Service-Oriented Dynamic Software Product Lines with DyBPEL

Liliana Pasquale*, Sam Guinea†, Luciano Baresi†,
* *Lero - the Irish Software Engineering Research Centre, Limerick, Ireland*
Email: liliana.pasquale@lero.ie
† *Politecnico di Milano, Milano, Italy*
Email: {guinea|baresi}@elet.polimi.it

***Abstract*—Software systems are becoming more and more dynamic. New requirements, context-awareness, and intrinsic complexity are demanding for solutions that allow software systems to change themselves while in operation. This shift has imposed dynamic software product lines (DSPL) to support late variability in systems that need to cope with changes at runtime. Since Service Oriented Architecture (SOA) have proven to be a cost-effective solution to the development of flexible and dynamic software systems, this paper discusses the convergence between SOA and DSPL. The proposed solution is based on CVL and on DyBPEL, our extended BPEL solution for managing variability at runtime. The proposal has been validated through a simple scenario in the context of smart homes. Obtained results are promising and interesting.**

I. Introduction

Modern software systems are becoming more and more flexible and dynamic, as they increasingly exploit runtime adaptations to cope with changes in their context of execution and in their requirements. Often times, however, the sources of change cannot be predicted in advance, making it impossible to engineer systems that incorporate all possible variants. Furthermore, managing a huge number of configurations can be extremely costly.

In the past the alternative features would be identified and selected before deploying the system; nowadays the actual features need to be selected, deployed, and operated while the system is in execution. This shift towards runtime solutions has imposed the advent of dynamic software product lines (DSPL): they extend software product lines (SPL) to support late variability. Thanks to DSPL, software product lines are becoming *situational*-aware, flexible and capable of adapting their features.

SPL exploit the fundamental notion of feature modeling. Feature modeling is the design-time task in which the designer analyzes the software family as a whole and establishes (i) the common and reusable assets that form its basic platform, and (ii) the application-specific assets required for its specific customizations. The model shows the alternative variations that can exist for each feature, and describes the constraints that exist between them. In DSPL we need to be able to dynamically switch an executing system from one variant to another, without stopping its execution, without violating its feature model's constraints, and without degrading its functional and non-functional behavior. However, we believe that being able to dynamically switch between feature sets is still not enough to satisfy our needs. The feature model itself must become a runtime entity that can dynamically evolve to satisfy new variability dimensions in the system.

Since Service Oriented Architecture (SOA) has proven to be a cost-effective solution to the development of flexible and dynamic software systems [1], we believe that the convergence of SOA and DSPL can provide significant mutual advantages. On the one hand, the loose coupling that is intrinsic to SOA techniques can provide DSPL with the technical underpinnings needed to provide flexible feature management. In this sense DSPL can benefit from the great body of work achieved in the realm of self-adapting SOA systems [2], [3], where monitoring and adaptation techniques have been extensively studied. On the other hand, DSPL can provide the modeling infrastructure required to understand a running SOA-based system, by highlighting the relationships that exist between its various parts. In particular, these models can be used to understand the implications of modifying a system's configuration at runtime.

This paper aims to encourage this convergence by focusing on BPEL compositions, and by enriching them with dynamic variability management. Our contribution is therefore twofold: on the one hand, we present a technical SOA-based solution for DSPL; on the other hand, we provide BPEL processes with the modeling support they need to understand and embrace higher degrees of variability. The approach defines variabilities for our BPEL processes using CVL (Common Variability Language) [4]. This choice allows us to easily generate a dynamic software product line starting from our models, and then to run and manage it using an enriched BPEL engine. In this sense we provide DyBPEL, an open-source BPEL engine augmented with adaptation capabilities that can be used to dynamically inject variability into processes. The framework exploits aspect-oriented programming [5] to dynamically change the features that are bound to their variation points, as well as the variation points themselves. Features consist of fragments of BPEL code that can access both the process' internal state and cooperate with remote partner services. This allows BPEL processes to cope with unexpected changes in requirements, in the

availability of resources, and in the execution environment. Even if the aspect-based solution has been validated on conventional BPEL engines, we claim that proposed concepts and solutions can be exploited in a wider context.

The proposal is validated on a simple example in the context of smart homes in which we automate the control of domestic systems (e.g., heating, lighting, presence detection) to save energy. For example, we may want to activate automatic lighting to limit energy consumption. The obtained results are interesting and promising.

II. Service Oriented Architectures (sidebar)

A Service Oriented Architecture (SOA) defines an architectural pattern that allows one to garner beneficial qualities in complex systems that require intra- and inter-enterprise integration and collaboration [6]. SOA systems build upon the notion of services, that is, of self-describing coarse-grained components that are accessed over the Internet using well-defined standards, such as WSDL (Web Service Description Language [7]) and SOAP [8]. Services are loosely coupled, independent entities that can be implemented using heterogenous technologies. They simplify cross-enterprise integration, and allow complex service-based systems to become more flexible with respect to change.

Integration is simplified because a service provider only needs to publish a service (description) once. There is no need to download and deploy the services one wants to use; they can simply be accessed remotely through the Internet. This also means that the providers are the ones responsible for their management. This, in turn, has profound implications on the adopted business model: services can be charged on a per-interaction basis, using a flat rate, or be provided freely and supported through advertising.

Modern software systems also become more flexible with respect to the changes that can occur in their context of execution, or in the requirements that their stakeholders impose. This is crucial for emerging domains such as ambient intelligence, context-aware applications, and pervasive computing, and has been an important research problem within the SOA community for some time. One identified solution is to take advantage of loose-coupling, and to use dynamic- or late-binding techniques to exploit the most appropriate services available at runtime during any given situation.

The main development task in SOA systems is composition. Although many composition models have been proposed, most of the them are choreographies implemented in BPEL (Business Process Execution Language) [9]. A BPEL process establishes the order in which message exchanges are performed between a centralized entity, called the BPEL engine, and its external partner services. Although initial traction has been high, time has shown that the BPEL standard is fundamentally flawed in its capability to support change. Theoretically, one could support dynamic endpoint selection, but this would require that the endpoint management be intertwined with the process' main business logic. However, this is often times unfeasible, especially when the change-space is large, or partially unknown at design time. BPEL engine vendors have provided alternative solutions under the guise of BPEL extensions, yet this defeats one of the standard's most important goals, which is to have portable business processes that can be run on diverse engines. Finally, as we show in this article, being able to support change may require capabilities that go beyond simple service rebinding. For example, a process may need to substitute entire streams of operation; the BPEL specification currently has no solution for such scenarios.

Although the main ideas behind SOA systems are not new, the focus on well-defined standards, a simplified development process, and a clearer understanding of distributed applications, has allowed them to flourish. Nevertheless, some of the initial promises, and in particular its ability to easily support change, have not yet been entirely met.

III. DyBPEL

DyBPEL extends ActiveBPEL [10], an open-source BPEL engine, with adaptation capabilities that can be used to dynamically inject variability into processes. DyBPEL's architecture is shown in Figure 1. It includes three main components: the *Coordinator*, the *Runtime Modifier*, and the *BPEL Modifier*. The Coordinator is a web service that exposes administrative operations that can be used to migrate running and future process instances to new process versions, and to coordinate the modifications that the Runtime Modifier and the BPEL Modifier need to perform. The former handles the migration of a running process instance, while the latter manages future process instances.

DyBPEL also includes a fourth component, which we refer to as *DB (MySQL)*. It contains all the data structures required to effectively operate the Runtime Modifier and the BPEL Modifier. This component traces the processes that are deployed on the engine, the versions associated with each process, and the number of running instances that comply with each version (see record T1 in Figure 1). It also stores the changes that will be applied on the executing instances of the process, in case they need to be migrated to a new version (see record T2 in Figure 1). Each change record specifies the `type of change` (i.e., add/remove an activity, a variable, or a partner link), and the point in the process in which execution should be temporarily blocked for the change to be applied (`block activity`). If the change consists in the addition or removal of a BPEL activity, the record must also contain the `target activity`, that is the activity to be removed, or the activity after which a new activity will be added; and finally, if the change consists in the addition of a variable, a partner link, or an activity, the record must also contain the corresponding `XML definition`.

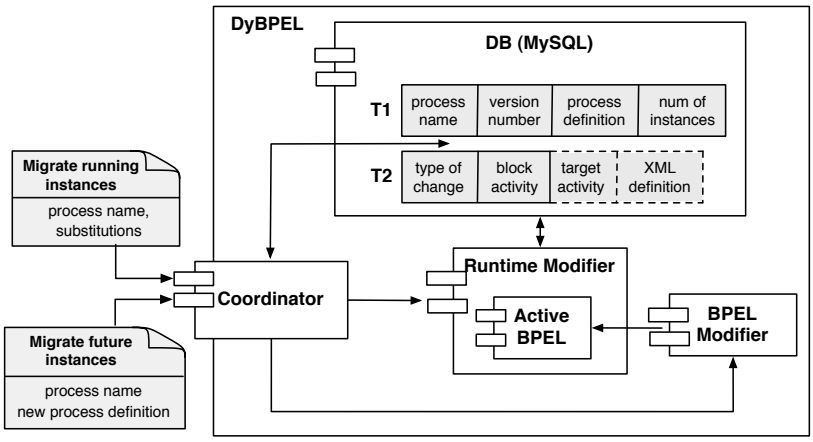

Figure 1. The Architecture of DyBPEL.

When the Coordinator receives a request to migrate a running process instance to a new version, it receives the process name, and the substitutions that the requestor wants to apply. For each substitution, the Coordinator generates a set of entries in the DB that specify the changes and the block activities in which they should be applied. Note that the block activities should guarantee that no conversation with partner services is interrupted, and that all internal state variables are in a consistent state. In other words, migrations cannot be performed while the process is executing activities in a transactional scope. For example, we can only add or remove partner links when scopes that incapsulate them have yet to be activated. If we want to add or remove an activity, the execution point should be immediately before the target activity and should not be contained in any transactional scope. If a process instance already passes the execution point where the first modification should be applied, it cannot be migrated anymore and it will continue to comply with the same version.

The Runtime Modifier is the component in charge of intercepting a running process to apply the changes stored in the DB. This is achieved exploiting AOP techniques (Aspect Oriented Programming [5]), and, in particular, AspectJ [11]. The process is stopped after each BPEL activity is executed. At this point the DB is checked to see whether it is necessary to perform changes in that point of execution. If it is the Runtime Modifier starts by retrieving the runtime object that is created by ActiveBPEL to represent the execution state and the activities of the intercepted process, i.e., the *AeBusinessProcess*. It then applies the requested modifications by adding and removing activities, partner links or variables.

The BPEL Modifier is the component in charge of migrating future instances to a new process version. It starts by extracting from the DB the id of the process' latest version. Then it creates a new endpoint that will be associated with the new version of the process, and modifies ActiveBPEL's deployment descriptor file for that process. Finally, the BPEL Modifier deploys the new version of the process in a way that is transparent to the users. It also creates a new record in the DB which contains the newest version's id and its definition.

IV. DSPL WITH DYBPEL

This Section describes how DyBPEL can dynamically support the deployment of new product lines associated with executing BPEL processes. Our approach to defining software variabilities and to generating a dynamic software product line is based on CVL. CVL is a technology independent language that proposes metamodels, semantics, and a concrete syntax for specifying variability. To illustrate DyBPEL and its interplay with CVL, this section provides a simplified example in which we illustrate a system that automates the control of the domestic appliances in a smart home (e.g., heating and lighting). The CVL representation of the software variabilities associated with this example is shown in Figure 2.

To express variabilities in CVL the variability designer needs to define what a *Base Model*, a *Feature Model*, and a `Product Realization Model`. In our approach the base model is provided in BPEL, and it contains the mandatory features of our application. However, we enrich it with *Fragments* and *Substitutions*. The former specify additional pieces for the base model (i.e., BPEL fragments), ranging from additional variables and partner links, to groups of activities of any given complexity. The latter define how pieces of the base model can be replaced by specific fragments. The feature model is provided in a technology independent language, and it contains the possible feature choices and relationships that exist between them. Finally, the realization model associates each choice in the feature model with the substitutions that have to be performed on the base model. Note that after a set of features is chosen, the

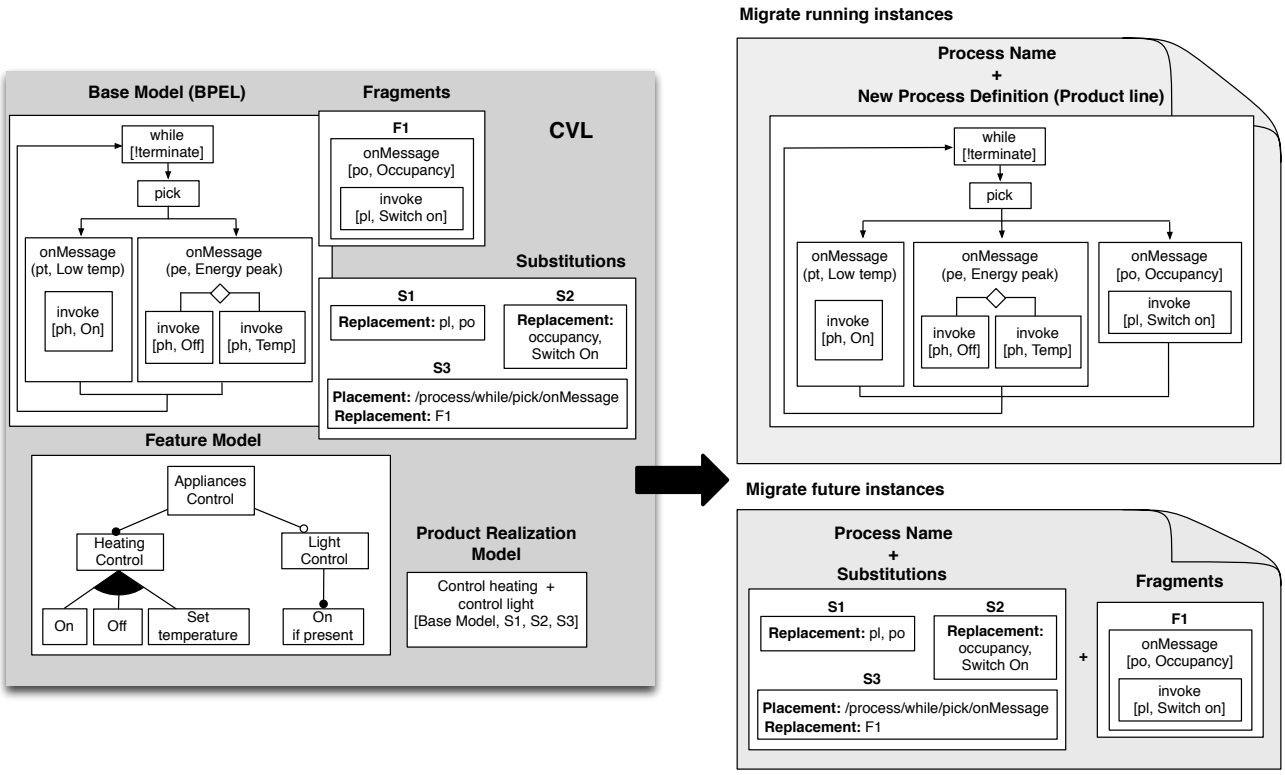

Figure 2. An example product line expressed in CVL.

generation of the new product line is automatically supported by CVL.

As described in the Feature Model of Figure 2, our example system is characterized by mandatory features (*Heating control*) and optional features (*Light control*). When providing heating control the system can switch the heater on (*On*) if the temperature is too low, or switch it off (*Off*) if it is too high. It can also reduce its target temperature (*Set temperature*) if the energy consumption is peaking. When providing light control the system can switch the light on (*On if present*) if a person is detected on premise.

In our example the base model only includes the process variables, activities and partner links necessary to support the mandatory features of the system. The process has three partner services to monitor the temperature (*pt*), control the energy consumption (*pe*), and manage the heater (*ph*). When the temperature is too low, the process receives message *Low temp* from *pt*, and consequently invokes *ph* to switch the heater on. When an energy peak is detected, the process receives message *Energy peak* from *pe*, and evaluates how near the temperature is to its target. If it is the process invokes *pt* to reduce the target temperature to *Temp*; alternatively, it invokes *ph* to switch the heater off.

The light control feature is implemented through fragment (*F1*). This fragment includes two new partner services. The first one (*pl*) is in charge of controlling the lighting, while the second one (*po*) represents a sensor capable of detecting human presence. Every time *po* discovers that a person is in the room it sends a message (*Occupancy*) to the process. Subsequently, the process invokes *pl* to switch the light on. To apply the light control feature, a set of substitutions (*S1*, *S2*, and *S3*) is specified. *S1* adds partner services *po* and *pl*, while *S2* adds variables *Occupancy* and *Switch on* needed to represent the messages exchanged between the process and its partner services within fragment *F1*. Since the locations in which the partner services and variables are added are fixed, it is not necessary to specify a placement in *S1* and *S2*. *S3* specifies that *F1* can be inserted right after the *onMessage* definition associated with the *pick* activity.

When we need to migrate a running process instance, the approach automatically generates the input that needs to be sent to DyBPEL's Coordinator component. It sends the name of the process, as well as the substitutions identified using the product realization model. In our example, the Coordinator reacts by creating a new record in DyBPEL's database for each variable and partner link that needs to be added. The modifications need to be applied before the *pick* activity. The Coordinator also creates a new record for adding fragment *F1*. The target activity, in this case, is one of the onMessage definitions defined in the context of the pick activity. The execution point where the process should be blocked is after the execution og the *pick* activity.

When we need to migrate future process instances, a new product line is generated directly through the CVL-based approach. A request containing the process name and its modified definition is subsequently sent to DyBPEL's Coordinator, the new version is deployed, and new process requests are transparently redirected to the newest version of the process.

V. Evaluation

This section briefly discusses the performance of our solution and its limitations. Table I provides data describing the time needed to migrate a running process instance. This includes the time needed to execute the (*Pick*) activity where the migration is performed. It includes the delay introduced when we retrieve a modification from the DB (*DB Check*) and when we apply the changes (*Changes*). The time needed to dynamically modify the process is very low ($\sim 11\%$ of the execution time of the pick activity); while the time needed to interact with the DB is higher ($\sim 25\%$ of the execution time of the pick activity).

	Average Time [s]	**Median [s]**	**Variance**
DB Check	0.0463	0.045	0.00002
Change	0.0198	0.019	0.00001
Pick	0.2472	0.2485	0.0001

Table I
PERFORMANCE TO MIGRATE THE RUNNING PROCESS INSTANCES.

Our approach has the drawback of being intrusive, since it needs to interrupt the process' execution at the end of each activity. This means that when we need to decide whether to migrate a running process or not, there is a tradeoff between the process delay and the criticality of the updates. Another limitation is that in some cases not all of the running process instances can be migrated to a new version. As a matter of fact, once a process instance has passed the execution point in which the changes need to be applied, it cannot be migrated anymore. For this reason, it would be beneficial to trigger suitable rollback activities to restore the execution to a previous state in which the changes can still be applied. However, this is a complex problem that involves distributed rollbacks, since we also need to compensate actions that have already been performed by the process' partner services. This will be part of our future work in this area.

Table II provides data describing the time needed by the Coordinator and the BPEL Modifier to migrate future process instances. Note that the time needed by the Coordinator also includes the time needed by the BPEL Modifier. The main overhead is generated by the BPEL Modifier which needs to deploy a new version of the process and create a corresponding entry in the DB. Note that this overhead is independent from the complexity of the process being deployed.

	Average Time [s]	**Median [s]**	**Variance**
Coordinator	0.1515	0.141	0.0035
BPEL Modifier	0.1302	0.115	0.0043

Table II
PERFORMANCE TO MIGRATE THE FUTURE PROCESS INSTANCES.

VI. Related Work (sidebar)

In [12], Gomaa and Hussein established that the advent of new and emerging domains, such as ubiquitous computing, house automation, and ambient intelligence would require higher degrees of adaptability not often provided by traditional SPLs. This same stance was further backed by Hallsteinsen et al. in their well-known paper on DSPL [13]. In this paper, they highlighted the need to support variations in requirements and in resource constraints, and discussed the properties a DSLP needs to possess to tackle these issues, such as dynamic variability, and support for dynamic variation points. They also stressed that DSPL could benefit from research being achieved in other related ares, yet they did not make a strong case for its convergence with Service-oriented technology.

The convergence with Service-based technology was emphasized by Krut and Cohen [14] and by Istoan et al. [15]. These works advocate that both DSPL and Service technology encourage the reuse of software assets, and that they both fostered productivity gains, decreased development costs, and competitive advantage. An example is given by Gomaa and Hashimoto [16]. They developed an extension to SASSY, a model-driven framework for Self-Architecting Software Systems. In their work, they create a mapping between features and services at design time, and use it to choose service substitutions at runtime. The approach is dynamic because the feature model and its mapping to services can change while the system is executing. Research on the convergence between DSPLs and SOA typically revolves around the idea that features should be mapped to atomic services. This is a simplistic approach, and in this paper we have shown that system designers can benefit from richer kinds of mappings.

Two further lines of DSPL research have emerged in the last few years: DSPL solutions that exploit Aspect-Oriented Programming techniques, and context-aware DSPL. Morin et al. [17] have developed K@RT, an aspect-oriented and model-driven DSPL framework. K@RT uses a reference model at runtime to navigate the system architecture using model-oriented languages, and then invoke the services in the running system. The model can be modified during execution, and checked against constraints to be sure that the reconfiguration is safe. Dinkelaker et al. [18] use a dynamic feature model to describe late variability in the DSPL. The approach uses dynamic aspects, runtime models of aspects, and detection and resolution of aspects interactions. The

result is a solution in which the designer does not need to create reconfigurations for every possible feature combination. Instead, the designer can focus on modeling the reconfigurations of interacting features. The runtime support then ensures that the DSPL is delivered appropriately.

Regarding context-aware solutions, Parra et al. [19] proposed CAPucine, a Context-Aware Service-Oriented Product Line. Using CAPucine designers can produce product derivations that monitor their context of execution and react by including appropriate software assets into the system. Their work exploits SCA (Service Component Architecture) models, and the dynamic binding and unbinding of referenced services provided by the FraSCAti runtime environment. In order to obtain information from the environment, the authors use COSMOS, a context-aware framework connected to the environment through appropriate sensors.

VII. Conclusions

We have presented a solution that encourages the convergence between DSPL and SOA technologies with a focus on BPEL processes. The solution uses CVL as a mechanism to define a BPEL process' base model, its feature variabilities and the constraints that exist between them, and the models needed for automatic product realization. The produced DSPL is then run and managed through DyBPEL, an open-source BPEL solution that exploits AOP techniques to provide dynamic addition and removal of features and variation points. The proposed solution was exemplified in the context of a simple home automation example, and has shown to provide interesting results. As already said, the aspect-based solution presented in the paper is not limited to SOA, and BPEL in particular, but we think it could be exploited within many different service- and component-based solutions. The analysis of different infrastructures and the customization of our solution for their particular needs is one of our main goals for the future.

References

[1] D. Krafzig, K. Banke, and D. Slama, *Enterprise SOA: service-oriented architecture best practices*. Prentice Hall PTR, 2005.

[2] H. Giese and B. H. C. Cheng, Eds., *Proceedings of the 6th Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2011.

[3] L. Baresi and S. Guinea, "Self-supervising bpel processes," *IEEE Transactions on Software Engineering*, vol. 37, pp. 247–263, 2011.

[4] O. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen, "Adding Standardized Variability to Domain Specific Languages," in *Proceedings of the 2008 12th International Software Product Line Conference*, ser. SPLC '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 139–148.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*. Springer, 1997, pp. 220–242.

[6] M. P. Papazoglou and W.-J. V. D. Heuvel, "Service Oriented Design and Development Methodology," *International Journal of Web Engineering and Technologies*, vol. 2, no. 4, pp. 412–442, 2006.

[7] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Service Description Language (WSDL) 1.1," http://www.w3.org/TR/wsdl, 2001.

[8] D. Box et al., "Simple Object Access Protocol (SOAP) 1.1," http://www.w3.org/TR/2000/NOTE-SOAP-20000508/, 2000.

[9] A. Alves et al., "Web Services Business Process Execution Language Version 2.0," http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf, 2007.

[10] Active Endpoints, "Activebpel engine," http://www.activebpel.org.

[11] "The AspectJ Project," http://www.eclipse.org/aspectj/.

[12] H. Gomaa and M. Hussein, "Dynamic software reconfiguration in software product families," *Software Product-Family Engineering*, pp. 435–444, 2004.

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

[14] R. Krut and S. Cohen, "Service-Oriented Architectures and Software Product Lines - Putting Both Together," in *Proceedings of the 12th International Conference on Software Product Lines*. IEEE Computer Society, 2008, p. 383.

[15] P. Istoan, G. Nain, G. Perrouin, and J.-M. Jézéquel, "Dynamic Software Product Lines for Service-Based Systems," in *Proceedings of the 9th International Conference on Computer and Information Technology*. IEEE Computer Society, 2009, pp. 193–198.

[16] H. Gomaa and K. Hashimoto, "Dynamic Software Adaptation for Service-Oriented Product Lines," in *Workshop Proceedings of the 15th International Conference on Software Product Lines*. ACM, 2011, p. 35.

[17] B. Morin, O. Barais, and J. marc Jzquel, "K@RT: An Aspect-Oriented and Model-Oriented Framework for Dynamic Software Product Lines," in *Proceedings of the 3rd International Workshop on Models@Runtime, colocated with MODELS'08*, 2008.

[18] T. Dinkelaker, R. Mitschke, K. Fetzer, and M. Mezini, "A Dynamic Software Product Line Approach Using Aspect Models at Runtime," in *Proceedings of the 1st Workshop on Composition and Variability, colocated with AOSD'10*, 2010.

[19] C. Parra, X. Blanc, and L. Duchien, "Context Awareness for Dynamic Service-Oriented Product Lines," in *Proceedings of the 13th International Software Product Line Conference*. ACM, 2009, pp. 131–140.