Designing an Adaptive User Interface According to Software Product Line Engineering

Yoann Gabillon and Nicolas Biri and Benoît Otjacques

Luxembourg Institute of Science and Technology (LIST)
5, avenue des Hauts-Fourneaux
L-4362 Esch/Alzette
Email: {yoann.gabillon, nicolas.biri, benoit.otjacques}@list.lu

***Abstract*—An adaptive User Interface (UI) is a UI that is able to adapt itself to a change of the context of use (user, platform, environment). Designing an adaptive UI remains a difficult and time consuming task that needs the use of common and variability parts between the different UI adaptations. Software Product Line (SPL) engineering is a software engineering approach that aims to develop a collection of similar software systems by using software assets and a variability model: the Feature Model. Dynamic Software Product Line is an adaptation of the SPL approach in order to design an adaptive software system. This paper proposes a method to design an adaptive UI according to a DSPL process. This method is implemented through the UI ADAPTOR prototype. This first implementation underlies the several benefits of the proposed method.**

Keywords–Adaptive User Interface; Software Product Line; Dynamic Software Product Line; Feature Model; Context of use.

I. INTRODUCTION

With the increasing amount of devices and mobile platforms as well as new user profiles, designers need to design a software system adapted to the current context of use. A software system encompasses functional core and User Interface (UI) parts. An adaptive User Interface (adUI) is a UI that is able to adapt itself or to be adapted, automatically at run time, to a context change. A context of use varies according to the properties regarding the *users*, the *platforms* and the *environment* of interaction. Because there are a huge amount of possible combinations, designing an adUI is a difficult and time consuming task.

Model-based approaches for the UI development (MBUID) are mainly used in adUI development in order to decrease the development costs. MBUID approaches promote the modelling of adaptation rules to make the adUI evolve according to a change of context [1] [2]. The designer must understand and identify the adaptation rules (variations) and its effects [3], i.e., the adaptation rules must be transparent (understandable and reusable) for developers. However, the huge amount of possible combinations make it difficult to understand and lead to a lack of reusability and design errors.

Software Product Line Engineering (SPLE) aims to develop a collection of similar software systems from a shared set of software assets. This approach is used to design a set of software systems that encompass common and variability parts by using a model of variability. Indeed, the variability model helps the designer to understand variations and reuse concepts and tools to check consistency of variations.

In order to decrease the development cost of MBUID approaches and to increase the understandability/reusability of adaptation rules, this paper provides a method to design an adUI according to SPLE. This method allows to reuse concepts and tools proposed by SPLE approach to manage and understand variabilities and its effect.

The paper begins with a background presentation of SPLE including the whole process and feature model. Based on related work on adUI design and on UI design by SPLE (Section 3), the paper proposes a method to design adUI according to SPLE in Section 4. In Section 5, the proposed method is implemented through a prototype called UI ADAPTOR. Based on UI components, this prototype composes a UI adapted to the current context of use. UI ADAPTOR aims to collect experimental lessons and to underlying the many benefits of the proposed method.

II. BACKGROUND

The *software product line (SPL)* development method separates the two following processes: the *domain engineering* and the *application engineering* [4]. Firstly, *domain engineering* is the process which is responsible for establishing the reusable artefacts and thus for defining the commonality and the variability of the product line. All types of software artefacts needed to develop the final products may be developed: requirements, design, realisation, tests, etc. Secondly, *application engineering* is the process which is responsible for deriving product line applications (products) from the artefacts established in domain engineering. A large part of application engineering consists of reusing artefacts of the domain engineering and binding the variability as required for the different applications.

The *Dynamic SPL (DSPL)* development method [4] is an adaptation of the SPL method in order to produce only one adaptive product, instead of a set of products. The DSPL process also separates the requirement engineering that is made at the design time and the domain engineering that is made automatically this time at runtime (i.e., dynamically) according to the current context of use.

Mostly, *Feature Models* (FM) are widely used to model the variability of requirements during the whole SPL and DSPL process. The *Feature Model* was first proposed by Kang and

 ISBN: 978-1-61208-382-7

al. as a part of Feature Oriented Domain Analysis (FODA) study [5]. Since then, feature models have been very popular in software product lines and have been widely accepted and used by the academic and industrial communities, although several dialects exists [6]. A FM is a tree that defines relationships between the features. The four following relationships specify the hierarchical decomposition of a feature into its sub-features: *Mandatory*, *Optional*, *Alternative* and *Or*. *Mandatory* means that the feature is included in every product that includes its parent. *Optional* means that the feature may or may not be included in a product that includes its parent. *Alternative* means that every product that includes the parent must include exactly one feature from the group. *Or* means that every product that includes the parent must include a non-empty subset of the group. Constraints are used to specify cross-tree relationships between features. A constraint consists of a boolean expression. For example, "$A \Rightarrow B$" means that if B is selected during the configuration phase, A must be selected as well.

III. Related work

This section presents a related work overview of approaches in adaptive UI development and in UI development according to SPLE.

A. Adaptive User Interface

According to [7], different software development approaches have already been investigated to design an adUI. For example, MBUID [1] [2] [8] promotes the modelling of transformation, Aspect-oriented programming [9] pushes forward insertion of aspects at runtime or Component-based programming [10] focuses on the (re)composition of available components. These approaches are based on the same principle: an adaptive system is in charge of adapting a UI by adaptation rules according to the current context of use. As reference example, the SERENOA project [3] [2] [11] identifies the main models needed to design an adUI: the UI, the rules of adaptation, the context of use and the adaptor. The adaptor is in charge of applying rules on the UI model according to the context of use to produce a new UI model expressing the adapted UI. The UI model is defined according to the four standard levels of abstraction in MBUID: Task Model, Abstract User Interface, Concrete User Interface and Final User Interface [12].

However, according to [3], a main issue is the lack of inspectability and understandability of adaptation rules that leads to possible side effects and lack of reusability.

B. User Interface development by Software Product Line

Even if the design of adaptive systems by SPLE has already been investigated [13] [14], this work focuses on the adaptation of the functional core part. In contrast, the literature concerning UI design according to SPL engineering focuses on the design of a set of UIs. [15]–[17] propose to model variants of UI features such as different whole UIs. As a consequence, UI variations cannot be reused because they are not traceable or saveable. [18] propose to model variations of interactors. Each interactor that varies is a variant. For example, a interaction variation can have two alternative variants: JComboBox or JList. This modelling has the advantage to be easily inspectable and traceable. [19] [20] argue in favour of the use of the standard levels of abstraction to model variability into the feature model. However, the selection of UI variations is based on functional features. Unfortunately, UI variations are necessary to improve usability even if functional features do not change [21]. In contrast, [22] propose a methodology to design a set of UIs from the selection of UI variants based on a feature model containing the four abstraction levels.

IV. Method to design Adaptive UI by DSPL

An overview of the SPLE-based Adaptive UI design method is sketched in Figure 1. We have adapted the DSPL process to design an adaptive UI. The main idea is to prepare UI components and to model their variations according to the targeted context of use during the domain engineering at design time. Then, at runtime, the UI adaptor selects and composes UI components according to the current context of use.

Firstly, at design time during the requirement engineering, context variations and UI variations must be modelled as features of the FM. The context features represent context variations such as recommended by [23]. The UI features represent variations between UI adaptations such as recommended by [22]. For example, the screen size variation can be expressed as two features: smallScreen and largeScreen. The UI variations can be defined at different abstraction levels. For example, two interactors can be expressed as two features: slider and textfield. In order to express the link between the context features and the UI features, we can add constraints. For example, the slider that is replaced by the textfield for space reasons can be expressed as two constraints: "slider $\Rightarrow$ largeScreen" and "textfield $\Rightarrow$ smallScreen".

Secondly, the UI components must be designed corresponding to each feature.

Thirdly, at run time during the application engineering, the appropriate context feature and UI features must be selected according to the current context of use. This selection must be made automatically to promote a context-aware adaptation of the composed UI. Many tools have been developed (such as Feature IDE [24]) in order to automatically selects the UI features according to the constraints and the selected context features.

Finally, the components corresponding to the selected features are composed to automatically design the adUI.

In order to promote context-aware adaptation of the composed UI, an adaptive system in charge of recomposing the adUI. From a context change detection, the UI adaptor updates the context of use model and, in consequence, selects the context feature. The context feature selection leads to a new UI feature selection and a new UI composition to produce the new adUI.

V. Applying the method: Implementation

Example application. Lets suppose a user, Orson, who want to visualize the consumption data of his house. To achieve his goal, he uses an adaptive visualization software called "*VisuData*" (the Figure 2 shows a possible UI of this software). *VisuData* provides four graphics in order to visualise a set a data: a horizontal barchart, a vertical barchart, a pie chart and a bubble chart. The *VisuData*'s user could use three data filters. The first, called "DataDisplayed", allows to choose the data displayed. For example, Orson could select the quantity

 ISBN: 978-1-61208-382-7

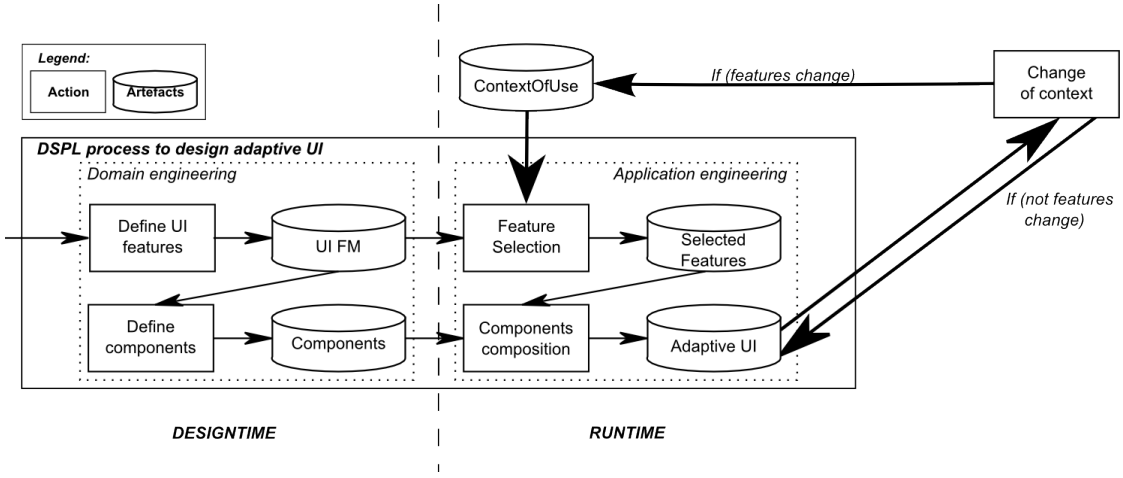

Figure 1. Overview of the Dynamic SPL process to design an adaptive UI.

of electric energy or the price consumed by the radiators. The second, "TimeLine", allows to select the time interval of the data displayed. For example, Orson could display the consumption between 2010 and 2012. The last filter ("Filter 1D") allows to select the value interval of the data. For example, Orson could choose to display the radiators that consume between 0 and 2000 dollars.

As requirements, the aim is to design this software adapted to three different contexts (for convenience, the design of the three adaptations according to three platforms are presented but the benefit of the approach increse as well as the number of adaptations):

1) if the user has a **personal computer with a large screen**, the graphic and the filters are displayed in a same frame (see Figure 2). The graphics selection is made by icons (see the"$SelectGraphic\_Icon$" red square in Figure 2). The selected data are displayed as a list widget (see the "$DataDisplayed\_Radio$" square in the Figure 2). The two other filters (see "$TimeLine\_Slider$" and "$Filter1D\_Slider$" red squares in Figure 2) use a slider to select intervals.
2) if the user has a **smartphone with a small screen** (see Figure 3), the graphics and the filters are displayed in two tabs. Because they use less space, the icons and the list are replaced by a combo box (see "$SelectGraphic\_ComboBox$" and "$DataDisplayed\_ComboBox$" in Figure 3), the Sliders are replaced by two textfields (see "$TimeLine\_2TextField$" and "$Filter1D\_2TextField$" in Figure 3).
3) if the user has a **personal computer with a large screen** and a **smartphone with a small screen**, the graphic is displayed on the PC (such as in the Figure 2 without the three filters) and the three filters are displayed on the smartphone (such as the second screen in the Figure 3 without the tab menu).

Prototype implementation. The prototype UI ADAPTOR is implemented in JAVA and JAVAFX. The two platforms are simulated according to their screen size (1280x1024 for the large screen and 640x960 for the small screen).

Feature model. These requirements are modelled by the feature model of Figure 4 according to the methodology proposed by [22]. The UI variations depend on the task. For example, the "$filter1D$" feature expresses the choice for the designer to use a slider or two textfields to design the filter 1D component. Each constraint informs the designer how the UI feature can be selected. For example, the "$Filter1D\_Slider$" feature is selected if there is large screen and no other platform available. Consequently, this feature is selected when Orson has a personal computer (case 1 of the requirements). The context of use variations depend on the screen size and the number of platforms. For example, a screen size can be large or small. The "$LargeScreen2$" feature is in red because it can be selected. Indeed, when Orson has a second platform, it is a smartphone such as specified by the requirements.

Components. The components are implemented in JAVA FX. The component model is not the focus of this paper, the component model is defined as in [10]. Each component corresponds to a concrete features of the feature model (Figure 4) and is represented by a red square in the Figures 2 and 3. The UI of these components are underlined in a red square in the Figures 2 and 3. A last component is developed corresponding to the "$VisuData2\_Frame$" feature. This component encompasses two frames in charge of displaying the graphic on the large screen and the three filters on the small screen.

Context-aware adaptation life-cycle. UI ADAPTOR supports the context-aware adaptation life-cycle. Firstly, the context of use perception is simulated by the designer, i.e., the context features are selected manually. Secondly, from the selected context features, Feature IDE deduces (thanks to a SAT solver [24]) the selected UI features automatically. Thirdly, from the list of selected features, the UI ADAPTOR composes the appropriate component in order to produce a composed UI displayed according to the corresponding size of the screen.

VI. CONCLUSION AND LESSONS LEARNED

This paper has proposed a method to design adaptive User Interfaces according to SPL engineering. The experience allows to learn lessons and benefits concerning the proposed methods.

 ISBN: 978-1-61208-382-7

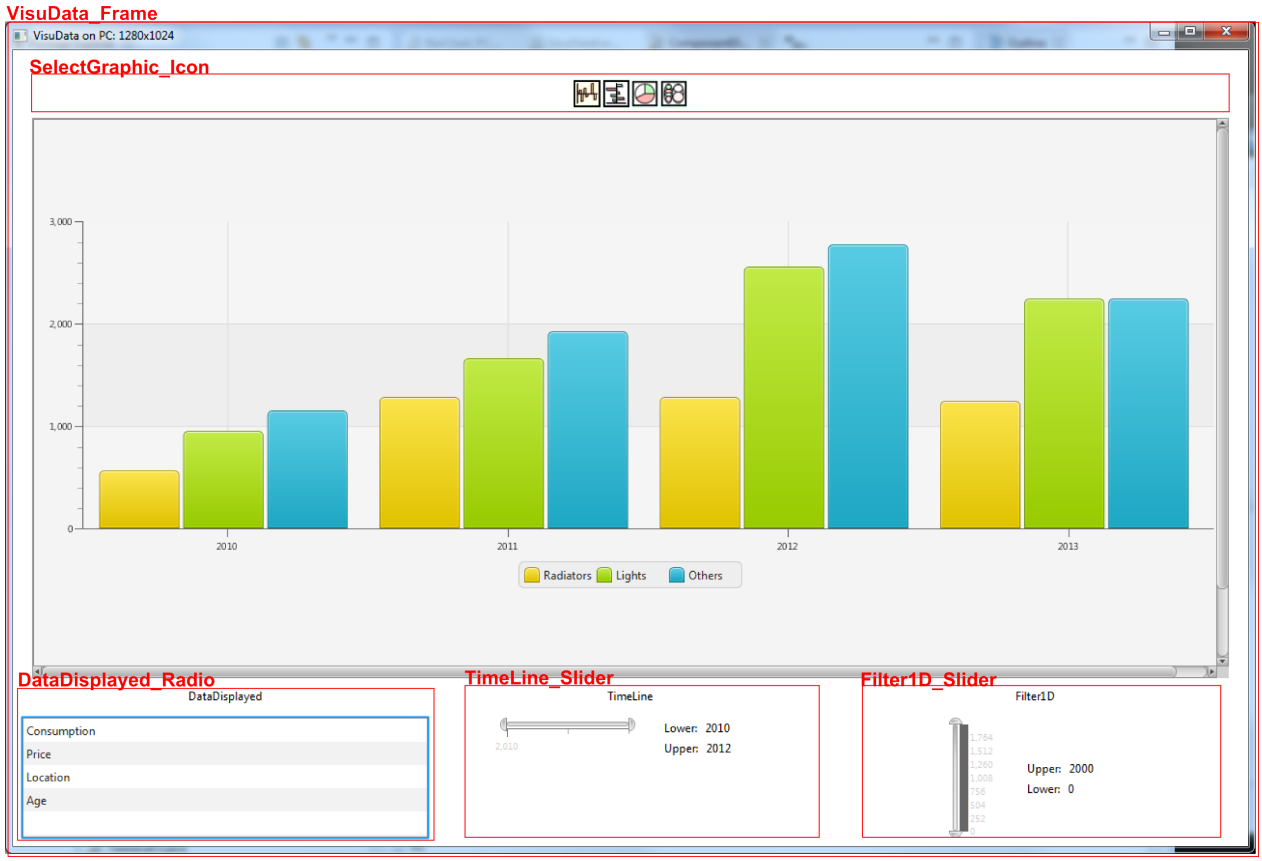

Figure 2. The composed User Interface adapted to a large screen. The red squares and the red labels are added to identify the UI components.

Figure 3. The composed User Interface adapted to a small screen. The red squares and the red labels are added to identify the UI components.

 ISBN: 978-1-61208-382-7

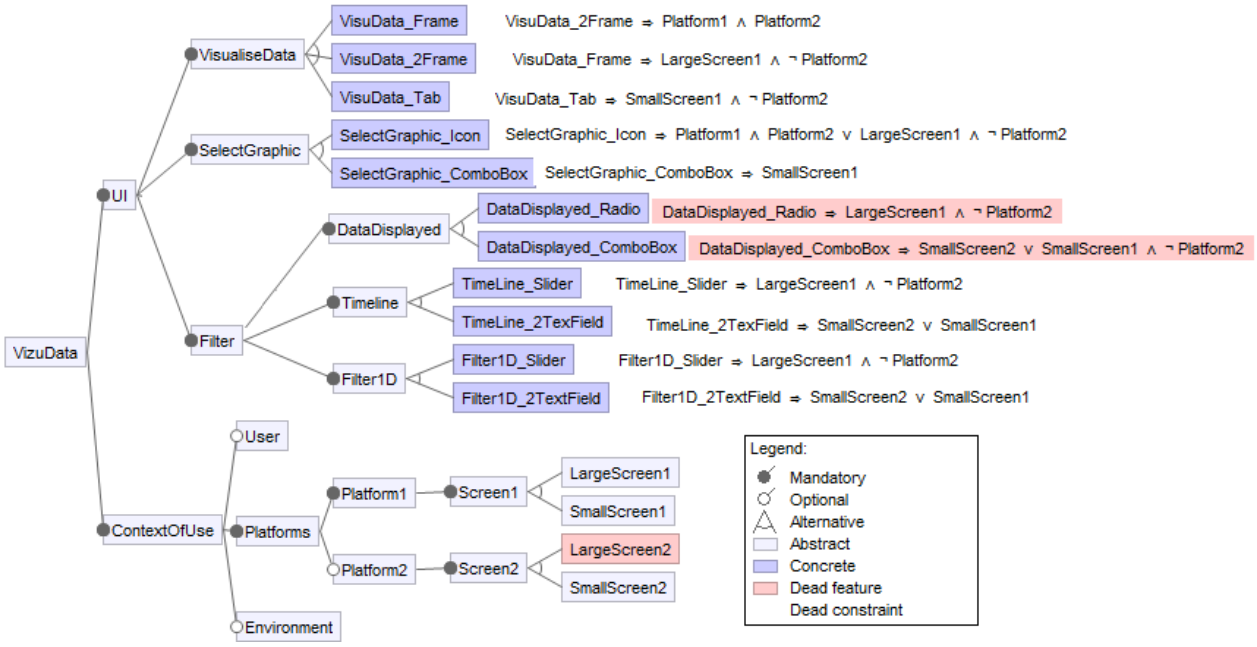

Figure 4. Feature Model of VisuData

Feature model. Because the feature model has a visual representation compared to adaptation rules, it increases transparency and understandability of the adaptation rules used by producing a formal and visual model of variability. Because the FM is formal, the FM also increase reusability between different adUIs and different designers. It is also particularly important to maintain and reuse adUI in order to design new adaptations to add new contexts. The constraints on the feature model increase the validity of the adaptation. For example, the dead features (LargeScreen2) and the inconsistent constraints are detected by Feature IDE. Moreover, the context of use can be easily modelled because there are other FM editors (such as CVL [25]) that handle the cardinality of the features. In consequence, these editors allow to model only one platform features without listing the number of platforms used.

Components. The use of components increase reusability and enhancement of quality because the UI components are reviewed and tested in many adaptive/composed UIs. Component-based programming can also be combined or replaced by other programming paradigms. For example, a component can be seen as a set of elements of the standard abstraction levels [12]. Moreover, when a UI component is maintained, the change can be propagated to all adUI with which the component is being used.

Process. The flexibility of the method allows to reuse pre-existing development approaches to design adUI. For example, the adaptation rules can be defined as aspects or models. The features can model the variability of a component, a task, an interactor but also the adaptation rules. In addition, the proposed method allows to improve the cost estimation of the adUI to design based on previous experience [4].

We plan to increase the number and type of artefacts by adding tests to design more complex adUI. We plan to design a complete framework to design AdUi, i.e., including context detection and multi programming languages (HTML and JAVA) in order to evaluate the designer effort and adaptation performance.

REFERENCES

[1] J.-S. Sottet et al., "Model-driven adaptation for plastic user interfaces," in Human-Computer Interaction–INTERACT 2007. Springer, 2007, pp. 397–410.

[2] V. G. Motti and J. Vanderdonckt, "A computational framework for context-aware adaptation of user interfaces," in RCIS, R. Wieringa, S. Nurcan, C. Rolland, and J.-L. Cavarero, Eds. IEEE, 2013, pp. 1–12.

[3] G. Meixner, F. Patern, and J. Vanderdonckt, "Past, present, and future of model-based user interface development." i-com, vol. 10, no. 3, 2011, pp. 2–11.

[4] K. Pohl, G. Bockle, and F. Van Der Linden, Software product line engineering. Springer, 2005, vol. 10.

[5] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (foda) feasibility study," DTIC Document, Tech. Rep., 1990.

[6] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps, "Generic semantics of feature diagrams," Computer Networks, vol. 51, no. 2, 2007, pp. 456–479.

[7] I. Jaouadi, R. Ben Djemaa, and H. Ben Abdallah, "Interactive systems adaptation approaches: A survey," in ACHI 2014, The Seventh International Conference on Advances in Computer-Human Interactions, 2014, pp. 127–131.

[8] F. Paterno, C. Santoro, and L. D. Spano, "MARIA: a universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments," vol. 16, no. 4, pp. 1–30.

[9] A. Blouin, B. Morin, O. Beaudoux, G. Nain, P. Albers, and J.-M. Jzquel, "Combining aspect-oriented modeling with property-based reasoning to improve user interface adaptation," in Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems. ACM, 2011, pp. 85–94.

[10] Y. Gabillon, M. Petit, G. Calvary, and H. Fiorino, "Automated planning for user interface composition," in Proceedings of the 2nd International Workshop on Semantic Models for Adaptive Interactive Systems: SEMAIS'11 at IUI 2011 conference. Springer HCI, 2011, pp. 1–5.

[11] V. G. Motti, D. Raggett, and J. Vanderdonckt, "Current practices on model-based context-aware adaptation," in CASFE, 2013, pp. 17–23.

 ISBN: 978-1-61208-382-7

[12] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt, "A unifying reference framework for multi-target user interfaces," Interacting with Computers, vol. 15, no. 3, 2003, pp. 289–308.

[13] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," Computer, vol. 41, no. 4, 2008, pp. 93–95.

[14] C. Parra, X. Blanc, and L. Duchien, "Context awareness for dynamic service-oriented product lines," in Proceedings of the 13th International Software Product Line Conference. Carnegie Mellon University, 2009, pp. 131–140.

[15] P. Trinidad, A. Ruiz-Cortés, J. Pena, and D. Benavides, "Mapping feature models onto component models to build dynamic software product lines," in International Workshop on Dynamic Software Product Line, DSPL, 2007, pp. 51–56.

[16] K. Geihs et al., "A comprehensive solution for application-level adaptation," Software: Practice and Experience, vol. 39, no. 4, 2009, pp. 385–422.

[17] H. Arboleda, A. Romero, R. Casallas, and J. Royer, "Product derivation in a model-driven software product line using decision models," in Proceedings of the Memorias de la XII Conferencia Iberoamericana de Software Engineering, CIbSE, vol. 2009, 2009, p. 59.

[18] K. Garcés, C. Parra, H. Arboleda, A. Yie, and R. Casallas, "Variability management in a model-driven software product line," Revista Avances en Sistemas e Informática, vol. 4, no. 2, 2007, pp. 3–12.

[19] A. Pleuss, G. Botterweck, and D. Dhungana, "Integrating automated product derivation and individual user interface design," Proceedings of VaMoS, vol. 10, 2010, pp. 69–76.

[20] A. Pleuss, B. Hauptmann, D. Dhungana, and G. Botterweck, "User interface engineering for software product lines: the dilemma between automation and usability," in Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems. ACM, 2012, pp. 25–34.

[21] Q. Boucher, G. Perrouin, and P. Heymans, "Deriving configuration interfaces from feature models: A vision paper," in Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems. ACM, 2012, pp. 37–44.

[22] Y. Gabillon, N. Biri, and B. Otjacques, "Designing multi-context uis by software product line approach," in Proceedings of the International Conference on Human-Computer Interaction (ICHCI'13). World Academy of Science, Engineering and Technology (WASET), 2013, pp. 628–637.

[23] A. S. Karataş, A. H. Doğru, H. Oğuztüzün, and M. Tolun, "Using context information for staged configuration of feature models," Journal of Integrated Design and Process Science, vol. 15, no. 2, 2011, pp. 37–51.

[24] C. Kastner et al., "Feature ide: A tool framework for feature-oriented software development," in Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on. IEEE, 2009, pp. 611–614.

[25] K. Czarnecki, P. Grunbacher, R. Rabiser, K. Schmid, and A. Wkasowski, "Cool features and tough decisions: a comparison of variability modeling approaches," in Proceedings of the sixth international workshop on variability modeling of software-intensive systems. ACM, 2012, pp. 173–182.

 ISBN: 978-1-61208-382-7