting algorithm can provide inappropriate solution, only when two independent (not dependent directly) features are both ordered in the same way by two adjacent ordering relationship edges. There exists a need for improvement of the tree search algorithm.

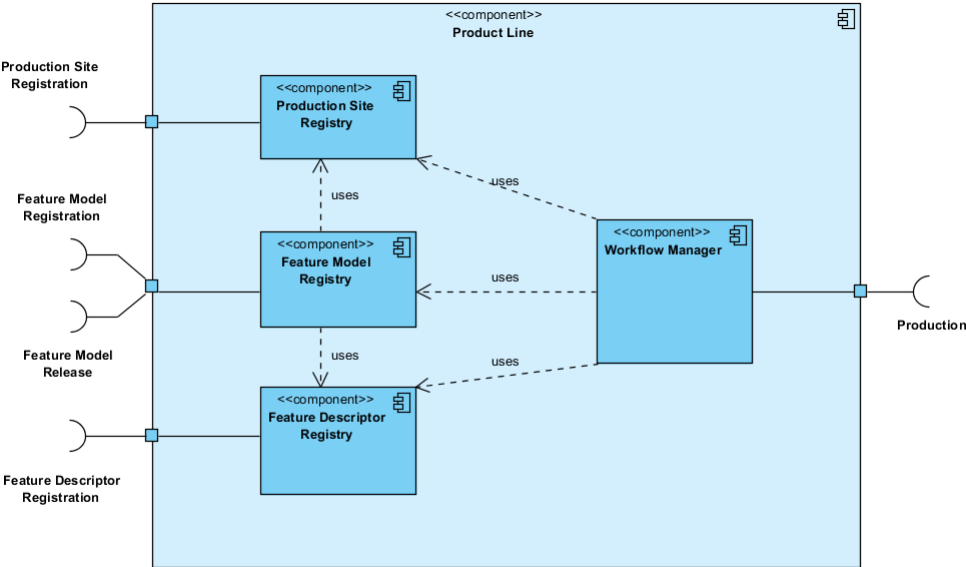
## 7.2 A concept of Feature Model - driven Software Product Line architecture

Previous section presents an abstract concept of Feature Model extension, which allows to equip the notation with some information needed to perform scheduling of production. The extension is based on introducing new relationship type into the model, and assuming that each feature is mapped to an atomic process of feature production. As it was discussed earlier, the reason for introducing this extension is the need to allow for derivation of production line architecture directly from the model. When we think of Feature Model not only in terms of visualizing product features, but also as a notation that can express dependencies of feature production processes, it seems that we have a tool for representation of product features in an intuitive way that allows for managing the means of production. This fact can be particularly valuable when our production line can change dynamically by adding new features (it can be noticed that similar situation took a place

7.2. A concept of Feature Model - driven Software Product Line architecture

in Cloudberries). As we virtually base our means of production on a single model, it seems that the architecture of production line variability can be managed using the information kept in Feature Model. This section presents a concept of production line system architecture that is based on extended Feature Model introduced in the previous section.

The component diagram presented in the Figure 7.9 shows the inner structure of core components of the proposed production line architecture. On the left side of the picture one can see the interfaces that are interesting from the perspective of means of production. Interface on the right side is dedicated to be used by the front-end for running the production line in order to create products.



**Figure 7.9.** The internal construction of the main production line component. The main inner component is the *Workflow Manager* that controls the production process. *Production Site* plug-ins, *Feature Models* and *Feature Descriptors* are registered in appropriate registries.

The production line architectural design is based on two basic artifacts: *Feature Model* containing dependencies between features of product and *Feature Descriptor* which maps features to the means of production (*Production Sites*). *Production Site* is an interface that has to be implemented to provide procedure of feature installation. More precise description of the production line architectural components is presented below.

### Feature Model

*Feature Model* describes the domain of products in terms of features and relationships between them. Relationships are identical to those presented in the previous section:

- parent - child relationship,
- dependency relationship,
- installation order relationship (can be represented as a variant of above two).

Each feature has to be represented by a unique identifier in order to associate it with an appropriate *Feature Descriptor*.

### Feature Descriptor

*Feature Descriptor* describes requirements of feature installation processes in form of input and output interfaces declaration. Moreover, *Feature Descriptor* connects a feature to the *Production Site* that provides means of the feature installation. Corresponding *Production Site* has to accept this declaration (see `isFeatureSupported()` of *ProductionSite* interface). *Feature Descriptor* can be thought of as a structure similar to the example below:

```
1 structure FeatureDescriptor {
2     Id : featureId;
3     Id : productionSiteId;
4     Set of IOPParameter : inputParameters;
5     Set of IOPParameter : outputParameters;
6 }
7 structure IOPParameter {
8     Id : parameterId;
9     Type : parameterType;
10 }
```

**Figure 7.10.** The concept of Feature Descriptor structure. This structure can be for example mapped to XML format.

For all input/output parameters identifiers must be unique among *Feature Descriptors* connected with a single model. Input/output interfaces will be used to create a contract between features. The process of binding features with contracts is introduced later in the description of the *Feature Descriptor Registry*.

## 7.2. A concept of Feature Model - driven Software Product Line architecture

### Production Site

Production Site is an interface for implementation of production line plugin. Production Site has to accept all of the features that declared association with it, before the feature model is validated and production line can be used (see the function `isFeatureSupported()` in the Figure 7.11). During the process of production, *Workflow Manager* prepares production schedule and invokes installation procedure (`installFeature()`) on each of *Production Site* corresponding to the features contained in the product configuration. *Production Site* interface is very simple and consists of two functions presented in the Figure 7.11:

```
1 interface ProductionSite {
2     getProductionSiteId () : Id;
3     isFeatureSupported ( featureId ) : Boolean;
4     installFeature (
5         FeatureDescriptor : featureDescriptor ,
6         ProductionSiteInput : input ,
7         ProductionSiteOutput : output ,
8         Set of IndependentSiteOutput
9         : ouptutsTheSiteDependsOn ) : Void;
10 }
11
12 structure IndependentSiteOutput{
13     FeatureDescriptor : featureDescriptor;
14     ProductionSiteOutput : siteOutput;
15 }
```

**Figure 7.11.** ProductionSite interface to be implemented in order to provide means of feature installation (see procedure `installFeature`). As a Production Site is controlled from the outside the function `isFeatureSupported` has to be implemented to declare support for production of feature with a given identifier.

- **getProductionSiteId** - *Production Site* returns its identifier.
- **ifFeatureSupported** - *Production Site* returns true if it can provide means of production of a given feature.
- **installFeature** - *Production Site* should perform installation after invocation of this function. Parameters: `featureDescriptor` - descriptor of feature being installed; `input` - *Production Site* can expect to be provided with input that meets `featureDescriptor` parameters; `output` - *Production Site* must fill in all callback fields declared by *FeatureDescriptor*; `ouptutsTheSiteDependsOn` - output of sites installing the features, this site de-

## 7. A CONCEPT OF FEATURE MODEL - BASED AUTOMATIC SPL GENERATION

---

depends on (by FM dependency relationship), which was installed before (installation order relationship).

### Feature Model Registry

*Feature Model Registry* allow the production line system to store feature models in order to handle multiple production lines. Each production line is based on a model managed by the registry. The lifecycle of a single production line starts with model registration. A production line can be used to create products after *releasing* corresponding feature model. *Feature Model Registry* uses *Feature Descriptor Registry* and *Production Site Registry* to validate the model before *releasing* it. Validation process determines whether the contract of interfaces between related features is satisfied. Another procedure performed by the *Feature Model Registry* before *releasing* a feature model is testing whether *Production Sites* declared in *Feature Descriptors* accept support installation of corresponding features. In order to do that, *Feature Model Registry* uses *Production Site Registry* to get the *Production Site* object and check if `isFeatureSupported()`.

### Feature Descriptor Registry

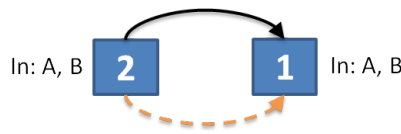
Collects *Feature Descriptors* for a given feature model. Before a feature model can be released, each feature has to be described by the corresponding *Feature Descriptor* stored in the registry.

Contracts of interfaces:

*Feature Descriptors* declare requirements of installation for each feature and output that is provided by processes of feature installation. This declarations form interfaces for *Production Sites* tied with features. These interfaces allow to connect inputs and outputs of installation procedures, in order to allow *Production Sites* to share some information. In general, there are two types of contracts that may bind feature installation interfaces, specified by the direction of feature installation order relationship that connects features.

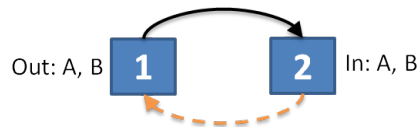
Let's see an example in the Figure 7.12. There is a pair of features connected with a dependency relationship (marked with the black edge - the dependent) and an orange relationship of installation order.

## 7.2. A concept of Feature Model - driven Software Product Line architecture



**Figure 7.12.** Sample contract of interfaces. Installation procedure of feature 2 depends on 1 and provides feature 1 *Production Site* with its own input. Feature 1 will be installed before feature 2.

In the Figure 7.12, relationship edges are at the same direction. The installation order edge points at the feature that will be installed first (as a natural analogy to the dependency arrow direction). It means that dependent feature 2 will be installed after 1. So that, when the feature model expresses a will to have the feature 1 installed before the installation of feature 2 starts, the descriptor of feature 2 must declare input interface that satisfy the input interface of feature 1. Feature 1 requires two input parameters - A and B. Because of installation order, feature 2 cannot provide feature 1 with its own output, which is not ready at the time of installation of feature 1. That is why feature 2 has to declare A and B input parameters as well.



**Figure 7.13.** Sample contract of interfaces. Feature 1 depends on 2. Feature 1 will be installed before feature 2 and provides feature 1 with output emerged from its own installation.

In the Figure 7.13 the situation is just the opposite. Feature 1 will be installed before the installation of feature 2 starts. So that, feature descriptor of 1 must declare that the production site will provide 2 with parameters A and B right after it installs the feature. If two features depends equally on the third, either the relative order must be specified by the user or some additional order rule has to be applied.

Another interesting issue is a behavior of "independent" inputs. Even if dependency is specified it does not mean that dependent feature will be selected by the user if the independent is chosen. Therefore, each input which is not provided by a dependent feature has to be prompted by the user. Because of the uniqueness of parameter identifiers it is easy to propagate manually provided input among all of the features dependent on each parameter. It should be noted at this point that any output of independent *Production Site* is available for the dependent *Production Site* as an installation function parameter named `outputsTheSiteDependsOn`.

### **Production Site Registry**

Manages *Production Sites*. In order to use *Production Site* in the production line it has to be registered in context of a given feature model. Once a *Production Site* is registered, it can be used for installation of feature in a product. In order to assign production of feature to *Production Site*, appropriate *Feature Descriptor* has to be registered in the *Feature Descriptor Registry*.

### **Workflow Manager**

*Workflow Manager* is the heart of the product line system. It takes care of several processes, in order to perform product creation:

- Uses released feature model to create production schedule.
- Invokes appropriate Production Sites in order to install features.
- Passes inputs and outputs from/to appropriate Production Sites.

The *Workflow Manager* connects all of the previously mentioned components into consistent production line. The *Workflow Manager* has to deal with production request handling. So that, from the top level point of view it is the component that is closest to the front end.

## **7.3 Further thoughts**

There are some issues that was not covered by the previous section because of its introductory character. In this paragraph there are presented some additional considerations in terms of presented architectural concept.

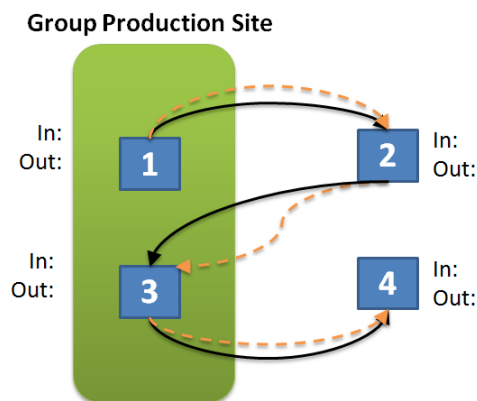
### **7.3.1 Group production site**

During the experimentation with application of Software Product Line to automation of experiment environment deployment, the author realized that there are some cases in which installation of multiple features as a group is very convenient. There may also exist some circumstances that require features to be installed as a group. As an example the process of installation performed by Chef can be recalled. When the Chef runs, it installs its client on the target machine and then applies all deployment procedures of selected cookbooks. There is no way to suspend Chef's operation between deployment of two cookbooks in order to install another in the meantime (using method of deployment other than Chef). It appears that in this case there is a need for a dedicated approach, in order to deal with modeling dependency of installation processes.

In order to present this approach let's define a problem once again, using more generic example. In the picture 7.14 one can see four features of which



installation processes can be ordered using appropriate relationships. As it was mentioned before in this example exist a *Group Production Site*, which is implemented in a way that does not allow to split installation of feature 1 and 3 into separate installation processes. So that, there is no way to create a product containing all of the features (1,2,3,4). Furthermore, following the ordering algorithm presented in the section 7.1 there is no way to find out if the configuration containing these four features, is correct in terms of installation schedule.

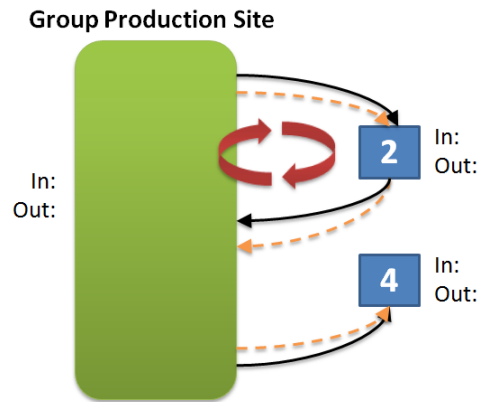


**Figure 7.14.** An illustration of the problem of scheduling the installation performed by *Group Production Site*. The *Group Production Site* is a *Production Site* that does not allow for separate installation of supported features. The process of subordinated features installation has to be performed as an atomic, indivisible group.

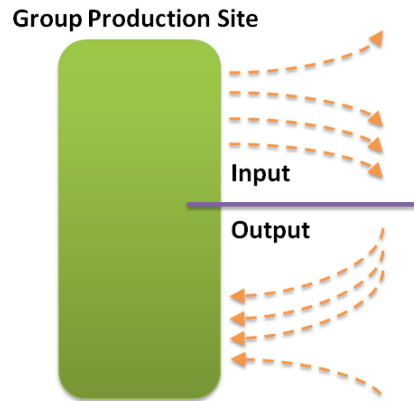
The solution for the problem is quite simple, and does not require any serious changes in the method presented before. We will simply treat the whole group of features as a single feature that inherits all of the relationship edges from the inner features as it is presented in the picture 7.15. Now, if the configuration of features contains a cycle, we can see that the scheduling will not end up successfully. The scheduling will be feasible if ordering relationship between the features contained in the group and the rest of the features will be split into two separate groups, as it is visible in the picture 7.16

## 7. A CONCEPT OF FEATURE MODEL - BASED AUTOMATIC SPL GENERATION

---



**Figure 7.15.** The figure presents exactly the same situation as in the previous picture, but all of the features installed by a *Group Production Site* are treated as a single feature (the green rectangle). This approach allows to check if there are any cycles which causes scheduling unfeasible. In this figure we can see a cycle of the installation ordering relationship graph, so the model that contain features arranged in this way, should be considered invalid.



**Figure 7.16.** A symbolic representation of a feature model part with features assigned to a *Group Production Site* in a way that is correct in terms of installation scheduling. Scheduling is feasible when features connected to a group with installation ordering relationships can be split into two independent groups. The first contains only features that are installed before the features in green rectangle (installed by *Group Production Site*). Second group contains only features installed afterwards.

### 7.3.2 Scheduling algorithm

The algorithm presented in the section 7.1 is not perfect. First of all it does not allow to handle the problematic situation presented in the Figure 7.8. Another factor that should be taken into account while improving the algorithm, is the use of *Group Production Sites* presented in the previous section. The actual algorithm will be the subject of further work.

## 7.4 Summary

This chapter presents reasoning on Software Product Line inspired by the research and tool implementation. In order to implement the idea of automatic generation of production line from Feature Model, the author introduced the model extension. Described modifications affect relationships specified by the model syntax, in the way that allows for scheduling of the installation procedures. Moreover, this chapter describes an architectural concept based on the extended Feature Model, which may be used to implement a framework for software products creation supporting Software Product Line paradigm. Presented architectural concept constitutes an extensible application layer that allows to link user-side software product configuration with the means of production. The framework may be expandable in terms of new product features as well as the procedures of feature installation. As certain details of the solution presented in this chapter still need polishing up it will be a subject for future work.



## Chapter 8

---

# Summary and Future Work

---

The main objective of this thesis was creation of a tool for installation of e-Science applications in heterogeneous cloud infrastructure. The presented solution was based on provisioning system (Chef) and allows for software deployment on an instance of virtual machine. The tool had to satisfy the needs of scientific user, so the design was focused on minimizing the complexity of configuration process, in order to facilitate the use. The tool was deployed in the production environment of VPH-Share project. The system has been assessed and found to comply with the requirements originating in the project.

Another goal of the implementation was investigation of Software Product Line application in the domain of e-Science. The tool proves the concept of implementation of an environment configuration, as a process guided by dependency management based on automatic reasoning on Feature Model. During the work there was also elaborated a concept of extensible production line architecture that is derived from a feature model. The concept may constitute a base for implementation of a framework for creation of extensible production lines.

There are some tasks connected with the work executed in scope of this thesis that were considered to be worth continuation. The list below presents subjects for the further work in two groups. The first group is connected with possibilities of the tool enhancements:

- Implementation of multiple node deployment tasks. For now the tool allows for running deployment on a single virtual machine. Extension of the user interface for deployment of a single configuration on multiple nodes, would make the process of environment preparation even easier.
- Implementation of filtering mechanism in the user interface. In the current version of the system, the user finds the interesting configurations

## 8. SUMMARY AND FUTURE WORK

---

and other entities by browsing elements stored by all of the users. Entity filtering mechanism would facilitate the use of the web interface.

- Implementation of a user interface allowing for feature model loading and edition. The implementation of the tool was focused on providing the interface for a scientific user. In order to allow to easily manage the system, specialized administration interface can be implemented.
- Implementation of mapping between Cloudberries user accounts and accounts of Chef server. For now, Cloudberries uses a dedicated Chef account. In order to monitor activities of users, each of the should use its own Chef credential.
- Implementation of on-demand VM instance creation. There is a plan for integration with a library for VM instantiation in the private cloud of VPH-Share.

The other group addresses some tasks concerning the reasoning on Software Product Line and presented production line architectural concept:

- There is a need for improvement of production line scheduling algorithm presented in this thesis. The algorithm has to cope with issues specific for ordering of the phases of production, using a directed graph composed of relationships contained in extended Feature Model.
- Implementation of a service-oriented framework for production line creation. There is a plan of software product realization based on the presented reasoning. The application may ease the process of creation of dedicated production line, based on an architecture derived from a feature model.

The further work presented in this chapter will be executed in scope of VPH-Share project.

# Appendices





# Appendix A

---

## Glossary

---

**Binary Decision Diagram** - a compact encoding for Boolean formulas used to implement numerous reasoning algorithms. The structure of a diagram is defined by a Directed Acyclic Graph representing whole combinatorial space.

**Boolean satisfiability problem (SAT)** - a mathematical problem defined as a set of Boolean variables and constraints the variables must satisfy.

**Chef** - Chef is an open-source systems integration framework built for automating the cloud. Chef belongs to the group of provisioning tools. The experiment environment deployment tool implemented in the scope of this thesis is based on Chef.

**Cloudberries** - the codename of the e-Science application deployment tool built in scope of this thesis. This thesis describes several aspects of Cloudberries design, implementation and evaluation.

**Cloud Computing** - a computing paradigm based on virtual infrastructures delivered as hosted services over computer network.

**Conjunctive Normal Form** - a formula of Boolean logic, that is a conjunction of clauses, where a clause is a disjunction of literals. The Conjunctive Normal Form is particularly useful in automated reasoning.

**Cookbook** - the notion used in the nomenclature of Chef to describe an installation package that contains resources and scripts included in the process of environment configuration.

**Deployment Task** - an entity representing the process of experiment environment deployment that is managed by Cloudberries.

**Deployment Template** - a stored selection of environment components and predefined installation attributes that is managed by Cloudberrries.

**Distributed Shell** - a class of software products for operating system configuration based on parallel usage of multiple remote shells.

**e-Science** - computationally intensive science carried out in highly distributed network environments. Most of the e-Science research activities are focused on the development of tools to support scientific discovery.

**e-Science application** - a scientific experiment built as a computer simulation. e-Science application are composed from various components including software and data.

**Feature Model** - a notation used in Software Product Line to define the domain of product features. Features are connected into a hierarchy which represent mutual dependencies between features.

**In silico** - a term denoting "performed on computer or via computer simulation" in terms of scientific experiment.

**Jetspeed** - Open Portal Platform and Enterprise Information Portal, written entirely in open source under the Apache license in Java and XML and based on open standards. Cloudberrries is a portlet application deployed in Jetspeed portal.

**Knife** - a tool from the Chef family used to manage server and clients in order to perform installation and configuration of environment components.

**Portlet** - pluggable user interface software component that is managed and displayed in a web portal.

**Provisioning tools** - a family of software products that allow for automatic deployment of environment components in distributed computer infrastructure.

**Software Product Line** - a paradigm for systematic reuse of common assets, from which different programs of a domain can be assembled. Reuse of code, data and configuration, together with reproducible assembly methods resemble the operation of production line in a factory.

**Software Product Line Automated Reasoning** - a Java-based reasoning library used in the implementation of the tool developed in scope of this thesis.

**Unattended Installation** - installation that is performed without user interaction during its progress or with no user present at all. There exist a group

---

of software that support unattended installation for numerous operating systems.

**Virtual Machine** - virtualization technology that enables running operating system instances in isolated environments managed by a hypervisor. Hypervisor performs emulation on hardware resources, and enable multiple Virtual Machines to run simultaneously.



# Appendix B

---

## Installation Guide

---

*This guide presents the process of Cloudberries installation by the example. Section B.1 presents installation of prerequisites needed to run Cloudberries. Section B.2 describes installation of the Cloudberries portal. All of the files needed for installation are delivered on the CD attached to this thesis.*

### B.1 Installing prerequisites

In order to install Cloudberries we have to go through several steps of prerequisites installation. In this example it is presented installation on Ubuntu. Cloudberries depend on several components:

- Chef server
- relational database (MySQL is used by default)
- JetSpeed portal (deployed on Apache TomCat web application server)

These three components can be deployed on the same machine or separated from each other. In this example all of the components will be installed on a single operating system. The process of Cloudberries installation is presented below:

#### B.1.1 Chef installation

First, we have to have running instance of Chef server. In order to install Chef refer to its documentation:

`http://wiki.opscode.com/display/chef/Installing+Chef+Server`

In a nutshell, Chef server can be installed via apt-get, after updating Debian packages (refer to the documentation). In order to do that use the below command:

## B. INSTALLATION GUIDE

---

```
shell> sudo apt-get install chef chef-server-api chef-expander
```

Once we have Chef installed, we have to configure *Knife* command line tool in order to manage Chef server. The procedure is described on the same page of the Chef documentation. In order to test whether our client is configured properly we can list all of the clients registered in the server:

```
shell> knife client list
  chef-webui
  bob
  chef-validator
```

Now we can create our user which will be used to manage the server (it is also described in the mentioned documentation). The key path `/tmp/my-username.pem` will be used later in Cloudberry's configuration file.

```
shell> knife client create myuser -d -a -f /tmp/myuser.pem
```

Chef runs by default on two ports - 4000 for API and 4040 for WebUI. The first one will be used by Cloudberry to communicate with Chef.

### B.1.2 Database installation

In fact, Cloudberry can use any relational database supported by Hibernate, but is by default configured to use MySQL (configuration of Cloudberry from the perspective of database will be presented later). Installation of MySQL on Ubuntu is easy. We simply use `apt-get` command:

```
shell> sudo apt-get install mysql-server mysql-client
```

By default MySQL listens on port 3306. In order to use custom database installation refer to the MySQL documentation:

<http://dev.mysql.com/doc/refman/5.1/en/installing.html>

We need to create single user and a single database for Cloudberry. In order to do that we log into database using privileged account.

```
shell> mysql -user=root -password
mysql> CREATE USER 'myuser'@'localhost' IDENTIFIED BY 'mypass';
mysql> CREATE DATABASE cloudberry
mysql> GRANT ALL PRIVILEGES ON cloudberry.* TO
'myuser'@'localhost'
```

Now, our database is ready to be used by Cloudberry.

### B.1.3 JetSpeed installation

JetSpeed is a portal running in a Apache TomCat web application server. As Cloudberries is a portlet application it is deployed in running instance of portal. Apart from JetSpeed, it is possible to use virtually any other portal that implements Portlet API specification [29]. However, this guide covers only installation in JetSpeed. The easiest way to install JetSpeed is to use its dedicated installer which copes with installation of TomCat. The other way of installation is to deploy JetSpeed in an existing TomCat instance. For information regarding the second approach refer to the JetSpeed documentation:

```
http://portals.apache.org/jetspeed-2/getting-started-installer.html
```

In order to install JetSpeed in a brand-new TomCat web application server, we need to get the installer binaries. The list of mirrors is included in the documentation. Once we have downloaded the installer we have to run it and follow installation steps.

```
shell> wget http://ftp.ps.pl/pub/apache/portals/jetspeed-2/binaries/jetspeed-installer-2.2.2.jar
shell> java -jar jetspeed-installer-2.2.2.jar
```

When our JetSpeed is installed successfully, we can run it and test whether the portal is working. Dedicated TomCat is by default installed in the HOME directory. So, in order to run the server we execute the command:

```
$HOME/Jetspeed-2.2.2/bin/catalina.sh run
```

The server may be also executed as a daemon using `startup.sh` script. In order to test if the portal is working paste this address into a Web browser (the default user of administration panel - admin, password - admin):

```
http://localhost:8080
```

## B.2 Cloudberries portlet installation

The installation package of Cloudberries consists of three elements: portlet application archive (`cloudberries-pa.war`), portal prerequisites (a folder named `jetspeed`), and sample feature model (`sample-model.xml`). First, we have to open our archive `cloudberries-pa.war`, get to `/WEB-INF/classes` and edit files `cloudberries.cnf.xml`, `hibernate.cnf.xml`.

### B.2.1 Cloudberries settings file (`cloudberries.cnf.xml`)

`cloudberries.cnf.xml` one looks as follows:

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <config>
3   <chef>
4     <identity>chef-user-name</identity>
5     <endpoint>http://chef-host:4000</endpoint>
6     <pem>/location/of/the/chef/user/credential.pem</pem>
7   </chef>
8   <runtime>
9     <model-repo>/repository/location</model-repo>
10    <model-file>repo:sample-model.xml</model-file>
11    <log-parent>/tmp/logs</log-parent>
12    <portlet-decoration>/name-of-the-portal/decorations/
13    portlet/cloudberries</portlet-decoration>
14    <fake-deploy>>false</fake-deploy>
15  </runtime>
16 </config>
```

We can see two sections that apply to communication with Chef server and runtime configuration of Cloudberryes. In order to communicate with the Chef instance, in our example we have to change `chef-user-name` to the name of our newly created Chef user `myuser`, `chef-host` will be `localhost` (port remains the same) and set `credential` location to point to our `/tmp/myuser.pem`.

Now we can adjust Cloudberryes runtime parameters:

- `model-repo` is a path that will be searched to find feature models. We can specify absolute path of the folder containing `sample-model.xml`.
- `model-file` - model locator. We leave the section as it is.
- `log-parent` - is a folder where will be kept installation logs. The user running application server has to have rights to create new folders there.
- `portlet-decoration` - relative path to the portal decoration folder. In a newly installed JetSpeed, we will only change `name-of-the-portal` to the portal name (the name is `jetspeed` by default).
- `fake-deploy` - this option allows for testing the application without running installation processes. If set to `true` Cloudberryes pretends that installation steps finish immediately. We leave `false`.



The resulting configuration for our example should look as follows:

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <config>
3   <chef>
4     <identity>myuser</identity>
5     <endpoint>http://localhost:4000</endpoint>
6     <pem>/tmp/myuser.pem</pem>
7   </chef>
8   <runtime>
9     <model-repo>/tmp/cloudberries-guide</model-repo>
10    <model-file>repo:sample-model.xml</model-file>
11    <log-parent>/tmp/logs</log-parent>
12    <portlet-decoration>/jetspeed/decorations/
13    portlet/cloudberries</portlet-decoration>
14    <fake-deploy>>false</fake-deploy>
15    </runtime>
16 </config>
```

### B.2.2 Database connection settings (hibernate.cfg.xml)

Cloudberries connects to the database using Java Persistence API provider - Hibernate. In order to support other databases than MySQL there is a need for providing JDBC driver implementation specific to the type of the database. Appropriate JAR has to be copied into the /WEB-INF/lib/ folder of the application. In this guide we will simply use default MySQL driver. To get more details about Hibernate configuration refer to its documentation. The configuration is quite obvious. In our example resulting configuration file has to contain the lines listed below:

```
1 <property name="connection.driver_class">
2   com.mysql.jdbc.Driver</property>
3 <property name="connection.url">
4   jdbc:mysql://localhost:3306/cloudberries</property>
5 <property name="connection.username">myuser</property>
6 <property name="connection.password">mypass</property>
```

### B.2.3 Portal prerequisites

In the jetspeed folder, which is included in the installation package, you can find two following folders: decorations and pages,

contents of which should be placed in appropriate locations. Both of them represent hierarchy of folders ending with a folders named cloudberries. The structure of the folders is identical to the following two: `$HOME/Jetspeed-2.2.2/webapps/jetspeed/decorations` and `$HOME/Jetspeed-2.2.2/pages/` (ATTENTION - it can be alternatively located at `$HOME/Jetspeed-2.2.2/webapps/jetspeed/pages`). In order of install prerequisites just reproduce the structure and copy folders named cloudberries into appropriate locations.

### B.2.4 Running the application

In order to run Cloudberries, make sure that the JetSpeed portal is running. JetSpeed supports hot-deploy so the portlets can be seamlessly deployed without restart. In order to deploy Cloudberries just copy the archive `cloudberries-pa.war` to the deployment folder `$HOME/Jetspeed-2.2.2/webapps/jetspeed/WEB-INF/deploy`. Then you should see logs of the server:

```
Creating war /home/bartek/Jetspeed-2.2.2/webapps/cloudberries-pa.war
War /home/bartek/Jetspeed-2.2.2/webapps/cloudberries-pa.war
created
Aug 24, 2012 3:18:45 AM org.apache.catalina.startup.HostConfig
deployWAR
INFO: Deploying web application archive cloudberries-pa.war
JetspeedContainerServlet: starting initialization of Portlet
Application at: cloudberries-pa
JetspeedContainerServlet: initialization done for Portlet
Application at: cloudberries-pa
```

When Cloudberries is succesfully deployed, it writes its own log:

```
Reading Cloudberries configuration from file
"/home/bartek/Jetspeed-2.2.2/webapps/cloudberries-pa/WEB-INF/classes
/cloudberries.cfg.xml"
=====
Cloudberries configuration:
=====
chefIdentity = myuser
chefEndpoint = http://localhost:4000
pemFileAbsolutePath = /tmp/myuser.pem
repoLocation = /tmp/cloudberries-guide
logParentPath = /tmp/logs
portletDecorationBase = /jetspeed/decorations/portlet/cloudberries
fakeDeploy = false
=====
```

## B.2. Cloubderries portlet installation

---

In order to access the tool paste following link into a Web browser:

`http://localhost:8080/jetspeed/portal/cloubderries`

The sequence jetspeed is a name of the portal and should be replaced if the portal name has changed. As Cloubderries is a portlet it can be made accessible from the main portal page - to get more information refer to the JetSpeed documentation.



---

# Bibliography

---

- [1] *Anaconda/Kickstart - FedoraProject*. URL: <http://fedoraproject.org/wiki/Anaconda/Kickstart>.
- [2] Henrik R. Andersen. *An introduction to binary decision diagrams*. Tech. rep. Course Notes on the WWW, 1997.
- [3] *Architecture Introduction - Chef - Opscode Open Source Wiki*. URL: <http://wiki.opscode.com/display/chef/Architecture+Introduction>.
- [4] *Automatic Installation - Ubuntu Linux*. URL: <http://help.ubuntu.com/12.04/installation-guide/i386/automatic-install.html>.
- [5] Don Batory, Jacob N. Sarvela, and Axel Rauschmayer. "Scaling Step-Wise Refinement". In: *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING* 30.6 (2004), p. 2004.
- [6] *Bcfg2 project web page*. URL: <http://trac.mcs.anl.gov/projects/bcfg2/>.
- [7] *Bcfg2 Windows support*. URL: <http://trac.mcs.anl.gov/projects/bcfg2/wiki/FeatureWindows>.
- [8] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. "Automated analysis of feature models 20 years later: A literature review". In: *Information Systems* (Mar. 4, 2010). ISSN: 03064379. DOI: 10.1016/j.is.2010.01.001. URL: <http://dx.doi.org/10.1016/j.is.2010.01.001>.
- [9] *CFEngine - Distributed Configuration Management*. URL: <http://cfengine.com/>.
- [10] *Chef - Opscode Open Source Wiki*. URL: <http://wiki.opscode.com/display/chef/Home>.
- [11] *Chef Opscode*. URL: <http://www.opscode.com/chef/>.
- [12] *Choco solver web page*. URL: <http://www.emn.fr/z-info/choco-solver/>.

## BIBLIOGRAPHY

---

- [13] choco Team. *choco: an Open Source Java Constraint Programming Library*. Research report 10-02-INFO. École des Mines de Nantes, 2010. URL: <http://www.emn.fr/z-info/choco-solver/pdf/choco-presentation.pdf>.
- [14] E. Ciepiela et al. "Exploratory Programming in the Virtual Laboratory". In: *Proceedings of the International Multiconference on Computer Science and Information Technology*. Wisla, Poland 2010, pp. 621–628.
- [15] *ClusterIt project web page*. URL: <http://www.garbled.net/clusterit.html>.
- [16] *ClusterSSH project web page*. URL: <http://clusterssh.sourceforge.net>.
- [17] *Comparison of open source configuration management software - Wikipedia, the free encyclopedia*. URL: [http://en.wikipedia.org/wiki/Comparison\\_of\\_open\\_source\\_configuration\\_management\\_software](http://en.wikipedia.org/wiki/Comparison_of_open_source_configuration_management_software).
- [18] *Detailed Bcfg2 Architecture à Bcfg2 1.2.3 documentation*. URL: <http://docs.bcfg2.org/architecture/>.
- [19] *Dish project webpage*. URL: <http://directory.fsf.org/wiki/Dish>.
- [20] *DSH - dancer's shell / distributed shell project web page*. URL: <http://www.netfort.gr.jp/~dancer/software/dsh.html.en>.
- [21] *euHeart project website*. URL: <http://www.euheart.eu/>.
- [22] *FaMa project web page*. URL: [http://www.isa.us.es/fama/?FaMa\\_Framework](http://www.isa.us.es/fama/?FaMa_Framework).
- [23] *FAMA tool suite web page*. URL: <http://www.isa.us.es/fama/>.
- [24] *FaMa user manual*. URL: <http://famats.googlecode.com/files/FaMaUserManual.pdf>.
- [25] *FogProject web page*. URL: <http://www.fogproject.org/>.
- [26] *FreeMarker: Java Template Engine Library - Overview*. URL: <http://freemarker.sourceforge.net/>.
- [27] Hassan Gomaa and Michael Shin. "Automated Software Product Line Engineering and Product Derivation". In: *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. Waikoloa, HI, USA: IEEE, 2007, 285a. DOI: 10.1109/HICSS.2007.95. URL: <http://dx.doi.org/10.1109/HICSS.2007.95>.
- [28] Tarik Hadzic, Rune M. Jensen, and Henrik R. Andersen. *Calculating valid domains for bdd-based interactive configuration*. 2006.
- [29] *Introducing Java Portlet Specifications: JSR 168 and JSR 286*. URL: <http://developers.sun.com/portalserver/reference/techart/jsr168/>.

- 
- [30] Nicholas W. Jankowski. "Exploring e-Science: An Introduction". In: *Journal of Computer-Mediated Communication* 12.2 (Jan. 2007), pp. 549–562. ISSN: 1083-6101. DOI: 10.1111/j.1083-6101.2007.00337.x. URL: <http://dx.doi.org/10.1111/j.1083-6101.2007.00337.x>.
- [31] *Java Servlet Technology*. URL: [http://java.sun.com/j2ee/tutorial/1\\_3-fcs/doc/Servlets.html](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets.html).
- [32] *JavaBDD - Java Binary Decision Diagram library*. URL: <http://javabdd.sourceforge.net/>.
- [33] *Jclouds-chef Wiki - GitHub*. URL: <http://github.com/jclouds/jclouds-chef/wiki/Quick-Start>.
- [34] *Jetspeed 2 documentation*. URL: <http://portals.apache.org/jetspeed-2/>.
- [35] K. C. Kang et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. Carnegie-Mellon University Software Engineering Institute, 1990.
- [36] Kyo C. Kang et al. "FORM: A feature-oriented reuse method with domain-specific reference architectures". In: *Annals of Software Engineering* 5 (1998).
- [37] Vipin Kumar. "Algorithms for Constraint Satisfaction Problems: A Survey". In: *AI MAGAZINE* 13.1 (1992).
- [38] Daniel Le et al. *The Sat4j library, release 2.2 system description*. 2010.
- [39] Marc'ilio Mendonca. "Efficient Reasoning Techniques for Large Scale Feature Models". PhD thesis. Waterloo: University of Waterloo, 2009, p. 184. URL: <http://hdl.handle.net/10012/4201>.
- [40] Marc'ilio Mendonça, Andrzej Wasowski, and Krzysztof Czarnecki. "SAT-based analysis of feature models is easy." In: *SPLC*. Ed. by Dirk Muthig and John D. McGregor. Vol. 446. ACM International Conference Proceeding Series. ACM, 2009, pp. 231–240. URL: <http://dblp.uni-trier.de/db/conf/splc/splc2009.html#MendoncaWC09>.
- [41] *Multivax: Bcfg2 vs Puppet*. URL: <http://multivax.blogspot.com/2009/12/bcfg2-vs-puppet.html>.
- [42] *@neurIST (Integrated Biomedical Informatics for the Management of Cerebral Aneurysms) - project website*. URL: <http://www.aneurist.org/aneurist1/index.php>.
- [43] *Omnitty SSH multiplexer*. URL: <http://omnitty.sourceforge.net/>.
- [44] *PSSH project web page*. URL: <http://www.theether.org/pssh/>.
- [45] *Puppet Labs Documentation*. URL: <http://docs.puppetlabs.com/>.
- [46] *Puppet versus Chef: 10 reasons why Puppet wins*. URL: <http://bitfieldconsulting.com/puppet-vs-chef>.

## BIBLIOGRAPHY

---

- [47] *SAT4J web page*. URL: <http://www.sat4j.org/>.
- [48] *Server API - Chef - Opscode Open Source Wiki*. URL: <http://wiki.opscode.com/display/chef/Server+API>.
- [49] *Software Product Lines Online Tools architecture*. URL: [http://gsd.uwaterloo.ca:8088/SPLIT/splot\\_open\\_source.html](http://gsd.uwaterloo.ca:8088/SPLIT/splot_open_source.html).
- [50] *Solaris 10 Installation Guide: Custom JumpStart and Advanced Installations*. URL: <http://docs.oracle.com/cd/E19253-01/817-5506/>.
- [51] *SPLIT - Software Product Line Online Tools*. URL: <http://www.splot-research.org/>.
- [52] *SXFM Feature Model format*. URL: <http://www.splot-research.org/sxfm.html>.
- [53] *TakTuk project webpage*. URL: <http://taktuk.gforge.inria.fr/>.
- [54] *The AHEAD Tool Suite*. URL: <http://www.cs.utexas.edu/users/schwartz/ATS.html>.
- [55] "Thoughts on cfengine, bcfg2, and puppet | The Changelog". In: (). URL: <http://changelog.complete.org/archives/519-thoughts-on-cfengine-bcfg2-and-puppet>.
- [56] Thomas Thüm, Don Batory, and Christian Kästner. "Reasoning about edits to feature models". In: *In Proc. Int'l Conf. on Software Engineering*, p. 2009.
- [57] *Unattended, A Windows deployment system*. URL: <http://unattended.sourceforge.net/>.
- [58] *VPH-Share project web page*. URL: <http://vph-share.org/>.