ABS Microservices and Ontology-Zotonic Integration for SPL Implementation in Information System

Andri Kurniawan
Faculty of Computer Science
Universitas Indonesia
West Java, Indonesia
Email: andri.kurniawan31@ui.ac.id

Iis Afriyanti
Faculty of Computer Science
Universitas Indonesia
West Java, Indonesia
Email: iisafriyanti@ui.ac.id

Ade Azurat
Faculty of Computer Science
Universitas Indonesia
West Java, Indonesia
Email: ade@cs.ui.ac.id

***Abstract*—Software Product Line (SPL) promises to accelerate the development process with higher quality of product and low budget of production. The essential key of SPL is feature diagram which describes the relation between features for the domain and captures the commonalities and variabilities. The approach gains numerous attention in research and industry area. However, the implementation of SPL approach for information system development is still inadequate. Some works explain the inclusion of ontology for SPL. On the other side, the executable modelling language such as Abstract Behavioural Specification (ABS) can be utilized to implement the feature diagram for distributed object-oriented systems. The trend of ABS expands to accommodate microservices-based software variabilities, that is ABS Microservices Framework. We propose an adaptor to integrate the ABS microservices and ontology-based information system to produce automated business logics into the system. We show that by using the ontology as its basis, the system is semantically structured and the business logics required by the system is updated automatically.**

I. Introduction

Software Product Line (SPL) gains a huge attention nowadays since it claims that using the paradigm can reduce the cost production, enhance the quality of product, and shorten time of distribution [1]. The fundamental key of SPL paradigm is feature model to capture the variabilities and commonalities of different product requirements for different stakeholders; some of previous product components can be used for other requirements. This paradigm has successfully adopted in industry area that several success stories can be found in Product Line Hall of Fame[1]. However, to the best of our knowledge, the automated SPL implementation in information system to have the system more adaptable is still inadequate. The main reason is that the feature model as a key part in SPL paradigm can not be executed as it is a model-oriented. Therefore, to realise automated SPL from requirement gathering to deployment phase, we need an executable model such as Abstract Behavioral Specification (ABS).

ABS becomes more promising to solve the problem in the SPL. It is a formal modelling language and executable for distributed object-oriented systems [2] [3]. The study of [4] is to create a framework to accommodate requirement changes in SPL-based system by realizing ABS Microservices [4]. The feature model is implemented into ABS Microservices and results a web service including the required business logic. The web service supports software development that meets the domain of feature model.

In addition, the previous study [5] presents the use of OWL ontology as a feature model representation to produce an application as part of SPL realization. By using such mechanism, it is possible to run inconsistency checking upon the feature model [6]. The feature model is mapped into *classes* and *object properties* of OWL ontology and can be transformed into Zotonic-based information system; a CMS which adopt pragmatic semantic web for its data model. The approach has already been adopted to automate a charity organization information system in [7]. However, the business logic for the system is still manually developed.

Since the web service produced by ABS Microservices provides business logic for the domain related with the feature model and can be retrieved via its web service, we can build an adaptor to integrate the web service resulted by ABS Microservices and the Zotonic-based system with ontology inside. Therefore, in this paper we attempt to develop the adaptor, in such a way the information system is semantically structured and the business logic is automatically developed and meets the requirement .

The remainder of this paper is structured as follows. Section II describes some studies related with this paper. Section III explains the adaptor along with its implementation process. We also provide a case study of the adaptor for a charity organization in Section IV. The last section concludes our work and explain the benefits of the adaptor.

II. Related Works

SPL becomes a promising paradigm in software development. In recent years, there has been an increasing amount of literature on SPL realisation. Some attempts utilize Semantic Web in SPL process to gain more benefits of the paradigm.

[1] http://splc.net/fame.html

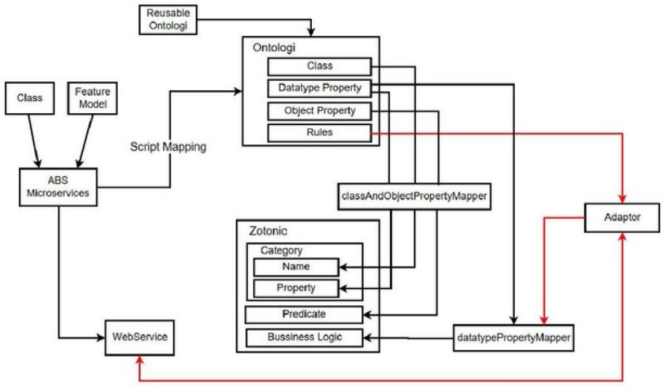

Fig. 1. Architectural Design SPL-based Information System

In [5] points out that ontology, as a main pillar of Semantic Web, accommodates application realisation production in SPL process with the help of Zotonic; a CMS with pragmatic semantic web for its data model. The study argues that within that it is possible to provide open and reasuable linked data of features.

The approach of study [5] is to translate every feature in the feature model into OWL classes. In addition, the relation between features is converted into OWL object property. The translation is implemented in Zotonic with BSMI charity organization as a case study (see Fig. 1). Each OWL class and object property are subsequently mapped into *Category* and *Predicate* of Zotonic system. The translation from OWL class and object properties runs automatically and results an information system. However, the system requires some business logic to be implemented, such as the total donation for charity organization system. In the previous study [7] the business logic of the system is implemented manually in Javascript. Although, business logic on one charity organizational with another may vary.

On the other hand, the development of executable modelling language such as ABS [2] also benefits the advancement of SPL. It provides a mechanism to implement the feature diagram into executable codes. A recent study by Afifun [4] develops a microservices for ABS modelling language to support requirement changes and product variabilities in a product line. The microservices can also provide some business logic for the domain described in the feature model.

Thus, we attempt to implement an adaptor to integrate the ABS Microservices and Zotonic system that already included ontology of feature diagram as shown in Fig. 1. We employ Zotonic and the ontology as a representation of a feature model is mapped into Zotonic data model. Moreover, the adaptor employs the rules and the web service to provide business logic for the information system. By using the adaptor, the business logic required by the system will be retrieved from the web service, for example to calculate the total of donation for a certain charity program.

III. Adaptor for ABS Microservices and Ontology-Zotonic Integration

The adaptor is specifically built to call the web service produced by ABS Microservices. However, the adaptor needs a rules table which contains a list of methods along its endpoints in JSON format (see Fig 2). When accessing a page that has business logic, the template engine generated by Zotonic which contains business logics will triggers the *m_abs* as illustrated in Fig. 2. The m_abs reads the rules table to identify the endpoint and total parameter in order to check whether the total parameter received is equalt to the web service requirement. Afterwards, the m_abs will send the data to the web service to be processed. The full source code of this adaptor implementation can be found in https://gitlab.com/andrikurniawan/adaptor-ontology-to-webservices.

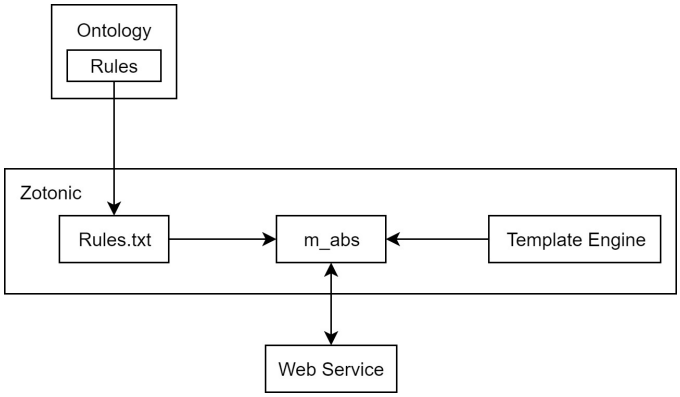

Fig. 2. Architecture Design of Adaptor

A. Adaptor Interface

The adaptor can be called by two options: *template engine* and a *model*.

1) Calling adaptor over template engine: For this reason, we developed a new model inserted in Zotonic installation folder, we named it as m_abs. Since it is a new model for Zotonic, we need to export the required built in models that is `m_find_value`, `m_to_list` and `m_value`. We did not define `m_to_list` and `m_value` since they will not be called when the adaptor is running.

However, we specify `m_find_value` to do pattern matching (see Listing 1). When the template engine runs, it automatically generates some commands structured as `m.abs.functionName[query param=value]` and triggers the adaptor to run. For example `m.abs.totalDonation[query id=id]` means that it calls function `m_abs` in the adaptor for *totalDonation* method or *key* and `query id=id` is the data which will be delivered to the web service. Both *key* and *data* are inputs for the parameters in `m_find_value` function.

Listing 1
Implementation of m_find_value function

```
m_find_value({query, Query}, #m{value=Q} =
 _, _Context) when is_list(Q) ->
```

```
 [Key] = Q,
 [Url, Param] = lookup_rules(Key),

 case validate_params(Param, Query) of

  false ->
   [{error, "Num of Params not same"}];

  true ->
   {DecodeJson} = fetch_data(binary_to_list
   (Url), jiffy:encode ({Query})),
   lager:info("ABS result : ~p", [DecodeJson]),
   proplists:get_value(<<"data">>, DecodeJson)

end;
```

2) Calling adaptor by a model: The adaptor is also can be called from other models, such as `m_src` model; one of Zotonic built in model. Regarding that requirement, we implemented a new function in the adaptor that is `call_api_controller` which can be accessed by other models. The function receives two paramaters: *key* and *data*. The *key* indicates the method name in the web service and Data is to pass data to the web service. For example, `m_abs:call_api_controller(deleteDonation, Json);` means that it calls function `call_api_controller` in the adaptor for *deleteDonation* key and `Json` is the data which will be delivered to the web service.

Listing 2
IMPLEMENTATION OF CALL_API_CONTROLLER

```
 case proplists:get_value(<<"status">>,
   DecodeJson) of
   200 ->
     proplists:
       get_value(<<"data">>, DecodeJson),
   201 ->
     Message = proplists:
       get_value(<<"message">>, DecodeJson),
     lager:info("[ABS] status 201 ~p",
       [binary_to_list(Message)]);
   400 ->
     Message = proplists:
       get_value(<<"message">>, DecodeJson),
     lager:error("[ABS] status 400 ~p",
       [Message]);
       _Other ->
     lager:error("[ABS] status undefined ~p",
       [_Other])
end
```

B. Adaptor Implementation

Aforementioned, the adaptor needs a table of rules. The rules is a JSON format and structured as in Listing 3. The rules table must be located in root folder of Zotonic.

Listing 3
STRUCTURE OF RULES

```
{
 function name: [endpoint, total of parameter]
}
```

We also implemented `lookup_rules` function to read the rules table then returns a list of endpoints along with its total of parameters with *key* as an input. The function calls `read_file` function which read the whole rules file. The adaptor also run `validate_params` function which returns a boolean. If *true* then the total parameter in rules table is same in the Query where query is data which will be delivered to the web service. If it returns true, the adaptor runs `fetch_data` to request data from the web service.

There are two parameters in `fetch_data` function: `Url` and `Query`. The `Url` is the endpoint of web service and `Query` is parameter in JSON format. However, the function needs `post_page_body` to directly interact with the web service (see Listing 4).The function returns data in JSON format and decode with `jiffy:decode/1`

Listing 4
IMPLEMENTATION OF POST_PAGE_BODY FUNCTION

```
post_page_body(Url, Body) ->
 case httpc:request(post, {Url, [],
    "application/json", Body},
    [], []) of
  {ok,{_, _, Response}} ->
    Response;
  Error ->
    {error, Error}
end.
```

```
{% javascript %}
  var predId = [];
  var total = 0;
{% endjavascript %}
{% with m.search.paged[{referrers id=id page=page}] as result %}
  {% for id, pred_id in result %}
    {% javascript %}
      if (predId.indexOf("{{ pred_id }}") == -1) {
        predId.push("{{ pred_id }}");
    {% endjavascript %}
    {% for amountholder in r.s[m.rsc[pred_id].title | lower] %}
      {% if amountholder.amount %}
        {% javascript %}
          total += {{ amountholder.amount }};
        {% endjavascript %}
      {% endif %}
    {% endfor %}
    {% javascript %}
      }
    {% endjavascript %}
  {% endfor %}
{% endwith %}
```

Fig. 3. Total Donation function in Javascript

IV. CASE STUDY: BSMI CHARITY ORGANIZATION

BSMI (Bulan Sabit Merah Indonesia) is a charity organization that works to help others in Indonesia. To make it easier for the community to participate in providing assistance through BSMI, BSMI needs a website to facilitate their operations. In this paper, we employ BSMI as a case study to demonstrate the adaptor and its usage.

BSMI has many features, but in this paper we focus on *program*, *donation*, *beneficiary* and *donor* features. Program

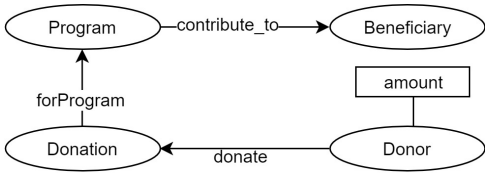

Fig. 4. Part of ontology translated from the feature diagram of charity organization [7]

feature provides charity activities undertaken by BSMI. Donation is a feature to store donations given by the communities for a charity program. Donor feature is to represent people who donate. Target feature are target of a program. The features have some business logic, for example, to calculate the total donation of a program. All of the features are represented in OWL Classes (see Fig. 4).

We added a table of rules to run the adaptor (see Listing 5). For example, *totalDonation* is a rule to calculate the total of donation comes to the BSMI website, the endpoint of web service is "http://sites.com/abs/program/totaldonation", and the total parameter is 1 is represented in Listing

Listing 5
EXAMPLE OF TABLE OF RULES

```
{
 "totalDonation" :
  [
   "http://sites.com/abs/program/totaldonation":
    1
  ]
}
```

When we open a page containing business logic, it will trigger the adaptor. In this example, when opening a page with the category of *Program*, the total donation of that program will be automatically updated as well as when updating the program page. Firstly, we need to create a page of Donation category and input the amount of donation and choose what program the donation is for. We also can create a page containing the donor of donation and who become the beneficiary of the donation. For example, Fig. 6 illustrates the program page of "Pelantikan dan Seminar BSMI Brebes" which the total donation is Rp5.030.000 from "Donasi Bpk. Anto untuk Pelantikan Brebes" and "Donasi Pusat untuk Pelantikan Brebes". The donation is distributed for "Dadang" and "Budi". The pages are the representation of instances of the category, or so-called OWL classes that already been translated.

V. CONCLUSION

In this paper we have presented an adaptor to integrate ABS Microservices and the ontology implemented in Zotonic-based information system. We explained the implementation process and show a case study in charity organization by adopting ontology from the previous study. The adaptor reads the *rules* related with the ontology as a mapping table to call the web service. The adaptor can be called by the *template engine* and other *models*.

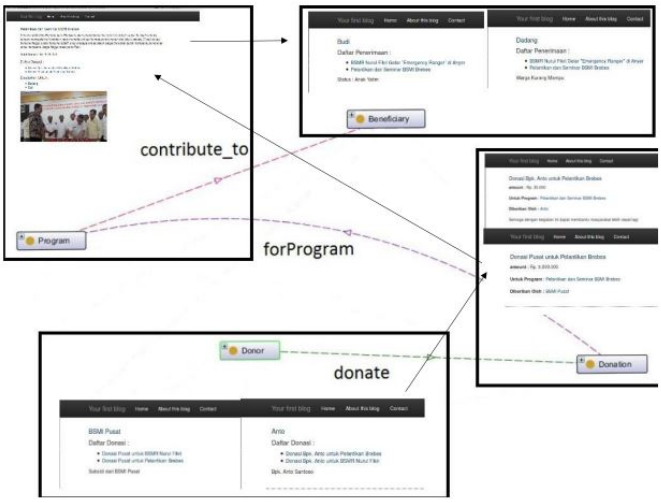

Fig. 5. An example of page "Pelantikan dan Seminar BSMI Brebes" program

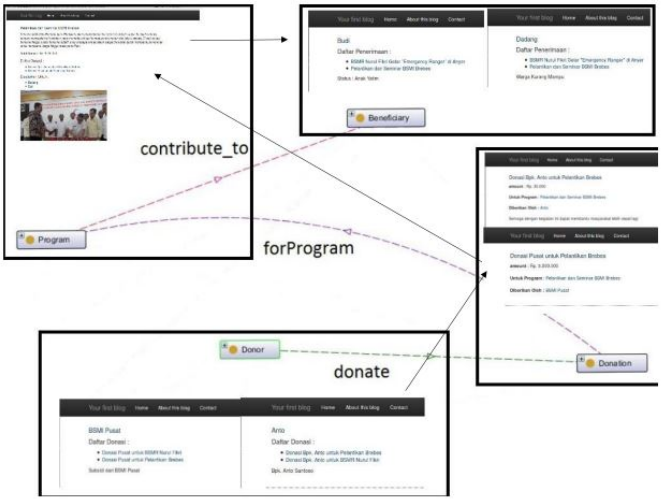

Fig. 6. Relationship between pages which represents the ontology of Charity Organization

By utilising this adaptor, the business logic required by the information system can be generated automatically without manual development. In addition, the adaptor facilitates dynamic business logic that meets the requirement of the system. Therefore, we can generate several well-structured web pages with different business logics. Furthermore, we can deliver resources of the system into the web service by calling the adaptor from the models. From this scenario, it became apparent that an external global database is possible to be produced since the resources from the information system can be transferred via the web service.

ACKNOWLEDGMENT

This work was fully supported by Reliable Software Engineering (RSE) Laboratory, Fasilkom UI and funded by Universitas Indonesia under Hibah PITTA no. 395/UN2.R3.1/HKP.05.00/2017

REFERENCES

[1] K. Pohl, G. Böckle, and F. Van Der Linden, *Software Product Line Engineering. Foundations, Principles, and Techniques*. Springer, 2005.

[2] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte and M. Steffen, "ABS: A Core Language for Abstract Behavioral Specification," in *Formal Methods for Components and Objects: 9th International Symposium*, Graz, Austria, November 29 - December 1, 2010. Springer Berlin Heidelberg. Berlin, Heidelberg, 2012. pp. 142-164.

[3] R. Hähnle. *The abstract behavioral specification language: A tutorial introduction*. in *Formal Methods for Components and Objects*. Springer, 2013. pp. 1-37.

[4] M. A. Naily, M. R. A. Setyautami, R. Muschevici, "A Framework for Modelling Variable Microservices using the Abstract Behavioral Specification Language", to appear in *Microservices: Science and Engineering Workshop co-located with The 15th International Conference on Software Engineering and Formal Methods*, Trento, Italy, September 4-8, 2017.

[5] I. Afriyanti, F.M. Falakh, A. Azurat and B. Takwa, "Feature Model-to-Ontology for SPL Application Realisation", submitted for publication *The 16th International Conference on Ontologies, DataBases, and Applications of Semantics*. Available: https://arxiv.org/abs/1707.02511

[6] H. H. Wang, Y. F. Li, J. Sun, H. Zhang, and J. Pan, *Verifying feature models using OWL*, Web Semantic, vol. 5, no. 2, pp. 117-129, 2007.

[7] B. T. Pangukir, *Pembentukan Otomatis Aplikasi Berbasis Web dengan Masukan Berupa Ontologi: Status Kasus Web BSMI*, Bachelor final report, Faculty of Computer Science, Universitas Indonesia, Depok, Indonesia, January 2017.