A Framework for Modelling Variable Microservices as Software Product Lines

Moh. Afifun Naily[1], Maya R. A Setyautami[1], Radu Muschevici[2], and Ade Azurat[1]

[1] Faculty of Computer Science, Universitas Indonesia
[2] Dept. of Computer Science, Technische Universität Darmstadt
{afifunnaily@cs.ui.ac.id, mayaretno@cs.ui.ac.id,
radu.muschevici@cs.tu-darmstadt.de, ade@cs.ui.ac.id}

Abstract. Microservices architecture is a software development style that divides software into several small, independently deployable services. Every service can be invoked by standard protocols such as HTTP, so it can be used on a variety of platforms (e.g. mobile, web, desktop). The diversity of users of microservices-based software causes an increased variation in software requirements. In order to accommodate this variability, we propose a framework for microservices-based software based on the Software Product Line Engineering (SPLE) approach. We call this framework *ABS Microservices Framework*, as it relies on the Abstract Behavioral Specification (ABS) language development platform that readily supports SPLE. The framework created in this research has shown more flexibility to accommodate software variability than other microservices frameworks. Hence, the ABS Microservices Framework can support the software industry to distribute variable software of high quality and reliability.

Keywords: Microservices, Framework, Software Product Line Engineering, Abstract Behavioral Specification

1 Introduction

Microservices are an emerging architectural style of software development. This style changes the development paradigm to creating an application as a set of (small) services instead of a single unit [4]. Each unit of system functionality is built as an independent service that can be accessed by other multi-platform applications. In this architectural style, an application could even be written using different programming languages for each service, while the data representation will be standardized as per Representational State Transfer (REST) and encoded in the JSON format.

As software evolves over its lifetime, requirement changes are all but inevitable. In microservices architecture, the impact of such changes – the effort to adapt the system to the new requirements – can be mitigated, as each service is modular. Of course, if a service is implemented with the help of several modules, all these

might be subject to implementation changes. Requirement changes could also directly affect a whole range of services and necessitate complex implementation changes as a consequence. Thus, we argue that a more structured mechanism to manage requirement changes in microservices architecture is required.

The Software Product Line Engineering (SPLE) paradigm allows the development of multiple software products in a single development cycle by defining commonalities and differences between products [11]. To help do this systematically, SPLE uses *features* and expresses products as compositions of features. Delta Oriented Programming (DOP) [14] is a software design methodology aligned with the SPLE paradigm, where the features are implemented using *delta modules* (deltas). Deltas define the modification of a *core* module, which describes the system commonality. The Abstract Behavioral Specification (ABS) language [7] is an executable modeling and programming language that supports SPLE by implementing the DOP approach.

In this paper, we propose a new framework to build microservices-based software with the Software Product Line Engineering approach. We aim to minimize the effort in accommodating requirement changes by using the rigorous approach to variability management provided by SPLE. By using this approach, our framework offers *flexibility* to accommodate changes combined with the confidence that changes are always applied consistently, as specified by the variability model.

The paper is structured as follows: Section 2 provides background information on microservice architecture, software product line engineering, and the Abstract Behavioral Specification language; Section 3 describes the design of the framework; Section 4 explains how the framework works and details its implementation. We conduct analysis by comparing our framework with other microservice framework (Spring Boot) in Section 5. Section 6 discusses related work. Conclusions and some possible future work are presented in Section 7.

2 Background

2.1 Microservices Architecture

The microservices architecture was first discussed in 2011 at a workshop on software architecture in Venice, Italy [4]. The following year at the same event, a set of microservices terminology was defined. This was followed by a presentation by James Lewis on "Micro Services - Java, the Unix Way" at 33rd Degree in Krakow in 2013. In recent years microservices architecture has gradually become more popular, which is apparent from the number of forums and special conferences on this topic.

Being relatively new, there is no standard definition of microservices architecture. Martin Fowler defines microservices architecture as an approach for developing single applications into small services (microservice), where each service runs independently in different processes and can be called through simple communication mechanisms, such as the HTTP protocol [4]. According to microservices.io, microservice architecture is divided in four components [12]:

1. Presentation components, that is responsible for handling HTTP requests and responding with either HTML or JSON/XML (for web services APIs),
2. Business logic, that consist of application functionalities,
3. Database access logic, that is responsible to access persistent data,
4. Application integration logic, that is responsible for exchanging messages with other systems.

2.2 Software Product Line Engineering

Pohl et al. [11] defines Software Product Line Engineering (SPLE) as a paradigm for developing software applications using platform and mass customization. In SPLE context, platform is a common structure of software used to produce derived products efficiently. Mass customization is achieved through variability management, that is, the precise modelling of similarities and differences between applications, which enables the automatic derivation of customized software products.

The main concept of SPLE is to capture commonality and variability into software features [10]. Commonality is a property or features that can be shared and used by all applications (software) in a product line. Variability is the difference of features used by all or several applications in a product line.

The development process of SPLE is divided into two processes, domain engineering and the application engineering [11]. Domain engineering is a process to define or model commonality and variability in the product line, while application engineering is the process of building concrete products based on the model constructed in the domain engineering process.

2.3 Abstract Behavioral Specification (ABS)

ABS is a formal modeling language, executable, object oriented, concurrent and can model software with a high level of variability, such as software product lines [6]. ABS was developed by a European consortium since 2008 in a project called Highly Adaptable and Trustworthy Software (HATS). There are five language layers in the ABS model that together support SPLE [5]:

- Core ABS: provides functional and object-oriented programming constructs, used to specify the "core" product with functionality common to all products of the product line.
- Feature modeling: defines the SPL's features and dependencies among them.
- Delta modelling: defines the implementation of features. Delta Model modifies core model by adding, removing, or modifying classes, interfaces, attributes, or methods.
- Product line configuration: defines the connection between Delta Model and Feature Model. One delta can be used by many features and one feature can be implemented by many deltas. The configuration thus defines which deltas are applied for specific feature combinations.
- Product selection: defines products as sets of features and naming them for convenient reference.

3 Methodology

This section describes the methodology used in the paper, explaining how we design and construct our framework.

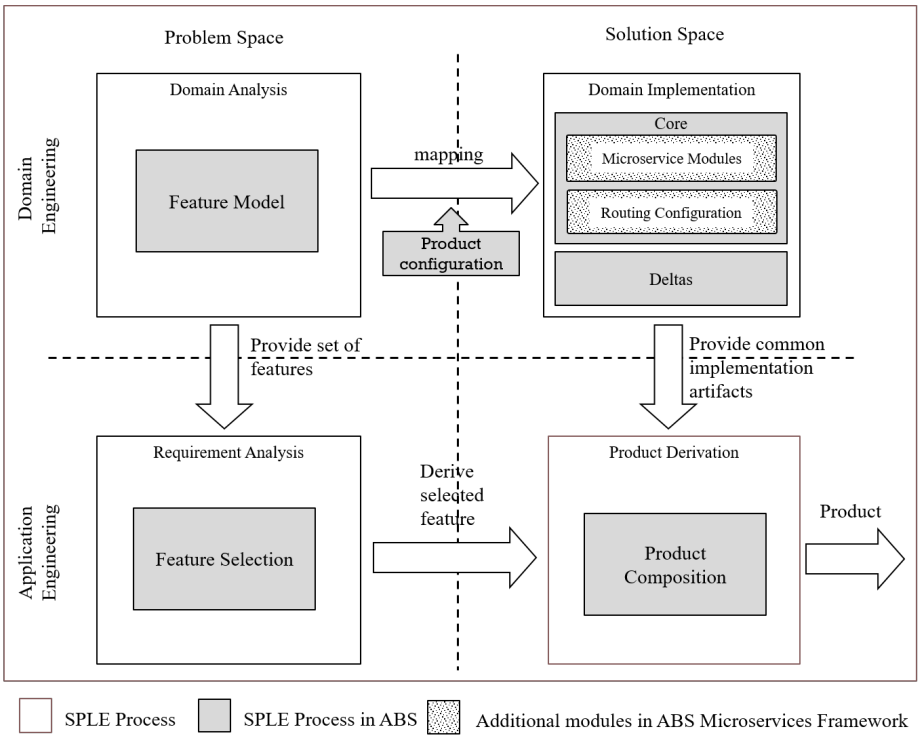

Fig. 1. SPLE Process in ABS Microservices Framework

Figure 1 shows an overview of the SPLE process in ABS Microservices Framework which adapted from [1]. Czarnecki and Eisenecker [3] distinguish between *problem space* and *solution space*. The problem space is the perspective of users. Problem space comprises domain-specific abstractions that describe the requirements on a software system and its intended behavior [8]. The solution space is the developers' perspective. Solution space comprises implementation-oriented abstractions, such as code artifacts [8].

According to [1], there are two engineering process, *Domain Engineering* and *Application Engineering*. Domain Engineering is a process to analyze potential requirement of application (commonalities and variants) and build reusable artifacts based that analysis. In this process, there two sub-process, *domain analysis* and *domain implementation*. Domain analysis takes place in problem space in domain engineering. Domain analysis is process to analyze scope and

all possibility requirement of domain, the output from this process is set of features. In ABS, this process in called feature modeling, and it implemented in *feature model*. Domain implementation is solution space of domain engineering. Domain implementation is process to implement reusable artifact based on domain analysis, the output of this process is source codes. In ABS, this process is implemented by *core* and *deltas*. Core is basic product that contains commonalities implementation of product. Deltas is set of delta modules that implements variation of product. Deltas modules is used to implement features in feature model. Correlation between delta modules and features is coordinated by *product configuration*.

Application Engineering is process to compose and reuse artifacts from *Domain Engineering* which divided into two sub-process, *requirement analysis* and *product derivation*. Requirement analysis is process to get requirement of specific user. In this process, we select appropriate features from domain analysis based on users' need. In ABS, this process is called feature selection or product selection. Product derivation is process to compose the product which selected features from *feature selection* process are verified. If the selected features is valid (follow the rules in feature model), then list of appropriate artifacts will be composed to generate microservice-based product according to product configuration.

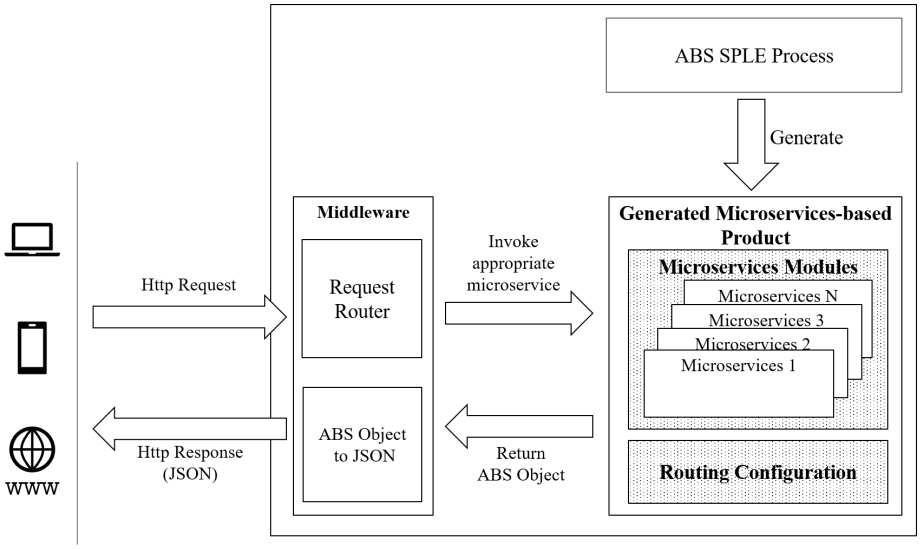

Fig. 2. Design of ABS Microservices Framework

In this paper, we use the SPLE process above to create a microservice-based product by using ABS. All those process are part of our framework, which we call *ABS Microservices Framework*. In here, we add *microservices modules* and their *routing configuration* in core. A microservices module does not merely

represent a feature. In ABS, a feature is implemented by one or more delta modules. Therefore, in this framework we could implement a microservice by using delta modules.

A microservices module consist of layers i.e. resource layer, service layer, domain layer, repository layer and persistence layer. The resource layer handles the incoming request and messages to objects representing the domain. Service layer contains logic implementation and coordinates across multiple domain activities. Domain layer represents model or entity that related with microservice. Repository layer provides a collection of operation to access persistent data through ORM. Persistence layer maps persistent data to domain. This structure is adapted from [2]. Figure 3 shows the structure and the connectivity between layers.

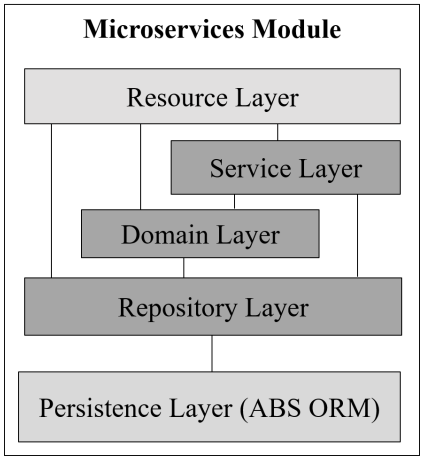

Fig. 3. The anatomy of microservices module in ABS Microservices Framework

Figure 2 shows the design of the ABS Microservices Framework that consists of three parts: ABS SPLE process, the generated microservice-based product and middleware. The ABS SPLE process is modeling part of the system using ABS modeling language as well as illustrated in Figure 1. Furthermore, the generated microservice-based product is a product that produced from the ABS SPLE Process. The product contains the selected microservices module and its routing configuration. The middleware is a module that acts as an interface for the microservice modules so it can be accessed from the external system using the HTTP protocol. There are two sub-modules in the middleware, the *Request Router* and *ABS Object to JSON*. The request router receives *http request* and maps the incoming requests to invoke the appropriate microservices, based on the routing configuration. Each microservices will return data as ABS object type. The data will be transform into JSON format by ABS object to JSON module.

4 Implementation

This section explains how to generate microservices-based products using the ABS Microservices Framework step by step, as outlined in Figure 1. We use a simple bank account example as toy case study. The simple bank account has three features, `Type`, `Check` and `Save`. For this case study, we generate two microservices-based products, *CheckingAccount* and *SavingAccount*. *CheckingAccount* is a product that has the `Type` and `Check` features. *SavingAccount* is a product with the `Type` and `Save` features.

4.1 Feature Model

First we need to define a feature model for our product line. A feature model in ABS is essentially a textual representation of a feature diagram. Figure 4 shows the feature diagram of the simple bank account SPL. It has a `Type` feature that has two child features `Check` and `Save`.

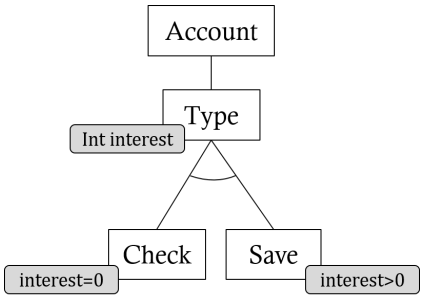

Fig. 4. Simple Bank AccountFeature Diagram

If the `Check` feature is selected, the value of the `interest` attribute must be 0, whereas with the `Save` feature it must be greater than 0. The arc between `Check` and `Save` means that we only allow to choose one feature; in other words, the account can be either a checking or savings account. The corresponding ABS feature model of simple bank account is shown in the code below.

```
root Account {
  group allof {
    Type {
      Int interest;
      group oneof {
        Check { ifin: Type.interest == 0; },
        Save { ifin: Type.interest > 0; }
      }
    }
  }
}
```

4.2 Microservices Module

Microservices module is a module that contains the implementation of microservices. A microservices module implements one or more microservices. For example, in our bank account, we have `Account` module that contains two microservices i.e. `withdraw` and `deposit`. `Account` consists of `MAccountResource` as resource layer, `MAccountService` as service layer, `MAccountModel` as domain layer, `MAccountDbImpl` and persistence layer is implemented using a library called ABS ORM. In this paper, we just focus on the service layer.

The service layer implementation of the `Account` module provides two microservices, "withdraw" and "deposit". The implementation of "withdraw" is illustrated below. It performs a query on the database based on the account ID to get the account model. Then it calls the withdraw method on the account model and saves it back to the database. The deposit service follows the same pattern to perform the deposit operation.

```
class AccountServiceImpl implements AccountService {
  Account withdraw(String id, Int amount) {
    AccountDb orm = new local AccountDbImpl();
    String qry = "id=" + id;
    Account a = orm.findByAttributes("MAccountModel.AccountImpl_c", qry);
    a.withdraw(amount);
    orm.update(a);
    return a;
  }
}
```

4.3 Delta Modules

In the next step we create a delta module for each feature defined in the feature model; we name these deltas `DType`, `DSave` and `DCheck`. Below we show the implementation of delta `DSave`. It modifies class `AccountImpl` by removing its `interest` variable and re-adding it, initializing it with the value provided by the delta parameter `i`.

```
delta DSave (Int i);
  uses MAccountModel;
  modifies class AccountImpl {
    removes Int interest;
    adds Int interest = i;
}
```

4.4 Routing Configuration

Routing configuration is used to configure URL for each microservice. The routing configuration is defined by the following format:

```
"<URL>" => "<resource_class_name>@<method_name>"
```

URL defines address called by the users if they want to invoke a microservice. Resource class name is the name of class that has responsibility to handle request of microservice. Then, method name is the name of method that serves the microservice. The routing configuration of Simple Bank Account is shown in the following code. For example, `/account/withdraw.abs` is the URL to access withdraw microservices.

```
module ABS.Framework.Route;
class RouteConfigImpl implements RouteConfig {
  String route(String url) {
    String result = case url {
       "/account/withdraw.abs"
               => "MAccountResource.AccountResourceImpl@withdraw";
       "/account/deposit.abs"
               => "MAccountResource.AccountResourceImpl@deposit";
    }
    return result;
  }
}
```

4.5 Product Configuration

In this step we define the product configuration of the simple bank account. It is used to define the relationship between feature and delta modules. For example, delta module `DType` will be applied to `Type` feature, if it is selected.

```
productline Accounts;
  features Fee, Overdraft, Check, Save, Type;
  delta DType(Type.interest) when Type;
  delta DSave(Type.interest) after DType when Save;
  delta DCheck after DType when Check;
```

4.6 Product Selection

The next step is to define all products to be generated. In this case, we will generate two products i.e. **CheckingAccount** and **SavingAccount**. CheckingAccount is a product that have `Type` and `Check` features. SavingAccount is a product with `Type` and `Save` features.

```
product CheckingAccount (Type{interest=0},Check);
product SavingAccount (Type{interest=1},Save);
```

4.7 Generate and Run Product

If all the steps has been completed, next we generate the product by using the command `ant -Dabsproduct=<product_name> abs.deploy`. The prod-

uct name is the name of product, that has been defined in product selection before. For example, if we want to generate CheckingAccount, we type `ant -Dabsproduct=CheckingAccount abs.deploy`.

After product generation has been successful, the next step is running the product by using `java -jar absserver.jar` command. By default, products run on `localhost` on port 8081.

5 Analysis

For analysis, we compared the requirement changes handling between this framework and other framework. For comparison we used Spring Boot Framework - the most popular JAVA microservices framework-. We modified the requirement of microservices-based application that developed before by adding two features i.e. `Overdraft` and `Fee`. `Overdraft` is a feature that allowed Bank Account to withdraw money more than its balance and `Fee` is a feature to add fee to Bank Account every deposit. In this case, we generate three products:

1. AccountWithFee : product with feature `Type`, `Check` and `Fee`.
2. AccountWithOverdraft : product with feature `Type`, `Check` and `Overdraft`.
3. AccountWithFeeAndOverdraft : product with feature `Type`, `Check`, `Fee` and `Overdraft`.

After that, we applied the requirement changes to each frameworks. First, we simulate it in Spring Boot. Then, we simulate it in ABS Microservices Framework. We will compare how many changes that is done to overcome the requirement changes.

5.1 Simulation in Spring Boot

AccountWithFee Product - We create `FeeAccount` class that extends `Account` class. `FeeAccount` class override `deposit` method of `Account`. And then, we can see that we have to add its associate class in each layer (See Figure 5a).

```
package com.rse.domain;
...
@Entity
public class FeeAccount extends Account {
  ...
  public int deposit(int x){
    Int result = x;
    if (x>=fee) { result = super(x-fee)};
    this.balance = result;
    return this.balance;
 }
}
```

AccountWithOverdraft Product - We create `OverdraftAccount` class that extends `Account` class and override `withdraw` method of `Account` by removing the mechanism of checking balance and the amount of withdrawal. And then, we add its associate class in each layer ((See Figure 5b).

```
package com.rse.domain;
...
@Entity
public class OverdraftAccount extends Account {
  ...
  public int withdraw(int x){
    this.balance = this.balance - y;
    return this.balance;
  }
}
```

AccountWithFeeAndOverdraft Product - We do same as before. we create `OverdraftFeeAccount` class that extends `Account` class and override `withdraw` and `deposit` method of `Account`. And then, we add its associate class in each layer (See Figure 5c).

```
package com.rse.domain;
...
@Entity
public class OverdraftAccount extends Account {
  ...
  public int withdraw(int x){
    this.balance = this.balance - y;
    return this.balance;
  }
```

```
 public int deposit(int x){
   Int result = x;
   if (x>=fee) { result = super(x-fee)};
   this.balance = result;
   return this.balance;
  }
}
```

5.2 Simulation in ABS Microservices Framework

Before we start to build product, we have to create delta module for `Overdraft` and `Fee`. We create delta module `DOverdraft` for `Overdraft` and delta module `DFee` for `Fee`. After that, we have to update our product configuration and add three products above to product selection.

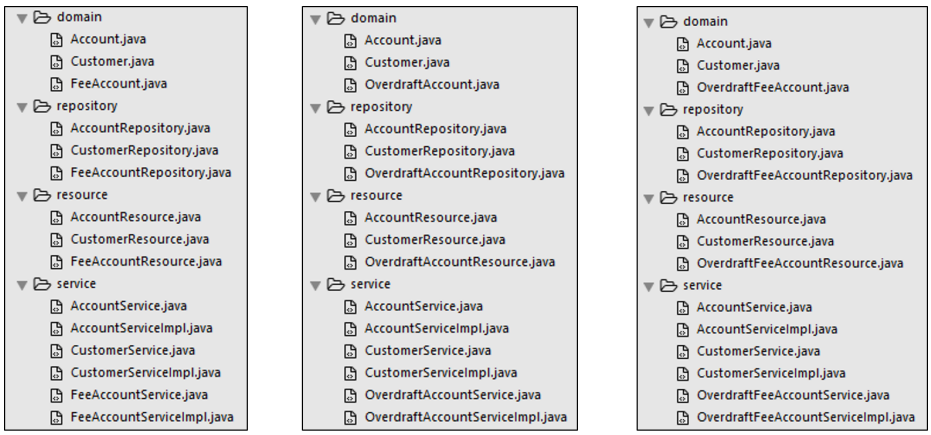

Fig. 5. Subclass for each product in every layer Spring Boot. (a) AccountWithFee product, (b) AccountWithOverdraft product and (c) AccountWithFeeAndOverdraft product.

```
delta DOverdraft;
uses MAccountModel;
modifies class AccountImpl {
  modifies Int withdraw(Int y) {
    balance = balance - y;
    return balance;
  }
}
```

The following snippet code is implementation of delta module `DOverdraft`, which modify `withdraw` method of class `AccountImpl` by removing balance checker.

```
delta DFee (Int fee);
uses MAccountModel;
modifies class AccountImpl {
  modifies Int deposit(Int x) {
    Int result = x;
    if (x>=fee) result = original(x-fee);
    return result;
  }
}
```

The implementation of delta module `DFee` is shown on the following snippet code. This delta modify `withdraw` method of class `AccountImpl` by adding `fee` parameter. `original` method means its method will call `deposit` method with original implementation.

```
productline Accounts;
  features Fee, Overdraft, Check, Save, Type;
  delta DType(Type.interest) when Type;
  delta DFee(Fee.amount) when Fee;
  delta DOverdraft after DCheck when Overdraft;
  delta DSave(Type.interest) after DType when Save;
  delta DCheck after DType when Check;
```

```
product CheckingAccount (Type{interest=0},Check);
product SavingAccount (Type{interest=1},Check);
product AccountWithFee (Type{interest=0},Check,Fee{amount=1});
product AccountWithOverdraft (Type{interest=0},Check,Overdraft);
product AccountWithFeeAndOverdraft (Type{interest=1},Save,Fee,Overdraft);
```

Finally, we generate (build) the products by executing the following commands on the console:

```
ant -Dabsproduct=AccountWithFee abs.deploy
ant -Dabsproduct=AccountWithOverdraft abs.deploy
ant -Dabsproduct=AccountWith FeeAndOverdraft abs.deploy
```

From both simulation above we found that in the Spring Boot, we have to do same effort for each product. We have to add new class in every layer. Otherwise, in ABS Microservices Framework, we can get three products on a single way. We just need to add delta module for each features, configure it, select it, and the last one generate the product. That simulation shows that in our framework, we need less effort to overcome the requirement changes than Spring Boot. In Sprint Boot, if there are requirement changes we have to modify each layer while in our framework only modify one layer (see Figure 6).

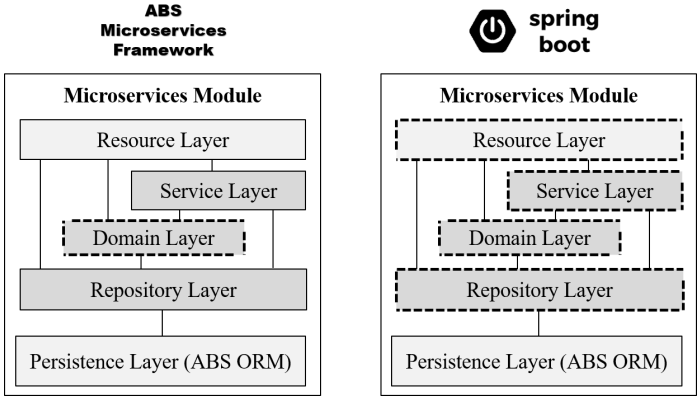

Fig. 6. The dash borderline box is modules that changed for handling requirement changes

6 Related Work

Our research is not the first effort to provide more structured support for microservices development. Safina et al. [13] extend Jolie, a programming language for the microservices paradigm, with a type system support for choices. While our framework is aimed more at managing evolution of microservices, by employing a programming language that provides broad support for variability, it could be also used to design a data-driven workflow. CIDE [9] is another microservices development environment that innovates in the domain of programming languages with an multi-agent-oriented programming style.

7 Conclusion and Future Work

The ABS Microservices Framework proposed in this research is a novel paradigm in microservices-based applications development; it's essential underlying idea is to structure related microservices as a software product line. It has been designed to be used in a similar way to other microservices frameworks such as Spring Boot, with the added benefit of more flexibility to handle requirements changes.

Future work will focus on finalising the framework design. For instance, the connection between features and microservices has not yet been fully explained. Furthermore, we need to add additional modules such as load balancer and security mechanism. This framework only support HTTP methods GET and POST, we have to add other methods, such as PUT and DELETE. In addition to JSON format data, we also can provide other data representations such as XML. Moreover, this framework have to be able to support multi-database and also integrate with software testing tools.

8 Acknowledgements

This work was supported by Reliable Software Engineering (RSE) Laboratory, Faculty of Computer Science, Universitas Indonesia and funded by Universitas Indonesia under PITTA Grant number 395/UN2.R3.1/HKP.05.00/2017.

Radu's contribution was supported by *Landesoffensive für wissenschaftliche Exzellenz* (LOEWE; initiative to increase research excellence in the state of Hessen, Germany) as part of the LOEWE Schwerpunkt CompuGene.

We thank the anonymous reviewers for their constructive comments, which helped us to improve the manuscript.

References

1. S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer, 2016.
2. T. Clemson. Testing strategies in a microservice architecture., 2014.
3. K. Czarnecki, U. W. Eisenecker, and K. Czarnecki. *Generative programming: methods, tools, and applications*, volume 16. Addison Wesley Reading, 2000.

4. M. Fowler and J. Lewis. Microservices. *ThoughtWorks. http://martinfowler.com/articles/microservices. html [last accessed on February 17, 2015]*, 2014.
5. R. Hähnle. The abstract behavioral specification language: A tutorial introduction. In *Formal Methods for Components and Objects*, pages 1–37. Springer, 2013.
6. R. Hähnle, M. Helvensteijn, E. B. Johnsen, M. Lienhardt, D. Sangiorgi, I. Schaefer, and P. Y. Wong. Hats abstract behavioral specification: The architectural view. In *International Symposium on Formal Methods for Components and Objects*, pages 109–132. Springer, 2011.
7. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th Intl. Symp. on Formal Methods for Components and Objects FMCO)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011.
8. C. Kästner and S. Apel. Feature-oriented software development. In *Generative and Transformational Techniques in Software Engineering IV*, pages 346–382. Springer, 2013.
9. D. Liu, H. Zhu, C. Xu, I. Bayley, D. Lightfoot, M. Green, and P. Marshall. CIDE: An integrated development environment for microservices. In *2016 IEEE International Conference on Services Computing (SCC)*, pages 808–812, 2016.
10. A. Metzger and K. Pohl. Software product lline engineering and variability management: Achievements and challenges. In *Proceedings of the on Future of Software Engineering*, pages 70–84. ACM, 2014.
11. K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Springer, 2005.
12. C. Richardson. Pattern: Microservices architecture. *Microservices. io. http://microservices. io/patterns/microservices. html [last accessed on February 17, 2015]*, 2014.
13. L. Safina, M. Mazzara, F. Montesi, and V. Rivera. Data-driven workflows for microservices: Genericity in Jolie. In *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, pages 430–437, 2016.
14. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proc. 14th Intl. Conf. on Software Product Lines: Going Beyond (SPLC)*, pages 77–91. Springer, 2010.