Model-to-Code transformation from Product-Line Architecture Models to AspectJ

Jessica Díaz, Jennifer Pérez, Carlos Fernández-Sánchez, Juan Garbajosa
Technical University of Madrid (UPM) - Universidad Politécnica de Madrid
Systems & Software Technology Group (SYST), E.U. Informática, Madrid, Spain
Email: yesica.diaz,carlos.fernandez-at-upm.es,jenifer.perez,jgs-at-eui.upm.es

***Abstract*—Software Product Line Engineering has significant advantages in family-based software development. The common and variable structure for all products of a family is defined through a Product-Line Architecture (PLA) that consists of a common set of reusable components and connectors which can be configured to build the different products. The design of PLA requires solutions for capturing such configuration (variability). The Flexible-PLA Model is a solution that supports the specification of external variability of the PLA configuration, as well as internal variability of components. However, a complete support for product-line development requires translating architecture specifications into code. This complex task needs automation to avoid human error. Since Model-Driven Development allows automatic code generation from models, this paper presents a solution to automatically generate AspectJ code from Flexible-PLA models – previously configured to derive specific products. This solution is supported by a modeling framework and validated in a software factory.**

I. Introduction

Software Product Line Engineering (SPLE) has proved to have significant advantages in family-based development [1], [2], but also implies an upfront design in product-line architecture (PLA) from which individual product applications can be engineered. PLAs consist of a set of common and reusable building blocks (components and connectors from the structural viewpoint) that can be configured to build the different and/or customized products that make up a product-line. Such configuration is an expression of *variability*. In fact, variability is one of the most important drivers of these kinds of architectures that support a set of members of a product-line. However, the modeling of the architectural variability has become a challenge for SPLE [3].

The *Flexible-PLA Model* [4] extends previous approaches for modeling architectural variability [5], [6], [7], [8] not only at the level of the external architecture configuration, but also at the level of internal specification of components. However, a complete support for product-line development requires translating architecture specifications into code. This complex task needs traceability from architecture to code and automation to avoiding human error. Model-Driven Development (MDD) is presented as a solution for dealing with these issues through the use of models, model transformations, and traceability between models. In this way, architecture and code models can be linked and inconsistencies due to evolution are avoided by updating PLA models, and then re-generating code.

To provide the advantages of MDD in the product-line development, this paper presents a solution to (i) automatically generate code from Flexible-PLA models that are configured to derive specific products, and (ii) guarantee the traceability among architecture and code. This solution is supported by a modeling framework called FPLA[1] which supports the description of Flexible-PLA models ready to be configured and involved in a MDD process by automatically transforming their outputs into code, specifically AspectJ [9]. AspectJ is an extension of the Java programming language to support Aspect-Oriented Programming (AOP) [10] that aims to increase code modularity through separation of crosscutting concerns. Therefore, FPLA automatically generates the AspectJ code of one or more product applications from a product-line after configuring and deriving the variability defined in a Flexible-PLA model, and after populating and/or importing code from external sources. Specifically, FPLA automatically generates the code skeletons from Flexible-PLA models and composes the code from external sources by defining a *model-to-code* transformation.

The FPLA Modeling Framework has been developed by using the infrastructure provided by the Eclipse Modeling Framework (EMF) [11], and the model-to-code transformation is implemented by using the Epsilon Generation Language (EGL) from the Epsilon Generative Modeling Technologies (GMT) research project [12]. We have empirically validated the code generation in a project developed in a software factory [13]. We refer to this project as OPTIMETER, which is part of several projects[2] focused on Smart Grids. The validation consisted of the design of the PLA of a family of power metering management systems and the automation of code generation.

This paper is structured as follows: Section II describes the Flexible-PLA Model upon which the model-to-code transformation is performed. Section III describes a case study used to validate the transformation. Section IV presents the transformation from Flexible-PLA models to

[1]https://syst.eui.upm.es/FPLA/home (Eclipse Public License v1.0)
[2]IMPONET http://innovationenergy.org/imponet and NEMO&CODED http://innovationenergy.org/nemocoded/

AspectJ code. Section VI discusses related work. Finally, conclusions and further work are presented in Section VII.

II. Background

A. Variability in Software Architecture

The integration of variability concepts into architectural specifications is essential in order to successfully develop SPLs [6], [3]. Most mechanisms for describing architectural variability specify what we have called *external variation* of the architecture. External variation allows the specification of flexibility points by modifying the structural configuration of the architecture [5], as well as of composite components (e.g., through the use of *representations* [8]). However, external variation is not enough to completely define all kinds of variabilities [6] and trace these variabilities from requirements to the architecture [14], or even code. Sometimes variations may happen inside components. As a result, it is necessary to specify *internal variation*, i.e., variations of non-composite components.

To support internal variability, the Flexible-PLA Model is based on a novel concept called *Plastic Partial Component (PPC)* [4]. PPC's approach applies the advantages of *aspect-oriented software architectures* [15] (see Figure 1.a) by defining components as fragment boxes that hook a set of reusable fragments of code—*aspects*—, which makes components easier to be maintained, and by extension software architectures. Specifically, PPCs take advantage of aspect-oriented concepts to specify the internal variation of components, in such a way that part of its behavior corresponds to the core functionality of a SPL and part of its behavior is specific of a product or set of products from that SPL. The variability of a PPC is specified using *variability points*, which hook fragments of code known as *variants* to the PPC (see Figure 1.b). The specification of a variability point must include the definition of *weavings* between the PPC and the variants (see Figure 1.b). The notion of weaving originally comes from AOP and provides the functionality needed to specify *where* and *when* to extend components through the use of aspects. Unlike AOP, our weavings specify *where* and *when* to extend PPCs through the use of variants. AOP defines a set of weaving primitives: *pointcuts* and *advices*, which are applied to weave a PPC with variants. Pointcuts define where the code of a variant is going to be inserted. Specifically, a pointcut is the call to one or a subset of services that a PPC provides. The services of a PPC, which can be intercepted during their execution to insert code, are called *services for derivation*. The fragment of code to be inserted in a PPC, widely named *advice* in AOP terminology, is provided by variants. Each variant is composed of several advices, and each advice is modeled as a *service for derivation*. The definition of the weaving operator consists of establishing *when* to insert the advice of the variant in regard to a pointcut. It could be *before*, *after* or *insteadOf* the call of the pointcut.

B. Flexible-PLA Ecore

To specify Flexible-PLA models it is necessary to define a domain-specific (modeling) language (DSL). The DSL abstract syntax has been specified through a metamodel. Specifically, the Flexible-PLA metamodel has been specified through an Ecore Model in order to be deployed in a framework for DSL definition, such as the Eclipse Modeling Framework. The Flexible-PLA Ecore Model has been described in terms of (E)Classes, relationships between classes, and their cardinality. The Flexible-PLA Ecore Model is described in Figure 2.

Components are described by the EClass *Component*. A component defines an attribute *name* which is inherited from the EClass *NamedElement*. A component is characterized by a set of properties (see the EClass *Property* and the aggregation relationship *characterizedBy* in Figure 2), and offers a set of services (see the EClass *ServiceForCore* and the aggregation relationship *offersServices* in Figure 2). A component has a set of ports that publish its services (see the EClass *AbsPort* and the aggregation relationship *hasPorts* in Figure 2). Ports publish the services that a component provides or requires through interfaces. This relationship is represented by the aggregation *publishesInterfaces* between the EClasses *AbsPort* and *AbsInterface* (see Figure 2). Ports can be mandatory or optional while interfaces can be required or provided (see Figure 2).

Connectors model interactions among components. Although we advocate the approach of considering connectors as first-class entities in software architecture representations [16], the Flexible-PLA Ecore Model implements a simplification of connectors as simple attachments. Hence, connectors define the communication channels between the ports of two components. This connection between components is represented in the metamodel by the association relationships *attachesMandatory* and *attachesOptional*, which are recursive with the EClasses *PortMandatory* and *PortOptional* respectively (see Figure 2). Such relationships determine whether the connection is optional, i.e., whether it belongs to the core of the PLA, or whether the connection is specific to a product (or subset or products) from the SPL that is being modeled. As a result, a component that has all their ports optional, is naturally optional. Therefore, external variability is realized by adding or removing optional components (and their ports) to/from PLAs, i.e., by modifying their structural configuration.

PPCs are a specialization of components, as the EClass *PlasticPartialComponent* inherits all the properties and behavior from the EClass *Component* (see Figure 2). Internal variations are specified using PPCs, i.e., components that define a set of variability points which hook variants. Hence, a PPC is characterized by the definition of a set of variability points, i.e., the place where the different variants are hooked to one or more PPCs. This relationship is modeled by means

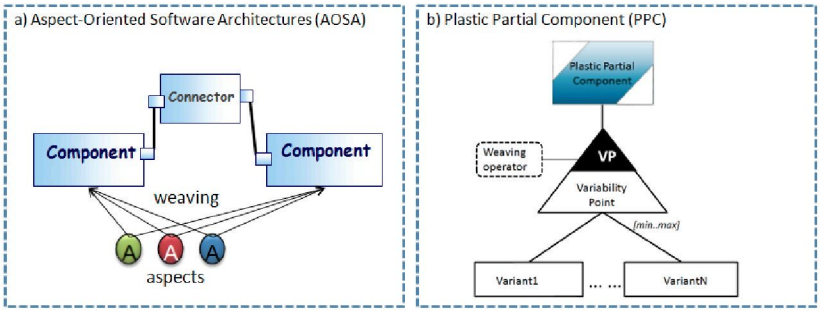

Figure 1. AOSA applied to the specification of internal variability of components: PPCs

of the association relationship called *defines* (see Figure 2), which relates the EClass *PlasticPartialComponent* to the EClass *VariabilityPoint*.

The set of variants that a variability point provides is specified through the association relationship *hooks* between the EClasses *VariabilityPoint* and *Variant* (see Figure 2). The EClasses *VariabilityPoint* and *Variant* define an attribute *name* which is inherited from the EClass *NamedElement*. Additionally, the EClass *VariabilityPoint* defines an attribute to specify the cardinality of the variability point (see the attribute *cardinalitySelection* in Figure 2), i.e., optional, alternative or multiple variants.

Variability points are characterized by the weaving that pinpoints where and when to extends PPCs through the use of variants. The weaving is defined by the EClass *Weaving*. The EClass *Weaving* is part of the variability point as it specifies the weaving between PPCs and variants, and it is dependent on the linking context. As a result, a variability point must specify all the weavings that can be applied to the PPC(s) that define it and its variants. This relationship is defined in the metamodel through the aggregation relationship *weaves*, which is inclusive. The EClass *Weaving* has three attributes: *name*, (weaving) *operator*, and *selection*. The first one allows the identification of the weaving, whereas the second one establishes when to insert the advice of the variant in regard to a pointcut: before, after, instead (see Figure 2). The attribute *selection* determines if a weaving has to be applied to or not. This means, at the time of deriving or configuring a specific product from a SPL, this attribute determines which pointcuts—service(s) of a PPC or set of PPCs—are intercepted by the advice of a variant. The pointcut and the advice are represented by the EClass *ServiceForDerivation*. This is why the EClasses *PlasticPartialComponent* and *Variant* are composed by *ServicesForDerivation* (see the aggregation relationships *composedof* and *constitutedby* in Figure 2).

ServiceForDerivation are the services that participate in the weaving to specify where to insert the code of the *advice* in the *pointcut* (see the aggregation relationships *pointcut* and *advice* in Figure 2). The EClass *ServicesForDerivation* is a specialization of services that inherits all the properties and behavior of a common service of the EClass *Service* (see Figure 2). They are a specialization because they are services that participate in a variability point. Hence, common components (non-PPCs) cannot be composed of this kinds of services, only PPCs and variants can be composed of services that participate in a variability point, i.e., *ServicesForDerivation*.

Finally, it is necessary to specify the DSL concrete syntax by defining a graphical language representation. A graphical modeling language is defined as this kind of language is usually more intuitive. This language reuses common graphical metaphors of components [17], as well as variability points [1]. Its usage has been pursued to be as friendly as possible for use in the PLA community. Figure 3 shows the graphical representation of the main concepts of the Flexible-PLA model.

III. Case Study

With the purpose of validating the code generation resulting of configuring Flexible-PLA models, we have conducted a case study within a project in an experimental i-smart software factory (iSSF [13]). The iSSF is a software engineering research and education setting in close cooperation with the top industrial and research collaborators in Europe. Indra Software Labs leads this initiative at the corporate level in Spain in conjunction with the Technical University of Madrid (UPM).

The case study consists of a project, called OPTIMETER, to develop a SPL of metering management systems in electric power networks. OPTIMETER is part of several projects[2] focused on the Smart Grids. Metering management systems capture meter data from a huge number of distributed energy resources, load these data in a database, support data querying and processing, and provide these data to other systems for billing, forecasting or purchasing. Data loading and querying require to leverage high performance through the use of data clustering technologies. To manage meter data with high performance it is necessary to manage several of the large data storing technologies available in the market (such as Berkeley DB, Oracle 11g, together

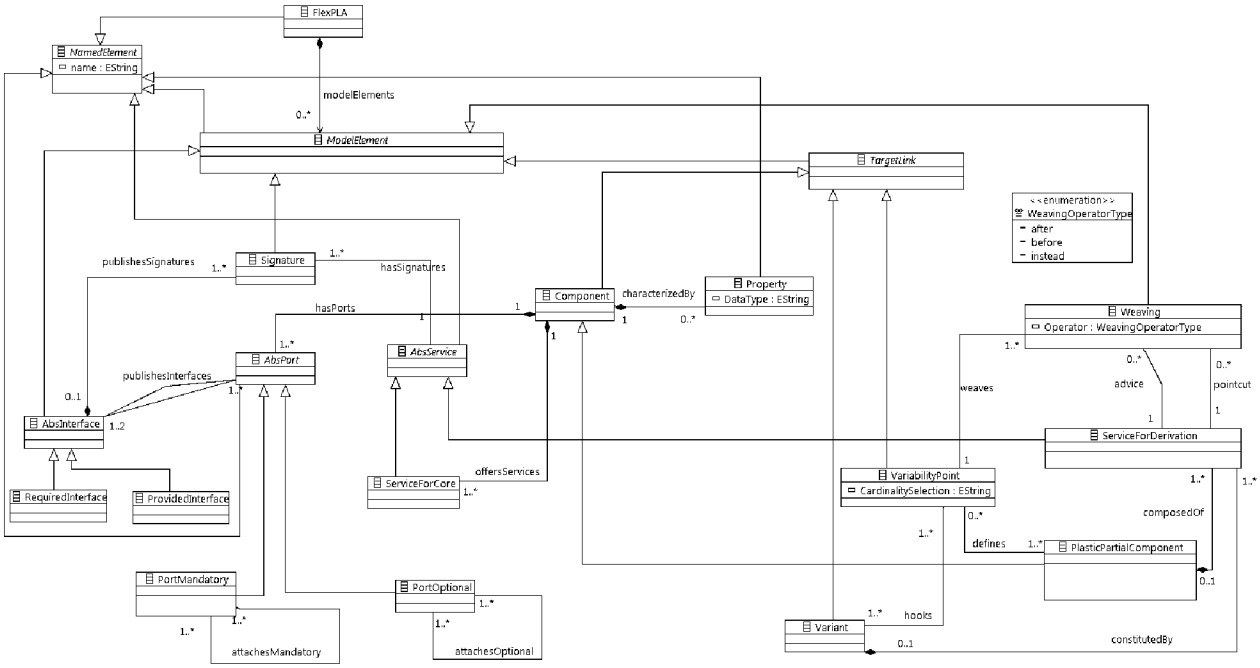

Figure 2. Flexible-PLA Ecore Model

clustering technologies such as Hadoop and Oracle Real Application Clusters) to use those more suitable in each particular case. Therefore, these technologies are alternative variants to develop a family of metering management systems.

The case study here described focused on the development of two products from OPTIMETER SPL. These products respectively implement the data storing technologies Berkeley DB over Apache Hadoop clustering and Oracle 11g over RAC. To that end, the architects designed two PPCs for data loading and querying respectively (see *DataLoader* and *DataQuerying* in Figure 3.a). These components are defined as PPCs to implement the variability for the different data storing technologies. Both PPCs define the variability point *clustering* which hooks the variants *HadoopMAP/REDUCE* and *RAC* (see Figure 3.a). Additionally, the architects defined the weavings to specify where and when to extend the code of the PPC DataLoader using the code of these variants (see the weavings *WeavingClusteringHadoop*, *WeavingRunJobHadoop*, *WeavingClusteringRAC*, *WeavingRunJobRAC* in Figure 3.a).

Figure 3.b shows the PPC *DataLoader* which provides the service *load* that is published through the interface *IDataLoader*, so that other components can require it. This service is a *ServiceForCore*, i.e., all products of the OPTIMETER SPL has a component *DataLoader* that provides the service *load*. The PPC *DataLoader* also defines other two services: *initializeCluster* and *runJob* (see Figure 3.b). Both services are not provided or required to/from other components, and are *ServiceForDerivation* which means that these services participate in a weaving definition —pointcuts—, and therefore the code of these services can be completed (through the weavings operators *after* and *before*) or replaced (through the weaving operator *instead*) by the services defined by variants. The variant *HadoopMAP/REDUCE* is composed of two services: *hadoopcluster* and *hadoopRunJob* (see Figure 3.b). Both services are *ServiceForDerivation* which means that these services participate in a weaving definition – advices.

Finally, Figure 3.c shows the weaving definition (pointcuts and advices) to derive a specific product application from OPTIMETER SPL. Specifically, to derive a metering management system working with Berkeley DB running over Apache Hadoop clustering. To that end, the code of the service *hadoopcluster* is injected instead of the service *initializeCluster* and the code of the service *hadoopRunJob* is injected instead of the service *runJob*.

IV. Transformation: From Flexbile-PLA Models to AspectJ Code

The FPLA Modeling Framework provides modeling primitives for describing PLAs conforming to the Flexible-PLA metamodel (see marker 1 in Figure 4). Next, FPLA supports the selection of a particular configuration out of the multitude of available architectural configurations that the PLA implements (see marker 2 in Figure 4). The result is a product architecture model. Finally, FPLA automatically

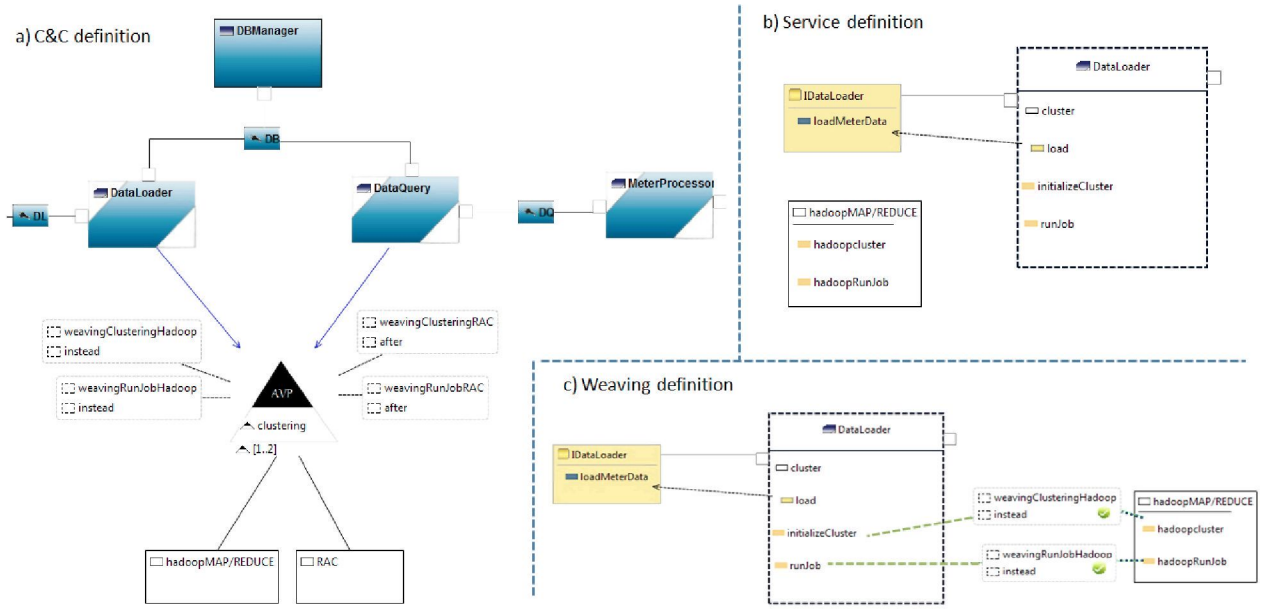

Figure 3. OPTIMETER SPL - Flexible-PLA model

transforms this model into AspecJ code (see marker 3 in Figure 4). This transformation consists in the generation of code skeletons and the composition of external source code.

To transform a Flexible-PLA model into AspectJ code, we have used the Epsilon Generation Language (EGL) to define a model-to-text transformation (see a fragment of the EGL transformation in Code 1). This fragment shows how the code of aspects is generated from the information modeled in the weavings (see Figure 3.c) as follows. First, the header is generated (see line 4 in Code 1). Then, the transformation iterates on the services of a weaving to locate both the pointcut and the advice (see lines 6-7 and 11-12 in Code 1). Line 23 prints the poincut, line 32 prints the aspect operator, and finally line 33 prints the advice of the aspect – i.e., this line prints the code that has been previously imported in a Flexible-PLA model.

The generated code is structured into three packages: components, interfaces, and variants. Hence, for each component and PPC of a Flexible-PLA model the transformation that we have implemented generates a Java class (file .java in the package components), although complex components— or subsystems— could be structured in sub-packages. For each interface, that the components or PPCs of a Flexible-PLA model *provide*, the transformation generates a Java interface (file .java in the package interfaces). Finally, for each weaving, that the variability points of a Flexible-PLA model *weave*, the transformation generates an aspect (file .aj in the package variants).

The code on the left in Figure 5 shows a class generated by applying the transformation to the PPC *DataLoader* shown in Figure 3. As this code shows, the transformation generates:

1) Headers for *package* and *import* definitions.
2) Class headers.
3) Interfaces implementation (see the interface *IDataLoader*).
4) Constructor(s) definition.
5) Attributes definition, as well as *setter* and *getter* methods for each attribute (see the property *cluster*).
6) Implementation of those *ServicesForCore* that are provided through an interface or that are private (see the service *load*).
7) Partial definition of those *ServicesForDerivation* that are provided through an interface or that are private (see the services *initializecluster* and *runJob*).

The code on the right in Figure 5 shows an aspect generated by applying the transformation to the weaving *WeavingClusteringHadoop*. As this code shows, the transformation generates:

1) Pointcut definition (see the poincut *initializeCluster* or *runJob*).
2) Weaving operator (see the operator *around*).
3) Advice definition (this is the code imported by the variant *HadoopMAR/REDUCE*, specifically by the service *hadoopcluster*, see Figure 3).

V. Discussion

The people involved in the case study were interviewed to check whether Flexible-PLA modeling primitives were effective in providing the capabilities for (i) specifying most common kinds of variations that the SPL engineers required to define, (ii) configuring two different metering

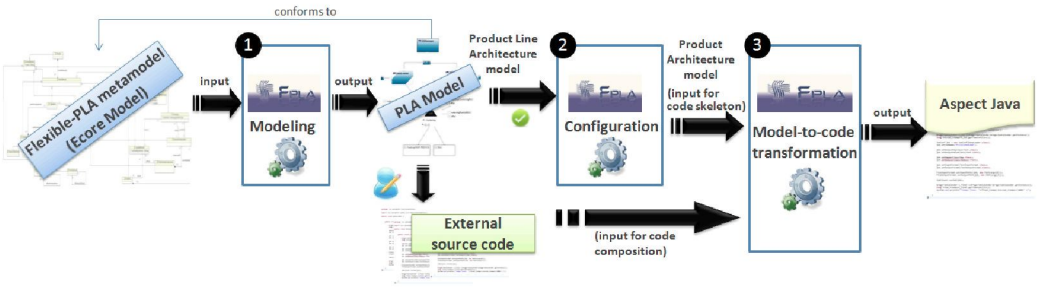

Figure 4. FPLA Modeling Framework: the Model-Driven Development process of FPLA

Code 1 EGLTransformation

```
1  package [%=directoryVariant%];
2  import [%=directoryComponent%].*;
3
4  public aspect [%=weaving.name%]{
5  [%
6  for (variant  in Variant.all.select(v|v.hasService.exists(sv| sv.Advice.exists(p|p.name=weaving.name)))){
7     for (serviceVar in variant.hasService.select(sv| sv.Advice.exists(p|p.name=weaving.name))){
8        var weavingAdvice ="";
9        var weavingPointCut ="";
10       var weavingCode =serviceVar.getCodeMethod();
11       for (components  in PlasticPartialComponent.all.select(v|v.composedOf.exists(sc| sc.Pointcut.exists
         (p|p.name=weaving.name)))){
12           for (serviceCom in components.composedOf.select(sc| sc.Pointcut.exists(p|p.name=weaving.name))){
13              var method = "execution (* " + components.name + "." + serviceCom.name.getNameMethod() + "(..))";
14              if (weavingPointCut=""){
15                 weavingPointCut = method;
16                 weavingAdvice =serviceCom.name.getNameMethod() + "Method";
17              }else {
18                 weavingPointCut = weavingPointCut + " || " + method;
19              }
20     }
21       }
22  %]
23       pointcut [%=weavingAdvice%] () : [%=weavingPointCut %];
24       [%
25       var weavingOperator = weaving.Operator;
26       var weavingReturnType ="";
27       if (weaving.Operator.name.isSubstringOf('instead')){
28          weavingOperator = "around";
29          weavingReturnType = "void";
30       }
31       %]
32   [%=weavingReturnType%] [%=weavingOperator%]() : [%=weavingAdvice%] () {
33   [%=weavingCode%]
34   }
35 [% }
36 }%]
37 }
```

managements system applications, and (iii) generating their code. Most of the people involved in the case study reported their satisfaction. They asserted they did not incur a big cost, and that automatic code generation was possible. However, the use of the Flexible-PLA Model requires to know and understand the modeling concepts on which it is based on, as well as to learn the usage of the FPLA Modeling Framework. The learning curve of these concepts as well as the usage of FPLA could slow down the process of putting model-driven SPL development into practice. In fact, the SPL engineers expressed reluctance at the time of putting the Flexible-PLA Model into practice, although later, the product engineers found code generation and (architecture-to-code)traceability essential to do their work during the configuration of variability to derive the metering management system applications.

However, the major limitation in case study research concerns external validity because only one case is studied. On the contrary, case studies allow one to evaluate a phenomenon, a model, or a process in a real setting. This is something important in software engineering in which a multitude of external factors may affect to the validation results, and that other techniques such as formal experiments, although they permit replication and generalization, do not consider as they are conducting under controlled settings.

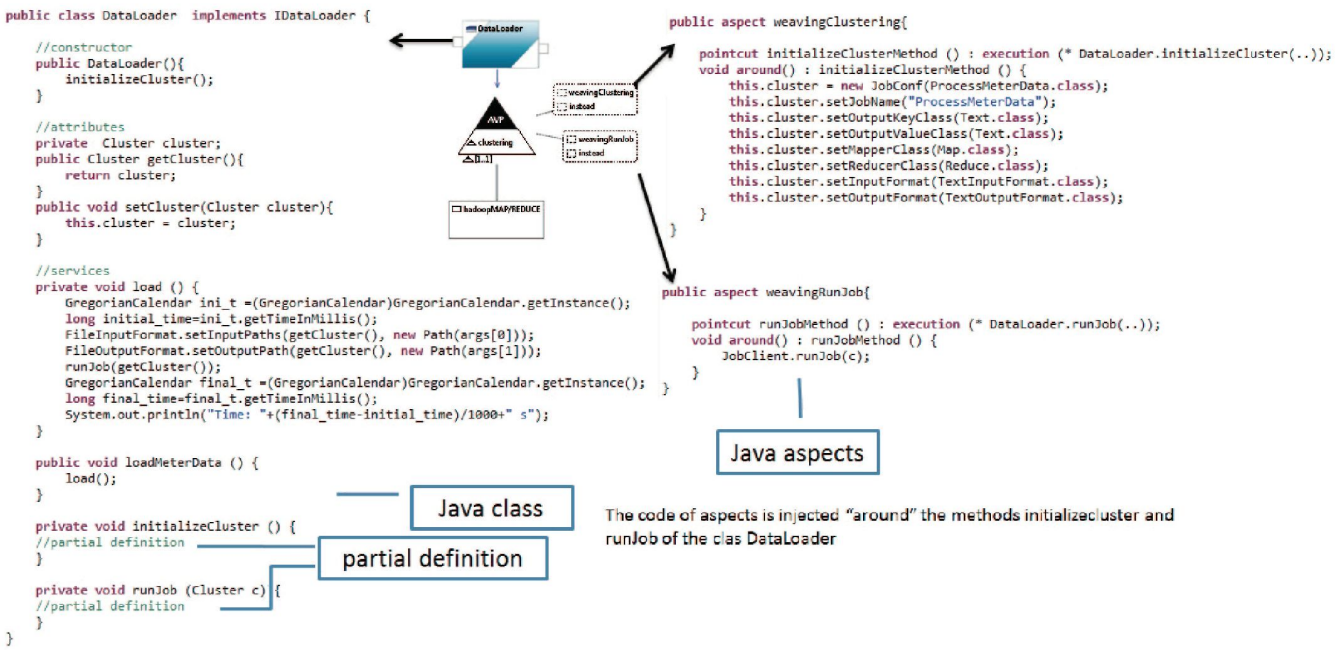

Figure 5. OPTIMETER SPL - AspectJ code

VI. Related Work

Most approaches and tools for specifying PLAs only account for external variability. Thus, PL-Aspectual ACME [8] is based on the compositionality mechanism of software architectures to define multiple representations for a given component. The Koala Component Model [18] is also based on the compositionality mechanism through hierarchical specification of subcomponents; then selection between subcomponent variants is realized by variation points called *switches*. The frameworks Mae and Dradel [5], [19] propose variant components, variant connectors, and multi-versioning connectors.

When internal variability is addressed, mostly is based on UML compositionality and inheritance mechanisms [6], [20], [14], [21]. Hence, Bachmann & Bass [6] laid the basis for addressing internal variability by means of a root component, from which hangs components that implement the variation. The works [21], [20], [14] extend this contribution and formalize it by using UML profiles. However, all these approaches use inheritance and aggregation patterns to establish relationships between components and their variants. Therefore, this notation is closer to class diagrams rather than to architectural descriptions.

There is a growing number of approaches that combine AOP and SPLs [7], [22], [23]. The Aspect-Oriented Modeling (AOM) [7] proposes a PL-ADL where everything is an aspect, but it rejects the main concepts of formal *Architecture Description Languages*. Finally, a work that is very close to the notion of PPC is proposed by Lee et al. [23]. They implement variable features that crosscut several modular units by using aspects. These aspects modify the internal behavior of components following invasive composition. This work is defined at the implementation level using the programming languages AspectJ [9]. However, the use of AspectJ makes their aspects dependent on the linking context (i.e., the component), and thus, they are not reusable. The Flexible-PLA Model also defines variants by using the concepts of aspect-oriented, but at an architecture-level. These variants specify their pointcuts and weaving operators outside the variants, thus they are reusable. And the most important advantage, Flexible-PLA takes an step forward providing an MDD support to PLAs by generating AspectJ code from Flexible-PLA models as a result of a model-to-code transformation.

VII. Conclusion and Further Work

This paper presents a model-to-code transformation to put model-driven SPL development into practice. We take the advantages of the Flexible-PLA Model and the concept of PPC to specify the external variability of the PLA configuration and the internal variability of components. The variability of Flexible-PLA models is configured to derive specific products, and then, the code for specific products is automatically generated. Thanks to this MDD process, architecture and code are automatically traced, and software evolution can be well-supported by employing high-level abstractions and automating code generation to reduce human error. As future work we plan managing constraints of validation in model transformations, the specification of

internal services through the UML activity diagrams to offer more automation, as well as to apply these advances in more case studies.

ACKNOWLEDGMENT

The work reported in here has been partially sponsored by the Spanish fund: INNOSEP (TIN200913849), IMPONET (ITEA 2 09030, TSI-02400-2010-103), iSSF (IPT-430000-2010-038), NEMO&CODED (ITEA2 08022, IDI-20110864) and ENERGOS (CEN-20091048) and UPM (Researcher Training program).

REFERENCES

[1] K. Pohl, G. Bckle, and F. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Germany, 2005.

[2] F. Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action*. Springer Berlin Heidelberg, 2007.

[3] I. Schaefer et al., "Software diversity: state of the art and perspectives," *International Journal on Software Tools for Technology Transfer*, vol. 14, pp. 477–495, 2012.

[4] J. Pérez, J. Díaz, C. C. Soria, and J. Garbajosa, "Plastic partial components: A solution to support variability in architectural components," in *WICSA/ECSA '09: Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. IEEE Computer Society Press, 2009, pp. 221–230.

[5] A. van der Hoek, D. Heimbigner, and A. L. Wolf, "Capturing architectural configurability: Variants, options, and evolution," Department of Computer Science, University of Colorado, Boulder, Colorado, Tech. Rep., 1999.

[6] F. Bachmann and L. Bass, "Managing variability in software architectures," in *SSR '01: Proceedings of the 2001 symposium on Software reusability*. New York, NY, USA: ACM, 2001, pp. 126–132.

[7] N. Noda and T. Kishi, "Aspect-oriented modeling for variability management," in *SPLC '08: Proceedings of the 2008 12th International Software Product Line Conference*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 213–222.

[8] E. Adachi Barbosa, T. Batista, A. Garcia, and E. Silva, "PL-AspectualACME: An aspect-oriented architectural description language for software product lines," in *Software Architecture*, ser. LNCS, vol. 6903. Springer Berlin / Heidelberg, 2011, pp. 139–146.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 2001, pp. 327–353.

[10] G. Kizcales, J. Lamping, A. Mendhekar, and C. Maeda, "Aspect-oriented programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science, vol. 1241. Springer-Verlag, 1997.

[11] Eclipse.org, "Eclipse Modelling Framework," http://www.eclipse.org/gmt/emf.

[12] Epsilon, "Extensible Platform for specification of Integrated Languages for mOdel maNagement," http://www.eclipse.org/gmt/epsilon.

[13] J. L. Martin, A. Yague, E. Gonzalez, and J. Garbajosa, "Making software factory truly global: the smart software factory project," in *Software Factory Magazine. Available on http://www.softwarefactory.cc/magazine*, F. Fagerholm, Ed., March 2010, p. 19.

[14] T. Weiler, "Modelling architectural variability for software product lines," in *SVM'03: Proceedings of the Software Variability Management Workshop*, 2003, pp. 53–61.

[15] J. Pérez, N. Ali, J. Carsí, and I. Ramos, "Designing software architectures with an aspect-oriented architecture description language," in *Component-Based Software Engineering*, ser. LNCS, vol. 4063. Springer Berlin / Heidelberg, 2006, pp. 123–138.

[16] M. Shaw, "Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status," in *Studies of Software Design*, ser. LNCS, vol. 1078. Springer Berlin / Heidelberg, 1996, pp. 17–32.

[17] Object Management Group, "Unified Modeling Language (OMG UML), Superstructure Version2.2," http://www.omg.org/spec/UML/2.2/Superstructure, 2009.

[18] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The koala component model for consumer electronics software," *Computer*, vol. 33, no. 3, pp. 78–85, 2000.

[19] A. van der Hoek, M. Mikic-Rakic, R. Roshandel, and N. Medvidovic, "Taming architectural evolution," in *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2001, pp. 1–10.

[20] D. L. Webber and H. Gomaa, "Modeling variability in software product lines with the variation point model," in *ICSR-7: Proceedings of the International Conference on Software Reuse*, ser. LNCS, vol. 2319. Springer, 2002, pp. 109–122.

[21] M. Razavian and R. Khosravi, "Modeling variability in the component and connector view of architecture using uml," in *AICCSA '08: Proceedings of the International Conference on Computer Systems and Applications*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 801–809.

[22] G. Kakarontzas, P. Katsaros, and I. Stamelos, "Elastic components: Addressing variance of quality properties in components," in *EUROMICRO-SEAA'07: Proceedings of 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE Computer Society, 2007, pp. 31–38.

[23] K. Lee, K. C. Kang, M. Kim, and S. Park, "Combining feature-oriented analysis and aspect-oriented programming for product line asset development," in *SPLC '06: Proceedings of the 10th International on Software Product Line Conference*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 103–112.