Alférez Salinas, GH.; Pelechano Ferragud, V.; Mazo, R.; Salinesi, C.; Díaz, D. (2014). Dynamic adaptation of service compositions with variability models. Journal of Systems and Software. 91:24-47. doi:10.1016/j.jss.2013.06.034.

Dynamic Adaptation of Service Compositions with Variability Models

G.H. Alférez[a,*], V. Pelechano[b], R. Mazo[c], C. Salinesi[c], D. Diaz[c]

[a] *Facultad de Ingeniería y Tecnología, Universidad de Montemorelos, Apartado 16-5, 67500 Montemorelos, Mexico*
[b] *Centro de Investigación en Métodos de Producción de Software (ProS), Universitat Politècnica de València, Camí de Vera s/n, E-46022 Valencia, Spain*
[c] *CRI, Panthéon Sorbonne University, 90 rue de Tolbiac, 75013 Paris, France*

Abstract

Web services run in complex contexts where arising events may compromise the quality of the whole system. Thus, it is desirable to count on autonomic mechanisms to guide the self-adaptation of service compositions according to changes in the computing infrastructure. One way to achieve this goal is by implementing variability constructs at the language level. However, this approach may become tedious, difficult to manage, and error-prone. In this paper, we propose a solution based on a semantically rich variability model to support the dynamic adaptation of service compositions. When a problematic event arises in the context, this model is leveraged for decision-making. The activation and deactivation of features in the variability model result in changes in a composition model that abstracts the underlying service composition. These changes are reflected into the service composition by adding or removing fragments of Business Process Execution Language (WS-BPEL) code, which can be deployed at runtime. In order to reach optimum adaptations, the variability model and its possible configurations are verified at design time using Constraint Programming. An evaluation demonstrates several benefits of our approach, both at design time and at runtime.

Keywords: Variability, Models at runtime, Autonomic computing, Dynamic adaptation, Dynamic software product line, Web service composition, Constraint programming, Verification

1. Introduction

Software is executed in complex, heterogeneous and highly intertwined computing infrastructures in which a diversity of events may arise. For example,

∗. Corresponding author
Email addresses: harveyalferez@um.edu.mx (G.H. Alférez), pele@dsic.upv.es (V. Pelechano), raulmazo@gmail.com (R. Mazo), camille.salinesi@univ-paris1.fr (C. Salinesi), daniel.diaz@univ-paris1.fr (D. Diaz)

security threats, network problems, performance reduction in one of the servers, etc. In these situations, it is desirable to adapt the software to continue offering the required functionality. *Software adaptation* can be seen as the ability for humans to reconfigure the software and then restart it, or the ability of the software to reconfigure itself during execution (Akkawi et al., 2007). The first case can be referred to as *static adaptation* and the second one as *dynamic adaptation*. It is possible to carry out static adaptations in cases where the system can be shut down in order to make the required manual adaptations. However, there are critical systems that cannot be stopped to implement the adaptations, e.g. software that run power grids and software for global banking. In such cases, software needs to dynamically adapt its behavior at runtime in response to changing conditions in its supporting computing infrastructure (McKinley et al., 2004, Cetina et al., 2009, Alférez and Pelechano, 2011a). *Dynamic adaptation* of software behavior refers to the act of changing the behavior of some part of a software system as it executes, without stopping or restarting it (Keeney, 2004).

In order to carry out dynamic adaptations, we argue that software needs to take the following key issues into account :

- ***Context Awareness :*** For the purpose of supporting dynamic adaptations, software should be aware of changes in its context. The *context* is any information that can be used to characterize the situation of an entity (Dey, 2001). Context-aware systems are concerned with the acquisition of context, the abstraction and understanding of context, and application behavior based on the recognized context (Schmidt, 2002).
- ***Adaptation Policies :*** *Adaptation policies* change the behavior of the system during execution (Morin et al., 2008). They state in a declarative manner the actions required to adapt the running system to a configuration that better fits its current context.
- ***A Supporting Infrastructure :*** It is unthinkable to depend on manual adaptations because of the inherent intricacy of today's systems and the desired prompt responses. Furthermore, critical systems cannot be stopped in order to carry out the necessary adaptations. Thus, a computing infrastructure should provide support for dynamic adaptations to face context events (Alférez and Pelechano, 2011a, Cetina et al., 2009).
- ***Verification :*** Adapting the system according to changes in the context is not enough. It is necessary to ensure that new system configurations are not invalid in a given situation.

1.1. The Need for Dynamic Adaptation of Service Compositions

A good example of systems that require dynamic adaptations are the ones based on Web service compositions (or simply called service compositions). Web services have evolved as a standardized and technology-agnostic interoperable way of integrating processes and applications (Little, 2003). Basically, a *Web service* is a special software component that is searched, bound, and executed at runtime and allows systems to interact through standard Internet protocols (Koning et al., 2009). In order to reach the full potential of Web services, they

can be combined to achieve specific functionalities. If the implementation of a Web service business logic involves the invocation of other Web services, it is called a *composite service*. The process of assembling a composite service is called *service composition*.

Web services run in a context (e.g. their operating computing infrastructure). In an ideal scenario, Web service operations would do their job smoothly. However, several exceptional situations may arise in the complex, heterogeneous, and changing contexts where they run. For instance, a Web service operation may have greatly increased its response time or may have become unavailable. Cases like these make evident the need for dynamic adaptations in critical systems that are based on service compositions. These adaptations may be triggered in order to do the following at runtime : keep certain contracts known as service-level agreements (SLAs), offer extra functionality depending on the context, protect the system, or make the system more usable.

Related work about dynamic adaptation of service compositions can be classified into three groups. The first group supports dynamic adaptation at the language level (Colombo et al., 2006, Koning et al., 2009, Baresi and Guinea, 2011, Narendra et al., 2007, Sonntag and Karastoyanova, 2011, Moser et al., 2008). This approach can hinder reasoning about adaptations with complex and error-prone scripts (Fleurey and Solberg, 2009). The second group focuses on low-level implementation mechanisms for self-adaptation (Erradi and Maheshwari, 2005, Cardellini et al., 2010, Mosincat and Binder, 2008). This approach lacks support for analyzing the inherent variability of dynamic adaptation at design time. The third group deals with modeling variability in service compositions that support business processes (BPs) (Nguyen et al., 2011, Sun et al., 2010, Hadaytullah et al., 2009, Razavian and Khosravi, 2008, Rosemann and Van der Aalst, 2007, Gottschalk et al., 2008, Puhlmann et al., 2005). Research works in this group propose the creation of variability models that are only used at design time. We argue that the knowledge in variability models should be leveraged at runtime to guide adaptations and hide the complexity of the adaptation space. Moreover, the approaches in the aforementioned groups lack verification of possible service composition configurations caused by dynamic adaptations.

1.2. Contribution

In this paper, we propose a framework that uses variability models at runtime to support the dynamic adaptation of service compositions. This framework spans over design time and runtime, and states the models, tools, and artifacts that can be used to support dynamic adaptations.

At design time, the framework provides tool-supported steps for creating the models that guide autonomic changes. In general terms, a service composition can be viewed as the assembly of pieces to deliver functionality; those pieces can be Web services offered by different providers or composite services themselves. We argue that in the advent of problematic events, functional pieces can be added, removed, replaced, split or merged from a service composition at runtime, hence delivering a new service composition configuration. To this end,

we propose that service compositions be abstracted as a set of features in a variability model. A *feature* can be defined as "a logical unit of behavior specified by a set of functional and non-functional requirements" (Bosch, 2000). Thus, adaptation policies describe the dynamic adaptation of a service composition in terms of the activation or deactivation of features in the causally connected variability model (i.e., changes in this model cause the service composition to adapt and vice versa). The variability model and its possible configurations are verified by means of Constraint Programming (CP) prior execution to ensure safe recompositions.

At runtime, a computing infrastructure detects problematic events that arise in the context and carries out the necessary adjustments on the service composition. The activation and deactivation of features in the variability model result in changes in a composition model, which is an abstract representation of the underlying service composition. These changes are reflected into the service composition by adding or removing fragments of Business Process Execution Language (WS-BPEL) code, which are deployed at runtime. WS-BPEL is a standard language for specifying BP behavior based on Web Services (OASIS, 2007). Flexible service composition updates are possible through Dynamic Software Product Line (DSPL) engineering (Hallsteinsen et al., 2008).

This paper offers several novel contributions beyond those of our previously published works (Alférez and Pelechano, 2011a,b, Cetina et al., 2009, Alférez and Pelechano, 2012b): 1) in our previous work, the generation of variability model configurations was carried out manually. This was an error-prone and time-consuming task. As a result, in this paper we propose a tool that automatizes the creation of variability model configurations at design time; 2) our previous work did not support the verification of the variability model and its possible configurations. As a result, some undesirable dynamic adaptations could emerge at runtime. Therefore, in this paper we incorporate a constraint logic programming solver to automatize the verification of the variability model and its configurations. Although this solver has been used in our previous work (Mazo et al., 2012), it is the first time that it is used to verify the variability model configurations that are used at runtime; 3) context reasoning has been further exploded by means of inference rules and more expressive adaptation policies; 4) we propose an strategy to avoid the saturation of the system when several context events arise in tight time frames; 5) in our previous work, the translation of changes in the composition model into WS-BPEL code was slow (Torres et al., 2012). In this paper, we propose a faster solution based on fragments of WS-BPEL code; and 6) in this paper, our approach supports dynamic adaptations on an enterprise orchestration engine. Instead of extending the functionality of the orchestration engine, we offer a transparent solution in which the engine does not have to be modified.

The remainder of this paper is structured as follows: Section 2 describes a running example that illustrates the need for dynamic adaptation of service compositions. Section 3 gives an overview of our framework for dynamic adaptation of service compositions. Section 4 describes the models that are created at design time to support dynamic adaptations. Section 5 describes the com-

puting infrastructure to deal with dynamic adaptations. Section 6 introduces a demonstration of our framework. Section 7 presents the evaluation of our framework. Section 8 presents related work, and Section 9 presents conclusions and future work.

2. Running Example

To illustrate the need for self-adaptive service compositions, we introduce a composite service that supports online book shopping at Orange Country Bookstore. The example is specified with the Business Process Model and Notation (BPMN) in Figure 1. BPMN tasks express Web service operations (e.g. UPS Shipping service); and BPMN subprocesses express composite service operations (e.g. Barnes & Noble Books composite service).

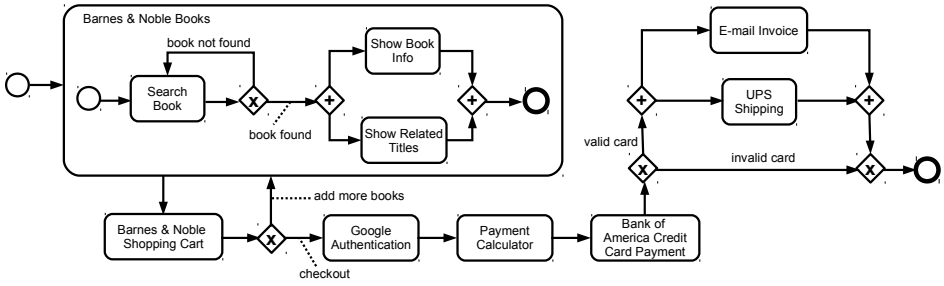


Figure 1: A BPMN model that represents a composite service for online book shopping

The business process starts when a customer looks for a book on the website of Orange Country Bookstore. The first thing the customer wants to do is identify the books to purchase. The searching operation is provided by the Search Book Web service, which is part of the Barnes & Noble Books composite service. When a book is found, then the book information is returned to the customer by the Show Book Info Web service while at the same time the information for other related books is listed by the Show Related Titles Web service. If no book is found, then the customer must refine the search, e.g. using supplementary or different search criteria, or undertake another search. In the next step, the customer adds books into the shopping cart through the Barnes & Noble Shopping Cart Web service. The process can start over again until the customer is satisfied with his or her selection. When the customer is ready to checkout, he or she has to be authenticated by the Google Authentication Web service. Then, the in-house Payment Calculator Web service calculates the total amount to be paid. The payment is done through the Bank of America Credit Card Payment Web service. Finally, if the credit card information is valid, the in-house E-mail Invoice Web service sends an e-mail to the customer with the invoice while the UPS Shipping Web service is invoked to deliver the book. Otherwise, the process terminates.

Different context events may arise in this heterogeneous infrastructure, which call for dynamic adaptations. For example, any third-party Web service operation may fail or perform below required SLAs. The drivers to carry out dynamic adaptations in this example are as follows: 1) since this service composition supports a critical short-running BP (i.e., a process that is contained within a single transaction), it is impossible to shut down the system to make adaptations; 2) as a business differentiator, the online book shopping process requires high availability and high performance. Availability deals with the readiness for correct service in a specific time (Cotroneo et al., 2002). Performance can be measured by observing the current response time to access a Web Service, or by observing the execution time that a Web service takes to execute a job (response time plus execution time) (Ameller and Franch, 2008). Service operations that violate these quality attributes trigger service recompositions; and 3) any necessary adaptation should be verified beforehand to detect undesirable results, such as inconsistencies or delivery of suboptimal solutions in terms of availability or performance.

3. A Framework for the Dynamic Adaptation of Service Compositions

We propose the following strategy to offer a solution for the dynamic adaptation of service compositions. First, the service composition is modeled at design time. Then, we introduce mechanisms to express where and how service compositions can be adapted to face arising context events. These mechanisms are expressed as easy-to-understand and as highly-abstract as possible. At runtime, we provide an infrastructure that detects changes in the context and enables dynamic adaptation. To make this strategy a reality, we propose a framework that states the models, tools, and artifacts that can support dynamic adaptation of service compositions (see Figure 2). This framework consists of two phases: Preparation and Dealing with Dynamic Adaptation.

In order to support dynamic adaptations in our framework, it is necessary to count on abstractions that represent the context, the dynamic configurations of the service composition, and the service composition itself. Also, it is necessary to create the adaptation policies that move the service composition to new configurations. The Preparation Phase covers the creation of these artifacts (see the top of Figure 2). A *composition model* describes the service composition. A *variability model* describes the dynamic configurations of the service composition in terms of activation or deactivation of features (thus, the knowledge that is captured by this model is the basis for adaptation policies). We also propose the creation of two additional supporting models. First, a *context model* formalizes collected context knowledge. Second, since changes in the variability model guide adaptations in the service composition, which is represented in the composition model, we propose a *weaving model* to connect these two models.

We have developed two building blocks to provide variability reasoning at design time: 1) the Configuration Generator uses a variability model and a set of adaptation policies to automatically generate the adaptation space with all the possible configurations of the variability model; and 2) the Verifier uses

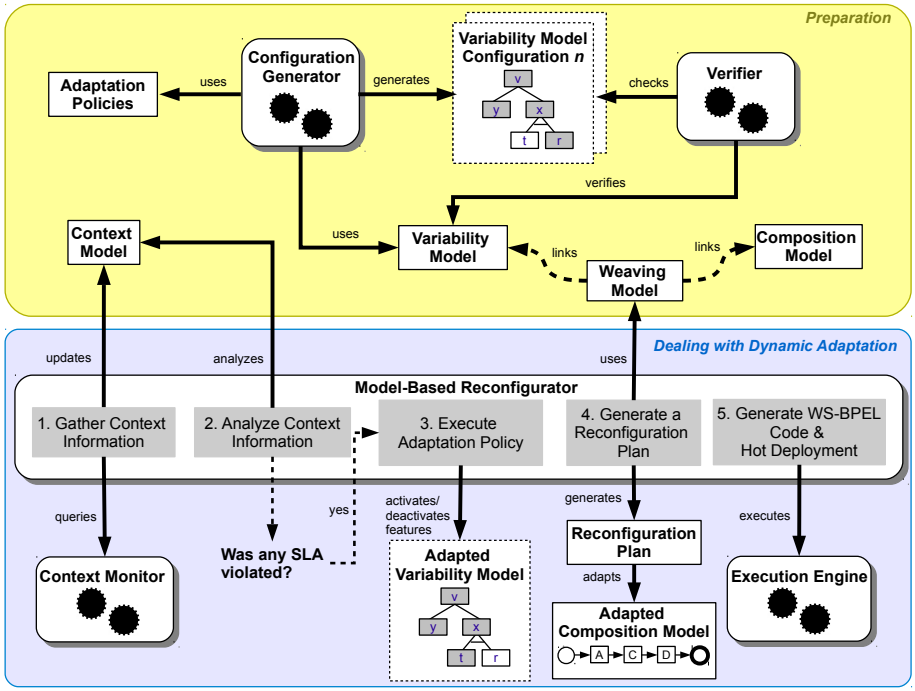


Figure 2: A framework for the dynamic adaptation of service compositions

CP to verify the variability model and check that the generated configurations respect the constraints imposed by the variability model. Verification of the variability model entails finding undesirable properties, such as contradictory information or the impossibility to offer a valid configuration for a particular context. If there are errors in the variability model, they will inevitably spread to an undefined number of configurations, which can drastically diminish the quality and outcome of the entire adaptation.

In the Dealing with Dynamic Adaptation Phase, the models and adaptation policies that are created in the Preparation Phase are used to guide the self-adaptation of the service composition (see the bottom of Figure 2). The proposed infrastructure carries out the following steps to support dynamic adaptations. First, the Model-Based Reconfigurator queries the context information that is collected by the Context Monitor and updates the context model accordingly. Then, the Model-Based Reconfigurator determines if any SLA has been violated in the context according to the information in the context model. If any SLA has been violated, the Model-Based Reconfigurator executes an adaptation policy that indicates the activation or deactivation of features in the variability model (i.e., to move to a new configuration). Then, the set of active features in the new configuration of the variability model is used to generate a *reconfiguration plan*, which is used to modify the elements in the composition model accordingly. Modifications in

the composition model are reflected into the service composition by adding or removing fragments of WS-BPEL code, which are hot deployed in the EXECUTION ENGINE. The EXECUTION ENGINE uses the adapted WS-BPEL code to orchestrate the service composition.

Our solution is framed in the *closed-world assumption*, in which necessary adaptations are fully known at design time. Therefore, it is possible to know beforehand the whole set of service operations that can be used during execution. Our ongoing work presents preliminary results about the evolution of variability models in the open world (Alférez and Pelechano, 2012b).

3.1. Underpinnings of Our Framework

This section provides the background for the four major topics our approach relies on, which are: autonomic computing, models at runtime, dynamic software product lines, and constraint programming.

- ***Autonomic Computing:*** Autonomic Computing (Horn, 2001) has evolved as a discipline that covers the broad spectrum of computing in domains as diverse as mobile devices (White et al., 2007) and home-automation (Cetina et al., 2009). The DEALING WITH DYNAMIC ADAPTATION PHASE is supported by a computing infrastructure that is in charge of autonomously reconfiguring the service composition at runtime according to arising context events.
- ***Models at Runtime:*** A *model at runtime* is a "causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective" (Blair et al., 2009). In our approach, a set of models, in which the variability model plays a fundamental role, is used at runtime to automatically determine how the service composition should be adapted.
- ***Dynamic Software Product Lines:*** Software Product Line (SPL) engineering supports prescribed reuse by selecting the features that are part of a product while removing others that are not part of it (Clements and Northrop, 2001). In our framework, SPL features are used to represent Web service operations that can be selected or deselected in a service composition (Alférez and Pelechano, 2011b). DSPL engineering goes a step further from SPL with the investigation of development issues for reusable and dynamically reconfigurable core assets. Basically, DSPL binds variation points at runtime when software is launched to adapt to the current context and during operation to adapt to context changes (Hallsteinsen et al., 2008). When features are activated or deactivated at runtime due to changes in the context, our framework's DSPL architecture supports the dynamic service recomposition.
- ***Constraint Programming:*** CP has proven to be successful in many relevant application areas such as scheduling, planning, vehicle routing, and resource allocation (Rossi et al., 2006). CP is a declarative programming paradigm to solve Constraint Satisfaction Problems (CSP). A CSP is defined by a set of problem variables (i.e., the unknowns), each associated with a domain of values, and a set of constraints. A *constraint* is a logical

relation between several variables restricting the values these variables can simultaneously take. Solving a CSP consists in finding an assignment of variables satisfying all the constraints. A constraint program mainly states the constraints (incrementally) and asks the constraint solver to find a solution. The solver is then used as a "black-box"' responsible for ensuring the consistency of the constraints. In this work, CP is used to verify the variability model and the generated variability model configurations.

4. Preparation Phase

In the Preparation Phase, we propose to create at design time the models and adaptation policies that shall be leveraged during execution for dynamic adaptation (see Figure 3). Besides creating a composition model to represent a service composition at a highly-abstract level, we propose that systems analysts create a set of additional models to support adaptations. A variability model describes the *variants* (representations of variability objects within domain artifacts (Pohl et al., 2005)) in which a service composition can change at runtime. Thus, the variability model is appointed to solve the need for expressive and easy-to-understand adaptation policies. Nevertheless, in order to replicate the changes that are carried out in the variability model in the composition model, it is necessary to count on a weaving model, which works as a bridge between the elements in these models. Finally, a context model can be used for formal analysis of context information for the service operations, which are represented in the composition model.

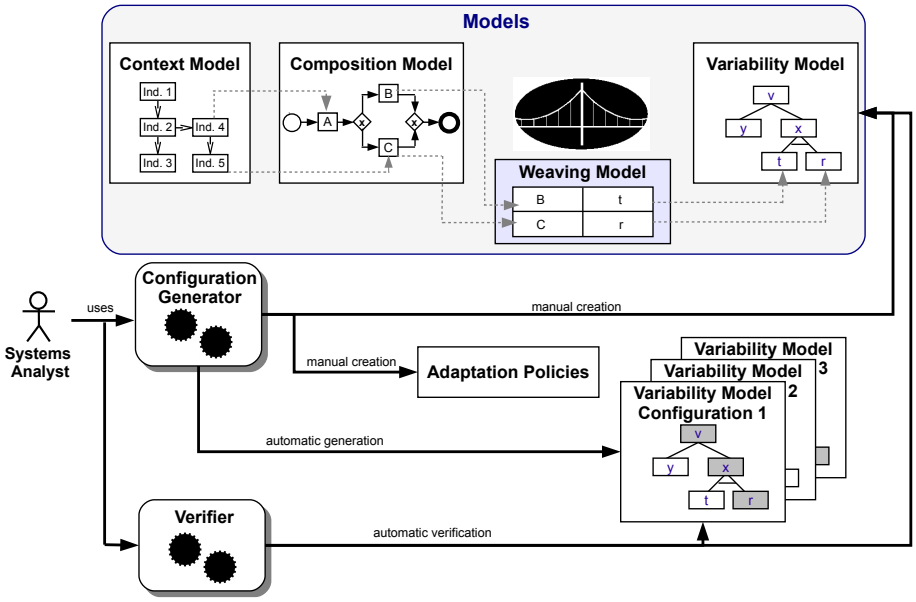


Figure 3: Preparation Phase

Two tools support this phase. First, our MOSKitt4SPL tool [1] implements the Configuration Generator. It supports the manual creation of variability models and adaptation policies. With this information, it automatically generates the set of variability model configurations. Afterwards, we use our GNU Prolog tool [2] (Diaz and Codognet, 2001, Diaz et al., 2012) to implement the Verifier that is in charge of verifying the variability model and the generated configurations against the variability model.

4.1. Preparation Phase Steps

The importance of software design has been discussed for quite a long time (Miller, 1989). A good software design is even more important when the created models are used to guide dynamic adaptations. Therefore, it is necessary to count on a strong basis at design time to adequately support necessary adjustments at runtime. In this section, we propose a detailed set of steps in the Preparation Phase to create the abstractions that will guide dynamic adaptation of service compositions (see Figure 4).

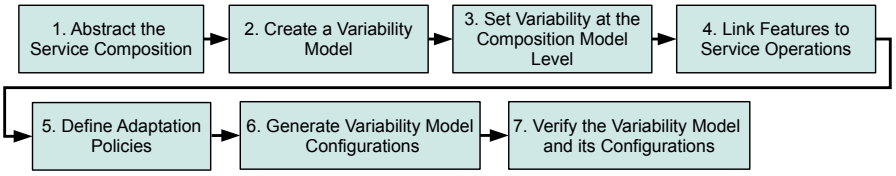


Figure 4: Preparation Phase steps

Step 1: Abstract the Service Composition

In this step, we propose the creation of an initial composition model to abstract the underlying service composition (such as the one in Figure 1). The composition model is causally connected to the underlying service composition. In this work, a BPMN model was chosen to represent the elements in a service composition because BPMN is suitable to express sequences and dependencies among Web services and composite services (Ayora et al., 2012). In order to specify queries against this model at runtime, it is expressed in XMI format and processed by the software infrastructure provided by the Eclipse Modeling Framework (EMF) [3].

Step 2: Create a Variability Model

Even though the initial composition model represents the underlying service composition, it lacks semantics for variability. Therefore, it is necessary to count

1. http://www.pros.upv.es/m4spl/.
2. http://www.gprolog.org/.
3. http://www.eclipse.org/modeling/emf/.

on feasible, semantically-rich, and coarse-grained variability representations in service compositions.

In this step we propose to create a variability model to describe variants in which a service composition can evolve. These variants may provide better Quality of Service (QoS), offer new services that did not make sense in the previous context, or discard some other services (Morin et al., 2009, Cetina et al., 2009). Our approach requires a variability modeling technique to implement the variability model, such as feature modeling (Kang et al., 1990), the Common Variability Language (CVL) (Haugen et al., 2008), or any domain-specific language to express variability. Feature modeling was chosen for variability modeling and analysis because it can offer coarse-grained variability management of service compositions and it has good tool support for variability reasoning (Mazo, 2011).

In a *feature model*, features are hierarchically linked in a tree-like structure through variability relationships such as optional, mandatory, and alternative (Kang et al., 1990). Also, in this kind of model there is only one *root feature* on which all the other features depend, and primitive features are the *leaves* and compound features are the *interior nodes*. In our approach, features represent the functionalities of the Web-service-based system in a coarse-grained fashion. Therefore, dynamic adaptations are carried out to keep the features of the system at runtime when context changes are faced.

Figure 5 shows the feature model for our running example. Certain features are appointed as variants that may be used to solve problematic context events and preserve the functionality of the service composition at runtime. For example, the UPS Shipping, the FedEX Express, and the DHL Delivery features are variants that can be used during execution to accomplish the shipment functionality. The variability model also has variation points that express decisions leading to different variants at runtime. Since only one variant can be chosen at a time in a particular variation point, there is an alternative relationship between a variation point and its variants (e.g. the Shipment variation point).

Different techniques can be used to create a feature model that is aligned with the elements in a composition model (Alférez and Pelechano, 2011b, Bae and Kang, 2007). The feature model can be manually created with MOSKitt4SPL. As the composition model, the feature model is also specified in the XMI format in order to reason about it at runtime (Cetina et al., 2009). The *current configuration* concept expresses the set of features in the feature model with "active" state at a particular time. Thus, the current configuration indicates the functionalities that are provided by a composite service at a specific moment. For instance, the features in gray in Figure 5 express the current configuration when the system starts in our running example.

Step 3: Set Variability at the Composition Model Level

The initial composition model created in Step 1 does not have semantics for variability. Nevertheless, this model should be able to change at runtime when facing arising context events. Also, a service composition dictates an ordered main workflow that has to be preserved in the composition model after

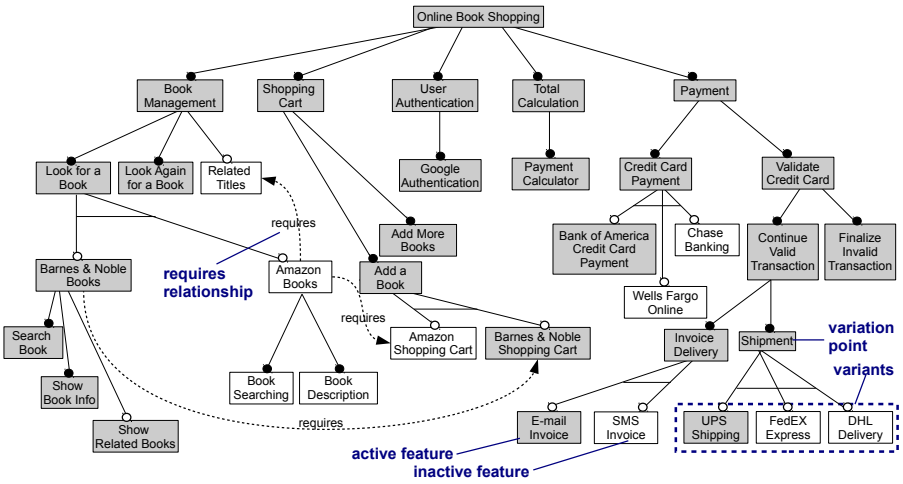


Figure 5: Feature model for the online-book-shopping running example

adaptations have taken place (e.g. user authentication is carried out first, then payment). In order to support these situations, we propose the creation of a *base composition model* that extends the initial composition model with semantics for variability and preserves the main workflow during adaptations. In our previous work (Ayora et al., 2012), we specify a set of commonalities (i.e., elements that are shared by all the configurations of the service composition) and variation points in the base composition model. The reasoning about what is common and variable in the base composition model is supported by the knowledge in the variability model created in Step 2. In addition, it is necessary to specify variants that can be bound into the variation points of the base composition model. For instance, Figure 6 shows the base composition model and two variants in our running example. At runtime, commonalities and the main workflow remain constant. However, the variation points can be bound with different BPMN variants.

Step 4: Link Features to Service Operations

In our approach, the features in the variability model are dynamically activated or deactivated to reach a particular configuration of the service composition. Therefore, it is necessary to count on a connection between the elements in the variability model and low-level service operations. Since the composition model represents the operations in the service composition at any moment, the definition of a bridge between the elements in the variability model and the elements in the composition model could be used to support dynamic adaptations in the underlying service composition.

In order to define this bridge, different techniques can be used, such as the template approach based on superimposed variants (Czarnecki and Antkiewicz, 2005). In this work, we propose the creation of a weaving model (Alférez and

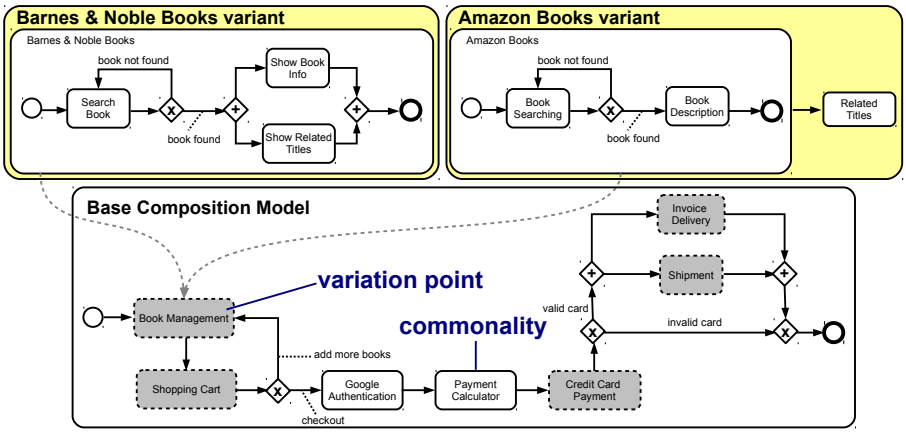


Figure 6: Base composition model and two BPMN variants in our running example

Pelechano, 2011a, Del Fabro et al., 2006) since it is supported by the ATLAS Model Weaver [4] tool. The weaving model can be considered as a *mapping model* (Czarnecki and Eisenecker, 2000) that defines the mapping relationships between a *problem space model* (i.e., the variability model) and a *solution space model* (i.e., the composition model). In our running example, each link (or mapping) has the following endpoints: the first endpoint refers to features in the feature model; the second endpoint refers to BPMN variation points and commonalities in the base composition model, and BPMN variants. Figure 7 shows a fragment of the weaving model for the running example. It highlights the relationship between the Barnes & Noble Books feature and a set of BPMN elements in the Barnes & Noble Books variant [5].

Step 5: Define Adaptation Policies

In order to define adaptation policies, first it is necessary to define the context events that may trigger adaptations. Then, it is necessary to define the actions to be carried out in the service composition to solve particular context events. These two substeps are described as follows.

Step 5.1: Define Context Conditions

In order to solve the need for expressing the context in a way that supports formal reasoning of its current status and possible arising situations, we propose an ontology-based context model that leverages Semantic Web technology. Specifically, we make use of the Web Ontology Language (OWL) [6] to support

4. http://www.eclipse.org/gmt/amw/.
5. The ATLAS Model Weaver tool requires only one destination model. Therefore, at the implementation level we have created one destination BPMN model with a *pool* that contains the base composition model, and additional *pools* for each variant.
6. http://www.w3.org/TR/owl-ref/.

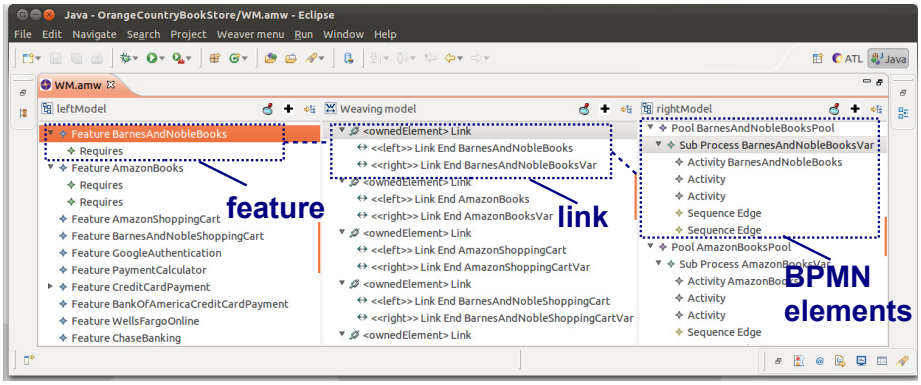


Figure 7: A fragment of the weaving model for the running example

the formal analysis of the contextual information that is captured by the Context Monitor. OWL extends the expressivity of the Resource Description Framework (RDF) [7] by adding an additional layer of semantics on top of RDF. With RDF, the contextual knowledge can be decomposed into small pieces, with some rules about the meaning (or semantics) of those pieces.

Figure 8 shows the ontology for our running example visualized in OntoGraf [8]. The *Thing* superclass has two classes, namely *CompositeWebService* and *WebService*, which respectively represent composite services and Web services. Classes are interpreted as sets of individuals. Individuals represent specific composite service and Web service operations. Individuals keep runtime information for service operations represented in the base composition model and in variant models (see Step 3) to collect contextual information no matter what the current configuration of the service composition is.

Each individual has a set of datatype properties (i.e., relations between instances of classes and RDF literals or XML schema datatypes). These datatype properties are used to keep track of context information. Since dynamic adaptations are triggered when a particular SLA is at risk, datatype properties keep information about qualitative properties in the *QoS context category* proposed by (Bandara et al., 2009). In (Bandara et al., 2009), dynamically relevant qualitative properties in the *QoS context category* are organized into the following four groups based on the type of measurement performed by each attribute: *runtime attributes* such as availability and performance, *financial/business attributes* such as execution cost, *security attributes* such as non-repudiation, and *trust attributes* such as good reputation of service operations.

Since our running example is concerned with the availability and performance of the service composition, each individual of the *WebService* class has the following datatype properties (or QoS runtime attributes): isAvailable

7. http://www.w3.org/RDF/.
8. http://protegewiki.stanford.edu/wiki/OntoGraf.

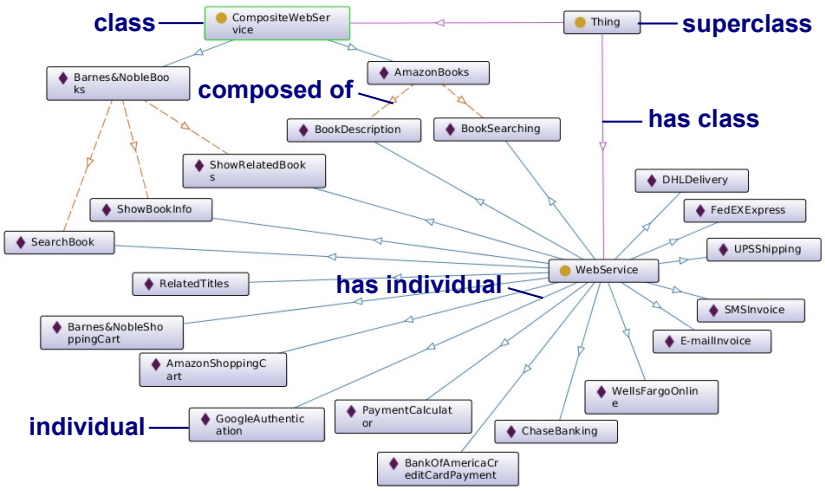


Figure 8: Context model for the running example

indicates if the service operation is currently available (it is a Boolean value); hasResponseTime indicates the current response time in milliseconds to have access to a particular Web service operation; and hasExecutionTime indicates the current execution time in milliseconds that a Web service operation takes to execute a job. The individuals of the *CompositeWebService* class have the same datatype properties of the *WebService* class' individuals. In this case, the isAvailable datatype property is TRUE if all the compound service operations are available. Also, the values of the hasResponseTime and hasExecution-Time datatype properties are the averages of the response times and execution times of the compound service operations, respectively. The information that can be kept in datatype properties is very versatile. For instance, they can be also used to track network performance when orchestrating remote service operations (e.g. latency, throughput, and bandwidth).

In this work, we propose that the value of particular datatype properties in complicated scenarios (e.g. when it is necessary to figure out if a service composition is under attack) can be inferred by means of logic rules over context information. For simple service compositions, rules can be obtained from human experts by collecting empirical data from the current service composition or by analyzing collected data to discover the symptoms of problematic situations. In complex service compositions, it is even possible to use methods for generating rules from data (e.g. with heuristics or neural networks). For instance, it is possible to infer with Jena 2 [9] that a service operation is under attack in a T1 network when the execution time is extremely high, the latency is higher than five milliseconds, and the bandwidth is lower than 1.544 Mbps (see Listing 1).

9. http://incubator.apache.org/jena/.

```
@prefix j.0: http://my.ontology#
[underAttack: (?s rdf:type j.0:WebService)
        (?s j.0:hasExecutionTime ?c) greaterThan(?c,30000)
        (?s j.0:hasLatency ?c) greaterThan(?c,5)
        (?s j.0:hasBandwidth ?c) lessThan(?c,1.544)
        -> (?s rdf:type j.0:underAttack)
```

Listing 1: Rules to infer that a service operation is under attack

In order to solve the need for examining the compliance of certain situations in the context, *context conditions* are extracted from the context model as Boolean expressions. A context condition works as an SLA. If a context condition is accomplished (i.e., an SLA is violated), then an adaptation is triggered on the service composition to deal with the arising situation. Each context condition is represented as a RDF triple in the form of (subject, predicate, object). The *subject* (i.e., an ontology individual) denotes a resource (i.e., a Web service or a composite service operation), and the *predicate* expresses a relationship between the subject and the object (i.e., the value of a datatype property). Two context conditions in our running example are the following: 1) ***B&NUnavailable*** *= (Barnes&NobleBooks, isAvailable, false)*, which is triggered when the Barnes & Noble Books composite service is currently unavailable; and 2) ***UPSHiExecTime*** *= (UPSShipping, executionTime, >1,500 ms)*, which is triggered when the current execution time of the UPS Shipping Web service operation is greater than 1.5 seconds.

The aforementioned context conditions are ideal for situations where the user is expecting an immediate response in short-running BPs. Nevertheless, context conditions can also be used to check context situations in long-running BPs that execute over an extended period of time and involve the coordination of different people (e.g. the process to return a book) (Torres et al., 2012). Context conditions can be manually defined in MOSKITT4SPL (see Figure 9).

Step 5.2: Define Resolutions

In SPL engineering, variability models focus on the efficient derivation of customized product variants that, once created, retain their properties throughout their lifetime. We argue that a service composition can activate or deactivate its own features dynamically at runtime by fulfilling certain context conditions. Therefore, we propose to use the *resolution* concept to represent the set of changes in a feature model triggered by a context condition. In other words, resolutions are the adaptation policies that express the transitions among different configurations of the service composition in terms of activation or deactivation of features. Basically, a resolution (R) can be expressed as a list of pairs (F, S) where each pair is made up of a feature (F) in a feature model (FM) and the state (S) of the feature. Each resolution is associated to a context condition (C). A feature's state is set to active or inactive: $R_C = \{(F, S) \mid F \epsilon [FM] \wedge S \epsilon \{Active, Inactive\}\}$ (Cetina et al., 2009).

In a different approach, adaptations are triggered to reach a configuration with the maximum overall utility (Esfahani et al., 2011). To this end, utility

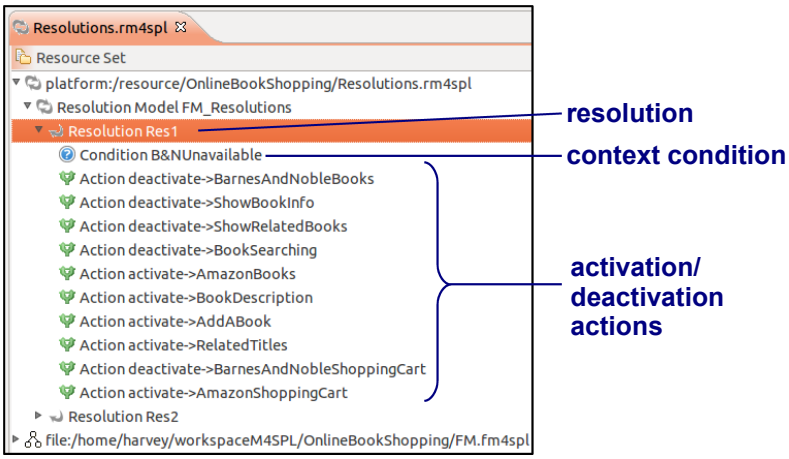


Figure 9: Defining context conditions and resolutions in MOSKitt4SPL

functions provide the objective function for self optimization, mapping each possible state (e.g. the set of active service operations in a particular configuration) of an entity (e.g. a service composition) into a real scalar value (Tesauro and Kephart, 2004). For instance, the utility function of a service composition can be calculated with the quality attributes of each individual service operation in the service composition (i.e., the utility of combining service operations). The major problem with utility functions is that they can be extremely hard to define, as every aspect that influences the decision by the utility function must be quantified. On the other hand, our approach offers an easy, effective, and abstract way to define adaptation policies.

For example, the resolution for the *B&NUnavailable* context condition is as follows: $R_{B\&NUnavailable}$ = *{(Barnes & Noble Books, Inactive), (Search Book, Inactive), (Show Book Info, Inactive), (Show Related Books, Inactive), (Amazon Books, Active), (Book Searching, Active), (Book Description, Active), (Related Titles, Active), (Barnes & Noble Shopping Cart, Inactive), (Amazon Shopping Cart, Active)}*. Although the Barnes & Noble Shopping Cart may not be unavailable when *B&NUnavailable* occurs, $R_{B\&NUnavailable}$ deactivates this functionality and activates the Amazon Shopping Cart in order to respect *requires* relationships in Figure 5. MOSKitt4SPL provides a resolution editor to ease the specification of resolutions (see Figure 9).

In this work, we increase the expressiveness of resolutions with *composite context conditions* (C_{comp}), which can be defined as follows:

$$C_{comp} = \{(C_1 \vee C_2) \wedge (C_3 \vee C_4) \wedge \ldots \wedge C_n\}$$

Each C_n in the C_{comp} represents a specific context condition. AND and OR logical operators are used to connect each C_n. For example, the following resolution deactivates the UPS Shipping functionality and activates the DHL Delivery functionality:

$$R_{UPSHiExecTime \wedge DHLExeTimeLowerThanFedEX}$$

This resolution is triggered when both context conditions occur. In case the *UPSHiExecTime* context condition occurs, only one of two variant features can be activated in the Shipment variation point (FedEX Express or DHL Delivery in Figure 5). Therefore, the *DHLExeTimeLowerThanFedEX* context condition is used to activate the DHL Delivery functionality when it has lower execution time than the FedEX Express service operation.

Step 6: Generate Variability Model Configurations

The execution of a DSPL's architecture, which describes a dynamic service composition, can be abstracted as a highly connected state machine where *states* are the possible variability model configurations and *transitions* the migration paths among configurations. In order to facilitate the generation of variability model configurations, we implemented in MOSKITT4SPL the functionality to fully generate the implicit adaptation space as a state machine model from the variability model and the set of resolutions. The generated adaptation space for the running example can be downloaded from our website (Alférez et al., 2012).

Step 7: Verify the Variability Model and its Configurations

Manual reasoning of DSPLs is an error-prone, tedious and sometimes infeasible task (Benavides et al., 2005). Therefore, we propose to use CP to help automate this reasoning task. In order to do so, we transform the feature model, the context model, and the constraints that relate them, into a constraint program over finite domains. Details about the transformation rules and an algorithm that transforms feature models into constraint programs can be found in (Mazo et al., 2011). Details about the transformation of context models into a constraint program and their relationships with variability models are provided in (Sawyer et al., 2012). The composition model is not represented as a constraint program because its relationships with the variability model are implemented by means of the weaving model.

Once the variability model is transformed into a constraint program, the model and its configurations can be automatically verified with a solver (Salinesi et al., 2010). In this step we present four criteria to verify, at design time, variability models intended to be used for dynamic adaptation: accuracy of the variability model, non-existence of dead features, stability, and semi-aliveness. If the verification process finds any anomaly in the variability model, it should be fixed at design time to avoid the undesirable effects of these anomalies in the service composition at runtime. The verification of the composition model and the mappings between the variability model and the composition model are out of the scope of this paper (cf. (Gröner et al., 2011, Van der Aalst et al., 2010) for details).

We take advantage of GNU PROLOG to implement our collection of verification criteria by means of algorithms that avoid expensive computations and reuse the precedent results to avoid wasting time in unnecessary operations (Salinesi and Mazo, 2012). GNU PROLOG is a programming language that

includes a powerful constraint solver over finite domains. It uses similar algorithms as other constraint programming solvers and the unification operation to work with Prolog facts. GNU Prolog uses very efficient algorithms based on consistency, propagation, and backtracking techniques. The non-determinism used by GNU Prolog allows to calculate several solutions for the same constraint problem and obtain the best solution (based on several criteria, such as computing time, number of backtracks, and a maximization function) that satisfies the constraints.

1. Accuracy of the variability model

This operation verifies that the variability model is not void and has enough variability to be considered a real variability model. On one hand, a void model does not permit any legal configuration. On the other hand, a variability model that allows only one legal configuration, is a model that has not enough variability to be considered a variability model; it is instead the model of one configuration. In other words, this operation verifies that there are at least two configurations that satisfy the constraints of the variability model. So far, two alternative techniques have been proposed to implement this operation: 1) calculating the number of configurations (Van den Broek and Galvão, 2009), and 2) asking for two configurations that meet the constraints of the variability model (Trinidad et al., 2008, Salinesi and Mazo, 2012). The former approach is unnecessarily costly (if possible at all). In fact, there is no need to compute all solutions to prove that the model has at least two solutions. To implement this verification operation, GNU Prolog is queried for any two configurations.

Listing 2 shows the algorithm for this operation. If the variability model is not void, the algorithm checks if the solver finds another valid solution and produces a corresponding message. The execution of this algorithm on our running example gives an answer, which says it is not void.

```
accuracy(VariabilityModel VM, Solver S) {
  S.load(VM);
  Answer1 = S.getOneSolution();
  If (Answer1 != FALSE) {
    Write("The variability model is not void");
    Answer2 = S.nextSolution();
    If (Answer2 != FALSE) {
      Write("The variability model has enough variability");
    } Else {
      Write("The variability model has only one valid
          solution");
    }
  } Else {
    Write("The variability model is void");
  }
}
```

Listing 2: Algorithm of the verification criterion for accuracy

2. Non-existence of dead features

This operation verifies that there are not any dead features in the variability model. A feature is dead if it cannot appear in any configuration of the variability model. Features can become dead by: 1) the exclusion with another feature appearing in all configurations (or core features) (Von der Massen and Lichter, 2004, Trinidad et al., 2008, Van den Broek and Galvão, 2009); 2) the exclusion and requires dependency, of a feature with another one, at the same time (Von der Massen and Lichter, 2004, Van den Broek and Galvão, 2009); and 3) the feature or its attributes are wrongly constrained (e.g. < 1) (Mazo et al., 2012).

In this verification operation, we assume the convention that a feature is selected in a particular configuration when it is set to 1, and is not selected when it is set to 0. Thus, in order to implement this operation, we evaluate if each feature can take the value of 1 in any configuration, and if it is not the case, the corresponding feature is dead. In order to find dead features, we query GNU Prolog for one result where each feature is different to 0. Then, if the solver finds a solution, we reuse these results to avoid querying features that we already know (from the solution) are not dead.

Listing 3 shows the algorithm for this operation. First, this algorithm creates a list of the features whose dead or non-dead condition is yet to be assessed (*DeadFeatureList*). Then, it queries for a solution (based on features for which we still ignore if they are dead or not) and sieves the selected (and thus alive) elements from this list. The test is repeated until all alive features are sieved.

```
deadFeatures(VariabilityModel VM, Solver S) {
  S.load(VM);
  DeadFeatureList = all features in VM;
  For (each feature F in DeadFeatureList) {
      Solution = S.getOneSolution(F = 1);
      If (Solution = FALSE) {
        Write ("The feature " + F + " is dead");
      } Else {
        Erase F and all the other features
        with values equal to 1,
        which are obtained in
        Solution from DeadFeatureList;
      }
  }
}
```

Listing 3: Algorithm of the verification criterion for non-existence of dead features

For example, in order to know if the Amazon Shopping Cart feature is dead or not, it is sufficient to query the solver for a solution with Amazon Shopping Cart = 1. The algorithm provides the following solution: E1 = [Shopping Cart = 1, Amazon Shopping Cart =1...]. This result means not only that the Amazon Shopping Cart feature is not dead, but also that the other features with values equal to 1 are not dead. Therefore these features can be sieved from the list

of dead features. The purpose of the *DeadFeatureList* is to reduce the number of queries by reusing the results obtained from the solver. For instance, in our running example only eight queries were necessary to evaluate all the features, in contrast to 36 queries (there are as many queries as features in the variability model).

3. Stability

This operation verifies that for every combination of context variable values that represent a context, there is at least one legal configuration that satisfies all the variability constraints. However, in complex systems there may be thousands of possible configurations (Khan et al., 2008), which can lead to configurations that are mutually inconsistent, contradictory, or simply unachievable. In our approach, features are represented as Boolean variables that can be satisfied or not by variability dependencies and context values. If no configuration exists that satisfies all the variability dependencies for a given context, the systems analyst either needs to rethink the context model or the variability model. If there are several possible configurations for a particular context, the system respects our stability verification criterion.

Listing 4 shows the algorithm for this operation. This operation activates each possible combination of context variables (a context variable is activated when its value is greater than 0) and queries the solver for one solution that satisfies the extra constraints imposed by the activated context variables. No finding a solution for a particular combination of context variables means that the variability model cannot offer a valid configuration for the context at hand and a corresponding message is shown to the user.

```
stability(VariabilityModel VM, ContextModel CM, Solver S) {
  S.load(VM);
  S.load(CM);
  C1...Cn is the list of variables in CM;
  Solution = S.getOneSolution(C1>0,...,Cn>0);
  If (Solution = FALSE) {
    Write (VM + " is NOT stable for all the possible values
        that can take the variables of "+ CM);
  } Else {
    Write (VM + " is stable for the combination of the
        context variable values of "+ CM);
  }
}
```

Listing 4: Algorithm of the verification criterion for stability

For example, let us suppose the following current configuration: E1 = [DHL Delivery = 1, UPS Shipping = 0, FedEX Express = 0...]. All of a sudden, there is a new context event: DHLDelivery_HasExecutionTime = 25,000 (all the other context variables keep their current values). According to this information, the following configuration constraint should be triggered: (DHLDelivery_HasExecutionTime > 20,000 $\bigwedge$ UPSShipping_HasExecutionTime < FedEX-Express_HasExecutionTime) ==> (DHL Delivery = 0 $\bigwedge$ UPS Shipping = 1). In

this case, UPS Shipping is 1 in the variability model (if UPSShipping_HasExecutionTime < FedEXExpress_HasExecutionTime). In this example, the combination of context variables results in a legal configuration that satisfies the variability constraints.

4. Semi-aliveness

This operation verifies whether or not the variability model is sensitive to changes in the context (i.e., it verifies that the variability model is able to respond to changes in the context). To this end, this operation verifies that there is a way to change from one configuration *C(E1)* to another *C(E2)*, such that *E1-E2 = [0, 0, 0, x, ... , 0]* (i.e., when one context variable has changed its value, the configuration should also change to adapt itself to the new context). E is a given context represented as a tuple of N context variables, where each variable takes a particular value of its domain, e.g. *TransmissionRate = 2 Mbps*. Thus, when a value of the tuple *E1-E2* is equal to 0, it means that the corresponding domain variables remain the same.

For instance, the current configuration of the variability model in Figure 5 changes to a new one when the *B&NUnavailable* context condition is accomplished (i.e., when the context changes). However, the variability model configuration does not have to change always when the context changes since the same configuration can satisfy (even in an optimal way) the constraints induced by the context and of course, those imposed by the variability model. We simulate these changes, variable by variable, and produce the different configurations when the context changes. Then, the systems analyst can define, according to his or her expertise, the adaptation ability of the service composition and corrects it if necessary.

Listing 5 shows the algorithm for the semi-aliveness criterion. However, this operation is computationally complex because it combines the values of a collection of variables. In order to improve its performance, we divide the collection of context variables into partitions as small as possible. For instance, in the case where context variables are Boolean, our algorithm searches for 2^k * n, instead of 2^n solutions, where k is the maximum number of variables into n partitions. Each partition corresponds to a collection of context variables, which do not depend from the rest of context variables. For instance, two partitions in our running example are as follows (each one corresponds to a collection of related functionalities): *Credit Card Payment Partition* – Bank of America Credit Card Payment, Wells Fargo Online, and Chase Banking; and *Invoice Delivery Partition* – Email Invoice, SMS Invoice*;* Shipment Partition: UPS Shipment, FedEX Express, DHL Delivery.

Let us suppose a current configuration when the Barnes & Noble Books feature is active (equal to 1) and the Amazon Books service is inactive (equal to 0). Thus, the current configuration is as follows: E1 = [Barnes & Noble Books = 1, Barnes & Noble Shopping Cart = 1, Amazon Books = 0, Amazon Shopping Cart = 0, Related Titles = 0...]. At a certain moment, the system perceives that the transmission rate of the Barnes & Noble Books service operations is inferior to 2 Mbps. In this case, the Amazon Books feature is activated (equal to 1) and the

Barnes & Noble Books feature is deactivated (equal to 0), as represented in the following configuration constraint: (TransmissionRate_BarnesAndNobleBooks < 2) ==> (Amazon Books = 1 ⋀ Barnes & Noble Books = 0). Thus, the new configuration that is proposed by the solver is as follows: E2 = [Barnes & Noble Books = 0, Barnes & Noble Shopping Cart = 0, Amazon Books = 1, Amazon Shopping Cart = 1, Related Titles = 1...] [10]. The difference between E1 and E2 can be calculated by taking the absolute value of each result ([1, 1, 0, 0, 0...] - [0, 0, 1, 1, 1...]). It produces the following list of results: [1, 1, 1, 1, 1...]. The sum of this list of values is different from zero. This means that our running example is sensitive to the changes in the context (decreasing transmission rate). It is worth noting that when all possible changes in the context model are simulated, and the solution that is given by the solver is always the same, the variability model is insensitive to the context. The semi-aliveness operation also identifies this problem.

```
semi-aliveness(VariabilityModel VM, ContextModel CM, Solver
    S) {
  S.load(VM);
  S.load(CM);
  Vector currentSolution = S.getOneSolution();
  For each Partition P in CM {
    For each combination C of context variables values into
        P {
      Solution newSol = S.getOneSolution (C);
      If (newSol != FALSE) {
        Vector delta = currentSolution - newSol;
        If(the sum of values of delta != 0){
          Write ("The variability model is sensitive to the
              Partition " + P);
          currentSolution = newSol;
          Exit from the current "For loop" and continue with
               the next Partition;
        }
      } Else {
        Write("The variability model is insensitive to the
            Partition " + P);
      }
    }
  }
}
```

Listing 5: Algorithm of the verification criterion for semi-aliveness

In case all the variables in the context and variability models are Boolean (which is not the case in our running example), it is even possible to propose an

10. The Amazon Shopping Cart feature and the Related Titles feature are also activated thanks to the following implication instruction: Amazon Books ==> Related Titles, Amazon Books ==> Amazon Shopping Cart.

improved algorithm. Instead of asking the solver for several solutions (until two solutions differ on at least one context variable and one variability variable), it is possible to find the first solution of a unique CSP. This algorithm is described on our website (Alférez et al., 2012).

5. Dealing with Dynamic Adaptation Phase

An ad-hoc approach to build dynamic adaptive service compositions is to use existing variability mechanisms (e.g. if-statements or method dispatch) directly in the architecture. However, the arising complexity of dynamic adaptable software can limit the number of dynamic reconfiguration points to a few well-defined ones due to the lack of appropriate approaches (Dinkelaker et al., 2010).

In the Dealing with Dynamic Adaptation Phase, we reuse the knowledge captured in variability models, which have proven useful in mass-production domains (Coplien et al., 1998), to describe the variants in which the service composition can be adapted. In response to changes in the context, the system itself can query these models to determine the necessary modifications in the service composition. Specifically, the use of variability models at runtime has the following benefits:

- The modeling effort made at design time is not only useful for producing the service composition but also provides a rich semantic base for autonomic behavior during execution.
- The variability model is causally connected to the underlying service composition. Therefore, this model provides up-to-date information to drive subsequent adaptation decisions.
- The same model representation that is used at design time is kept at runtime. This avoids the need for technological bridges, making it possible to apply the same technologies used at design time to manipulate variability models at runtime.

The Model-Based Reconfigurator leverages variability models at runtime for guiding dynamic adaptations of service compositions. When features are activated or deactivated at runtime in the variability model due to changes in the context, a DSPL architecture supports the dynamic service recomposition. The Model-Based Reconfigurator is implemented with our Model-based Reconfiguration Engine for Web Services (MoRE-WS) (Alférez and Pelechano, 2011a). MoRE-WS is an extension of MoRE (Cetina et al., 2009), which has been successfully applied to the smart-home domain. Although MoRE also uses variability models to guide dynamic adaptations, MoRE-WS extends MoRE in two aspects: 1) in MoRE, adaptation policies are not verified ahead. In this work, we use GNU Prolog at design time to detect problems in the variability model and in its configurations; and 2) MoRE makes use of low-level instructions for implementing and executing reconfiguration actions. Contrarily, MoRE-WS executes reconfiguration actions in a higher abstraction level (i.e., at the composition model).

5.1. Computing Infrastructure for Dynamic Adaptations

A computing infrastructure deals with the dynamic adaptation of service compositions (see Figure 10). This infrastructure is based on the components of IBM's reference model for autonomic control loops (which is sometimes called the MAPE-K loop) (IBM, 2006), namely Monitor, Analyze, Plan, Execute, and Knowledge. In our case, the knowledge that is managed in this loop is based on models at runtime. First, measurement instruments collect information from the context of the service composition. In the Monitor component, this information is processed by the Context Monitor. In the Analyze component, MoRE-WS updates the context model with the observed context information and decides if any context condition has been accomplished. If any context condition has been accomplished, then an adaptation is requested. In the Plan component, MoRE-WS executes adaptation policies that activate or deactivate features in the variability model. The adapted configuration of the variability model is used to automatically create a reconfiguration plan with the adaptation actions to be carried out on the composition model. The adapted composition model is dispatched to the Execute component. In this component, MoRE-WS provides the mechanisms to map the elements in the adapted composition model to WS-BPEL code fragments. The generated WS-BPEL code and other required artifacts are deployed at runtime in the Execution Engine.

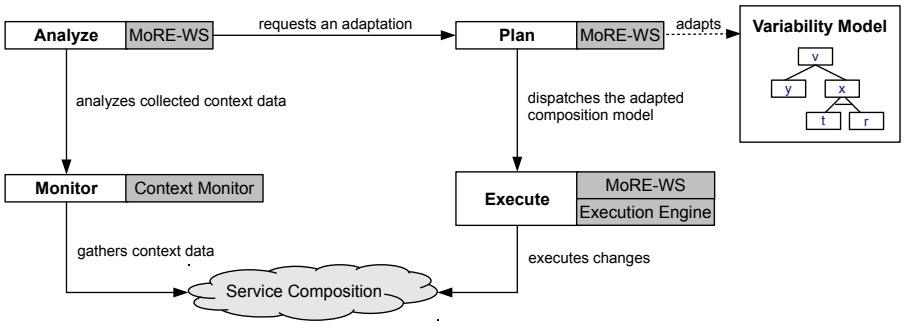


Figure 10: Computing infrastructure to deal with the dynamic adaptation of service compositions

In our previous research (Alférez and Pelechano, 2011a), we described how the Context Monitor is implemented with SALMon (Ameller and Franch, 2008). In order to count on a constant representation of the context, MoRE-WS periodically updates the context model according to the information that is collected by SALMon. We propose the following strategy to avoid the saturation of the system when several problematic context events arise in a tight time frame: 1) in each observation, MoRE-WS retrieves a list with an ordered sequence of new context events ($contEvent_1, contEvent_2, \ldots, contEvent_n$). It is possible to retrieve an ordered sequence of context events because every context event has a unique ascending identification number; 2) MoRE-WS sequentially evaluates the set of context conditions ($contCond_1, contCond_2, \ldots, contCond_n$), which

can be triggered by the set of context events. Therefore, only one adaptation is triggered at a time for each collected problematic context event.

MoRE-WS uses the updated context model as a base to determine whether or not any context condition has been accomplished. By means of the SPARQL Protocol and RDF Query Language (SPARQL) [11], we have implemented the following operations to modify the context model and reason about context conditions: 1) the *Insert Context Event* operation updates the context model according to new context events. This operation is implemented with the SPARQL INSERT form, which inserts new triples in the RDF graph of an ontology. For example, the following query inserts information about the availability of the Barnes & Noble Books service operation: PREFIX web: <http://my.ontology#> INSERT DATA {web: 'Barnes&NobleBooks' web:isAvailable 'false'}; 2) the *Evaluate Context Condition* operation evaluates if a context condition has been accomplished. This operation is implemented with the SPARQL ASK form to test whether or not a query pattern (e.g. a context condition) has a solution. For example, the following query evaluates the *B&NUnavailable* context condition: PREFIX web: <http://my.ontology#> ASK { web:Barnes&NobleBooks web:isAvailable 'false' }. When a context condition is TRUE, then MoRE-WS requests an adaptation on the variability model.

In the following subsections, we focus on the Plan and Execute components, where the variability model plays a fundamental role.

5.2. Planning the Adaptation

When an adaptation has been requested (i.e., after a context condition has been accomplished), MoRE-WS carries out the following steps to plan the adaptation of the service composition:

Step 1: Execute a Resolution

In this step, MoRE-WS looks for a resolution associated to a context condition (or a composite context condition), which has occurred, from the set of resolutions that have been defined at design time. Since EMF Model Query (EMFMQ) [12] can be used to manipulate and update models at runtime, MoRE-WS uses it to activate or deactivate features in a feature model at runtime according to resolutions. EMFMQ provides an API to construct and execute query statements in a SQL-like fashion. For instance, in Listing 6, the state of the model element that matches the *featureID* is set to either active or inactive.

In order to realize the new configuration of the variability model to transit to after a resolution has been applied, MoRE-WS uses the representation of the adaptation space (which has been generated at design time as a state machine). As a result, MoRE-WS has full control of the different possible variability model configurations and the transitions among them.

11. http://www.w3.org/TR/rdf-sparql-query/.
12. http://www.eclipse.org/modeling/emf/.

```
UPDATE statement =
    new UPDATE(
    new FROM(resource.getContents()),
    new WHERE(new EObjectAttributeValueCondition(
          featureModelPackagePackage.eINSTANCE.getID(),
          new StringValue(featureID))),
    new SET(setFeatureState());
```

Listing 6: EMFMQ operation for updating a feature's state

Step 2: Generate a Reconfiguration Plan

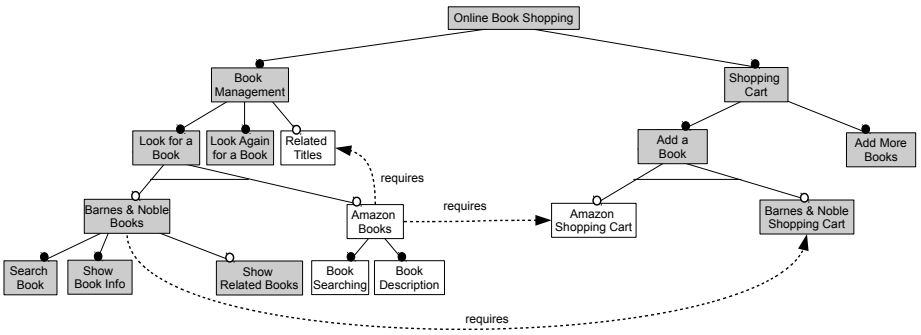
In this step, MoRE-WS creates a reconfiguration plan, which contains a set of reconfiguration actions to adapt the composition model according to the new configuration of the feature model (which has been modified by a resolution). Reconfiguration actions are stated as *composition model increments* ($CM\triangle$) and *composition model decrements* ($CM\nabla$). These operations take a new configuration of the variability model as input, and they calculate the modifications to the composition model by adding ($CM\triangle$) or removing ($CM\nabla$) variants from the base composition model. The modified composition model will eventually cause the adaptation of the WS-BPEL code that orchestrates the service operations.

In order to generate reconfiguration actions, MoRE-WS queries the weaving model to realize the mappings between the features that are active in the new configuration of the feature model and their related BPMN elements. In this way, a given service operation, which is represented in the composition model, will be invoked in the adapted service composition if and only if its related feature in the feature model configuration is active. That is, the composition model is adapted through the activation or deactivation of features. The queries on the weaving model are carried out by means of EMFMQ (Alférez and Pelechano, 2011a).

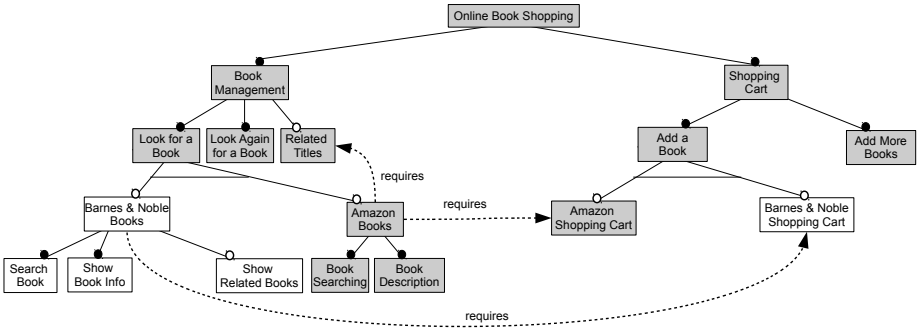
For example, when MoRE-WS applies $R_{B\&NUnavailable}$ to the initial configuration of the running example (depicted in Figure 1), the resulting reconfiguration plan is the following: $CM\nabla$ = *{Barnes & Noble Books variant, Barnes & Noble Shopping Cart variant}* and $CM\triangle$ = *{Amazon Books variant, Amazon Shopping Cart variant}*. These actions express how to reorganize elements in the composition model to move from one configuration when the Barnes & Noble Books composite service operation fails to another configuration when this operation is replaced by the Amazon Books, Related Titles, and Amazon Shopping Cart Web service operations (according to Figure 6).

Summarizing, Figure 11 shows the model adaptation process when a context condition has been accomplished. *Section a* shows a fragment of the initial variability model configuration in the running example. At a particular point in time, the *B&NUnavailable* context condition is fulfilled. As a result, MoRE-WS looks for a resolution to solve this situation and finds $R_{B\&NUnavailable}$ that triggers the activation and deactivation of features in the variability model (*section b*). The application of this resolution causes the system to transit from

a) Initial variability model configuration. Features in gray are active (*configuration 0*):



b) New variability model configuration after $R_{B\&NUnavailable}$ has been executed. Features in gray are active (*configuration 1*):



c) Modified composition model after a reconfiguration plan ($CM\triangle$, $CM\triangledown$) has been executed:

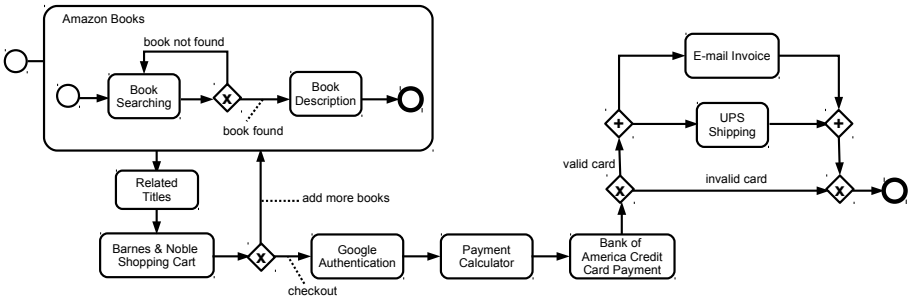


Figure 11: Model adaptation process

configuration 0 to *configuration 1*. Finally, MoRE-WS creates a reconfiguration plan and executes it to modify the composition model (*section c*).

5.3. Executing the Adaptation

In order to materialize the changes at the composition model level into the running service composition, this phase is divided into two aspects: 1) MoRE-

WS provides the mechanisms to map BPMN variants to WS-BPEL code fragments. These fragments can be added or removed from the WS-BPEL composition schema, which orchestrates the service composition; and 2) MoRE-WS deploys at runtime the generated WS-BPEL code and other required artifacts in the Execution Engine. These aspects are described in the following subsections.

5.3.1. Mapping BPMN Variants to WS-BPEL Code Fragments

The creation of the WS-BPEL composition schema is guided by the information contained in the adapted composition model. In our previous research, we used the BABEL Java tool [13] to translate BPMN models into WS-BPEL code (Torres et al., 2012). However, we discovered that model-to-model and model-to-text transformations in BABEL are unfeasible at runtime because they take around 95% of the total time required for the adaptation (from discovering a context condition to reconfiguration) (Alférez and Pelechano, 2012a).

In this section, we propose a faster solution to reflect the changes in the composition model into WS-BPEL code. This solution is composed of the following three steps (see Figure 12): 1) MoRE-WS uses EMFMQ to discover the variants that have been added to the base composition model. Each variant maps to a WS-BPEL code fragment, which is stored in a repository (i.e., a directory); 2) MoRE-WS selects the set of WS-BPEL code fragments, which map to the variants that have been used; and 3) MoRE-WS injects the WS-BPEL code fragments into variation points in the WS-BPEL template. These variation points (e.g. Shopping Cart in gray) can be implemented with different variant WS-BPEL code fragments (e.g. with code that invokes the Barnes & Noble or the Amazon Shopping Cart service operations). The WS-BPEL template also indicates commonalities (e.g. Payment Calculator) that do not vary at runtime and are common to all the versions of the service composition.

5.3.2. Hot Deployment

After the WS-BPEL template has been filled up with WS-BPEL code fragments, MoRE-WS creates a deployment directory for all the relevant deployment artifacts. This directory contains the deployment descriptor (an XML file), the composition schema (i.e., the filled WS-BPEL template), and the Web Services Description Language (WSDL) files, which describe the functionality offered by the Web services. This directory is put into the Web application directory of the Apache Orchestration Director Engine (Apache ODE) [14]. Apache ODE was chosen as the Execution Engine because it is compliant with WS-BPEL and offers mature hot-deployment support. Instead of extending the functionality of the WS-BPEL engine, our approach is transparent to the engine (i.e., it is unchanged). Therefore, our approach could be used with other WS-BPEL engines.

13. http://www.bpm.scitech.qut.edu.au/research/projects/oldprojects/babel/tools/.
14. http://ode.apache.org/.

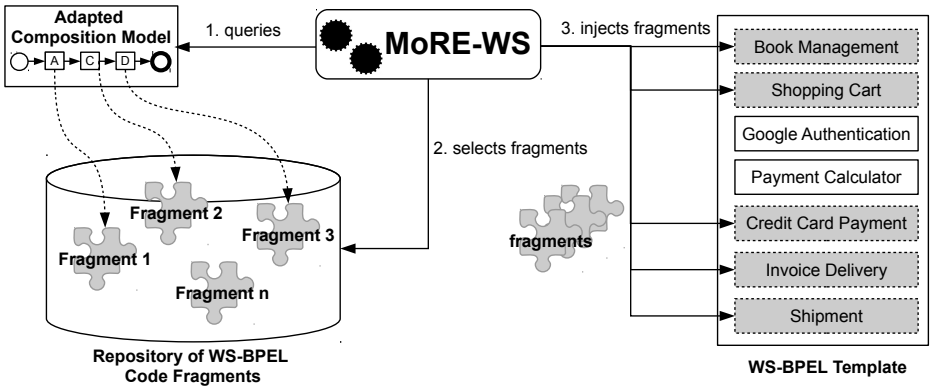


Figure 12: Mappings between BPMN variants and WS-BPEL code fragments

We have implemented a versioning strategy for the deployment directory to prevent Apache ODE from deleting all the running instances when a new composition schema is deployed. To this end, a new deployment directory with an increasing version number is deployed with every dynamic adaptation. New instances run according to the composition schema in the directory with the latest version. Figure 13 shows MoRE-WS and Apache ODE consoles during a dynamic adaptation according to $R_{B\&NUnavailable}$.

The dynamic adaptation of composition schema versions is one side of the coin. The other side is the dynamic adaptation of running instances. This is not an easy task since each instance may be running a different operation at the same time. For example, some instances are almost finishing their execution while others are just starting. Instead of migrating all instances to apply changes, we propose a set of strategies to decide whether or not instances should migrate to new versions of the composition schema (Alférez and Pelechano, 2012a). This is an important aspect because uncontrolled instance migration will lead to inconsistencies or errors (Weber et al., 2008). The migration of instances from an old composition schema to a new one is carried out when it is safe to do so. That is, only those instances are migrated which are compliant with the old version of the composition schema (Weber et al., 2008). Specifically, an instance I is *compliant* with a composition schema S, if the current execution history of I can be created based on S (Rinderle et al., 2004). All other instances remain running according to the old version of the composition schema. In our case, the execution history of every I is managed at the composition model level. Therefore, in addition to indicating the workflow to be followed by the service composition, the composition model keeps the updated information about the activity (BPMN task or subprocess) or event (start or end event) that is being executed.

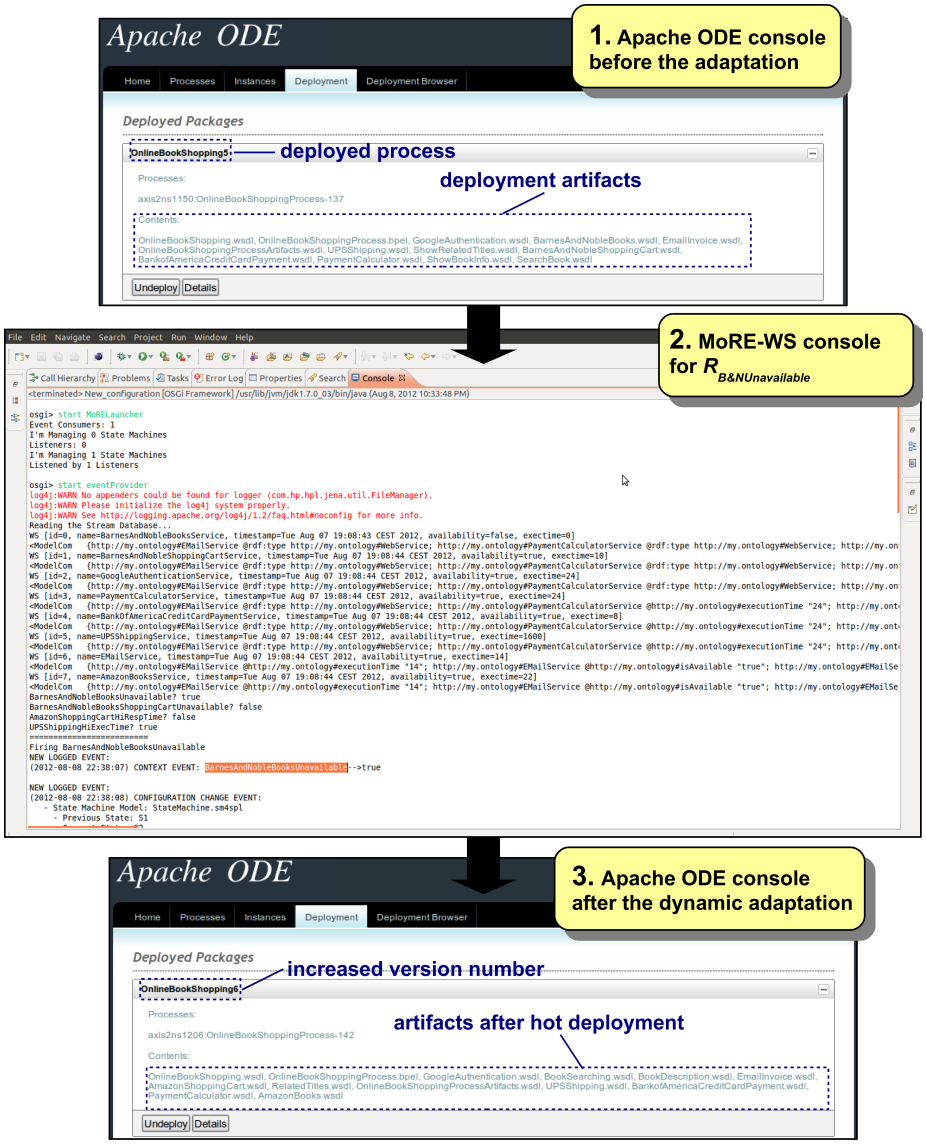


Figure 13: MoRE-WS and Apache ODE consoles during a dynamic adaptation

6. Framework Demonstration

We created a video demonstration of our framework (Alférez et al., 2012). In this demonstration, MoRE-WS used the non-trivial variability model in our running example (with 36 features and five variation points) to guide two dynamic adaptations when the *B&NUnavailable* and the *UPSHiExecTime* context conditions are fulfilled in a tight time frame: three milliseconds. Figure 14 shows

the UML deployment diagram that depicts the computing infrastructure that was used in this demonstration. The Web services ran on Apache Axis2 [15] version 1.6.1, which was deployed as a WAR (Web archive) distribution on Apache Tomcat [16] version 7.0.8. The hot deployment was carried out by MoRE-WS on Apache ODE version 1.3.5, which was deployed on a second instance of Apache Tomcat as a WAR distribution. The demonstration was carried out on a PC with an Intel Core 2 Duo 2.0 GHz processor, 4 GB RAM, Ubuntu version 10.04, and Kernel Linux version 2.6.32-36-generic.

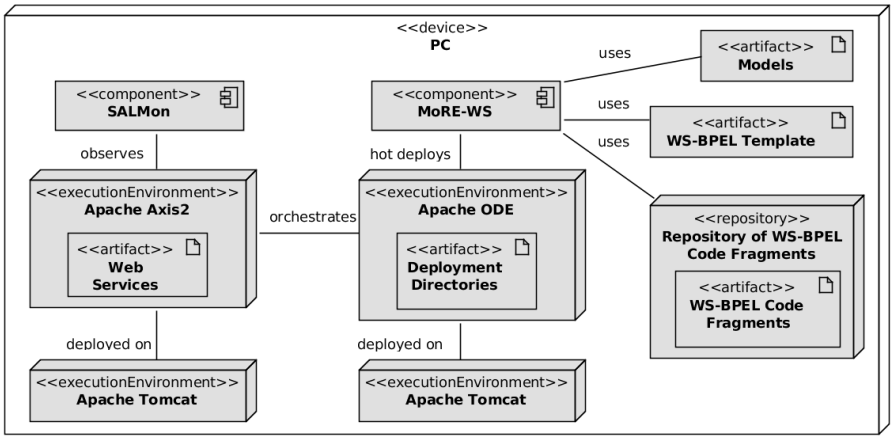


Figure 14: UML deployment diagram for the demonstration

7. Evaluation

In this section, we describe the evaluation results of our framework. Among the types of investigations (strategies), we chose to carry out experiments as the empirical investigation strategy. An experiment in software engineering is an empirical inquiry that manipulates one factor or variable of the studied setting (Wohlin et al., 2012). In order to develop the evaluation metrics, we used the Goal/Question/Metric (GQM) method (Basili et al., 1994). The GQM method was chosen because measurement is defined in a top-down fashion, from goals to metrics.

The GQM models that guided the evaluation are described in the following subsections. Each GQM model supports key aspects of our contribution (see Section 1.2). Since our contribution spans from design time to runtime, it was necessary to answer research questions related to these two phases. Regarding design time, the GQM models in Section 7.1 answer research questions that are related to generation efficiency and complexity reduction of the adaptation

15. http://axis.apache.org/axis2/java/core/.
16. http://tomcat.apache.org/.

space. Moreover, Section 7.2 answers research questions that are related to the verification of the variability model and its configurations: anomalies reduction and verification efficiency. Regarding runtime, the GQM models in Section 7.3 answer research questions that are related to model-driven dynamic adaptations: efficiency and capacity to avoid saturations under stress circumstances.

7.1. GQM Models that are Related to the Adaptation Space

This subsection presents two GQM models that have helped us to evaluate two aspects of our approach: generation efficiency and complexity reduction of the adaptation space.

GQM Model 1: Table 1 describes the GQM model for the following goal: *"Efficient generation time of variability model configurations from the systems analyst's viewpoint."* Efficiency is considered as performing or functioning in the best possible manner with the least waste of time and effort. In order to answer **Q1**, we used the variability model in our running example with 36 features (**M1**) and a set of nine resolutions specified for this model (**M2**). With this information, MOSKITT4SPL generated an adaptation space with 40 variability model configurations (**M3**) and 360 transitions among configurations (**M4**) in less than three seconds (**M5**) – find additional details about this experiment on our website (Alférez et al., 2012). The manual creation of an adaptation space with these proportions is clearly unfeasible.

Goal	**Purpose**	**Efficient**
	Issue	**generation time of**
	Object	**variability model configurations**
	Viewpoint	**from the systems analyst's viewpoint**
Question	**Q1**	What is the required time to generate the adaptation space?
Metrics	**M1 - M5**	(**M1**) number of features in the variability model, (**M2**) number of resolutions, (**M3**) number of generated variability model configurations, (**M4**) number of generated transitions among variability model configurations, and (**M5**) generation time of the adaptation space

Table 1: GQM model 1 for the "efficient generation time of variability model configurations from the systems analyst's viewpoint" goal

GQM Model 2: Table 2 describes the GQM model for the following goal: "*Reduce the complexity of the adaptation space from the systems analyst's viewpoint.*" **Q2** can be answered by looking at the variability model in our running example with 36 features (**M6**) and nine resolutions (**M7**) specified for this model. This small model and a small set of resolutions describe a large adaptation space with 40 variability model configurations (**M8**) and 360 transitions among configurations (**M9**) (Alférez et al., 2012). In our previous work (Cetina

et al., 2009) we introduced a variability model containing 18 features (**M6**) and three resolutions (**M7**) specified for this model. This model and the set of resolutions represented more than 200,000 variability model configurations (**M8**) and more than 600,000 possible transitions among configurations (**M9**). Therefore, it is possible to say that variability models can reduce (or hide) much of the complexity in the definition of the adaptation space from the point of view of systems analysts. Variability models can provide an intensional rather than extensional description of each possible configuration of the service composition.

Goal	**Purpose**	**Reduce**
	Issue	**the complexity of the**
	Object	**adaptation space**
	Viewpoint	**from the systems analyst's viewpoint**
Question	**Q2**	Do a small variability model and a set of resolutions can be used to describe a large adaptation space?
Metrics	**M6 - M9**	(**M6**) number of features in the variability model, (**M7**) number of resolutions, (**M8**) number of generated variability model configurations, (**M9**) number of generated transitions among variability model configurations

Table 2: GQM model 2 for the "reduce the complexity of the adaptation space from the systems analyst's viewpoint" goal

7.2. GQM Models that are Related to Verification

This subsection presents the GQM models that have guided us to evaluate aspects related to the verification of the variability model and its configurations. In order to evaluate the GQM models in this section, we implemented and tested the verification operations under the following conditions: PC with 32 bit Windows Vista, AMD Turion 64 bits X2 Dual-Core Mobile RM-74 2.20 GHz processor, 4 GB RAM, and GNU Prolog 1.3.0.

GQM Model 3: Table 3 describes the GQM model for the following goal: "*Reduce anomalies in the variability model and in its configurations from the systems analyst's viewpoint.*" In order to answer **Q3** and **Q4**, we evaluated 47 models, out of which 45 were taken from the SPLOT repository (Mendonca et al., 2009). The other two models were developed during industry collaboration projects (Lora-Michiels et al., 2010). The number of features in the models is distributed as follows (**M10** and **M12**): 30 models contained from nine to 49 features, four models from 50 to 99 features, four models from 100 to 999 features, and nine models from 1,000 to 2,000 features. The variability models covered various domains such as insurance, entertainment, Web applications, home automation, search engines, and databases (Mazo, 2011). Experiments

showed that our verification approach identified 100% of the anomalies (**M11**) with 0% false positives (**M13**).

Goal	**Purpose**	**Reduce**
	Issue	**anomalies in the**
	Object	**variability model and its configurations**
	Viewpoint	**from the systems analyst's viewpoint**
Question	**Q3**	What is the percentage of identified anomalies?
Metrics	**M10 - M11**	(**M10**) number of features in the variability model and (**M11**) percentage of identified anomalies
Question	**Q4**	What is the percentage of false positives?
Metrics	**M12 - M13**	(**M12**) number of features in the variability model and (**M13**) percentage of false positives

Table 3: GQM model 3 for the "reduce anomalies in the variability model and in its configurations from the systems analyst's viewpoint" goal

GQM Model 4: Table 4 describes the GQM model for the following goal: "*Efficient verification of the variability model from the systems analyst's viewpoint.*" In response to **Q5**, the response time for each verification criterion is as follows. We used the same variability models, with the same number of features per model (**M14**), that were used to evaluate GQM Model 3:

- *Accuracy of the Variability Model:* The response time of this operation on our benchmark takes an average of 0.8 milliseconds (**M15**).
- *Non-Existence of Dead Features:* For models from nine to 100 features, our approach verified dead features in 3.79 milliseconds in average. For models from 101 to 2,000 features, our approach took 3.45 seconds in average (**M16**).
- *Stability:* The evaluation on our benchmark shows that it takes 0.9 milliseconds in average to verify models between 9 and 100 variables and 16.2 seconds, in the worst case, to verify models between 100 and 2,000 variables (**M17**).
- *Semi-aliveness:* The performance of this verification criterion depends on the number of partitions and the number of variables per partition. Let us recall that a CSP is theoretically non-polynomial. However, modern solvers use highly optimized algorithms that are able to solve these CSPs efficiently (if needed, the programmer can help the solver with redundant constraints, alternative modeling, heuristics, etc.). Our preliminary tests show that the algorithm presented in Listing 5 scales well on our running example: 6.2 seconds for seven partitions and four context variables per partition (**M18**). Since we do not currently have an algorithm to find the partitions, further experiments are required to test semi-aliveness in our benchmark (we found partitions manually in our running example).

Goal	**Purpose**	**Efficient**
	Issue	**verification of the**
	Object	**variability model**
	Viewpoint	**from the systems analyst's viewpoint**
Question	**Q5**	What are the response times for each verification criterion?
Metrics	**M14 - M18**	(**M14**) number of features in the variability model, (**M15**) average response time for the Accuracy of the Variability Model criterion, (**M16**) average response time for the Non-existence of Dead Features criterion, (**M17**) average response time for the Stability criterion, and (**M18**) average response time for the Semi-aliveness criterion

Table 4: GQM model 4 for the "efficient verification of the variability model from the systems analyst's viewpoint" goal

7.3. GQM Models that are Related to Dynamic Adaptations

This subsection presents two additional GQM models. These models have allowed us to evaluate the following aspects about the dynamic adaptations that are carried out by MoRE-WS: efficiency during dynamic adaptations and capacity to avoid saturations under stress circumstances.

GQM Model 5: Table 5 describes the GQM model for the following goal: *"Efficient dynamic adaptation of service compositions from* MoRE-WS *'s viewpoint."* In order to answer **Q6**, we carried out an experiment on the same PC that is described in Section 6. We used the following randomly generated models: a MOSKitt4SPL feature model, an ATLAS Model Weaver weaving model, and a BPMN composition model. These models started with one element and they were populated with two hundred new elements each iteration up to 3,000 elements (**M19**).

In order to scale up this experiment, the number of features in the feature models exceeds the number of features in normal feature models found in literature (Mazo, 2011). The following model operations were performed: *getting current configuration*, *getting service operations mapped to features*, *mapping composition model elements to WS-BPEL fragments*, and *updating feature states*. The first two operations are performed to calculate $CM\triangle$ and $CM\nabla$. The *mapping composition model elements to WS-BPEL fragments* operation is carried out just before hot deployment. The *updating feature states* operation sets feature states to active or inactive. Figure 15 shows the response time of these operations (**M20 - M25**).

The *getting current configuration* operation exhaustively navigates a variability model configuration to look for active features. Although, the *updating feature states* operation also navigates the whole variability model to set the state of features, this operation gets significant different results. This is because

Goal	**Purpose**	**Efficient**
	Issue	**dynamic adaptation of**
	Object	**service compositions**
	Viewpoint	**from MoRE-WS's viewpoint**
Question	**Q6**	Is MoRE-WS efficient to carry out the dynamic adaptation of service compositions?
Metrics	**M19 - M25**	(**M19**) number of elements in models, (**M20**) response time of service composition increment, (**M21**) response time of service composition decrement, (**M22**) response time of getting the current configuration, (**M23**) response time of updating feature states, (**M24**) response time of getting service operations mapped to features, and (**M25**) response time of mapping composition model elements to WS-BPEL fragments

Table 5: GQM model 5 for the "efficient dynamic adaptation of service compositions from MoRE-WS's viewpoint" goal

the *updating feature states operation* has to make persistent the model changes. This operation is implemented by an UPDATE statement of the EMFMQ and a call to the save resource of the EMF API. The *getting service operations mapped to features* operation was slow. This is because of the Atlas Model Weaver metamodel. This metamodel specifies links between models by means of two levels. To get a BPMN element linked to a feature, first the operation has to navigate from an ElementEqual meta-element to a RightElement meta-element. Second, from this RightElement the operation has to navigate to an ElementRef meta-element. Furthermore, these two steps have to be performed for each link. Finally, the *mapping composition model elements to WS-BPEL fragments* got a good response time thanks to the strategy for managing variant WS-BPEL code fragments. Overall, even with a model population of 30,000 elements in each model, the model operations had a good time response ($<$ 300 milliseconds) that can be considered fast in the domain that we are addressing.

MoRE-WS carries out linear search on composition models to find out either the elements to be adapted in running instances are ahead or behind the current activity or event (Alférez and Pelechano, 2012a). Therefore, the performance of this operation does not depend on the model manipulation itself, but on the search algorithm. The worst case performance scenario is that MoRE-WS needs to loop through the entire model: *O(n)*. Further details about the performance evaluation during dynamic adaptation of running instances can be found in (Alférez and Pelechano, 2012a).

GQM Model 6: Table 6 describes the GQM model for the following goal: "*Avoid saturation under stress circumstances of* MoRE-WS *from the*

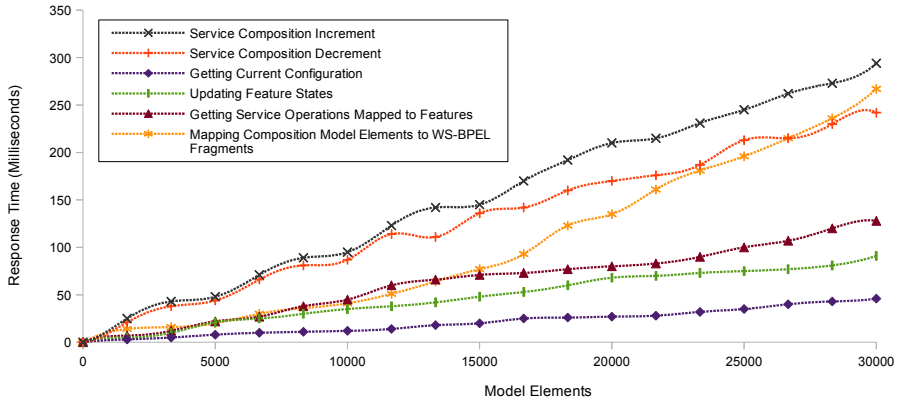


Figure 15: Response time of model operations

systems analyst's viewpoint." In order to answer **Q7**, we have manually injected 100 problematic context events ($contEvent_{1...100}$) into an execution log (**M26**), which is read by the Context Monitor. These events are separated by time frames of one to five milliseconds (**M27**). In the fist run (or observation), MoRE-WS retrieved the 100 problematic events at once because it queries the information collected by the Context Monitor every five seconds (**M28**). Then, MoRE-WS evaluated five context conditions ($contCond_{1...5}$) that could be affected by these events (**M29**). Since context conditions are evaluated in sequence (a dynamic adaptation for $contCond_1$ is always triggered before an adaptation for $contCond_2$), we did not experience performance decrease in **M30** (i.e., the response time is similar to the response time without stress). Only one dynamic adaptation is carried out at a time. An additional scenario is when several context events in a particular set of observed events may affect a particular context condition (e.g. $contEvent_1$ and $contEvent_{85}$ can affect $contCond_3$). Since MoRE-WS triggers a dynamic adaptation first for $contEvent_1$ than for $contEvent_{85}$, then $contEvent_{85}$ is discharged (i.e., a dynamic adaptation has been already triggered to solve $contCond_3$). We can conclude that MoRE-WS works well under stress circumstances when several problematic context events arise in tight time frames.

7.4. Discussion

By defining activation or deactivation of features by means of model-based resolutions, the adaptation paths between different variability model configurations (which describe service composition configurations) can be expressed in a declarative manner without the need for an exhaustive definition. The time to generate the adaptation space is short. The verification process finds anomalies and avoids false positives in variability models and its configurations at design time; hence, problematic situations can be fixed at design time to avoid negative effects at runtime. Also, the performance of our four verification criteria was

Goal	**Purpose**	**Avoid**
	Issue	**saturation under stress circumstances of**
	Object	**MoRE-WS**
	Viewpoint	**from the systems analyst's viewpoint**
Question	**Q7**	Does MoRE-WS have a good performance under stress circumstances?
Metrics	**M26 - M30**	(**M26**) number of problematic context events, (**M27**) time frame between problematic context events, (**M28**) frequency to observe the context, (**M29**) number of context conditions that could be affected by arising context events, and (**M30**) response time under stress circumstances

Table 6: GQM model 6 for the "avoid saturation under stress circumstances of MoRE-WS from the systems analyst's viewpoint" goal

notable. At runtime, our model-driven solution is efficient when dealing with dynamic adaptation of service compositions, even under stress circumstances.

The variability models used in the experiments for GQM models 1, 2, 5, and 6 were auto-generated. The variability models that were used in the experiments related to GQM models 3 and 4 came mostly from other studies. Therefore, we do not have enough empirical results yet about evaluating the human effort required at design time to create the variability model and the other models contained in our proposal. Nevertheless, we think our approach does not exponentially increase human effort at design time because this phase is fully tool-supported. Specifically, the provided tools support the creation of models and adaptation policies, and the automation of key tasks such as the generation and verification of variability model configurations. As part of our future work, we expect to quantify this effort in real service compositions at the Valencian Regional Ministry of Infrastructure and Transport [17].

8. Related Work

Our research is placed at the intersection of dynamic adaptation of services compositions, variability modeling, DSPL engineering, and verification of self-adaptive systems. Related work in these areas is described in the following subsections.

17. http://www.moskitt.org/eng/moskitt0/.

8.1. Dynamic Adaptation of Service Compositions

Several research works related to dynamic adaptation of service compositions have tended to implement variability constructs at the language level. For example, SCENE (Colombo et al., 2006) extends WS-BPEL with Event Condition Action (ECA) rules that define consequences for conditions to guide the execution of binding and rebinding self-reconfiguration operations. VxBPEL (Koning et al., 2009) is an adaptation of WS-BPEL that allows variation points, variants, and configurations to be defined for a process in a service-centric system. In (Baresi and Guinea, 2011), monitoring directives are expressed in the Web Service Constraint Language, and recovery strategies, which follow the ECA paradigm, are stated in the Web Service Recovery Language. Also, Aspect-Oriented Programming (AOP) has been proposed in several works for self-adaptive service compositions (Narendra et al., 2007, Sonntag and Karastoyanova, 2011, Moser et al., 2008). We argue that implementing and managing dynamic adaptations at the language level can become complex, especially in large systems. Our solution can guide dynamic adaptations at a more abstract level with models at runtime.

Another trend uses the Semantic Web to support capability-based discovery and interoperation of Web services at runtime. For instance, the DARPA Agent Markup Language for Services (DAML-S) tries to close the gap between the Semantic Web and Web services (Paolucci and Sycara, 2003). Services use the Semantic Web to support capability-based discovery and interoperation at runtime. Although Semantic Web services can use a DAML-S process model and grounding to manage their interactions with other Web services, variability management is not made evident in this model. More recent works focus on discovering Web service operations using the Semantic Web (Srinivasan et al., 2005, Klusch and Kapahnke, 2010, Pahl et al., 2011). Web services are semantically described in terms of capabilities offered. Then, inferences are performed to match the capabilities requested with the capabilities offered. A reconfiguration plan can be generated with the discovered services (using for example Artificial Intelligence (Moore et al., 2008)). We use an ontology to realize if arising context events can cause dynamic adaptations (similarly to (Bandara et al., 2009)). However, discovering Web service operations at runtime falls outside the current view of our approach.

In contrast to our approach, which addresses the reasoning of dynamic adaptation of service compositions through models, at both design time and runtime, several research covers self-adaptations only at runtime through implementation mechanisms (Erradi and Maheshwari, 2005, Cardellini et al., 2010, Mosincat and Binder, 2008).

We highlight four relevant related approaches that use models at runtime to support dynamic adaptation of service compositions. First, DySOA (Bosloper et al., 2005) is a tool that offers components for monitoring and reconfiguring Web service-centric systems. Even though models are used by these components for context analysis, QoS determination, and variability, these models and their related reconfiguration mechanisms at runtime are not presented in a

detailed way. QoSMOS (Calinescu et al., 2011) is a tool-supported framework for the QoS management of self-adaptive, service-based systems that combines existing techniques and tools. As in our case, QoSMOS autonomic architecture is based on the MAPE-K loop. It combines formal specification of QoS requirements, model-based QoS evaluation (through analytic solving of Markov Models), monitoring and parameter adaptation of the QoS models, and planning and execution of system adaptation. QoSMOS is focused on the translation of high-level QoS requirements into probabilistic temporal formulas and on the formalization of these QoS requirements. SASSY (Menasce et al., 2011) is a model-driven framework that provides runtime adaptation of service compositions in response to changing operating conditions. In this research, it is not clear how the service coordination logic can be deployed in a WS-BPEL engine. In (Morin et al., 2008), the authors propose combining model-driven and aspect-oriented techniques to support runtime variability from requirements to execution. Aspects are dynamically composed to produce configuration models, and these models are then used to generate the scripts needed to adapt a running system from one runtime configuration to another. However, this solution is not focused on Web service compositions. Moreover, in contrast to our approach, the adaptation model that captures the information about dynamic variability is presented without many implementation details.

The approaches that are presented above make evident the trend towards implementing dynamic adaptation of service compositions at the language level, which can be complex and time-consuming, and with low-level implementation mechanisms. The aforementioned model-driven approaches miss hot deployment in enterprise WS-BPEL engines.

8.2. Variability Modeling

There are several approaches that deal with modeling variability in service compositions that support BPs (Nguyen et al., 2011, Sun et al., 2010, Hadaytullah et al., 2009, Razavian and Khosravi, 2008). In this section we describe three relevant research works in this area. PESOA (Puhlmann et al., 2005) abstracts the BP in an unique model with a set of annotations that identify variable behavior. C-EPC (Rosemann and Van der Aalst, 2007) is a language extension to configure reference BP models that formalize recommended practices for specific domains. A single BP model contains configurable elements, alternatives that depend on the context of use, and context conditions. In (Gottschalk et al., 2008), the authors propose an approach to identify configurable elements of a workflow modeling language (such as YAWL, WS-BPEL or SAP WebFlow) with opportunities for predefining alternative model versions within a single workflow model. Although the aforementioned works have inspired ours, they are limited to variability modeling. Our approach goes a step forward by leveraging models at runtime to guide dynamic adaptations. Also, they integrate all possible process variants in a single model. It results in large and difficult-to-understand models. On the contrary, we propose to reason about variability separately. An approach in a similar direction proposes a separated variability descriptor document to describe the variability points and their properties and points into

the file to be customized (Mietzner and Leymann, 2008). This document makes up the Software as a Service (SaaS) process layer of an application. Instead of proposing the creation of a new variability descriptor, our approach reuses the proven potential of widely-used feature models to describe variability.

8.3. Dynamic Software Product Lines

A recent approach named Service-Oriented Product Line (SOPL) (Lee and Kotonya, 2010) has proven the viability of a DSPL application domain built on services and a service-oriented architecture. However, in spite of the growing amount of research related to context-aware self-adaptive services using SPL principles, it does not focus on supporting dynamic adaptation of Web service compositions. Some relevant related approaches are presented as follows.

In (Lee and Kang, 2006), Lee et al. propose a systematic method to develop dynamically reconfigurable core assets and a reconfigurator that monitors and manages product configuration at runtime. This method remains at the general level with respect to the models employed. Moreover, their specifications of reconfiguration actions are not as flexible as our model-based adaptation policies. This method has been applied to the development of home service robot control software. In (Hallsteinsen et al., 2006), Hallsteinsen et al. present the MADAM approach to build adaptive systems as component-based systems families with variability modeled explicitly as part of the family architecture. They target distributed applications accessed through hand-held networked devices which have to adapt to context changes. In (Parra et al., 2009), Parra et al. propose a Context-Aware Dynamic Service-Oriented Product Line (DSOPL) named CAPucine. CAPucine is based on a model-driven approach as ours is. Their approach is divided into two different processes for product derivation. In the first process, for every selected feature of the product family, there is an associated asset that corresponds to a partial model of the product itself. These models get composed and transformed to generate the product. The second process relates to dynamic adaptation. FraSCAti, which is a Service Component Architecture (SCA) platform with dynamic properties, enables binding and unbinding of components at runtime. SCA domains are built on single-vendor infrastructure, while Web services are not. This fact coupled with the popularity of Web services makes it relevant to propose self-adaptation mechanisms for Web service compositions.

The aforementioned approaches show that the focus has been placed on methods and mechanisms for runtime adaptability of services in areas such as domotics and robotics but not specifically on Web service compositions. In contrast to some of these works, we make an intensive use of variability models as fundamental drivers for dynamic adaptations.

8.4. Verification of Self Adaptive Systems

Calinescu et al. (Calinescu et al., 2012) and Salifu et al. (Salifu et al., 2012) explore the self-adaptation paradigm based in the relations among contextual information (W), requirements (R) and the specifications of the required solution

(S) as a logic entailment $W, S \models R$ (which means that the use of specification S in context W ensures the satisfaction of the requirements R). Both approaches explore how to extend (Zave and Jackson, 1997) to verify self-adaptive systems. On one hand, (Calinescu et al., 2012) outlines a range of complementary approaches that use formal verification techniques in runtime scenarios and presents several research challenges about verification of self-adaptive systems (SASs). For instance, one of the main challenges is to develop a "*repertoire of techniques that provides timely reaction to detected violations of the requirements*" of SASs. On the other hand, (Salifu et al., 2012) presents an approach that enables to: 1) represent and reason about changes in the physical environment of software systems and assess their impact on requirements satisfaction; and 2) specify monitors and switchers that can detect changes and adapt in response to requirements satisfaction violation. Salifu et al. follow the same direction for monitoring described in (Wang et al., 2009): to encode monitoring and switching problems into propositional logic constraints that are analyzed with a SAT solver. Wang et al.'s approach (Wang et al., 2009) is suitable for analyzing internal state changes of software systems. However, physical context changes are not analyzed. In another research (Sama et al., 2008), rule-based analysis algorithms operate on extended finite state machines that detect potential inconsistencies among the paths of the states under varying situations corresponding to different switches. However, their treatment does not handle conditions for monitoring problems.

Other research works (Bencomo et al., 2008, Lee and Kang, 2006, Cetina et al., 2009, Sawyer et al., 2012) use variability models to specify the legal combination of requirements to satisfy adaptation needs elicited from a monitoring process. However, they neither deal with verification of SASs nor focus on service compositions.

Autili et al. (Autili et al., 2009) presents the PLASTIC approach to support context-aware adaptive services. In PLASTIC, adaptations happen at discovery time. Therefore, the deployed application is customized with respect to the context at binding time but it does not adapt at runtime. Our approach goes a step further with dynamic adaptations guided by variability models, and by verifying service composition configurations at design time.

Iftikhar and Weyns (Iftikhar and Weyns, 2012) present a case study in which model checking is used to verify behavioral properties of a decentralized SASs. In order to model the main processes of the system, they use timed automata, and for the specification of the required properties they use timed computation tree logic. Also, they propose to use the Uppaal tool (Behrmann et al., 2006) to specify the system and verify flexibility and robustness properties. In contrast, our approach is based on variability models and CP-oriented verification criteria to avoid the combinatorial explosion of states that limits the use of model checking on large variability models.

Xu et al. (Xu et al., 2012) identify five categories of errors in SASs at runtime: predictability error, stability error, reachability and liveness error, and consistency error. They propose the ADAM approach, which takes context changes from the environment, and adapts an application based on user-defined

rules for each one of their applications. Then, ADAM can run these user-specified rules, automatically report errors of each one of the aforementioned categories when the rule is violated at adaptation-time, and helps to identify potential defects in the rule design. Our approach is different to this one in two aspects: 1) we use a variability model to represent the adaptation rules of several configurations at the same time (i.e., as a family of configurations) and not in rules that are specified for each configuration; and 2) our approach tries to identify defects at design time to allow designers to correct them before the system is released and to avoid runtime verification, which hinders correction, and in most of cases does not scale to industrial models (Tamura et al., 2012). Further works about verification of SASs are discussed in (Weyns et al., 2012, Villegas et al., 2011, Tamura et al., 2012).

9. Conclusions and Future Work

In this paper, we proposed a solution based on semantically rich variability models to support the dynamic adaptation of service compositions. In order to face problematic context events, our MoRE-WS tool activates and deactivates features in a variability model at runtime. New variability model configurations are used to modify a composition model that abstracts the service composition. Changes in the composition model are reflected into the service composition by adding or removing fragments of WS-BPEL code that are deployed at runtime. The variability model and its possible configurations are verified at design time with CP to reach optimum adaptations.

The use of variability models in our approach has the following advantages: 1) instead of programming complex scripts to describe adaptation policies, easy-to-understand and technology-independent variability models can be used to express dynamic adaptations with abstract concepts over the underlying technologies; 2) variability models can hide the complexity of the adaptation space, thus facilitating the reasoning of dynamic adaptations; and 3) the verification of the variability model and its configurations at design time can avoid inconsistent service composition adaptations.

As future work, we will extend our approach to be used in the open world, in which the service composition should react to continuous and unanticipated changes in complex and uncertain contexts. We have preliminary results in this area in which variability models are able to evolve at runtime for better functioning and system "survival" (Alférez and Pelechano, 2012b). Ontological reasoning is extended in the open world to realize the requirements that can be affected by arising unknown context events (unforeseen at design time). To this end, we use forward chaining to evaluate arising context facts against rule premises. Accordingly, we will extend the set of verification operations at runtime to avoid inconsistent service recompositions in the open world. For example, settling time will be verified to avoid overheads, and changes in the service composition when dealing with unknown context events will be verified during execution to avoid a negative impact in the expected requirements.

References

Akkawi, F., Akkawi, K., Bader, A., Ayyash, M., Fletcher, D., Alzoubi, K., March 2007. Software adaptation: A conscious design for oblivious programmers. In: Proceedings of the IEEE Aerospace Conference. pp. 1–12.

Alférez, G., Pelechano, V., November 2012a. Towards dynamic service compositions with models at runtime. Tech. Rep. ProS-TR-2012-05, Research Center on Software Production Methods.
URL `http://pros.webs.upv.es/technicalreports/PROS-TR-2012-05.pdf`

Alférez, G. H., Pelechano, V., 2011a. Context-aware autonomous web services in software product lines. In: Proceedings of the 2011 15th International Software Product Line Conference. SPLC '11. IEEE Computer Society, Washington, DC, USA, pp. 100–109.

Alférez, G. H., Pelechano, V., sept. 2011b. Systematic reuse of web services through software product line engineering. In: 2011 Ninth IEEE European Conference on Web Services (ECOWS). pp. 192 –199.

Alférez, G. H., Pelechano, V., 2012b. Dynamic evolution of context-aware systems with models at runtime. In: France, R., Kazmeier, J., Breu, R., Atkinson, C. (Eds.), Model Driven Engineering Languages and Systems. Vol. 7590 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 70–86.

Alférez, G. H., Pelechano, V., Mazo, R., Salinesi, C., Diaz, D., 2012. Dynamic adaptation of service compositions with variability models.
URL `http://www.harveyalferez.com/dynamicadaptation/`

Ameller, D., Franch, X., 2008. Service level agreement monitor (SALMon). In: Proceedings of the Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008). IEEE Computer Society, Washington, DC, USA, pp. 224–227.

Autili, M., Benedetto, P., Inverardi, P., 2009. Context-aware adaptive services: The PLASTIC approach. In: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009. FASE '09. Springer-Verlag, Berlin, Heidelberg, pp. 124–139.

Ayora, C., Alférez, G. H., Torres, V., Pelechano, V., 2012. Applying CVL to business process variability management. In: Proceedings of VARiability for You. MODELS 2012. pp. 24–29.
URL `http://vary2012.irisa.fr/VARY2012Proceedings.pdf`

Bae, J., Kang, S., oct. 2007. A method to generate a feature model from a business process model for business applications. In: Computer and Information Technology. pp. 879–884.

Bandara, K. Y., Wang, M., Pahl, C., 2009. Context modeling and constraints binding in web service business processes. In: Proceedings of the first international workshop on Context-aware software technology and applications. CASTA '09. ACM, New York, NY, USA, pp. 29–32.

Baresi, L., Guinea, S., March 2011. Self-supervising BPEL processes. IEEE Trans. Softw. Eng. 37, 247–263.

Basili, V. R., Caldiera, G., Dieter, R. H., 1994. Goal question metric paradigm. In: Encyclopedia of Software Engineering. Vol. 2. John Wiley & Sons, Inc., pp. 528–532. URL http://www.cs.umd.edu/~basili/publications/technical/T89.pdf

Behrmann, G., David, A., Larsen, K. G., Hakansson, J., Petterson, P., Yi, W., Hendriks, M., 2006. Uppaal 4.0. In: Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems. QEST '06. IEEE Computer Society, Washington, DC, USA, pp. 125–126.

Benavides, D., Trinidad, P., Ruiz-Cortés, A., 2005. Automated reasoning on feature models. In: Proceedings of the 17th international conference on Advanced Information Systems Engineering. CAiSE'05. Springer-Verlag, Berlin, Heidelberg, pp. 491–503.

Bencomo, N., Sawyer, P., Blair, G. S., Grace, P., 2008. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In: SPLC (2). pp. 23–32.

Blair, G., Bencomo, N., France, R. B., October 2009. Models@ run.time. Computer 42, 22–27.

Bosch, J., 2000. Design and use of software architectures: adopting and evolving a product-line approach. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Bosloper, I., Siljee, J., Nijhuis, J., Hammer, D., 2005. Creating self-adaptive service systems with DySOA. In: Proceedings of the Third European Conference on Web Services. ECOWS '05. IEEE Computer Society, Washington, DC, USA, pp. 95–104.

Calinescu, R., Ghezzi, C., Kwiatkowska, M., Mirandola, R., Sep. 2012. Self-adaptive software needs quantitative verification at runtime. Commun. ACM 55 (9), 69–77.

Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G., 2011. Dynamic QoS management and optimization in service-based systems. IEEE Transactions on Software Engineering 37, 387–409.

Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F., 2010. Adaptive management of composite services under percentile-based service level agreements. In: Maglio, P., Weske, M., Yang, J., Fantinato, M. (Eds.), Service-Oriented Computing. Vol. 6470 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 381–395.

Cetina, C., Giner, P., Fons, J., Pelechano, V., October 2009. Autonomic computing through reuse of variability models at runtime: The case of smart homes. Computer 42, 37–43.

Clements, P., Northrop, L., 2001. Software product lines: practices and patterns. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Colombo, M., Di Nitto, E., Mauri, M., 2006. SCENE: A service composition execution environment supporting dynamic changes disciplined through rules. In: Dan, A., Lamersdorf, W. (Eds.), Service-Oriented Computing – ICSOC 2006. Vol. 4294 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 191–202.

Coplien, J., Hoffman, D., Weiss, D., November 1998. Commonality and variability in software engineering. IEEE Softw. 15, 37–45.

Cotroneo, D., Gargiulo, M., Russo, S., Ventre, G., 2002. Improving the availability of web services. In: 22nd International Conference on Software Engineering (ICSE 2002. pp. 59–63.

Czarnecki, K., Antkiewicz, M., 2005. Mapping features to models: a template approach based on superimposed variants. In: Proceedings of the 4th international conference on Generative Programming and Component Engineering. GPCE'05. Springer-Verlag, Berlin, Heidelberg, pp. 422–437.

Czarnecki, K., Eisenecker, U. W., 2000. Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Del Fabro, M. D., Bézivin, J., Valduriez, P., 2006. Weaving models with the Eclipse AMW plugin. In: Eclipse Modeling Symposium, Eclipse Summit Europe.

Dey, A. K., January 2001. Understanding and using context. Personal Ubiquitous Comput. 5, 4–7.

Diaz, D., Abreu, S., Codognet, P., Jan. 2012. On the implementation of GNU Prolog. Theory Pract. Log. Program. 12 (1-2), 253–282.

Diaz, D., Codognet, P., 2001. Design and implementation of the GNU Prolog system. Journal of Functional and Logic Programming 2001 (6).

Dinkelaker, T., Mitschke, R., Fetzer, K., Mezini, M., March 2010. A dynamic software product line approach using aspect models at runtime. In: Proceedings of the 1st Workshop on Composition and Variability.
URL `http://www.stg.tu-darmstadt.de/media/st/publications/a_dynamic_software_product_line_approach_using_aspect_models_at_runtime.pdf`

Erradi, A., Maheshwari, P., 2005. wsBus: QoS-aware middleware for reliable web services interactions. In: Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05) on e-Technology, e-Commerce and e-Service. EEE '05. IEEE Computer Society, Washington, DC, USA, pp. 634–639.

Esfahani, N., Kouroshfar, E., Malek, S., 2011. Taming uncertainty in self-adaptive software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ESEC/FSE '11. ACM, New York, NY, USA, pp. 234–244.

Fleurey, F., Solberg, A., 2009. A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems. MODELS '09. Springer-Verlag, Berlin, Heidelberg, pp. 606–621.

Gottschalk, F., van der Aalst, W. M. P., Jansen-Vullers, M. H., Rosa, M. L., 2008. Configurable workflow models. Int. J. Cooperative Inf. Syst. 17 (2), 177–221.

Gröner, G., Wende, C., Bošković, M., Parreiras, F. S., Walter, T., Heidenreich, F., Gašević, D., Staab, S., 2011. Validation of families of business processes. In: Proceedings of the 23rd international conference on Advanced information systems engineering. CAiSE'11. Springer-Verlag, Berlin, Heidelberg, pp. 551–565.

Hadaytullah, Koskimies, K., Systa, T., july 2009. Using model customization for variability management in service compositions. In: Web Services, 2009. ICWS 2009. IEEE International Conference on. pp. 687 –694.

Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K., 2008. Dynamic software product lines. Computer 41, 93–95.

Hallsteinsen, S., Stav, E., Solberg, A., Floch, J., 2006. Using product line techniques to build adaptive systems. In: Proceedings of the 10th International on Software Product Line Conference. IEEE Computer Society, Washington, DC, USA, pp. 141–150.

Haugen, O., Møller-Pedersen, B., Oldevik, J., Olsen, G. K., Svendsen, A., 2008. Adding standardized variability to domain specific languages. In: Proceedings of the 2008 12th International Software Product Line Conference. SPLC '08. IEEE Computer Society, Washington, DC, USA, pp. 139–148.

Horn, P., 2001. Autonomic computing: IBM's perspective on the state of information technology.
URL `http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf`

IBM, 2006. An architectural blueprint for autonomic computing. Tech. rep., IBM.
URL `http://www.eecs.harvard.edu/~chaki/bib/papers/autonomic.pdf`

Iftikhar, M. U., Weyns, D., 2012. A case study on formal verification of self-adaptive behaviors in a decentralized system. In: FOCLASA. pp. 45–62.

Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S., November 1990. Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University.
URL `http://www.sei.cmu.edu/library/abstracts/reports/90tr021.cfm`

Keeney, J., 2004. Completely unanticipated dynamic adaptation of software. Ph.D. thesis, Trinity College Dublin.
URL `http://www.tara.tcd.ie/bitstream/2262/30726/1/TCD-CS-2005-43.pdf`

Khan, M., Reichle, R., Geihs, K., july 2008. Architectural constraints in the model-driven development of self-adaptive applications. Distributed Systems Online, IEEE 9 (7), 1.

Klusch, M., Kapahnke, P., 2010. isem: Approximated reasoning for adaptive hybrid selection of semantic services. In: Proceedings of the 2010 IEEE Fourth International Conference on Semantic Computing. ICSC '10. IEEE Computer Society, Washington, DC, USA, pp. 184–191.

Koning, M., Sun, C.-a., Sinnema, M., Avgeriou, P., February 2009. VxBPEL: Supporting variability for web services in BPEL. Inf. Softw. Technol. 51, 258–269.

Lee, J., Kang, K. C., 2006. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In: Proceedings of the 10th International on Software Product Line Conference. IEEE Computer Society, Washington, DC, USA, pp. 131–140.

Lee, J., Kotonya, G., May 2010. Combining service-orientation with product line engineering. IEEE Softw. 27, 35–41.

Little, M., October 2003. Transactions and web services. Commun. ACM 46, 49–54.

Lora-Michiels, A., Salinesi, C., Mazo, R., January 2010. A Method Based on Association Rules to Construct Product Line Models. In: Fourth International Workshop on Variability Modelling of Software-intensive Systems. pp. 147–150.
URL http://www.wi-inf.uni-duisburg-essen.de/FGFrank/download/icb/ICBReportNo37.pdf

Mazo, R., November 2011. A generic approach for automated verification of product line models. Ph.D. thesis, Université Paris 1 Panthéon - Sorbonne, Paris, France.
URL https://dl.dropbox.com/u/11314786/Thesis-RaulMAZO.pdf

Mazo, R., Salinesi, C., Diaz, D., Lora-Michiels, A., 2011. Transforming attribute and clone-enabled feature models into constraint programs over finite domains. In: Proceedings of the 6th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE). Springer Press, pp. 188–199.

Mazo, R., Salinesi, C., Djebbi, O., Diaz, D., Lora-Michiels, A., Apr. 2012. Constraints: the Heart of Domain and Application Engineering in the Product Lines Engineering Strategy. International Journal of Information System Modeling and Design IJISMD 3 (2), 50.

McKinley, P. K., Sadjadi, S. M., Kasten, E. P., Cheng, B. H. C., July 2004. Composing adaptive software. Computer 37, 56–64.

Menasce, D., Gomaa, H., Malek, S., Sousa, J., 2011. SASSY: A framework for self-architecting service-oriented systems. IEEE Software 28, 78–85.

Mendonca, M., Branco, M., Cowan, D., 2009. S.P.L.O.T.: software product lines online tools. In: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. OOPSLA '09. ACM, New York, NY, USA, pp. 761–762.

Mietzner, R., Leymann, F., 2008. Generation of BPEL customization processes for SaaS applications from variability descriptors. In: Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 2. SCC '08. IEEE Computer Society, Washington, DC, USA, pp. 359–366.

Miller, A., May 1989. Engineering design: its importance for software. Potentials, IEEE 8 (2), 14–16.

Moore, C., Xue Wang, M., Pahl, C., November 2008. An architecture for autonomic web service process planning. In: Binder, W., Dustdar, S. (Eds.), 3rd Workshop on Emerging Web Services Technology.
URL http://www.inf.usi.ch/faculty/binder/wewst08/wewst08_proceedings.pdf

Morin, B., Barais, O., Jezequel, J.-M., Fleurey, F., Solberg, A., October 2009. Models@ run.time to support dynamic adaptation. Computer 42, 44–51.

Morin, B., Fleurey, F., Bencomo, N., Jézéquel, J.-M., Solberg, A., Dehlen, V., Blair, G., 2008. An aspect-oriented and model-driven approach for managing dynamic variability. In: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems. MoDELS '08. Springer-Verlag, Berlin, Heidelberg, pp. 782–796.

Moser, O., Rosenberg, F., Dustdar, S., 2008. Non-intrusive monitoring and service adaptation for WS-BPEL. In: Proceedings of the 17th international conference on World Wide Web. WWW '08. ACM, New York, NY, USA, pp. 815–824.

Mosincat, A., Binder, W., 2008. Transparent runtime adaptability for BPEL processes. In: Proceedings of the 6th International Conference on Service-Oriented Computing. ICSOC '08. Springer-Verlag, Berlin, Heidelberg, pp. 241–255.

Narendra, N. C., Ponnalagu, K., Krishnamurthy, J., Ramkumar, R., 2007. Run-time adaptation of non-functional properties of composite web services using aspect-oriented programming. In: Proceedings of the 5th International Conference on Service-Oriented Computing. ICSOC '07. Springer-Verlag, Berlin, Heidelberg, pp. 546–557.

Nguyen, T., Colman, A., Han, J., 2011. Modeling and managing variability in process-based service compositions. In: Proceedings of the 9th international conference on Service-Oriented Computing. ICSOC'11. Springer-Verlag, Berlin, Heidelberg, pp. 404–420.

OASIS, April 2007. Web services business process execution language version 2.0. URL `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`

Pahl, C., Gacitua-Decar, V., Wang, M., Bandara, K., 2011. Ontology-based composition and matching for dynamic service coordination. In: Salinesi, C., Pastor, O. (Eds.), Advanced Information Systems Engineering Workshops. Vol. 83 of Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, pp. 464–477.

Paolucci, M., Sycara, K., Sep. 2003. Autonomous semantic web services. IEEE Internet Computing 7 (5), 34–41.

Parra, C., Blanc, X., Duchien, L., 2009. Context awareness for dynamic service-oriented product lines. In: Proceedings of the 13th International Software Product Line Conference. SPLC '09. Carnegie Mellon University, Pittsburgh, PA, USA, pp. 131–140.

Pohl, K., Böckle, G., Linden, F. J. v. d., 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Puhlmann, F., Schnieders, A., Weiland, J., Weske, M., June 2005. Variability Mechanisms for Process Models. Tech. rep.
URL `http://frapu.de/pdf/PESOA_TR_17-2005.pdf`

Razavian, M., Khosravi, R., 2008. Modeling variability in business process models using UML. In: Proceedings of the Fifth International Conference on Information Technology: New Generations. ITNG '08. IEEE Computer Society, Washington, DC, USA, pp. 82–87.

Rinderle, S., Reichert, M., Dadam, P., Jul. 2004. Correctness criteria for dynamic changes in workflow systems: a survey. Data Knowl. Eng. 50 (1), 9–34.

Rosemann, M., Van der Aalst, W. M. P., Mar. 2007. A configurable reference modelling language. Inf. Syst. 32 (1), 1–23.

Rossi, F., Van Beek, P., Walsh, T., 2006. Handbook of constraint programming. Elsevier Science.

Salifu, M., Yu, Y., Bandara, A. K., Nuseibeh, B., 2012. Analysing monitoring and switching problems for adaptive systems. Journal of Systems and Software 85 (12), 2829 – 2839.

Salinesi, C., Mazo, R., April 2012. Software Product Line - Advanced Topic. InTech, Ch. Defects in Product Line Models and how to Identify them, pp. 3–22.
URL `http://www.intechopen.com/books/software-product-line-advanced-topic/defects-in-product-line-models-and-how-to-identify-them`

Salinesi, C., Mazo, R., Diaz, D., Djebbi, O., 2010. Using integer constraint solving in reuse based requirements engineering. In: Proceedings of the 2010 18th IEEE International Requirements Engineering Conference. RE '10. IEEE Computer Society, Washington, DC, USA, pp. 243–251.

Sama, M., Rosenblum, D. S., Wang, Z., Elbaum, S., 2008. Model-based fault detection in context-aware adaptive applications. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. SIGSOFT '08/FSE-16. ACM, New York, NY, USA, pp. 261–271.

Sawyer, P., Mazo, R., Diaz, D., Salinesi, C., Hughes, D., oct. 2012. Using constraint programming to manage configurations in self-adaptive systems. Computer 45 (10), 56 –63.

Schmidt, A., November 2002. Ubiquitous computing - computing in context. Ph.D. thesis, Lancaster University.
URL `http://www.comp.lancs.ac.uk/~albrecht/phd/`

Sonntag, M., Karastoyanova, D., August 2011. Compensation of adapted service orchestration logic in BPEL'n'aspects. In: Proceedings of the 9th International Conference on Business Process Management (BPM 2011), Clermont-Ferrand, France, 2011. Springer-Verlag, pp. 1–16.

Srinivasan, N., Paolucci, M., Sycara, K., 2005. An efficient algorithm for OWL-S based semantic search in UDDI. In: Proceedings of the First international conference on Semantic Web Services and Web Process Composition. SWSWPC'04. Springer-Verlag, Berlin, Heidelberg, pp. 96–110.

Sun, C., Rossing, R., Sinnema, M., Bulanov, P., Aiello, M., 2010. Modeling and managing the variability of web service-based systems. Journal of Systems and Software 83 (3), 502 – 516.

Tamura, G., Villegas, N., Müller, H., P. Sousa, J., Becker, B., Pezzè, M., Karsai, G., Mankovskii, S., Schäfer, W., Tahvildari, L., Wong, K., Aug. 2012. Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems. In: de Lemos, R., Giese, H., Müller, H., Shaw, M. (Eds.), Software Engineering for Self-Adaptive Systems 2. Vol. 7475 of LNCS. Springer, pp. 108–132.

Tesauro, G., Kephart, J. O., 2004. Utility functions in autonomic systems. In: Proceedings of the First International Conference on Autonomic Computing. ICAC '04. IEEE Computer Society, Washington, DC, USA, pp. 70–77.

Torres, V., Giner, P., Pelechano, V., 2012. Developing BP-driven web applications through the use of MDE techniques. Software and Systems Modeling 11, 609–631.

Trinidad, P., Benavides, D., Durán, A., Ruiz-Cortés, A., Toro, M., Jun. 2008. Automated error analysis for the agilization of feature modeling. J. Syst. Softw. 81 (6), 883–896.

Van den Broek, P., Galvão, I., January 2009. Analysis of feature models using generalised feature trees. In: Third International Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2009. No. 29 in ICB-Research Report. Universität Duisburg-Essen, Essen, Germany, pp. 29–35.

Van der Aalst, W. M. P., Dumas, M., Gottschalk, F., ter Hofstede, A. H. M., Rosa, M. L., Mendling, J., May 2010. Preserving correctness during business process model configuration. Form. Asp. Comput. 22 (3-4), 459–482.

Villegas, N. M., Müller, H. A., Tamura, G., Duchien, L., Casallas, R., 2011. A framework for evaluating quality-driven self-adaptive software systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '11. ACM, New York, NY, USA, pp. 80–89.

Von der Massen, T., Lichter, H., 2004. Deficiencies in Feature Models. In: Männistö, T., Bosch, J. (Eds.), Workshop on Software Variability Management for Product Derivation - Towards Tool Support.
URL `http://www.soberit.hut.fi/SPLC-WS/AcceptedPapers/Massen.pdf`

Wang, Y., McIlraith, S., Yu, Y., Mylopoulos, J., 2009. Monitoring and diagnosing software requirements. Automated Software Engineering 16, 3–35.

Weber, B., Reichert, M., Rinderle-Ma, S., September 2008. Change patterns and change support features - enhancing flexibility in process-aware information systems. Data Knowl. Eng. 66, 438–466.

Weyns, D., Iftikhar, M. U., de la Iglesia, D. G., Ahmad, T., 2012. A survey of formal methods in self-adaptive systems. In: Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering. C3S2E '12. ACM, New York, NY, USA, pp. 67–79.

White, J., Schmidt, D. C., Wuchner, E., Nechypurenko, A., 2007. Automating product-line variant selection for mobile devices. In: Proceedings of the 11th International Software Product Line Conference. IEEE Computer Society, Washington, DC, USA, pp. 129–140.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., 2012. Experimentation in Software Engineering. Springer.

Xu, C., Cheung, S., Ma, X., Cao, C., Lu, J., 2012. ADAM: Identifying defects in context-aware adaptation. Journal of Systems and Software 85 (12), 2812 – 2828.

Zave, P., Jackson, M., Jan. 1997. Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol. 6 (1), 1–30.