Bartosz Wilk, Marek Kasztelnik
AGH Krakow, ACC Cyfronet
{b.wilk,m.kasztelnik}@cyfronet.pl
Marian Bubak
AGH Krakow,
Department of Computer Science and ACC Cyfronet
bubak@agh.edu.pl

Software for eScience: from feature modeling to automatic setup of environments

Abstract

To increase our productivity when setting up various software environments, we try to reduce the complexity of configuration tasks by managing components at different levels of abstraction and by automation of the process. This is particularly important (in terms of performance) when the configuration is not the direct objective of our activity. When deploying environments for eScience applications the researcher's main interest lies in executing experiments and obtaining results, not in tedious fine-tuning of the computational platform itself. Tackling the challenge of automatically setting up environments for *in-silico* experiments is the main motivation behind the discussion presented in this work. In such a task, clear representation and processing of component dependencies pose a challenge. In this work the Feature Model notation, popular in the Software Product Line methodology and successfully applied to configuration modeling, was examined for this purpose. This article shows a feasibility study of applying Feature Model to develop tools for automatic environment configuration on the basis of a prototype implementation. The presented discussion has led the authors to further extend this idea, covering a wider range of applications. The paper describes the architecture of an extensible framework automating various deployment and component installation tasks, based on the Feature Model.

Keywords: Feature Model, Software Product Line, eScience, automation, provisioning

Introduction

eScience [6, pp. 549 - 552] is more and more popular approach to scientific research. However, new research paradigms such as simulation and data intensive processing entails new challenges. Preparation of eScience application execution environment is a complex and time consuming process, which incorporates configuration of many cooperating components. As such components include various applications and datasets, their deployment requires extensive knowledge on operating systems administration, cloud platforms, communication protocols and others.

After requirements analysis and review of available technologies, there appeared a need to create a tool enabling selection of eScience application prerequisites in a simple and intuitive way, and then deploy them in a given environment. The tool presented here combines the use of the *Feature Model* for modeling components domain with the *Provisioning* system for application deployment. The *Feature Model* is a representation of product features relationships, known for its broad application in science and industry, in particular in the *Software Product Lines*.

The objective of this paper is to present the feasibility study of using the *Feature Model* for modeling eScience applications components dependencies, implemented as a tool for automatic deployment of execution environments. The architecture described in the section 1. was implemented as a prototype system using tools and libraries presented in the section 2. and evaluated in the case study (section 3.). Based on this evaluation a generalization of the idea was proposed (section 4.). We also consider a question – how appropriate is the presented approach for configuration of environments with wider range of applications (not only eScience) as well as broader spectrum of installation methods (not only *Provisioning Tools*). Architectural concept validation results in the design of an expansible software production line framework, which better fulfills presented criteria of evaluation. Related work is presented in the section 5.

1. Description of the proposed solution

eScience applications consist of many modules, often using separate technological stacks. In the framework of the VPH-Share [25] examples of such applications include workflows @neurIST [14] (neural system simulation), euHeart [19] (human heart simulation), Virolab [25] (virtual virological laboratory), VPHOP [27] (prediction of osteoporotic bone fracture risk). We are dealing with such components as applications being an operating systems' process, deployed in application containers, interpreted code in different programming languages and various types of databases. All of them have to be placed on a virtual machine being an execution environment. The variety of components leads to usage of a generic scripting approach for installation of prerequisites. We assume that each component is associated with installation unit comprised of deployment scripts and a set of configurable attributes. The user selects components which form deployment configuration. This approach gives flexibility, imposing at the same time the need to cope with component dependencies analysis and visualization. Achievements of the *Software Product Line* appear to be particularly helpful for this purpose. The *Feature Model* - successfully applied in *Software Product Line* - seems to perfectly fit in the gap between configuration elements domain modeling and mechanisms of instantiating execution environment.

The *Feature Model* [3, p. 3] is a notation, used to define a domain of objects as a set of features. By building tree-like hierarchy (parent-child relationships) and defining types of sibling relationships (and, or, xor), organizes dependencies and helps to identify commonalities and variabilities [1, p. 7]. It allows also for representing dependencies which does not fit into a hierarchical model (cross-tree constraints) and in some variants for defining even more complex requirements (extended Feature Models) [3, p. 3]. Thanks to the simple mapping to well-known decision problems, the *Feature Model* is well suited for representing dependency logic [9, p. 3] (it is understandable to a computer). There are various implementations of operations on the *Feature Models* allowing a.o. for easing the model configuration process, auto-completing decisions and error detection [2, p. 16]. Because of the simplicity of the graphical *Feature Model* representation - and hierarchical construction (which allows for reducing abstraction), it is also understandable to a human and well suited to visualization. Furthermore there are ready-to-use tools for the *Feature Model* visualization.

The *Software Product Line* (SPL) [1, pp. 4-6] is a group of methods that describes process of organizing software creation in a way that allows for increase the reusability of artifacts and leads to a partial automation of the software product creation. The problem solved by methods of SPL has much in common with automation of eScience application installation. In the process of building production line, there is a challenge of mapping a model of product features to production line architecture and defining methods of product instantiation. However, product instantiation is one of the less frequently studied activities in the domain of software product lines [4, p. 1]. Therefore, a challenge is to examine advantages and drawbacks of the chosen mapping method as well as the mechanism of eScience application instantiation on the basis on prototype tool evaluation.

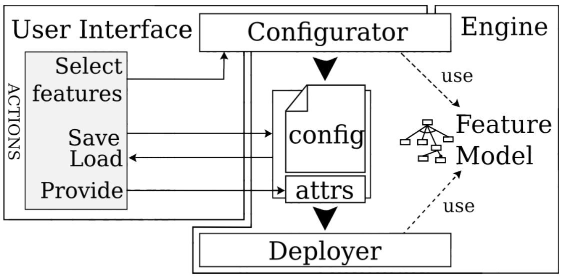

Fig. 1. The proposed architecture of tool. Model configuration links *Configurator* and *Deployer* modules. Partition between *User Interface* and *Engine* is also visible.

The architecture of the tool realizing presented task comprises two main components presented in the Fig.1. The *Configurator* is a module associated with the user interface, allowing for selection of the *Feature Model* elements step-by-step. The system automatically eliminates potentially conflicting decisions from the decision space. The elements of *Feature Model* selected in such a way are supplemented by a set of attributes and passed to the Deployer module. The Deployer's task is to deploy the configuration on a machine dedicated for an experiment execution. The Deployer validates configuration and creates acyclic dependency graph based on relationships defined in the Feature Model. This graph is sorted topologically to obtain the order of installation. It should be emphasized that each feature in a model is mapped to at most one installation unit, therefore, an element of a model is either a representation of installation unit or a group of other features.

2. Choice of technology

In order to choose an appropriate approach to execution environment automatic configuration a study of available technological solutions was performed [12, pp. 1-32]. Three classes of tools were taken into account - *Distributed Shell, Unattended Installation, Provisioning Tools*. Evaluation was influenced by factors such as possibility of application in private cloud infrastructure, ability for simultaneous configuration of multiple Virtual Machine instances and an ability for redeployment. Given such criteria the most promising are the tools from the latter group. That is why provisioning tool was chosen to be used in the implementation of the prototype. Suitability of four popular provisioning tools were evaluated – Bcfg2 [15], CFEngine [16], Chef [17], Puppet [22]. The comparison criteria were ease of integration, availability of ready-to-use installation packages, and essential compatibility with Java Enterprise Edition technological stack, support for various operating systems and license type (required for the application in VPH-Share project). As each of the reviewed tools represent slightly different approach and each might be usefull in a realization of the given task, it is hard to compare them fairly. Nevertheless the Chef platform meets selected criteria best for its' full Windows support, ease of using available Java API and activity of the community of users providing ready-to-use installation scripts. Using the Chef consist of two main tasks - maintaining installation unit repository and performing deployment tasks. The Chef repository is comprised of so called *cookbooks:* packages containing scripts and requisites needed to perform installation. Deployment is performed by the Chef after providing it with an ordered list of cookbooks with associated attributes.

We have also evaluated tools for automatic analysis of dependencies represented by the *Feature Model*. The use of libraries implementing various operations on *Feature Models* enables auto-completing configuration decisions, detecting conflicts etc. [2, p. 16]. Reviewed libraries (SAT4J [23], JavaBDD [21], Choco [18], AHEAD [13], FaMa [20], SPLAR [24]) realize mentioned tasks at different abstraction levels and implement operations on models on the basis of two main approaches to representation of Feature Model – mapping to SAT (Boolean Satisfiability Problem) or BDD (Binary Decision Diagram) trees [9, p. 3]. As there are differences in the efficiency and memory utilization of the processing depending on the choice of the data structure, BDD-based approach was chosen due to its' better performance in the operations associated with interactive configuration of a single model. In the prototype implementation the SPLAR library were used for its' ease of use and a possibility of reusing model vizualization code fragments of related open-source tool - SPLOT [24].

3. Evaluation of tool and architecture

The created tool, called *Cloudberries* was evaluated in a case study. It was integrated with the web portal as well as with the private cloud infrastructure of the VPH-Share project and tested in a case of euHeart [19] application deployment. The structure of euHeart allowed to completely automate its' deployment. Environment configuration process consists of twelve steps such as locating files in a target operating system, creation of user accounts, granting user rights, etc. Although the process is quite simple, configuration steps require some administrative knowledge and time. Time savings depend on individual skills so it is hard to be measured. Most of the mentioned deployment steps can be easily transformed to cookbooks (Chef installation units) and mapped to the Feature Model elements. Cookbooks were created for such components as python, python-pip, xvfb, wine, libxp6, openjdk-6-jdk, python-dev, libxslt1-dev. Procedures specific for euHeart installation were collected in a separate cookbook.

During the evaluation of the tool we came to some conclusions on the borderline of implementation and architecture design. Using the current mechanism of system extension (based on cookbooks) it is almost impossible to automate creation of a new Virtual Machine instance. It is connected with several limitations which cannot be bypassed. One of the most significant one is the fact that all the Chef deployment scripts are invoked on already existing machine available via SSH protocol. Although Chef can indeed be integrated with a cloud stack, it is not a procedure which can be performed by the user on their own by providing implementation for a new feature. Even if the VM instantiation were possible to be represented as an installation unit, passing attributes between installation units would prove to be an issue. Some attributes, such as IP address of the machine newly allocated by cloud hypervisor, cannot be defined by the user and have to be produced by the system.

Although the presented architecture is functioning well, it has some shortcomings. Firstly, the installation scheduling process is imperfect as it is based on the feature dependency model. The rules constituting coexistence of components in a single environment does not have to be connected with the order of their deployment in a single *product* (deployed application). In particular, a situation in which product feature dependencies form a cycle should be taken into account. Such cycle prevents us from considering dependency relationships as chronological order of deployment. Secondly, in the current situation there is no way to define a flow of attributes between installation units. An installation unit should be allowed to base its' behavior on a product of the previous one. Moreover, the architecture does not allow for sharing attributes and consequently the same attributes has to be specified multiple times for different installation units. Finally, all of the installation units subordinated to a tool managing their sequential execution (Chef), which limits system expansion to the mechanisms built into that tool.

By limiting the model-architecture mapping to the feature-component correspondence, we have lost the flexibility of defining product features presented in classic approaches to feature-based domain analysis [8, pp. 35-39]. However, thanks to automatic mapping between features to the installation rules, we gain flexibility of expanding our production line which is clearly an advantage of the presented approach. The question arise – how to modify the architecture to get a reasonable compromise.

4. Research result: a refined architecture

Overcoming the drawbacks presented in the previous section allows to broaden the range of served applications and create a more generic solution. As it is shown in the Fig. 2. A., the proposed refined architecture is composed of dynamically modifiable, layered core of production line, user interface and modules managing its' lifecycle.

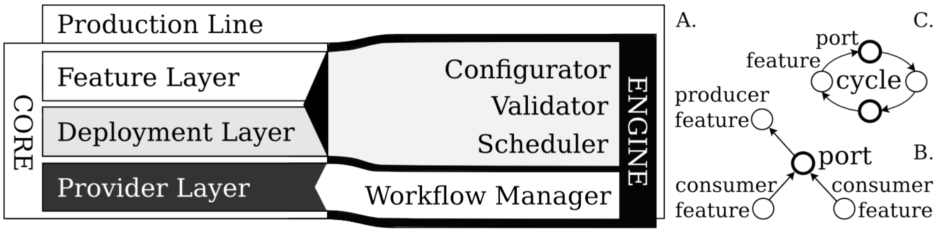

Fig. 2. Refined architecture – production line framework (A). Fragments of installation dependency graph - passing parameter between feature installation units via so called *port* (B), cycle of dependencies (C).

The *Feature Layer:* (the Feature Model is defined here) dependencies define sets of features which can constitute a correct product configuration. The Feature Model validation is more complex due to the influence of installation dependencies (*Deployment Layer*). An element of the Feature Model is treated as an entity being a feature of product configuration and a representation of an installation unit at the same time. Each installation unit provides an implementation of the feature instantiation as a plug-in in the *Provider Layer.* Installation may be implemented as any behavior in a chosen programming language. The *Deployment Layer:* the installation dependency graph is defined as a set of nodes (elements of the Feature Model and so called *ports*) and installation dependency edges. The graph defines the order of executing installation units representing elements of Feature Model; the ports define types of information transported between them [Fig. 2. B.]. Each port is either an attribute transmitted between installation units or a production line state. Each port has its type, zero or more producers, and zero or more consumers. Each element of the Feature Model may have zero or more input ports and zero or more outputs [Fig. 2.B.]. An input port is an attribute provided before unit installation. An output port defines attribute produced by the unit. A port without producer is processed as an attribute provided by the system user.

Each producer of a single port has to be a part of a different configuration: in a single cycle of installations each port is produced only once. Exception from this rule are production line states which can have multiple producers invoked in an unspecified order. The *Provider Layer:* a layer in which an implementation of installation units is provided in a form of plug-ins (eq. invocation of the Chef). Each element of the Feature Model is associated with a single *provider*. Each *provider* has to satisfy the interface defined by input and output ports in the *Deployment Layer.*

These three layers allow for dynamic expansion of the production line complemented by the other modules. The *Product configurator:* the Feature Model configuration panel for the end user. User selects components, defines required attributes and instantiates the product. The *Model configurator:* administration panel used for production line expansion. It enables modifications of the Feature Model, installation dependency graph and installing *provider* plug-ins. The *Validator:* a module for assessing correctness of the Feature Model. Validation is performed on the basis of installation dependency graph defined in *Deployment Layer*. The *Validator* seeks for cycles in the graph [Fig. 2. C.] and checks if all the features belonging to that cycle can together belong to any correct model configuration. The complexity of this operation is subordinated to the complexity of cycle detection and complexity of validating partial model configuration (dependent on implementation of the operation on the Feature Model). If a model configuration containing all of the cycle elements exists, feature selection was not performed properly and the feature model is incorrect. The *Validator* should also validate the dependency graph analyzing nodes which does not represent a state of production line. Each of them has own at most single producer in a single model configuration. The *Scheduler:* a module scheduling installation order on the basis of dependencies in the *Deployment Layer.* In order to create a schedule the *Scheduler* sorts topologically the subgraph of the dependency graph limited to the current configuration. The *Workflow manager:* the module managing running installations and passing attributes between units.

Defining installation dependency graph in an obvious way forms an overhead in the time of introducing new Feature Model element. However, identification of a configuration containing elements whose installation procedures cannot be ordered chronologically precludes model correctness. Therefore, introducing this additional formal description allows to reduce probability of erroneousness in modeling configuration domain.

5. Related work

This section presents some different approaches to mapping Feature Model to product line architecture and product instantiation. In [11] automatic creation of Java Enterprise Edition family applications was described. Configuration files used by the base production line components (Spring framework Object Factory configuration, application container Deployment Descriptor) are generated on the basis of a model configuration. The authors propose a solution tightly coupled with technological stack and much less generic than presented in this article. Instantiation process is partially manual what also differs both solutions. However an interesting aspect of this approach is the mechanism of component state probes allowing for monitoring of installation progress and state of running application lifecycle. The paper [5] presents an approach to automatic creation of applications based on aspect programming. Features are mapped to so called Object Teams - modules grouping sets of classes, whose behavior can vary depending on the aspect. The solution is limited to composing a software before its compilation. An idea worth considering in future work seems to be the substitution of behavior according to requirements represented as features. Similarly to this work a need for defining attributes was presented. [7] describes some ideas inspiring for the further research. Feature Model elements are mapped to components, as well as it was presented in this article. However, each of the components consists of a set of component states and a definition of its' inner construction – in a form of Feature Model. Component construction may vary in different component revisions. It is worth to notice that dependencies between of component states are considered at the Feature Model level, which may be a good alternative to the dependency graph concept of presented in the section 4. FArM presented in [10] strives to achieve a transformed FM where each feature can be implemented in an architectural component. At the beginning of the process model elements are described by additional semantics (eg. Quality

Feature) which allow to choose an appropriate approach for treating a feature. The notion of the *provider* presented in this work (section 4.) allows for specifying any semantics within the rules of production line operation.

Conclusions and future work

Although the tool presented in this work is a prototype one, it has proved to be successful in the realization of the given task. Cloudberries links benefits of software provisioning with the speciffic requirements of the scientiffic community, increasing productivity of researchers. Moreoveras it was shown in this paper, eliminating some architecture shortcoming allows for designing a generic framework of expansible production line, with a much wider range of applications. The developed solution presents an approach to mapping Feature Model to production line architecture slightly different than presented in other publications. A careful comparison and evaluation of its usefulness will be the subject of further work. Following research will focus on the selection on the best approach to modeling dependencies and automatic compilation of applications comprised of different types of components.

Acknowledgement. The work is partially based on the MSc thesis [12] and was partially funded by EC ICT VPH-Share Project [26] and KI AGH grant (contract 269978).

Bibliography

1. Benavides D.: *On the automated analysis of Software Product Lines using Feature Models. A framework for developing automated tool support*, PhD thesis, Department of Computer Languages and Systems, ETSI Informática, University of Seville, Spain, 2007
2. Benavides D., Segura S., Ruiz-Cortés A.: *Automated analysis of feature models 20 years later: A literature review*, Information Systems, Volume 35 Issue 6, DOI: 10.1016/j.is.2010.01.001, pp. 615-636, 2010

3. Benavides D., Trinidad P., Ruiz-Cort ́s A.: *Automated Reasoning on Feature Models*, Advanced Information Systems Engineering: 17th International Conference, Proceedings, DOI: 10.1007/11431855_34, pp. 491-503, 2005
4. Bosch J., Högström M.: *Product Instantiation in Software Product Lines: A Case Study*, Second International Symposium on Generative and Component-based Software Engineering, DOI: 10.1007/3-540-44815-2_11, pp. 149-163, 2000
5. Hundt C., Mehner K., Pfeiffer C., Sokenou D.: *Improving Alignment of Crosscutting Features with Code in Product Line Engineering*, Journal of Object Technology, Volume 6, Number 9, pp. 416-436, 2007,
6. Jankowski N. W.: *Exploring e-Science: An Introduction*, Journal of Computer-Mediated Communication, Volume 12, Issue 2, DOI: 10.1111/j.1083-6101.2007.00337.x, pp. 549–562, 2007
7. Jansen S., Brinkkemper S.: *Modelling Deployment using Feature Descriptions and State Models for Component-Based Software Product Families*, Proceedings of the Third international working conference on Component Deployment, DOI: 10.1007/11590712_10, pp. 119-133, 2005
8. Kang K., Cohen S., Hess J., Novak W., Peterson A., *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU/SEI-90-TR-021, 1990
9. Mendonça, M.: *Efficient Reasoning Techniques for Large Scale Feature Models*, PhD thesis, School of Computer Science, University of Waterloo, 2009
10. Sochos P., Riebisch M., Philippow I., *The Feature-Architecture Mapping (FArM) Method for Feature-Oriented Development of Software Product Lines*, Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, DOI: 10.1109/ECBS.2006.69, pp. 308-318, 2006
11. White J., Schmidt D. C., Czarnecki K., Wienands C., Lenz G., Wuchner E., Fiege L.: *Automated Model-based Configuration of Enterprise Java Applications*, 11th IEEE International Enterprise

Distributed Object Computing Conference, DOI: 10.1109/EDOC.2007.22, 2007

12. Wilk B.: *Installation of complex e-Science applications on heterogeneous cloud infrastructures*, MSc thesis, Faculty of Electrical Engineering, Automatics Computer Science and Electronics, AGH Kraków, 2012
13. AHEAD: http://www.cs.utexas.edu/~schwartz/ATS
14. @neurIST: http://www.aneurist.org
15. Bcfg2: http://trac.mcs.anl.gov/projects/bcfg2
16. CFEngine: http://cfengine.com
17. Chef: http://www.opscode.com/chef
18. Choco: http://www.emn.fr/z-info/choco-solver
19. EuHeart: http://www.euheart.eu
20. FaMa: http://isa.us.es/fama
21. JavaBDD: http://javabdd.sourceforge.net
22. Puppet: http://docs.puppetlabs.com
23. SAT4J: http://www.sat4j.org
24. SPLAR, SPLOT: http://www.splot-research.org
25. Virolab: http://www.virolab.org
26. VPH-Share: http://www.vph-share.eu
27. VPHOP: http://www.vphop.eu