PlugSPL: an Automated Environment for Supporting Plugin-Based Software Product Lines

Conference Paper *in* International Journal of Software Engineering and Knowledge Engineering · July 2012

CITATIONS
0

READS
186

6 authors, including:

Elder Rodrigues
Universidade Federal do Pampa
60 PUBLICATIONS **157** CITATIONS
SEE PROFILE

Avelino F. Zorzo
Pontifícia Universidade Católica do Rio Grande do Sul
138 PUBLICATIONS **1,212** CITATIONS
SEE PROFILE

Edson OliveiraJr
Universidade Estadual de Maringá
102 PUBLICATIONS **378** CITATIONS
SEE PROFILE

Itana Maria de Souza Gimenes
Universidade Estadual de Maringá
101 PUBLICATIONS **623** CITATIONS
SEE PROFILE

PlugSPL: An Automated Environment for Supporting Plugin-based Software Product Lines

Elder M. Rodrigues*, Avelino F. Zorzo*, Edson A. Oliveira Junior†, Itana M. S. Gimenes†,
José C. Maldonado‡ and Anderson R. P. Domingues*
*Faculty of Informatics (FACIN) - PUCRS - Porto Alegre-RS, Brazil
Email: {elder.rodrigues, avelino.zorzo}@pucrs.br
Email: anderson.domingues@acad.pucrs.br
†Informatics Department (DIN) - UEM - Maringá-PR, Brazil
Email: {edson, itana}@din.uem.br
‡Computing Systems Department (ICMC) - USP - São Carlos-SP, Brazil
Email: jcmaldon@icmc.usp.br

***Abstract*—Plugin development techniques and the software product line (SPL) approach have been combined to improve software reuse and effectively generate products. However, there is a lack of tools supporting the overall SPL process. Therefore, this paper presents an automated environment, called PlugSPL, for supporting plugin-based SPLs. Such environment is composed of three modules: SPL Design, Product Configuration, and Product Generation. An example of a PlugSPL application is illustrated by means of the PLeTs SPL for the Model-Based Testing domain. The environment contributions are discussed whereas future work is listed.**

***Keywords*-Software Product Lines, Plugin-based SPL, Model-based Testing.**

I. Introduction

In recent years, software product line (SPL) [1] engineering has emerged as a promising reusability approach, which brings out some important benefits, e.g., it increases the reusability of its core assets, in the meanwhile decreases time to market. The SPL approach focuses mainly on a two-life-cycle model [1]: domain engineering, in which the SPL core assets are developed for reuse; and application engineering, in which the core assets are reused to generate specific products. It is important to highlight that the success of the SPL approach depends on several principles, in particular variability management [2].

Although SPL engineering brings out important benefits, it is clear the lack of an environment aimed at automating the overall SPL life cycle, including: (i) configuration of feature model (FM); (ii) configuration of products; and (iii) generation of products. Literature and industry present several important tools that encompass part of the SPL development life cycle, e.g., SPLOT [3].

The plugin approach has also received an increasing attention in the development of SPLs [4]. In the SPL field, a plugin-based approach enables the development of different applications by selecting/developing different sets of plugins. Although the use of plugins to develop SPL products is a promising approach and several works have been published in recent years, to the best of our knowledge, there is no tool to support plugin-based SPLs. Therefore, this paper presents an automated environment for supporting the overall SPL engineering life cycle, the PlugSPL. Such an environment differs from existing tools as it aims at supporting plugin-based SPLs. Moreover, PlugSPL provides capabilities both to import/export FMs from/to other tools and to effectively generate products.

This paper is organized as follow. Section II discusses some concepts of SPL and plugin-based SPLs; Section III presents the PlugSPL environment and its main characteristics; Section IV illustrates the use of PlugSPL to manage a Model-Based Testing (MBT) SPL; and, Section V presents the conclusion and directions for future work.

II. Background

In recent years, the plugin concept has emerged as an interesting alternative for reusing software artifacts *de facto* [5]. Moreover, plugins are a useful way to develop applications in which functionalities must be extended at runtime.

In order to take advantage of the plugin concept for developing software, it is necessary to design and implement a system as a core application that can be extended with features implemented as software components. A successful example of the plugin approach is the Eclipse platform [6], which is composed of several projects in which plugins are developed and incorporated to improve both the platform and the providing services.

The plugin approach has also received an increasing attention in the development of SPLs [4]. The SPL approach has emerged over the last years due to competitiveness in the software development segment. The economic considerations of software companies, such as cost and time to market, motivate the transition from single-product development to the SPL approach, in which products are developed in a large-scale reuse perspective [1]. Whereas a SPL can be defined as a set of applications that share a common set of features and are developed based on a common set of core

assets, the plugin approach can be easily applied to build new applications by plugging different sets of plugins to a core application [7]. Although the use of plugins to develop products is a promising approach and several works have been published in recent years, there is no tool to support plugin-based SPLs.

III. PlugSPL Environment: Supporting Plugin-based Software Product Lines

Although there are many tools focused on SPL modeling, consistence checking [3], and product generation support [8], currently, there is no tool that integrates all SPL development phases. Moreover, there is no tool to support the automated product configuration and product generation from a plugin-based SPL. Therefore, in this section we present the PlugSPL environment that has been developed to support SPL design, product configuration and generation of plugin-based SPLs. Figure 1 presents the PlugSPL modules and activities, as follows:

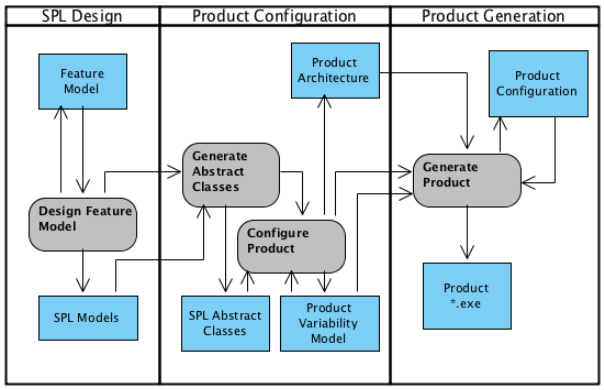

Figure 1: The PlugSPL modules.

a) the `SPL Design` module aims to design a FM by either creating it from scratch or importing a pre-existing FM from SPL tools. Such tools, usually, do not use a common format to represent FM elements and constraints, therefore, we conceived the PlugSPL `SPL Design` module to work with a wide FM representations and file formats. Thus, PlugSPL FMs can be seen as a starting point to automate the creation of the SPL architecture and then to generate products. Based on information extracted from a FM, the `SPL Design` module represents such information as SPL Models (Figure 1). Such models are taken as input to the `Product Configuration` module for composing the SPL architecture;
b) the `Product Configuration` module is responsible for automating the SPL architecture. Basically, this module has two activities - `Generate Abstract Classes` and `Configure Product`. The former receives the SPL Models provided by the `SPL Design` module and creates a set of abstract classes, one class for each feature. Each abstract class is a variation point and/or a variant with a specific type. Thus, each plugin may extend only one abstract class. The abstract classes might be used, according to the SPL documentation, by the core assets developer to build each plugin that composes the SPL. After the generation of the abstract classes, the SPL engineer is able to select the desired features for the target system (product configuration). PlugSPL checks the system consistency and generates the target system architecture (abstract classes).
c) the `Product Generation` module takes the target system architecture as input. This module graphically shows such architecture to the application engineer. Thus, this module retrieves from the plugin repository respective plugins that implement the types of the abstract classes (product architecture). After that, the plugins are linked to each of their respective abstract classes and the consistency between the generated architecture and FM is checked. PlugSPL shows graphically the set of plugins that is able to be selected for resolving the variability in each class and generates the target system. Them, the application engineer: (i) selects one or more plugins (which denote a feature in the FM) to resolve each class variability; (ii) gives a name to the target system; and (iii) clicks a button to generate the system.

IV. PlugSPL Application Example

This section presents an example of how PlugSPL can be used to design, develop and derive MBT products from PLeTs SPL [7]. PLeTs is a SPL aimed at automating the generation, execution and results collection of MBT processes. The MBT process consists in the generation of test cases and/or test scripts based on the application model. The MBT process main activities are [10]: *Build Model*, *Generate Expected Inputs*, *Generate Expected Outputs*, *Run Tests*, *Compare Results*, *Decide Further Actions* and *Stop Testing*. PLeTs goal is the reuse of SPL artifacts to make it easier and faster to develop a new MBT tool. Figure 2 shows the main features of the current PLeTs FM: `Parser`, `TestCaseGenerator`,`ScriptGenerator` and `Executor`.

Figure 2 shows several dependencies (denoted by propositional logic) between features. For instance, if feature `Executor` and its child feature `LoadRunnerParameters` are selected, then feature `ScriptGenerator` and its child feature `LoadRunnerScript` must be selected as the generated tool is not able to execute tests without test scripts.

The PLeTs FM can be extended to support new testing techniques or tools by adding new child features to its main features. For instance, if one adds new features for the SilkPerformer testing tool, new child features for the `ScriptGenerator` and `Parameterization` features must be included.

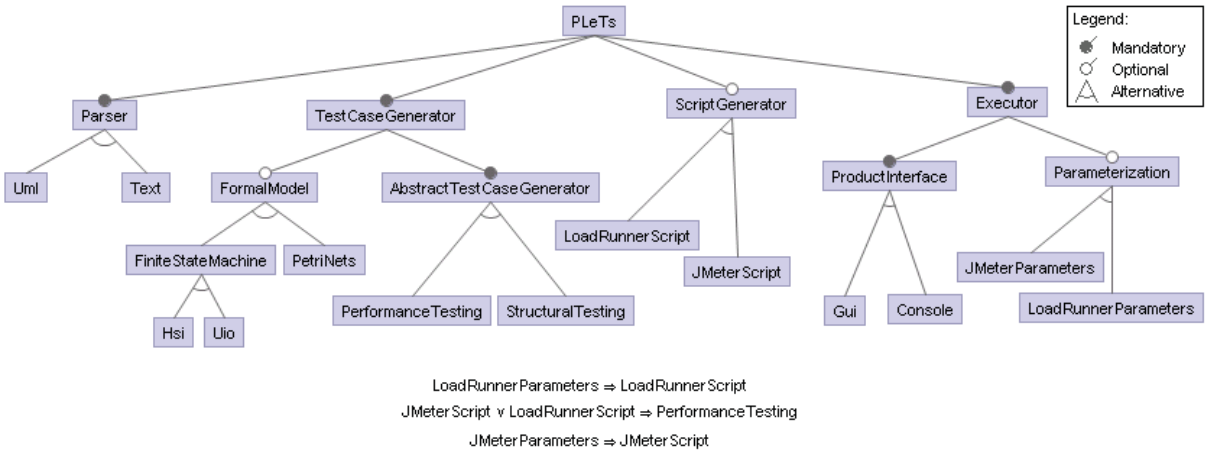

Figure 2: The PLeTs Feature Model [7].

A. *Using PlugSPL for Generating a MBT SPL*

Designing and development of SPLs supported only by FMs editors and SPL documentation itself might be error prone and time consuming activities. Moreover, checking features constraints manually is a hard task. Therefore, this section explains how PlugSPL can be used to automate the overall SPL process by using PLeTs as an example SPL.

PlugSPL imports PLeTs FM (Figure 2) to the `SPL Design` module. Thus, PlugSPL re-constructs and checks the FM and shows the result to the SPL architect using a tree notation. Therefore, the SPL architect can interact with the PLeTs FM to, for instance, add or edit feature relationships. The PLeTs FM is saved as SPL Models to support the Product Configuration phase.

During Product Configuration, PlugSPL generates the PLeTs architecture, formed by its abstract classes, e.g., Parser, UmlDiagrams and ScriptGenerator. Thus, the SPL architect might export such classes, by clicking on the `Deploy Development Library` button, and send it to the plugin development team. Based on the architecture, such team develops a plugin by extending a specific abstract class (e.g. Parser), and sends it back to the SPL architect to store it in the SPL plugin repository. As shown in Figure 3, the product architect might define each product architecture by selecting the abstract classes in the tree structure. A product is a performance MBT tool that realizes the following activities; a) Accepts as input an UML model (Parser, UmlDiagrams); b) Transforms the model in a formal model (TestCaseGenerator, FiniteStateMachine), and applies it a sequence method (HSI) to generate the testing sequence. Based on such a testing sequence, it generates the abstract performance test cases (AbstractTestCaseGenerator, PerformanceTesting); c) Uses the abstract performance test cases to generate scripts to the LoadRunner performance tool (ScriptGenerator, LoadRunnerScript); d) Executes product via command line (Execution, Console) and sets the LoadRunner parameters (Parameterization, LoadRunnerParameters).

PlugSPL allows the product architect to save each product architecture in a repository for reuse. It is important to highlight that in the PLeTs SPL each abstract class is a variation point that is resolved by selecting a variant (plugin).

The Product Generation module presents graphically the abstract classes structure of a product. It also links a plugin to classes by performing a search in the plugin repository to find what plugins are implemented by each product abstract class. Thus, the product architect selects a plugin, or a set of plugins, to resolve each variability. In the Product Generation activity each variability (abstract class) uses only one variant (plugin) to resolve a variability. However, it might be necessary to use two or more variants to resolve a variability. After resolving a variability, the product architect has two options: save the product configuration and/or generate the MBT product. In the first option, the tool asks for a product name and, then, saves its classes and plugins to generate a product later. In the second option, PlugSPL asks for a product name, and then generates the MBT product. In order to generate the product, PlugSPL: (i) selects the product abstract classes and its related plugins; (ii) packages them by using a "glue code"; and (iii) generates an executable product file. Although the "glue code" generation is only invoked and executed by PlugSPL, the piece of code that generates such a code is implemented in the SPL base plugin. This approach simplifies the development and evolution of the product as the complex information necessary to generate products from a wide amount of SPLs is stored as a SPL artifact.

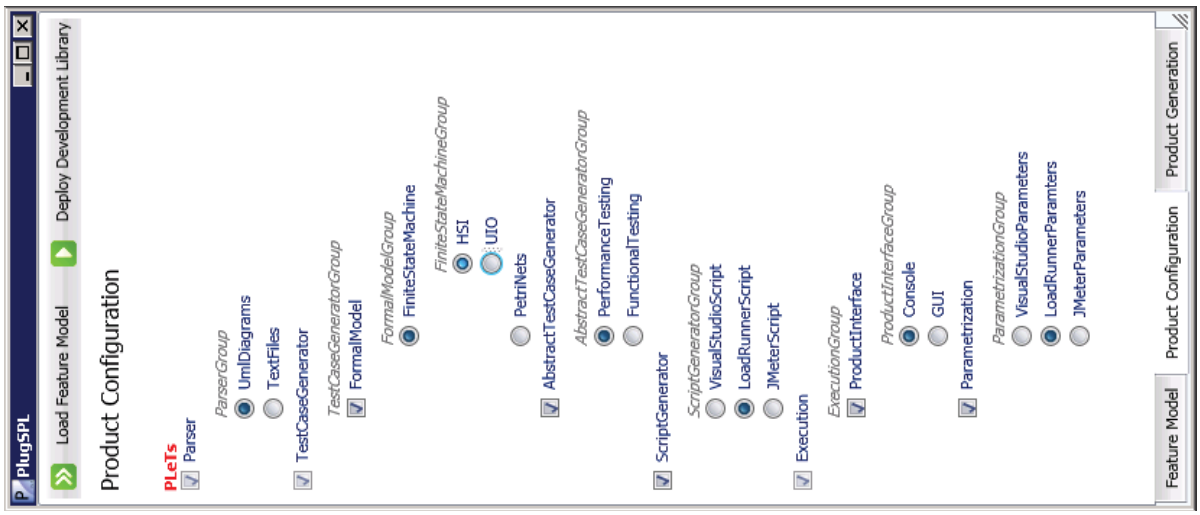

Figure 3: PlugSPL Feature Model Editing and Product Configuration.

V. Conclusion and Future Work

This paper presented PlugSPL, which is an automated environment to support the overall plugin-based SPL life cycle. Although there are tools to partially support the SPL life cycle as, for instance, `pure::variants`, there is no tool that supports plugin-based SPLs and the overall SPL life cycle. Furthermore, there are tools to design FMs, but most of them use different notations and file formats. PlugSPL provides capabilities with regard to create or import/export FMs from/to other tools and uses a wide file format. Therefore, there is no need to incorporate other tools/environments into PlugSPL.

Although PlugSPL is a flexible environment for modeling FMs, its most significant benefit is supporting the generation of SPL products based on its FM. Moreover, PlugSPL automatically generates an abstract class structure, which can be used to develop third-party plugins. A PlugSPL application example was presented for deriving MBT tools from the PLeTs SPL. Directions for future work are: (i) plan and conduct experiments for assuring the effectiveness of PlugSPL environment; (ii) extend PlugSPL functionalities to support different plugin-based SPLs; and (iii) include into PlugSPL an overall SPL evaluation module.

References

[1] F. J. v. d. Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[2] E. A. Oliveira Junior, I. M. S. Gimenes, and J. C. Maldonado, "Systematic Management of Variability in UML-based Software Product Lines," *Journal of Universal Computer Science*, vol. 16, no. 17, pp. 2374–2393, 2010.

[3] M. Mendonça, M. Branco, and D. Cowan, "S.P.L.O.T.: Software Product Lines Online Tools," in *Proc. Conf. Object Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2009, pp. 761–762.

[4] R. Wolfinger, S. Reiter, D. Dhungana, P. Grunbacher, and H. Prahofer, "Supporting Runtime System Adaptation through Product Line Engineering and Plug-in Techniques," *Int. Conf. Commercial-off-the-Shelf (COTS)-Based Software Systems*, pp. 21–30, 2008.

[5] J. Mayer, I. Melzer, and F. Schweiggert, "Lightweight Plug-In-Based Application Development," in *Proc. Int. Conf. NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*. London, UK, UK: Springer-Verlag, 2003, pp. 87–102.

[6] M. Kempf, R. Kleeb, M. Klenk, and P. Sommerlad, "Cross Language Refactoring for Eclipse Plug-ins," in *Proc. Workshop on Refactoring Tools*. New York, NY, USA: ACM, 2008, pp. 1–4.

[7] M. B. Silveira, E. M. Rodrigues, A. F. Zorzo, L. T. Costa, H. V. Vieira, and F. M. de Oliveira, "Model-Based Automatic Generation of Performance Test Scripts," in *Proc. Software Engineering and Knowledge Engineering Conf.* Miami, USA: IEEE Computer Society, 2011, pp. 258–263.

[8] D. Beuche, "Modeling and Building Software Product Lines with Pure::Variants," in *Proc. Int. Software Product Line Conf.* New York, NY, USA: ACM, 2011, pp. 358–.

[9] T. Thum, C. Kastner, S. Erdweg, and N. Siegmund, "Abstract Features in Feature Modeling," in *Int. Conf. Software Product Line*, 2011, pp. 191–200.

[10] I. K. El-Far and J. A. Whittaker, *Model-based Software Testing*. New York: Wiley, 2001.