An approach for managing a distributed feature model to evolve self-adaptive dynamic software product lines

Bruno Iizuka Moritani
Infolab21, Lancaster University
South Drive, Lancaster University
Lancaster, Lancashire, UK LA1 4WA
b.iizuka@lancaster.ac.uk

Jaejoon Lee
Infolab21, Lancaster University
South Drive, Lancaster University
Lancaster, Lancashire, UK LA1 4WA
j.lee3@lancaster.ac.uk

ABSTRACT

While maintaining core assets of a product line, product line engineers may need to adapt the assets to accommodate new requirements and new devices from the changing and newly emerging markets. Additionally, due to the emergence of new computing environments like autonomous systems (e.g., ubiquitous computing and the Internet of Things), there is increasing demand for dynamic adaptations of core assets, and this often needs to be managed with minimum human intervention. We propose an approach for managing a distributed feature model in order to facilitate the adaptation of self-adaptive dynamic software product lines (DSPLs). In addition, our approach allows the change of behaviours to promote the long lifecycle of self-adaptive DSPL. The case study applied in this paper is a baby care system (BCS). BCS assists people to monitor a baby while they are sleeping. We tested our BCS in the following scenarios: adding features, removing features and changing behaviours. All these scenarios showed that it is possible to use our approach to self-adapt DSPLs.

CCS CONCEPTS

• **Software and its engineering** → **Designing software**; *Software architectures*; **Software product lines**;

KEYWORDS

Dynamic Software Product Lines; Software Evolution; Feature Model

ACM Reference format:
Bruno Iizuka Moritani and Jaejoon Lee. 2017. An approach for managing a distributed feature model to evolve self-adaptive dynamic software product lines. In *Proceedings of SPLC '17, Sevilla, Spain, September 25-29, 2017,* 4 pages.
https://doi.org/10.1145/3109729.3109743

1 INTRODUCTION

New features and devices emerge continuously during the lifecycle of software systems to improve the quality of systems and to provide new functionalities. In some cases, the introduction of these features happens at runtime, i.e. they need to be adapted dynamically. This adaptation at runtime frequently exists in new computing environments like ubiquitous computing [12], the Internet of Things (IoT) [3] and autonomous computing [8]. In addition, the dynamic adaptation needs to be managed in an autonomous way.

In the literature, different approaches have been proposed to implement the dynamic adaptation, including autonomous computing [1], self-adaptive systems [5] and dynamic software product lines [7]. Of these approaches, Dynamic Software Product Lines (DSPLs) are based on the idea of software product line engineering (SPLE) [10] and extend SPLE to manage product configuration changes at runtime. DSPLs monitor their operating environments and adapt at runtime in response to environmental changes.

There is increasing interest in combining self-adaptation with DSPL to address new emerging challenges of distributed DSPLs [4][6][11]. That is, a product line of distributed systems without a central controller should have a self-organising capability so that each constituting node knows how to self-coordinate to deliver the overall functional/non-functional requirements. We have identified several challenges related to deploying new features and composing them with existing features at runtime in the context of self-adaptive DSPLs:

(1) Improved maintainability: the adaptation should occur with a minimum human intervention. One of the characteristics of self-adaptive systems is that a device may join and leave the system boundary at runtime and this should be managed autonomously, as users cannot be expected to continually get involved in system configuration.
(2) Improved availability: a self-adaptive DSPL should continuously deliver services with a minimum downtime. As we expect frequent reconfiguration with small and mobile devices, we should achieve seamless integration of new features with continuous operation of overall services.
(3) Improved lifespan: the longevity of systems implies that devices cannot be simply replaced when they become obsolete. This means that they should be able to evolve and change their behaviours to meet overall requirements that govern how the devices work together.

Based on our experience and literature survey, we have identified two critical issues to address the challenges. They are:

(1) An efficient and autonomous way to manage product configuration information after deployment - the system context we are targeting is highly distributed, and each constituting node may have limited resources. Therefore, we should find

SPLC '17, September 25-29, 2017, Sevilla, Spain

ACM ISBN 978-1-4503-5119-5/17/09…$15.00
https://doi.org/10.1145/3109729.3109743

a none-resource-intensive way to recognise a changed product context and adapt to it, if the new context is relevant; and

(2) Software architecture design to support seamless integration with a new configuration - each distributed node should be able to cope with changing contexts. As such, the architecture design of each node should support context awareness and autonomous behaviour changes.

In this paper, we address the first issue by introducing a concept of distributed feature model, called DisFM. This approach allows us to divide a feature model into distributed and deployable sets of features of a self-adaptive DSPL, and to deploy each set onto a distributed node. Therefore, each node only manages its own partial feature model (we call this set a feature unit) during its lifecycle. Each node can communicate with nodes in the vicinity and can perform matching and update operations on its feature unit whenever it finds comparable and related feature units nearby.

To address the second issue of software architecture, we introduce a software architecture model that supports two different categories of behaviours. One is an active behaviour that controls the interactions with other passive feature units (coordinator behaviour), and the other one is a passive behaviour that allows other active feature units to control their behaviours (subordinate behaviour). This architecture is called "Hybrid between Passive/ Active Behaviours" (Hy-PAB) [9] to refer to the software architecture design needed to support these two extreme sets of behaviour. Hy-PAB is a novelty in that it brings the two extreme feature unit behaviours as the main driver in architecture design and provides a basic mechanism to support the longevity of self-adapting DSPLs.

The remainder of this paper is organised as follows: section (2) introduces the case study (baby care system) to evaluate the approach. The next section 3 discusses the method concept, distributed feature model, and Hy-PAB. Section 4 presents the related work. Finally, Section 5 discusses and concludes this paper.

2 CASE STUDY - BABY CARE SYSTEM

We implemented a baby care system (BCS) to see the first results of our proposed approach. A BCS is a system to monitor a baby when parents are not around. BCS has characteristics of a self-adaptive DSPL, such as: dynamic variability (e.g. parents can select which function they want to monitor at runtime); distributed nodes (e.g. each function of the BCS can be implemented in different nodes such as room temperature detection, and sound detection can be implemented in different nodes); and, high availability requirements.

A BCS has the following features: *Sound detection* is responsible for detecting if a baby is making any sound; *Room temperature detection* is responsible for detecting if the baby's room is warmer or colder than the limit set by the user; *Light detection* is responsible for detecting if the baby's room is brighter or darker than the limit determined by the user. They trigger an alarm, in case the sensor reading (e.g., sound decibel, temperature, and brightness) is out of the user-set range. In addition, *Video monitor* is responsible for displaying on a screen and recording through a camera the baby's image.

The next section presents the concept that is going to be tested using BCS.

3 METHOD CONCEPT

We propose an approach for managing a distributed feature model to facilitate the adaptation when features are added, removed and changed at runtime in self-adaptive DSPLs.

3.1 Distributed Feature Model

DisFM is a concept that we divide a feature model into feature units, and each node of a self-adaptive DSPL manages its corresponding feature unit. A feature unit is a subset of a feature model of a product line and is determined by product line engineers. PL engineers should consider a feature unit in a feature model according to some suggested guidelines: (i) children of the root of a feature model and their child are different feature units; (ii) optional/alternative features and their child are considered a feature unit; (iii) features that are child of multiple features are a feature unit; These guidelines are a recommendation of how to find out a feature unit, so the engineers do not have to follow them strictly. For example, Figure 1 depicts the feature units found in BCS. First, engineers determine that *Sleep Light*, *Monitoring Alert*, and *Video Monitor* are three different feature units. After this division, engineers search for optional and alternative features that could be feature units, such as *Light Detection*, *Room Temperature Detection*. As shown in Figure 1, *Record Video*, *Zoom Camera*, *Rotate Camera*, *Night Vision* were not divided into different feature units due to the option of engineers to implement them. Finally, *Alarm* and *Data Communication* are children of different parents and are divided into two different feature units. After the feature unit identification, each feature unit can be implemented and deployed. A feature unit is converted into a text format and encoded as a configuration file of the system. This subset represents a service that is deployable in a node. As such, a node can provide its own service independently, but when the node discovers other nodes in its neighbourhood at runtime, they exchange their feature unit information and perform a feature-name matching operation by comparing the feature names in the feature units. If identical feature names are found, they recognise them as points for interoperation and establish a feature link so that one node controls other nodes or vice versa.

The feature matching happens at runtime when a feature unit detects a message that was sent from another feature unit. The message contains all the features that are in the feature unit. After the process has received the message, the received feature unit is analysed if the new features are part of the same product line, i.e. it compares the roots of the feature model to verify if they belong to the same product line by comparing the name of the root. The comparison contrasts the name of both roots. After this confirmation, the process determines if the emerged feature units are new or if they are already known. Another scenario is where the process identifies a feature unit that is no longer available. The process needs to verify what it is going to be affected. It investigates if there is any feature which is not going to work anymore.

3.2 Conceptual Architecture

In this section, we explain the Hy-PAB conceptual architecture that enables the communication between feature units and the change of feature unit behaviours. Hy-PAB proposes that the feature units can have two different types of behaviour (active and passive

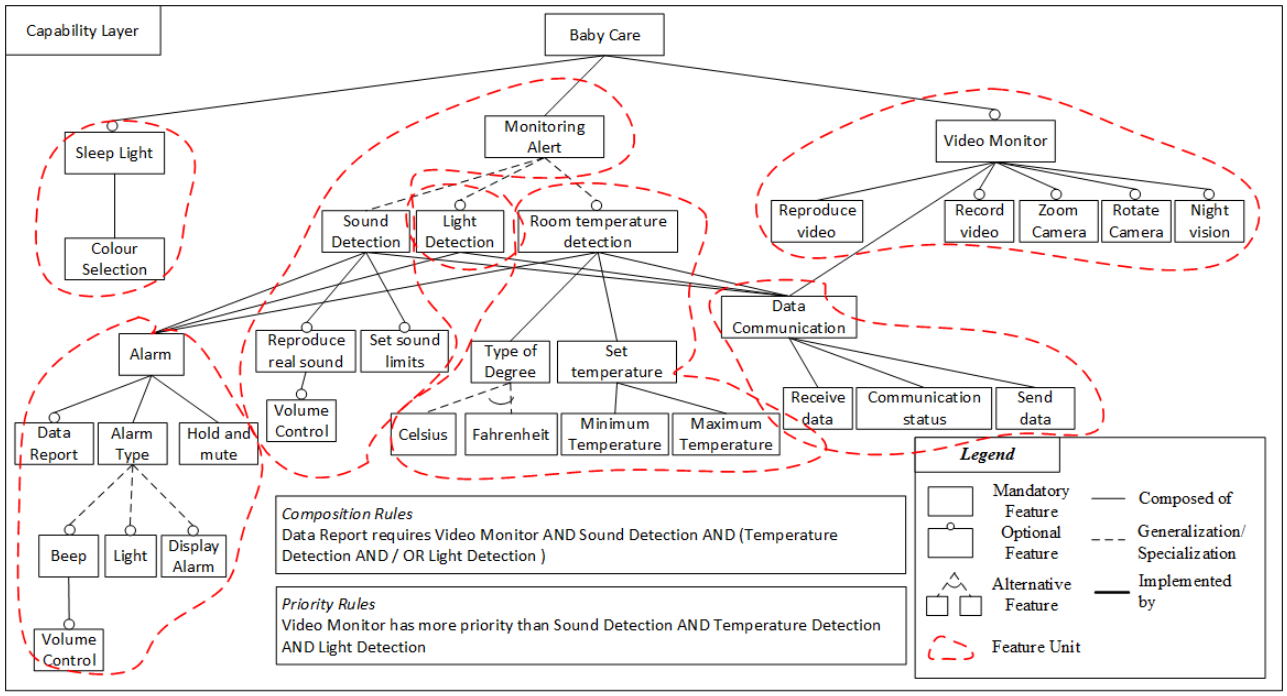

Figure 1: Partial feature model of the Baby Care System.

behaviours). While the active behaviour is to provide a service by itself and/or by using other available features in its vicinity, the passive behaviour lets other features use its produced data (e.g. sensor data) or control its behaviour (e.g. actuator control). The feature units start their execution in the active behaviour, and there is a possibility to change their behaviour when there is another feature unit in their neighbourhood. The trigger to change the feature unit behaviour is a priority rule. Priority rules that are under engineers responsibility supplement the feature diagram showing which features are going to be executed in active or passive behaviour when they are connected with certain feature units. For example, Figure 1 shows the priority rule for the BCS. In this case, if *Video monitor* is connected with *Sound detection* in the neighbourhood, *Video monitor* controls *Sound detection*, i.e. *Sound detection* acts as passive behaviour and *Video monitor* as active behaviour. Figure 2 depicts the conceptual architecture, and each component is explained in the following:

- *Neighbourhood Recogniser* is responsible for monitoring if a message has been sent from another node and to verify if there is a removed feature unit.
- *Hybrid Manager* checks if the new feature unit is part of the same product line and if a new feature and/or a new behaviour will emerge. After this check, the component determines which behaviour will be enabled/disabled to work with the new feature unit.
- *Active Manager* / *Passive Manager* are components responsible for executing the active and the passive behaviour of a feature unit.

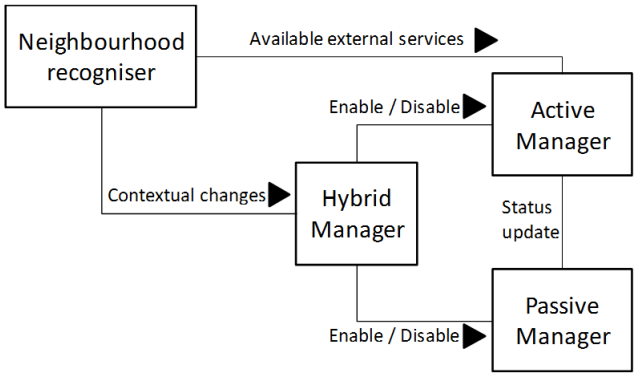

Figure 2: Hy-PAB Conceptual Architecture.

The case study dealt with several combinations of features that change dynamically at runtime. One of these combinations is the addition of *Video monitor* and *Sound detection*. The *Video Monitor* node has *Video Monitor* and *Data Communication* feature units; and, the *Sound detection* node has *Monitoring Alert*, *Alarm* and *Data Communication* feature units. The addition process occurs when *Sound detection* node is providing its service, and *Video monitor* node is switched on near the *Sound detection* node. Then Neighbourhood Recogniser (NR) of the *Sound detection* node finds out that *Video monitor* node is sending its feature unit. NR evaluates the information of *Video monitor*, i.e. it checks if *Video monitor* has already been detected or is a new feature unit. If it is a new feature unit, *Sound detection* adapts itself to connect to *Video monitor*. As a

result, *Sound detection* enables the passive behaviour and disables the active behaviour to work with *Video monitor*, due to the priority rules determined by the PL engineers. The same process occurs to *Video monitor* when it identifies *Sound detection*, but *Video monitor* will stay as active behaviour. It means that the control of *Sound detection* is now delegated to *Video monitor* and *Video monitor* will display the sound detection event instead of the alarm by *Sound detection*.

4 RELATED WORK

Acher et al. [2] propose the slicing of feature models. Their idea is to slice a feature model in order to decrease its complexity. Our concept is not only to slice to decrease the complexity, but also to divide a feature model into a distributed and deployable sets of features (feature units) to facilitate the evolution of DSPLs at runtime.

Abbas et al. [1] combine autonomic computing with the SPLs. They model an autonomic SPL architecture based on autonomic computing architecture supported by on-line training to deal with dynamic variation product line evolution. However, their focus is to show how the evolution should occur with the on-line training. Our approach allows the evolution using the distributed feature model and the adaptation at runtime by comparing feature units of each node; in this way is possible to evolve a dynamic product line without the training process. In addition, we contribute to increasing the lifecycle of the system using the different types of behaviours.

Quinton et al. [11] describe an evolution method based on DSPLs, which uses a variability model to specify how the reconfiguration is going to happen. Their idea is a more centralised variability model to enable the adaptation as the system evolves, i.e. their architecture maintains a variability model in one place and uses this to drive runtime adaptation of the product line. By contrast, we propose an approach (DisFM) to distribute parts of a feature model of the system in different nodes.

Gamez et al. [6] create a framework (FamiWare) to adapt product line in a mobile system context at runtime. FamiWare uses variability model to create the configurations to adapt the system at runtime. Each mobile has these configurations to allow the adaptation. Meanwhile, DisFM proposes the use of parts of a feature model that represents the services implemented in a node. Furthermore, DisFM introduces the change of behaviour between active (it controls itself and others) and passive (others control it) behaviours.

5 DISCUSSION AND CONCLUSION

We proposed an idea to distribute a feature model of a dynamic product line based on the concept of feature units in order to facilitate the dynamic adaptation. The use of DisFM reduces the maintenance effort of a DSPL by allowing the constituting nodes to self-adapt through exchanging the feature unit information and matching feature names. Also, it improves availability as the Hy-PAB architecture enables a feature unit to provide its service continuously without any downtime for reconfiguration. Finally, the change of behaviour (passive/active) of a feature unit enables its long lifecycle: a node can change its behaviour and be combined with new feature units. For example, *Sound detection* can be combined with *Room temperature detection* instead of *Video monitor* and the new combination will provide a user with more accurate reason for the sound (e.g., the baby could not sleep because the room was too hot).

DisFM presents the novelty that divides a feature model into distributed and deployable sets of features of a self-adaptive DSPL and deploys each set onto a distributed node. Each node can communicate with other nodes in the vicinity and can perform matching and update operations on its feature unit, whenever it finds comparable and related feature units nearby.

The initial experiments with the BCS showed that it is feasible to apply DisFM for self-adaptive DSPLs. The evaluation process of DisFM is still in progress and was not limited to the examples mentioned in section 2. The next steps of this work includes the comparison of the obtained results with the state-of-the-art techniques and the evaluation of them according to the challenges mentioned in Section 1.

ACKNOWLEDGMENTS

CAPES/Brazil supports Bruno de Abreu Iizuka (Process number: 9026-13-4).

REFERENCES

[1] Nadeem Abbas. 2011. Towards Autonomic Software Product Lines. In *Proceedings of the 15th International Software Product Line Conference, Volume 2 (SPLC '11)*. ACM, New York, NY, USA, 44:1—-44:8. https://doi.org/10.1145/2019136.2019187

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2011. Slicing feature models. *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings* (2011), 424–427. https://doi.org/10.1109/ASE.2011.6100089

[3] Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The Internet of Things: A survey. *Computer Networks* 54, 15 (2010), 2787–2805. https://doi.org/10.1016/j.comnet.2010.05.010

[4] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. 2009. Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer* 42, 10 (2009), 37–43. https://doi.org/10.1109/MC.2009.309

[5] Betty H C Cheng, Rogério De Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi a Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. 2009. Software Engineering for Self-Adaptive Systems: A Research Roadmap. *Software Engineering for Self-Adaptive Systems* (2009), 1–26. https://doi.org/10.1007/978-3-642-02161-9_2

[6] N Gamez, L Fuentes, and J Troya. 2015. Self-Adaptation of Mobile Systems with Dynamic Software Product Lines. *IEEE Software* 32, 2 (2015), 105 – 112. https://doi.org/10.1109/MS.2014.24

[7] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2008. Dynamic software product lines. *Computer* 41, April (2008), 93–95. https://doi.org/10.1109/MC.2008.123

[8] J O Kephart and D M Chess. 2003. The vision of autonomic computing. *IEEE Computer Society* 36, 1 (2003), 41–50.

[9] Jaejoon Lee. 2013. Dynamic Feature Deployment and Composition for Dynamic Software Product Lines. In *Proceedings of the 17th International Software Product Line Conference co-located workshops (SPLC '13 Workshops)*. ACM, New York, NY, USA, 114–116. https://doi.org/10.1145/2499777.2500717

[10] K. Pohl, G. Böckle, and F. Van Der Linden. 2005. *Software Product Line Engineering. Foundations, Principles, and Techniques.* Vol. 49. 467 pages. https://doi.org/10.1007/3-540-28901-1

[11] Clément Quinton, Rick Rabiser, Michael Vierhauser, Paul Grünbacher, and Luciano Baresi. 2015. Evolution in dynamic software product lines: challenges and perspectives. *Proceedings of the 19th International Conference on Software Product Line Pages - SPLC'15* (2015), 126–130. https://doi.org/10.1145/2791060.2791101

[12] German Sancho, Ismael Bouassida Rodriguez, Thierry Villemur, and Said Tazi. 2010. What about collaboration in ubiquitous environments?. In *2010 10th Annual International Conference on New Technologies of Distributed Systems (NOTERE)*. IEEE, 143–150. https://doi.org/10.1109/NOTERE.2010.5536749