A Model-driven Approach to Flexible Multi-Level Customization of SaaS Applications

Zakwan Jaroucheh, Xiaodong Liu, Sally Smith
School of Computing
Edinburgh Napier University, UK
{z.jaroucheh, x.liu, s.smith}@napier.ac.uk

Abstract—**Recently, Software as a Service (SaaS) has become a popular software service mode. Context-awareness and customizability are important and desirable properties for providing the same application for different customers. Most of existing approaches tackle application customization by explicitly specifying some form of variation points where parts of the application remain unspecified or are defaulted and can be customized by each customer to suit its particular needs. This, however, leads to a mismatch between how the architect or developer logically views and interprets differences in a SaaS application family and the actual modeling constructs through which the logical differences must be expressed. Hence, in order to capture the variability in SaaS applications in a more logical and independent way we propose the concept of change fragment and change primitives. A novel approach to effective customization of SaaS at levels of control flow and component framework is proposed and evaluated.**

Keywords-context-awareness; SaaS; MDD; SaaS customization

I. INTRODUCTION

SaaS, a new delivery model for software, can be characterized as *software deployed as a hosted service and accessed over the Internet* [2]. Indeed, moving from traditional "on-premise" software that is deployed and run in a data center at the customer's premise, to offering software as a service has a set of advantages for software customers [5] and requires software vendors to shift their thinking in business model and application architecture.

Changing the business model involves shifting the ownership of the software from the customer to an external provider, reallocating the responsibility for the management of the underlying hardware and software infrastructure for that software from the customer to the provider, and reducing the cost of providing software services, through specialization and economy of scale. From an application architect's point of view, the service provider provides the same application for several different customers; and thus the software must be built following the model of a multi-tenant architecture [2]. However, each individual tenant or customer has different requirements for the same application logic. To achieve this, the customer will be provided by an *application template* [5] in which some parts of the application remain unspecified or are defaulted and can be customized by each customer to suit their particular needs.

On the other hand, context-awareness refers to the capability of an application or a service being aware of its physical environment or situation (e.g. context) and to respond proactively and intelligently based on this awareness [1]. We define the context-aware SaaS application customization as: *the modification of an application behavior as to make it suitable to the special and unique needs of the customer and her context considered relevant to that application*.

Many different solutions have been proposed by researchers to the problem of context-aware customization during SaaS application development and provision. However, two main issues could be identified in the existing approaches.

Firstly, SaaS application modeling must be flexible enough to deal with constant changes – both at the business level (e.g. evolving business rules) and at the technical level (e.g. contextual information). The flexibility could be provided or addressed by incorporating variabilities into a system [5]. Most of the approaches tackle SaaS application customization by explicitly specifying some form of variation points. These approaches (e.g. [4][5]) first identify the variation points in the SaaS application and its associated alternatives (variants) that specify different implementation of the system. Second, either at design time or at runtime, they specify variants selection mechanisms (based on ranking rules, preferences, etc). These approaches enjoy the inherent power of a software product line in dealing with variability, automation and consistency. However, the problem is that, for example, each task in the application is modeled as a variation point in and of itself, each governed by its own clause to determine inclusion or exclusion. This is in contradiction with how the developer or architect logically views the application variant i.e. in terms of the features that determine the difference between application variants in each usage context. Moreover, these approaches are not scalable enough because of the difficulty in variation management when the number of variation points increase.

Secondly, SaaS are built following service oriented architecture (SOA). An application is a composition of services that is orchestrated by workflows such as the standard Business Process Execution Language (BPEL). Thanks to SOA, SaaS applications enjoy the power of programming in the large (i.e. service orchestration) which is separated from the programming in the small (i.e. service implementation). Thus, a deep customization can occur when considering the two basic levels: that of the *individual service* and the *service*

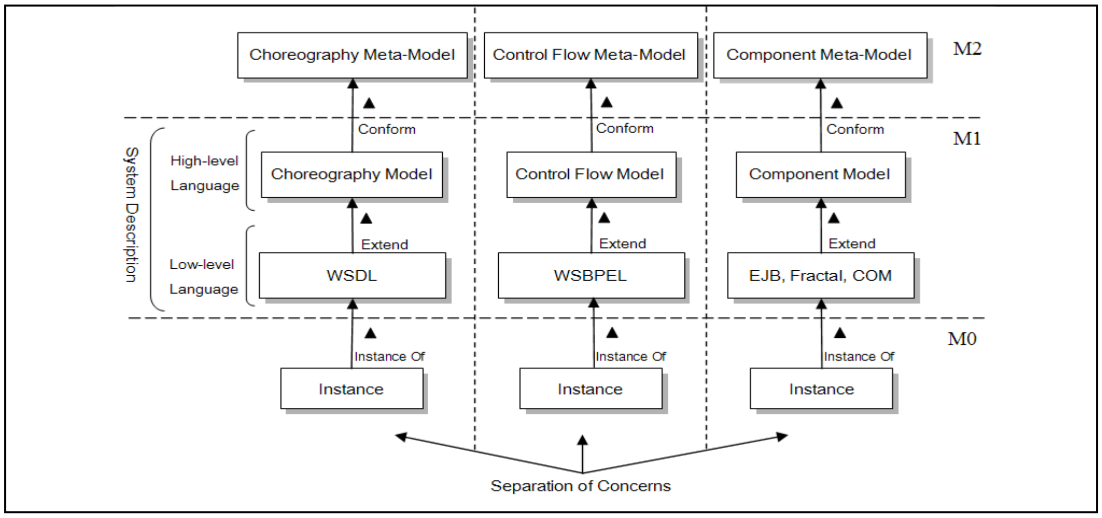

Figure 1. Leveled views of SaaS application

orchestration. The approach presented in this paper therefore could be applied to the layers of the SaaS application as long as this layer has been modeled as we will see later.

Motivated by these problems and directives in mind, we propose an MDD-based approach that introduces the notion of *change fragment* and *change primitive* to capture the variability in a more logical and independent form. The proposed approach contributes to a solution to automatically generating a customized SaaS application based on the relevant context.

The rest of the paper is structured as follows: Section II describes the rationale behind the proposed approach. Section III presents the proposed approach for the SaaS application customization. In Section IV, we describe how to instantiate a SaaS application. In section V we present the proof-of-concept prototype; and in section VI we illustrate the proposed approach by giving a simple example of an event advisor. The related work and concluding remarks end the paper.

II. The Rationale of The Proposed Approach

In the context of SaaS applications, the SaaS developer has to include not only business process in a process language (such as BPEL), but also business rules, polices, constraints, as well as customization mechanisms. Obviously, mixing process with business rules and customization issues weaken the modularity of the system. Typically, according the separation of concern principle, the SaaS application developer has to focus on the core application business logic and then define separately the customization and business rules, and weave them to the core application.

Therefore, modularization and separation of concerns are the driving principles of our approach to target the SaaS application customization. We propose a model-driven development (MDD) approach for developing such applications. MDD emphasizes using models to capture the application knowledge that are independently of any underlying computing infrastructure, e.g. middleware, programming languages operating systems etc; which will ease the reuse, adaptation, and evolution of applications.

As the number of services involved in a SaaS application grows, the complexity of developing and maintaining these applications also increases. One of the successful approaches to managing this complexity is to represent the application by different architectural views [8]. Examples of these views are orchestration view, control flow view, and component view (see Fig. 1). The idea is to give the developer the possibility of applying the necessary change fragments in each view and then the automated tool verifies the integrity of the changes and

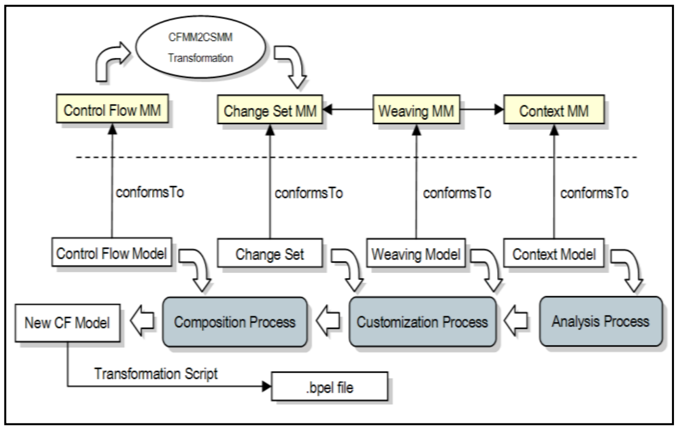

Figure 2. The proposed framework

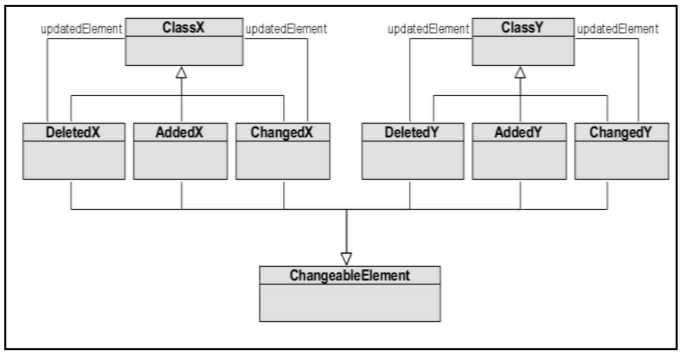

Figure 3. Deriving the change meta-model

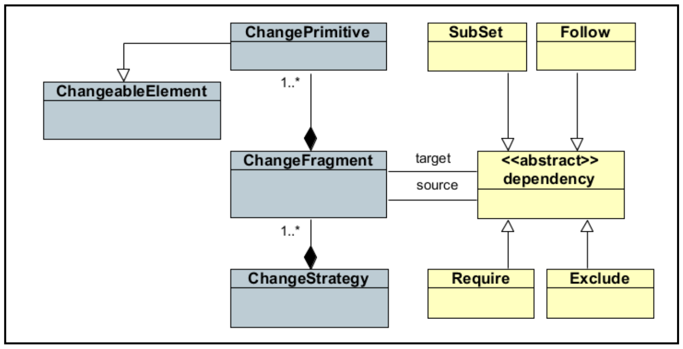

Figure 4. The change set meta-model

generates the customized application variant artifacts accordingly. This modelling respects the separation of concern principle so that we will have multiple views of the systems; each view will model a specific concern.

In this paper we focus on the control flow view; however the proposed approach could be extended to the other views. In addition, the approach aims to tackle context-aware customization without interfering with the core functionality of the application.

III. The Proposed Approach

Typically, the developer first focuses on the functional (business logic) aspect of the SaaS application which will yield a basic model of the system. Then, he defines the change set and the different possible context scenarios. "Weaving" a group of correspondent change fragments with the basic model will yield a new SaaS application instance.

The proposed conceptual model is structured in four main sections that address, respectively, the modeling of the control flow, context information, change set and the weaving model that links between change model and context model (Fig. 2).

During runtime, the customer and environmental context will be gathered when the SaaS application is invoked by the customer. The Context Analysis module evaluates all context constraints of the context model. Using the constraints elements evaluated to “true” and the weaving model we are able to determine the relevant change fragments (See Section IV) and the order in which they should be applied to the basic control flow model.

The customization process determines -according to the mapping between the change fragments and context elements- the set of change fragments to be applied to the application instance. The model composer combines the set of change fragments to the control flow model and verifies the integrity of the system. The result is a new control flow model which corresponds to the current context. All these operations are fulfilled in the model level. Thus, the resulting process model has to be translated to concrete artifacts e.g. BPEL. Now it is the role of the infrastructure to create a new instance corresponding to the new control flow model which satisfies the user requirements and context. This transformation from the model to the code is achieved using one of the model-to-text transformation tools.

```
create OUT : ChangeMM from IN1 : ControlFlowMM, IN2 :
MinimalChangeMM;
helper def: changeableElement: MinimalChangeMM!EClass =
MinimalChangeMM!EClass.allInstances()->select(i | i.name =
'ChangeableElement');

rule copyChangeEvolutionMM {
   from s : MinimalChangeMM!EClass
   to t: ChangeMM!EClass (
          name <- s.name,
          interface <- s.interface,
          eSuperTypes <- s.eSuperTypes,
          eStructuralFeatures <- Sequence {s.eStructuralFeatures}
          ...
   )
}
rule generateChangeMMElements {
   from s : ControlFlowMM!EClass (s.name <> 'Process' and not
s.abstract)
   to t: ChangeMM!EClass (
          name <- s.name,
          interface <- s.interface,
          eSuperTypes <- s.eSuperTypes,
          eStructuralFeatures <- Sequence {s.eStructuralFeatures}
          ...
    ),
      added_element: ChangeMM!EClass (
          name <- 'Added' + s.name,
          eSuperTypes <- Sequence {t, thisModule.changeableElement}
    ),
      changed_element: ChangeMM!EClass (
          name <- 'Changed' + s.name,
          eSuperTypes <- Sequence {t, thisModule.changeableElement}
    ),
      deleted_element: ChangeMM!EClass (
          name <- 'Deleted' + s.name,
          eSuperTypes <- thisModule.changeableElement
    )
}
```

Figure 5. Change metamodel generation script

The most interesting aspect of the proposed approach is that the relevant characteristics of the different concerns (context, change, application logic) will be abstracted in models and made observable to the application developers. Moreover, she will be able to do any type of change to the application model. In this way, developers will be able to manage customizability to a greater degree of flexibility.

A. *Context Model*

As in previous work [3], the main construct for representing context knowledge is the `ContextPrimitive` which represents the base context constructs (primitives): entity classes, entity attributes and entities associations. Also, we presented an approach for context-aware software development based on a flexible product line based context model which significantly enhances reusability of context information by providing context variability constructs to satisfy different application needs.

In addition, we introduce here the *context-dependent constraint* construct which allows us to specify conditions that must hold to introduce some kind of context-aware customization by specifying the change fragments (CFs) that should be applied to the basic model as described in the next sections. The Object Constraint Language OCL language is leveraged to express the constraint expression.

B. *Change Set Model*

Customizing SaaS applications usually involves adding, dropping and replacing tasks in the application. For instance, in component model this involves adding, dropping and replacing components. In this respect, and in order to achieve deep

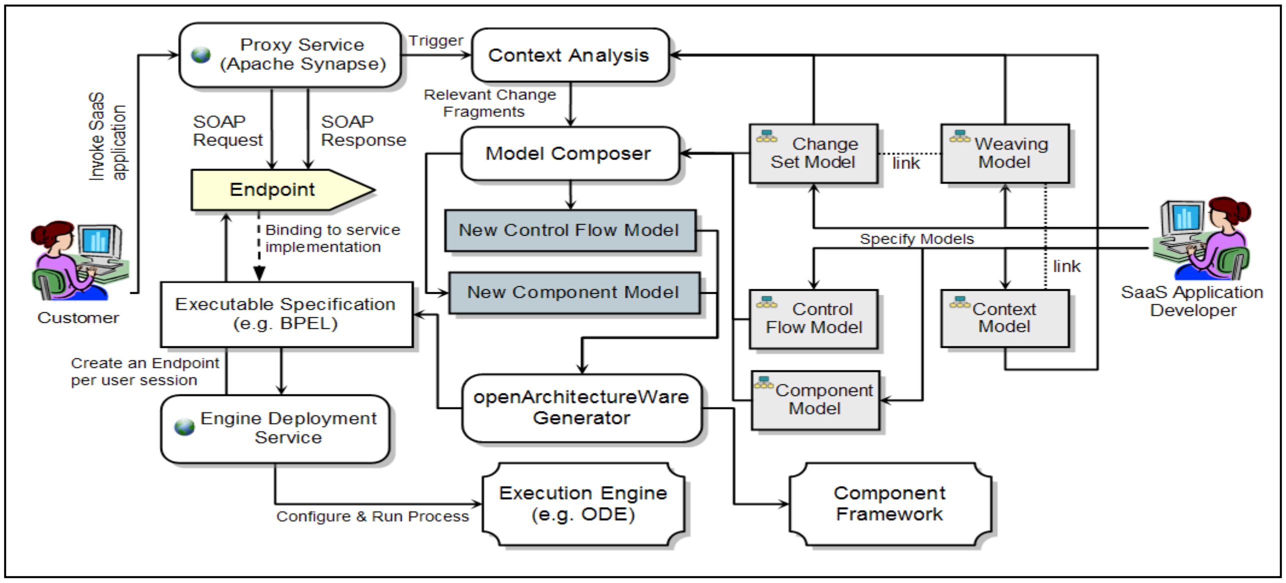

Figure 6. The prototype architecture

change ability, we propose to add for each component `X` three classes: `AddedX`, `DeletedX`, and `ChangedX` describing the difference between the basic component model and the respective variant model (Fig. 3). Other change types can be mapped to variations and combinations of these ones. For instance to achieve the plugin and plugout capability a combination of `DeletedX` and `AddedX` could be used. In the same manner, for each class `Y` in control flow metamodel we add three classes `AddedY`, `DeletedY`, and `ChangedY`. The change set metamodel (Fig. 4) consists of a `ChangeStrategy` class that contains one or more `ChangeFragments`. The `ChangeFragment` in turn consolidates related `ChangePrimitives` (a set of elements of type `ChangeableElement`) into a single conceptual variation. Our approach promotes CFs to be first-class entities consisting of closely-related additions, deletions and changes performed on the basic model. Dependencies are used to describe relations between CFs in order to constrain their use. The relations supported are as follows: dependency (*Require*), compatibility (*Exclude*), execution order constraint (*Follow*), and hierarchy (*SubSet*). Further, the CF concept is used to specify the application customization during runtime namely the customization strategy. But, what about the evolution of the customization strategy? This is the role of the change strategy concept. An example of strategy evolution is that the business owner may choose to apply a different customization strategy during the Christmas days and later to return to the basic strategy. To this end, the change strategy could also be linked to a specific context constraint.

The change set metamodel of each view could be automatically generated from the model of that view. One possible approach is to use the ATL transformation language[1] as in the script of Fig. 5.

On the other hand, in order to link the context model and change set model, and because in the MDD world everything should be a model, the mapping between the context constraints and the CFs will be represented by a weaving model. This mapping will be used as information for driving the model transformation.

IV. SAAS APPLICATION INSTANTIATION

The selection of a SaaS application variant in a particular context should be done automatically. Therefore the application context in which this selection takes place has to be considered. It is important to distinguish here between two type of changes: i) permanent change witch lead to change of the SaaS application specification (structure and behavior) due for instance to the business rules or application logic, and ii) instance-level change which affect only the current application instance. We generate the customized control flow model by applying a number of CFs and their related change primitives to the corresponding basic model as follows: i) Select the CFs whose context constraints associated with it evaluate to “true”, ii) Check CFs relations to ensure model consistency, iii) Apply the CFs to the basic model, and iv) Check for consistency to avoid any deadlock or data inconsistency in the resultant application variant. A consistency check is necessary and it is considered for our future work.

The proposed approach is flexible enough to accommodate the “permanent changes” that are due to changes of the regulation or the business rules by assigning them to a context constraint always evaluated to true. One of the advantages of this approach is that the change in the SaaS application specification can be easily documented.

V. PROTOTYPE ARCHITECTURE

We have developed a Java application for the SaaS application variant generation. The Eclipse Modeling Framework (EMF) was used to model the aforementioned models. In this prototype we consider both the control flow view model and component model of the SaaS application.

[1] ATL Language http://www.eclipse.org/m2m/atl/

Having specified these models, the developed application generates a context-aware customized control flow model or new plug-in/plug-out component model based on customer request (Fig. 6). The customer request for the SaaS application is intercepted by the Proxy service which in turn triggers the Context Analysis module which evaluates all context constraints of the context model. The Model Composer module applies only those CFs relevant in the SaaS application usage context to the basic control flow model and component model. The resultant control flow model and component model are automatically transformed, using a set of transformation rules, to generate the executable specification of the target platform i.e. BPEL, or component framework. They will be dynamically deployed to the execution engine; and the customer request is then transferred to the new deployed and personalized process.

VI. Case Study

To demonstrate the approach a small case study is done, namely, the Event Advisor application. This application provides the conference attendee (the customer) with a personalized suggestion for the conference events (i.e. paper, poster, and industrial demo presentations) according to the customer preferences and context. We consider a generic service application that customers can access through a wireless connection using their own portable devices. Fig. 7 depicts a part of the static structure of this application. This application could be enhanced by automatically filling in the `ClientType` parameter, using for this purpose information provided by the context infrastructure. Being a research-oriented customer means that she is not interested in getting suggestions for the industrial demos. Therefore there is a need to change the process structure so that the activity that invokes the `IndustrialDemo` is deleted.

Fig. 8 shows a simple example of the context model that contains two entities: Alice and Bob. The association elements assign the attributes to the entities so that Alice has an attribute `ClientType` whose value is `ResearchOriented` whereas Bob's `ClientType` is `IndustrialOriented`. The context constraint named `CustomerIsResearchOriented` is an example of the constraints having OCL-based parameterized expression. It

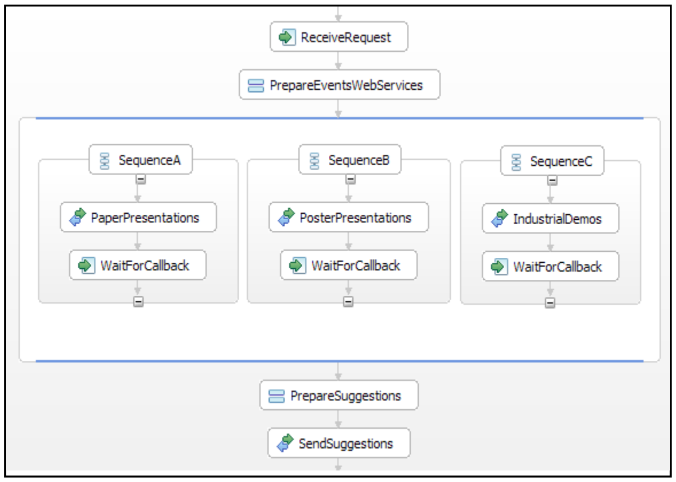

Figure 7. Event advisor application

contains a variable named $CustomerName whose value is extracted either from the customer request information or from any other data source. In either case the above-mentioned proxy service is responsible for assigning the variable value.

Fig. 9 shows a sample of the change fragments `cf1` that regroups different change primitives that should be applied when the customer type is research oriented. The weaving model (Fig. 10) contains one link element that links between the context constraint named `CustomerIsResearchOriented` and the CF named `cf1`. Finally, the developed prototype will generate the customized process which contains only the suggestions for paper and poster presentation events.

VII. Overhead Evaluation

In this experiment, the cost of generating the customized control flow model is evaluated in terms of response time using a Pentium4 PC with RAM of 3GB. For this purpose we use an example of a control flow model consisting of twenty activities. In each experiment we increment the number of change primitives to be applied to the basic model and measure the time required to derive the BPEL artifact and to deploy the new

```
<ctxt:ContextModel xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:ctxt="http://napier.ac.uk/context">
  <associations name="Alice_attributes"
entities="//@entities.0" attributes="//@attributes.0"/>
  <associations name="Bob_attributes" entities="//@entities.1"
attributes="//@attributes.1"/>
  <entities name="Alice"/>
  <entities name="Bob"/>
  <attributes name="ClientType" value="ResearchOriented"/>
  <attributes name="ClientType" value="IndustrialOriented"/>
  <contextconstraints expression="associations->select(a |
a.entities->exists(e | e.name='$CustomerName') and
a.attributes->exists(a1 |a1.name = 'ClientType' and
a1.value='ResearchOriented'))"
name="CustomerIsResearchOriented"/>
  ...
</ctxt:ContextModel>
```

Figure 8. The context model

```
<cs:ChangeStrategy xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:cs="http://napier.ac.uk/cs">
<changeFragments name="cf1">
 <children xsi:type="cs:DeletedSequence"
updatedElement="SequenceC"/>
 <children xsi:type="cs:DeletedCopy"
updatedElement="DemosSuggestion"/>
 <children xsi:type="cs:ChangedCopy"
updatedElement="SuggestionResponse">
   <to variable="..." part="suggestionsData"/><from
literal="..."/>
 </children>
</changeFragments>
</cs:ChangeStrategy>
```

Figure 9. The change strategy model

```
<weaving:WeavingModel xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:weaving="http://napier.ac.uk/weaving">
  <links name="l1"
contextConstraintName="CustomerIsIndustrialOriented"
changeFragmentName="cf1"/>
</weaving:WeavingModel>
```

Figure 10. The weaving model

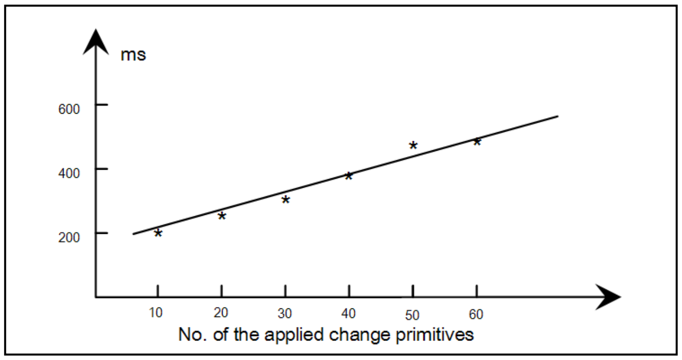

Figure 11. The overhead evaluation

process into the BPEL engine. Fig. 11 shows that the overhead is negligible. For 10 change primitives the overhead is around 200ms. For 60 change primitives the overhead is just less than 0.5s which still acceptable in SaaS kind of applications.

VIII. Related Work

One of the most successful research directions in the field of software engineering and particularly in software reuse was the software product line SPL (e.g. [6]). Variation points are one of the key concepts in SPL to express variability. However, as aforementioned capturing the application variability using the change fragments and primitives is more intuitive and logical from the developer point of view.

In [5] the authors present an approach that allows the generation of customization processes out of variability descriptors. The proposed approach is different in the way it presents the variation points and variants. It regroups the different variants into more abstract and meaningful constructs to ease the adjustments of the basic application.

Similar to the proposed approach, Provop [7] provides a flexible solution for managing process variants following an operational approach to configure the process variant out of a basic process. This is achieved by applying a set of well-defined change operations to it. However, the proposed approach deviates from Provop in that it uses the MDD approach and defines the CFs as change model elements not as change operations.

VxBPEL [4] is an adaptation language that is able to capture variability in processes developed in the BPEL language. VxBPEL provides the possibility to capture variation points, variants and realization relations between these variation points. Unlike the proposed approach, VxBPEL works on the code level and the variants are mixed with the process business logic which may add complexity to the process developer task. Further, unlike the proposed generative approach, VxBPEL is specific to BPEL language.

IX. Conclusion

The challenge for the SaaS architect is to ensure that the task of configuring applications is simple and easy for the customers, without incurring extra development or operation costs for each configuration. Therefore, we have described a MDD approach for managing and automatic generating customized SaaS application variants. Based on logically-viewed well-defined CFs and change primitive constructs; on the ability to regroup CFs in reusable components; and on the ability to regroup these components in a constrained way, necessary adjustments of the basic application can be correctly and easily realized when creating or configuring an application variant. The proposed approach may provide the possibility of "plugging" more easily within the same basic application different customization strategies tailored for different contexts. Future work includes providing the developer with verification tools to verify the change fragments composition regarding the application consistency and integrity at design time. This will give the developer the flexibility to define profound changes to the SaaS applications in different views.

References

[1] M. Baldauf, S. Dustdar, and F. Rosenberg, "A survey on context-aware systems," Int. J. Ad Hoc and Ubiquitous Computing, vol. 2, 2007.

[2] F. Chong and G. Carraro. "Architecture Strategies for Catching the Long Tail". MSDN Library, Microsoft Corporation, April, 2006.

[3] Z. Jaroucheh, X. Liu, and S. Smith, "CANDEL: Product Line Based Dynamic Context Management for Pervasive Applications," Int. Conference on Complex, Intelligent and Software Intensive Systems, 2010.

[4] M. Koning, C. Sun, M. Sinnema, and P. Avgeriou, "VxBPEL: Supporting variability for Web services in BPEL," Information and Software Technology, vol. 51, 2009, pp. 258-269.

[5] R. Mietzner and F. Leymann, "Generation of BPEL Customization Processes for SaaS Applications from Variability Descriptors," 2008 IEEE International Conference on Services Computing, 2008.

[6] K. Pohl, G. Bockle, and F. van der Linden. "Software Product Line Engineering: Foundations, Principles, and Techniques". Springer, 2005.

[7] M. Reichert, S. Rechtenbach, A. Hallerbach, and T. Bauer, "Extending a Business Process Modeling Tool with Process Configuration Facilities: The Provop Demonstrator," BPM'09 Demonstration Track, Business Process Management Conference, Germany, 2009.

[8] H. Tran, U. Zdun, and S. Dustdar, "View-based and Model-driven Approach for Reducing the Development Complexity in Process-Driven SOA," Int. Conference BPSC'07, 2007, pp. 105-124.