UNIVERSIDAD POLITECNICA DE VALENCIA

PROS
Centro de Investigación en Métodos de Producción de Software

# Informe Técnico / Technical Report




## Towards Dynamic Service Compositions with Models at Runtime

Germán H. Alférez and Vicente Pelechano






| | | | | | |
|---|---|---|---|---|---|
| Ref. #: | ProS-TR-2012-05 | | | | |
| Title: | Towards Dynamic Service Compositions with Models at Runtime | | | | |
| Author (s): | Germán H. Alférez and Vicente Pelechano | | | | |
| Corresponding author (s): | harveyalferez@um.edu.mx<br>pele@dsic.upv.es | | | | |
| Document version number: | 1.0 | Final version: | Yes | Pages: | 18 |
| Release date: | November 2012 | | | | |
| Key words: | Service Compositions, Models at Runtime, Dynamic Evolution of Composition Schemes, Dynamic Adaptation of Composition Instances | | | | |


Universidad Politécnica de Valencia · Camino de Vera s/n · Edificio 1F · 46022 Valencia Spain ·
T. +34 96 387 70 07 Ext. 83530 · M. + 34 619 543 623 · F. +34 96 387 73 59 · info@pros.upv.es · www.pros.upv.es

# Towards Dynamic Service Compositions with Models at Runtime

Germán H. Alférez[1] and Vicente Pelechano[2]

[1] Facultad de Ingeniería y Tecnología, Universidad de Montemorelos,
Apartado 16-5, Montemorelos N.L., 67500, Mexico
harveyalferez@um.edu.mx
[2] Centro de Investigación en Métodos de Producción de Software (ProS),
Universitat Politècnica de València, Camí de Vera s/n, E-46022, Spain
pele@dsic.upv.es

**Abstract.** Web services run in complex computing infrastructures where arising events may affect the quality of the system. Thus, it is desirable for service compositions to self-configure in order to deal with these events. Most research implements cumbersome variability constructs at the language level to implement dynamic service compositions. Also, research has focused on managing variability at the composition schema level. Nevertheless, the adaptation of running composition instances is an open research field. In this paper, we propose a framework that uses easy-to-understand models at runtime to: 1) dynamically evolve versions of the composition schema pro-actively; and 2) carry out strategies to dynamically adapt composition instances. Autonomic behavior is achieved by our Model-based Reconfiguration Engine for Web Services (MoRE-WS), which leverages models at runtime for decision-making. An evaluation demonstrates that our approach has good performance for an increasing number of evolved service operations.

## 1 Introduction

In nature, organisms adapt themselves to be more suitable to their environment. As organisms live in intricate, changing environments, software is executed in complex and heterogeneous computing infrastructures in which a diversity of events may arise (e.g. security threats and server failures). Therefore, it is desirable to translate the ideas of self-adjustment in the natural world to software in order to solve these situations.

A good example of systems that require adjusting themselves are the ones based on Web service compositions (or simply called *service compositions*). Web services run in a *context*, which is any information that can be used to characterize their situation [14] (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 example, the response time of a Web service operation may have greatly increased or may have become unavailable. Therefore, it is appropriate to count on *dynamic service compositions* that manage themselves at runtime to keep service-level agreements (SLAs), offer extra functionality depending on the deployment context, protect the system, or make the system more usable.

Most approaches for dynamic service compositions have tended to implement dynamic behavior on the language level [13,15,6], which can be complex and time-consuming as the system grows. In addition, the trend has been on the reactive dynamic evolution of the composition schema (e.g. described in WS-BPEL) to solve situations that arise in the context [4,16]. We define *dynamic evolution* as the modification at runtime of the *composition schema*, which specifies a business process. As expected, reactive dynamic evolutions may lead to unwanted consequences since they call for changes in the service composition when the problem is already evident to users or systems. Furthermore, several versions of the composition schema may be running at the same time because of unfinished composition instances (hereafter, *instances*) attached to old schemes. However, related work lacks mechanisms to dynamically evolve specific composition schema versions [13,15,6,4,16].

Despite the research trending on dynamic evolutions, we argue that it is also necessary to count on mechanisms for the dynamic adaptation of instances. We define *dynamic adaptation* as the migration of instances to a new version of the composition schema. However, the instance migration process is an open and complex research field [23]. For example, it is expensive to abort all ongoing instances that depend on a composition schema and replay them according to a new version of the schema.

In this paper, we propose a framework to support dynamic service compositions using easy-to-understand and semantically rich models at runtime. *Models at runtime* can be defined as causally connected self-representations of the associated system that emphasize the structure, behavior, or goals of the system from a problem space perspective [8]. The contribution of our model-driven framework is twofold. First, our framework generates actions to guide the proactive dynamic evolution of composition schema versions. Specifically, we propose that service compositions be abstracted as a set of *features* (logical units of behaviors that are specified by a set of functional and non-functional requirements [9]) in variability models. Thus, evolution actions are described in terms of the activation or deactivation of features in *causally connected* variability models (i.e., changes in these models cause the service composition to evolve and vice versa). Furthermore, the a priori analysis of variability models avoids problems before they affect the system. Second, our framework employs strategies for the dynamic adaptation of instances. These strategies decide whether to migrate to a new version of the composition schema or keep running on an old version of the schema. The analysis for dynamic adaptations is carried out on abstract and technology-independent representations of composition schema versions. The proposed framework is supported by our Model-based Reconfiguration Engine for Web Services (MoRE-WS), which generates abstract evolution and adaptation actions using models at runtime. Then, MoRE-WS uses these actions to create and deploy WS-BPEL code.

The remainder of this paper is structured as follows: Section 2 describes a motivating scenario. Section 3 presents an overview of our framework. Section 4 describes the models that support dynamic service compositions. Section 5 describes our model-driven approach for the dynamic evolution of composition schemes. Section 6 describes model-driven strategies for the dynamic adaptation of instances. Section 7 introduces a running prototype. Section 8 presents the evaluation of our framework. Section 9 presents related work, and Section 10 presents conclusions and future work.

## 2 Motivating Scenario

The BPMN model in Figure 1 represents a composition schema that supports online book shopping at Orange Country Bookstore. 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).

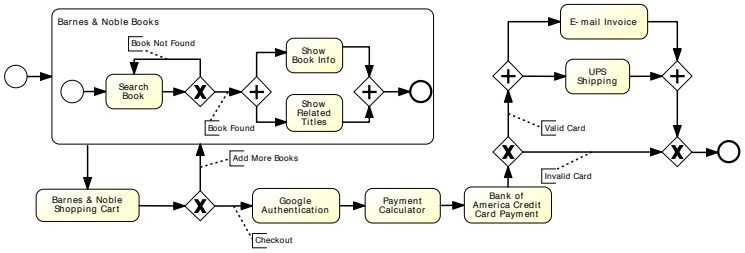


**Fig. 1.** A BPMN model that represents a composite service for online book shopping

The business process starts when a customer looks for a book on the website of Orange Country Bookstore. The searching operation is provided by the Search Book Web service, which is part of the Barnes & Noble Books composite service. If the 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 the book is not found, the customer must refine the search for the book. In the next step, the customer adds books into the shopping cart through the Barnes & Noble Shopping Cart Web service. 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.

As a business differentiator, Orange Country Bookstore requires that its online book shopping process be available 24/7. However, several context events may arise in a heterogeneous computing infrastructure, e.g. any third-party Web service operation may fail or perform below required SLAs. Also, since the service composition supports a critical business process, it is impossible to shut down the system with all the running instances to implement any necessary changes. Moreover, it is desirable to count on mechanisms to support dynamic behavior that are easy to understand by non-technical stakeholders to accelerate time-to-market and facilitate maintenance.

The aforementioned situations help us to identify the following *challenges* for dynamic service compositions: 1) reactive dynamic evolutions of the composition schema are triggered when problematic context situations have already affected the executing service composition. A better strategy is to proactively detect and solve problematic

situations in the context [5]; 2) different versions of the composition schema may be running concurrently. Therefore, it is necessary to count on mechanisms to dynamically evolve specific versions that may be affected by a context event; 3) instance migrations to a new version of the composition schema have to be done in a controlled way in order to avoid unexpected results; and 4) implementing the actions to carry out dynamic evolutions and dynamic adaptations on the language level can be complex and time-consuming. Expressing these actions as easy-to-understand abstractions can facilitate the development and maintenance of the logic behind dynamic management. Also, these actions should be autogenerated at runtime to reduce human workload.

## 3 Model-Driven Framework for Dynamic Service Compositions

Web services represent the most common realization of Service-Oriented Architecture (SOA) [18], making the development of inter-operable Internet applications possible. However, Web services run in heterogeneous and complex contexts. Thus, it is desirable to count on dynamic service compositions that can deal with context changes at runtime.

In order to support dynamic service compositions and solve the challenges presented in Section 2, we introduce the following strategy. First, we describe mechanisms to express where and how service compositions and their running instances can be adjusted to deal with arising context events. These mechanisms must be as easy to understand and as highly abstract as possible. Afterwards, we provide a supporting infrastructure that detects context changes and determines what to do with composition schema versions and instances. Evolution and adaptation actions are autogenerated at runtime by processing the knowledge in models at runtime. Model transformations are carried out to generate WS-BPEL code, which is hot-deployed in a WS-BPEL engine.

To make the aforementioned strategy a reality, we propose a framework that uses models at runtime to support dynamic service compositions. The underpinnings of our framework are as follows: 1) support for the dynamic evolution of composition schema versions that may be affected by context events; 2) use of highly-abstract *context conditions* to check for events arising in the current context. If a context condition is accomplished, then service composition adjustments are triggered. In other words, a context condition works as an SLA; 3) autogeneration of *evolution policies* that state the actions required to evolve composition schema versions to better fit the context; 4) use of dynamic adaptation strategies to guide the migration of running instances to new composition schema versions; and 5) self-adjustment of service compositions without having to restart the system. MoRE-WS is the most important component in the framework because it controls model-driven dynamic evolutions and adaptations to deal with arising context events. MoRE-WS is an extension of our previous work called MoRE [12]. MoRE translates context changes in the smart-home domain into changes in the activation and deactivation of features at runtime.

The framework is divided into two parts, namely dynamic evolution and dynamic adaptation (see Figure 2). In the dynamic evolution part, the Context Monitor periodically gets information from the context. Then, MoRE-WS updates the *context model* with the collected information in order to reason about the current context situation. If a context condition has been accomplished, then MoRE-WS activates or deactivates

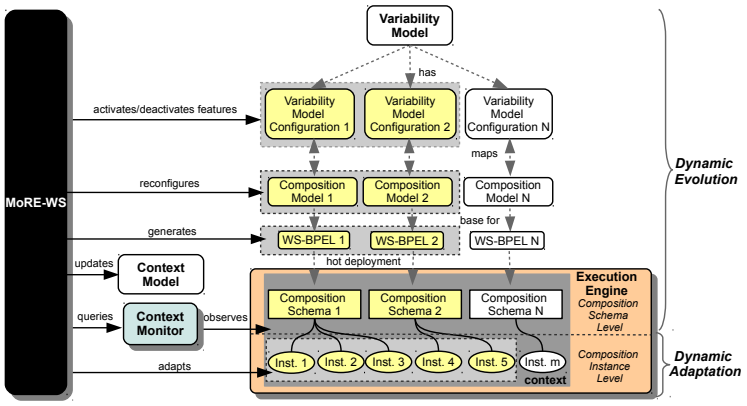


**Fig. 2.** A model-driven framework to support dynamic service compositions

features in variability model configurations. A *variability model* describes the *variants* (representations of variability objects within domain artifacts [21]) in which a composition schema can change. Since several versions of the composition schema may be running at the same time, there is a variability model configuration for each composition schema version. The changes that are carried out in variability model configurations are reflected in *composition models* that abstract the underlying composition schemes. Afterwards, composition models are used to generate and hot deploy WS-BPEL code that orchestrates service operations. In the dynamic adaptation part, MoRE-WS uses a set of strategies that are based on information in composition models to decide whether or not to migrate instances to new composition schema versions. Section 4 presents the models that are used in our framework. Then, Section 5 and Section 6 describe the dynamic evolution and dynamic adaptation parts.

## 4 Models that Support Dynamic Service Compositions

The models that are going to be leveraged during execution to support dynamic service compositions are created at design time (see Figure 3). Besides creating the composition model to represent the service composition (e.g. with a BPMN diagram, a Petri net, or a UML Activity diagram), we propose the creation of a set of additional models to support dynamic behavior. First, we propose the variability model to describe the variants in which a composition schema 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 [19]. In order to replicate the changes that are carried out in the variability model in the composition model, which represents the service composition, it is necessary to count on a weaving model. The weaving model works as a *bridge* between the elements in the composition and variability models. Finally, we propose a context model for the formal analysis of context information.

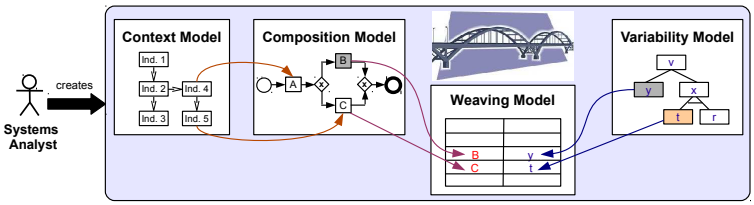


**Fig. 3.** Models that support dynamic service compositions

### 4.1 Composition Model

The composition model represents the service composition (such as the one in Figure 1). At runtime, this model is adjusted according to context changes. BPMN was chosen to represent the elements in a service composition for the following reasons: 1) BPMN models are suitable for representing sequences and dependencies between Web service operations; and 2) the use of languages such as WS-BPEL could be more difficult to understand by business analysts and managers who are involved in the development process and prefer to visualize the service composition in a flowchart format.

### 4.2 Variability Model

Even when the composition model can be used to represent the underlying service composition, it does not represent the variability that a service composition may have. Specifically, it is necessary to realize the set of variants that can be part of a service composition at a particular moment. Thus, we propose a variability model to describe the service composition variability [1].

Feature modeling was chosen for variability modeling and analysis because it can represent variants in a very concise taxonomic form, and it has good tool support for variability reasoning [7]. Figure 4 shows the feature model for our case study. Features represent the functionalities of the Web-service-based system in a coarse-grained fashion [1,2]. Therefore, adjustments in the service composition can be described in terms of the activation or deactivation of features in the variability model. *Variation points* are used to express decisions leading to different variants at runtime (e.g. the UPS Shipping, FedEX Express, and DHL Delivery features are variants that can accomplish the Shipment variation point). 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. Each variant is denoted with an *optional* feature because it can be added or removed according to specific needs. In line with customer requirements, a particular variant in a variation point will have a higher possibility of being chosen than the other variants at runtime. As a result, we propose that systems analysts assign a specific priority to each variant.

In order to give semantics to the variability model, a connection between the highly abstract elements in the variability model and the low-level service operations is necessary. 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

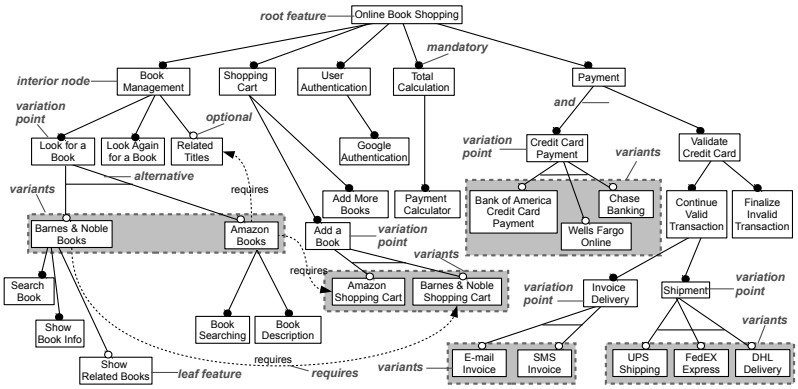


**Fig. 4.** Variability model for the online-book-shopping case study

and the elements in the composition model could be used to support dynamic adjustments in the underlying service composition. In order to define this bridge, we propose the creation of a weaving model [1,2]. The weaving model contains a set of links. Each *link* has the following endpoints: the first endpoint refers to model elements that make up a composition model. In our case study, these elements are BPMN tasks and subprocesses, which represent Web service and composite service operations, respectively; the second endpoint refers to features in the variability model.

### 4.3 Context Model

In order to express 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 [1]. The context model provides a strong semantic vocabulary for the representation of context knowledge and for describing specific situations in the context. The main benefit of the context model is to enable the analysis of the domain knowledge using first-order logic. Specifically, we make use of the Web Ontology Language[1] (OWL) to analyze the context information that is captured by the Context Monitor.

Figure 5 shows a fragment of the context model for our case study. Individuals have the following datatype properties: *hasAddress* indicates the address of the service operation; *isAvailable* indicates whether or not 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 (response time plus execution time).

At design time, systems analysts extract context conditions from the context model as Boolean expressions. Each context condition is represented as a triple in the form of *(subject, predicate, object)* [1]. Two context conditions in our case study are the following: 1) *Barnes&NobleBooksUnavailable = (Barnes&NobleBooks, isAvailable, false)*,

[1] http://www.w3.org/TR/owl-ref/: World Wide Web Consortium (W3C).

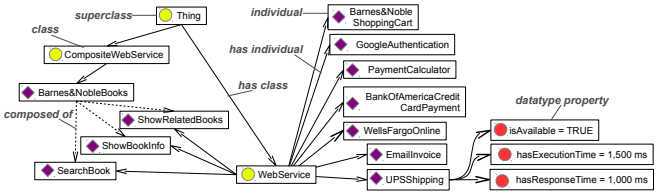


**Fig. 5.** A fragment of the context model for the online-book-shopping case study

which is triggered when the *Barnes & Noble Books* composite service is currently unavailable; and 2) *UPSShippingHiRespTime = (UPSShipping, hasResponseTime, >2,000 ms)*, which is triggered when the current response time of the *UPSShipping* Web service operation is greater than 2 seconds.

## 5 Dynamic Evolution

When dealing with arising context events, it is unthinkable to carry out manual adjustments in composition schema versions because of the inherent intricacy of service compositions and desired prompt responses. Furthermore, critical systems cannot be stopped in order to carry out the necessary changes. Therefore, an autonomic approach is a must for service compositions that have to adjust to the context.

In this section, we describe models at runtime as a means to proactively autogenerate evolution policies with simple and semantically rich instructions. To this end, the discovery and reasoning of any problematic context event is carried out in abstract models at runtime before the problem is evident to users or other systems. Specifically, *evolution policies* support the evolution of multiple composition schema versions by activating and deactivating features in variability model configurations according to context conditions. In order to reflect the changes in variability model configurations on the running service composition versions, we propose defining *reconfiguration plans*. These plans contain highly abstract reconfiguration actions to evolve composition models that represent the underlying composition schema versions. Autonomic mechanisms are defined to transform evolved composition models into WS-BPEL code. The generated WS-BPEL code is hot-deployed in the Execution Engine.

### 5.1 Generating Evolution Policies

Instead of manually creating evolution policies, which is a time-consuming process, our framework makes use of semantically rich models to autogenerate them. The model-driven generation of evolution policies has the following benefits: 1) project resources can be saved because the models created at design time are reused at runtime; 2) technology bridges are avoided between design and execution artifacts. Therefore, the effort is reduced because there is no need to build such bridges; 3) models are independent of the underlying technologies. Therefore, they can describe autonomic behavior through abstract and easy-to-understand concepts; and 4) models can hide the complexity of the evolution space, thus facilitating the management of autonomic behavior.

In our previous research [1], we showed how MoRE-WS queries the context information that is collected by the Context Monitor and updates the context model accordingly. With this information, MoRE-WS determines whether or not any context condition has been accomplished. Since a given context condition can trigger the activation or deactivation of several features at runtime, we propose using the *resolution* concept to represent the set of changes in the variability model triggered by a context condition. Thus, resolutions are the evolution policies that express the transitions among different configurations of a 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 the variability model (*VM*) and the state (*S*) of the feature. Each resolution is associated to a context condition (*C*).

$$R_C = \{(F, S) \mid F \epsilon [VM] \wedge S \epsilon \{Active, Inactive\}\} \quad (1)$$

The autogeneration of resolutions during execution is possible with the following steps: 1) MoRE-WS selects the problematic service operation that has launched the context condition (e.g. an unavailable service operation or a service operation with an execution time that violates an SLA); 2) MoRE-WS looks for the service composition versions that may be potentially affected by the problematic service operation; and 3) MoRE-WS tries to generate a resolution for each service composition version that may be negatively affected by the problematic service operation.

Models at runtime are used in the aforementioned steps to generate resolutions as follows. In the first step, MoRE-WS discovers the problematic service operation by observing the arising context condition, which is expressed in terms of elements in the context model. In the second step, since composition models abstract service compositions, MoRE-WS looks for all the composition model versions that define an abstraction of the problematic service operation (e.g. in terms of a BPMN activity). This is key information because we are interested in evolving the service composition versions that may be affected by particular service operations. In the last step, since resolutions are expressed in terms of features, MoRE-WS carries out the following steps to activate and deactivate features in variability model configurations:

1. MoRE-WS runs through the weaving model to find the mapping between the problematic service operation (e.g. a BPMN activity) and a feature in a variability model configuration.
2. MoRE-WS looks for variants in a variability model configuration to fix the feature that represents the problematic service operation. If this feature is a leaf feature, then MoRE-WS deactivates it and activates the variant that has the highest priority in the variation point on which the leaf feature depends. If the feature that represents the problematic service operation is an interior node (which maps to a composite service [2]), MoRE-WS deactivates its subfeatures and activates a variant together with its subfeatures.
3. If a variant has a "requires" relationship with a feature or interior node, then MoRE-WS also activates or deactivates the required feature or interior node.
4. Finally, resolutions are not generated if the feature to be adjusted is mandatory (i.e., there are no variants to be used during the evolution process).

For example, the *Barnes&NobleBooksUnavailable* context condition has been triggered. In this case, MoRE-WS uses the strategy above to generate a resolution for the initial configuration of the case study (in Figure 1). Figure 6 depicts the result after applying this resolution. Amazon Books and its required functionalities are activated.

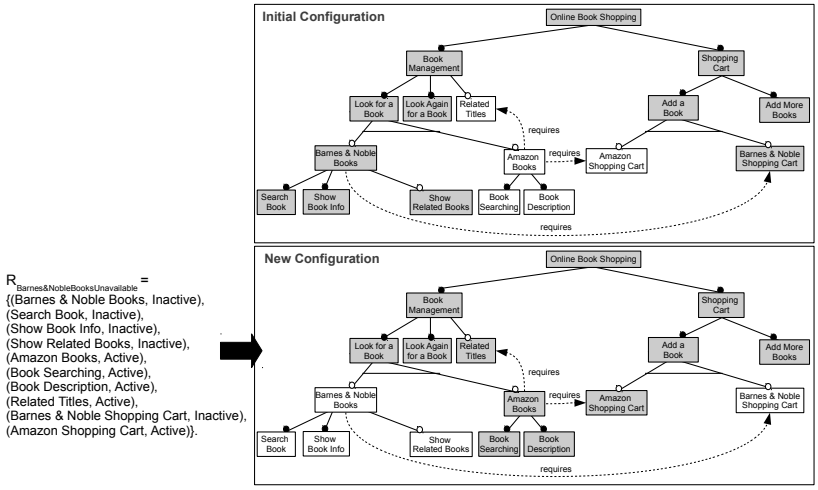


**Fig. 6.** Resolution application example. Highlighted features are activated

### 5.2 Reflecting Variability Model Changes in Composition Schemes

The changes that are carried out in variability model configurations have to be reflected into the underlying composition schema versions. Instead of programming complex instructions to modify composition schemes, we propose first carrying out evolutions at the composition model level. Afterwards, evolved composition models will be used to generate WS-BPEL code that orchestrates service operations.

We propose the creation of *reconfiguration plan*s to translate the changes in evolved variability model configurations into composition model versions. A reconfiguration plan contains a set of reconfiguration actions to modify a particular composition model version. Reconfiguration actions are stated as *composition model increments* ($CM\triangle$) and *composition model decrements* ($CM\nabla$). These operations take a resolution as input, and they calculate the modifications to a composition model version by adding ($CM\triangle$) or removing ($CM\nabla$) BPMN elements.

Instead of manually coding reconfiguration plans at design time (which can be time-consuming and error prone), we propose to autogenerate them at runtime using the knowledge in models at runtime. To this end, MoRE-WS queries the weaving model to find the mappings between the features that are expressed in resolutions and their related BPMN elements. In this way, a given service operation, which is represented by a BPMN element in a composition model version, will be invoked in an evolved

service composition if and only if its related feature in a resolution is *active*. That is, a composition model is evolved through the activation or deactivation of features.

For example, given $R_{Barnes\&NobleBooksUnavailable}$ for the initial configuration in Figure 1, the result is the following reconfiguration plan: *CM∇ = {Barnes & Noble Books, Barnes & Noble Shopping Cart}* and *CM△ = {Amazon Books, Related Titles, Amazon Shopping Cart}*. Figure 7 shows the evolved composition model.

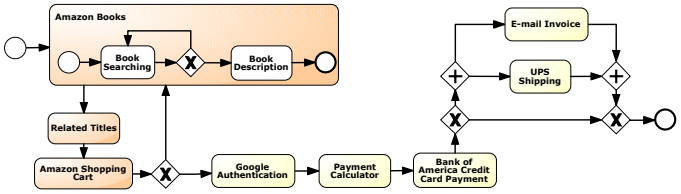


**Fig. 7.** Evolved composition model for the case study

### 5.3 Generating WS-BPEL Code and Hot Deployment

In order to realize our approach in an industrial environment, it is necessary to transform the evolved composition models into WS-BPEL code and hot deploy this code on a WS-BPEL engine. To this end, MoRE-WS converts composition model versions through model-to-text transformations into WS-BPEL files, which support service orchestrations. In order to translate a BPMN composition model into WS-BPEL code, MoRE-WS makes use of the Babel project[2]. Nevertheless, although the Babel tool uses model-to-text transformations to generate a WS-BPEL document from a BPMN model, the generated WS-BPEL document is not complete. For example, it lacks information about the partner links of services participating in the process and the variables used in the process. Thus, MoRE-WS injects pieces of XML code into particular points in the initial WS-BPEL file in order to obtain ready-to-run WS-BPEL code. Then, MoRE-WS puts the complete WS-BPEL file into the deployment directory. We have implemented a versioning strategy for the deployment directory to prevent the EXECUTION ENGINE 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 evolution. New instances run according to the latest version.

## 6 Dynamic Adaptation

The dynamic evolution of composition schema versions is one side of the coin. The other side is the dynamic adaptation of instances. However, 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.

[2] http://www.bpm.scitech.qut.edu.au/research/projects/oldprojects/babel/tools/: Babel tools.

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. This is an important aspect because uncontrolled instance migration will lead to inconsistencies or errors [24].

The applicability of the dynamic adaptation strategies is described in a set of instances in our case study. The left-hand side of Figure 8 depicts the first version of a composition model (according to Figure 1) and four instances that run conforming to this model. The right-hand side of the figure shows the second version of the composition model after a dynamic evolution has been triggered to deal with the *Barnes&Noble-BooksUnavailable* context condition. Each composition model version has a set of activities that have evolved ($EvolvedAc$), and running instances have the following sets of activities: 1) a set of already executed activities ($ExecutedAc$); 2) a current running activity or event – e.g. a start or end event ($CurrentAcEv$); and 3) a set of coming activities that have not yet been executed in the workflow ($ComingAc$).

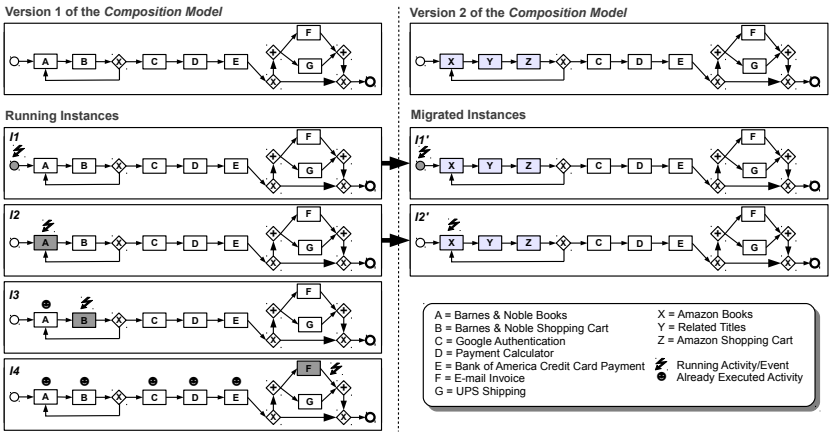


**Fig. 8.** Applicability of the strategies for dynamic adaptation

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 [24]. Specifically, an instance *I* is *compliant* with a composition schema *S*, if the current execution history of *I* can be created based on *S* [22]. 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 (it keeps the updated $CurrentAcEv$ information).

We propose migrating instances in two cases. The first case is if all the activities in the instance have not yet been executed (i.e., $ExecutedAc = \emptyset$). Therefore, adaptations will be done in $ComingAc$ and will not affect the previous results generated by $ExecutedAc$ (see *I1*). The second case is if $CurrentAcEv$ has evolved at the composition model and adaptations do not have to be carried out in $ExecutedAc$ (i.e., $ExecutedAc \bigcap EvolvedAc = \emptyset$) – see *I2*. Therefore, this strategy prevents the

adaptation of $ExecutedAc$ to avoid inconsistencies in $CurrentAcEv$ and $ComingAc$. During migrations, MoRE-WS moves the instances that depend on evolved composition schema versions from an active state to a passive state to avoid undesirable effects. An instance is kept in the passive state until a strategy has been applied on it (i.e., when MoRE-WS has decided whether or not the instance can be migrated). Afterwards, MoRE-WS reactivates passive instances.

On the other hand, instances do not migrate when it is unsafe or unnecessary to do so. Specifically, an instance keeps running on the old version of the composition schema in two cases. The first case is if $CurrentAcEv$ has evolved and any other activity in $ExecutedAc$ has also evolved (i.e., $ExecutedAc \bigcap EvolvedAc \neq \emptyset$). For example, *I3* is not migrated since the Barnes & Noble Books service operation (which had to evolve) has already been executed. The dynamic adaptation of the Barnes and Noble Shopping Cart service operation may cause an incongruence with the Barnes & Noble Books service operation (they cannot coexist). The second case is if all the activities that could be adapted have already been executed ($ExecutedAc \bigcap EvolvedAc = EvolvedAc$). In this case, it is better to let the instance finish its execution instead of spending resources on unnecessary migrations (see *I4*).

## 7 Prototype Implementation

A prototype system validates the feasibility of our approach. The composition model and the variability model are specified in the XML Metadata Interchange (XMI) format. They are processed by the software infrastructure provided in the Eclipse Modeling Framework (EMF)[3] to specify and execute queries against them at runtime. The composition model conforms to the BPMN metamodel, and the variability model conforms to the MOSKitt4SPL[4] metamodel. The weaving model is created in the ATLAS Model Weaver[5] tool. Context conditions are specified as SPARQL Protocol and RDF Query Language[6] (SPARQL) queries to the ontology in the context model [1]. Also, MoRE-WS uses the EMF Model Query (EMFMQ)[7] to activate or deactivate features in variability model versions during execution. SALMon [3] implements the Context Monitor because its components are mostly technology-independent and they act as services, making its architecture customizable for our purpose. Apache ODE[8] 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 engine, our approach does not require changes to the engine. The information about instances is retrieved by the BPEL Management API of Apache ODE and updated by MoRE-WS in the composition model versions. Figure 9 depicts the running prototype when the *Barnes&NobleBooksUnavailable* context condition has been triggered. The composition model shows the activities that have already been executed and the current running

[3] http://www.eclipse.org/modeling/emf/: EMF.
[4] http://www.pros.upv.es/m4spl: MOSKitt4SPL.
[5] http://www.eclipse.org/gmt/amw/: ATLAS Model Weaver.
[6] http://www.w3.org/TR/rdf-sparql-query/: SPARQL.
[7] http://www.eclipse.org/modeling/emf/: EMF Model Query.
[8] http://ode.apache.org/: Apache ODE.

activity in an instance. The figure also shows a variability model configuration, a console with evolution and adaptation actions, and the models that are used at runtime.

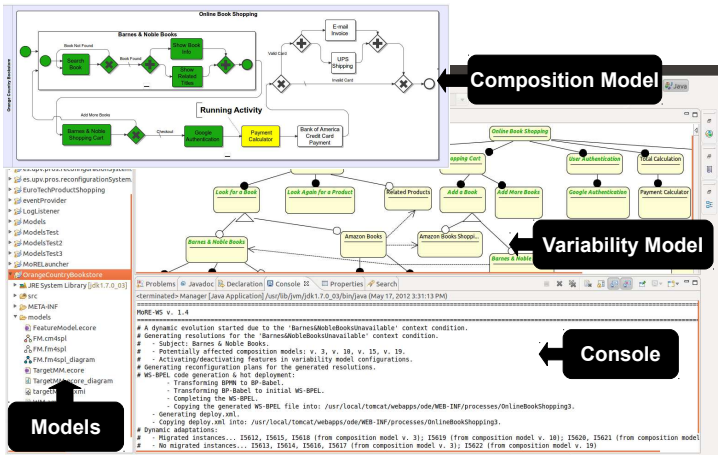


**Fig. 9.** Running prototype when facing the *Barnes&NobleBooksUnavailable* context condition

## 8 Evaluation

In this section, we describe the evaluation results of our framework. Specifically, we evaluate the following two aspects: 1) the dynamic evolution performance when the number of evolved service operations increases in service compositions of a growing size; and 2) the dynamic adaptation performance for an increasing number of evolved service operations. Since evolutions and adaptations are carried out at runtime, the performance of model-driven operations is a key aspect to be evaluated. Finally, we provide a brief discussion of the realizability of our approach during the development process.

The experiments were carried out on a PC with an Intel Core 2 Duo 2.0 GHz processor and 4 GB RAM with Ubuntu version 10.04 and Kernel Linux version 2.6.32-36-generic. The Web services ran on Apache Axis2[9] version 1.6.1, which was deployed as a WAR distribution on Apache Tomcat[10] 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. SALMon and MoRE-WS ran on the same device. Six composition schemes of increasing size were used in the experiments (see Table 1)[11]. The objective was to evolve the first version of each one of these composition schemes into a second version. The experiments were carried out with eighty instances running on each composition schema.

[9] http://axis.apache.org/axis2/java/core/: Apache Axis2.

[10] http://tomcat.apache.org/: Apache Tomcat.

[11] $CS_1$ implements our case study when the the *Barnes&NobleBooksUnavailable* context condition has been triggered.

| Composition Schema | CS1 | CS2 | CS3 | CS4 | CS5 | CS6 |
|---|---|---|---|---|---|---|
| Service Operations | 7 | 16 | 33 | 50 | 68 | 85 |
| Evolved Service Operation | 2 | 5 | 8 | 12 | 17 | 25 |

**Table 1.** Composition schema configurations that were used in the experiments

Figure 10 depicts the results of the experiments. Specifically, section a presents the execution time in milliseconds during dynamic evolutions. The results are divided into the four main tasks, which were carried out during dynamic evolutions. The *find a context condition*, *generate a resolution*, and *generate a reconfiguration plan* operations were fast, even for large composition schemes. The execution time of the *find a context condition* and *generate a reconfiguration plan* operations remained stable while the *generate a resolution* operation had an exponential growth. The *generate WS-BPEL & hot deployment* operation took the longest time. The most expensive task in this operation was the transformation from the evolved composition model to the initial WS-BPEL file using Babel, which took around 95% of the time in all cases. Therefore, in order to reduce the impact on the performance of instances, MoRE-WS starts the model-driven analysis of dynamic adaptation strategies before triggering WS-BPEL code generation and hot deployment. As a result, instances do not have to be kept in the passive state longer than necessary.

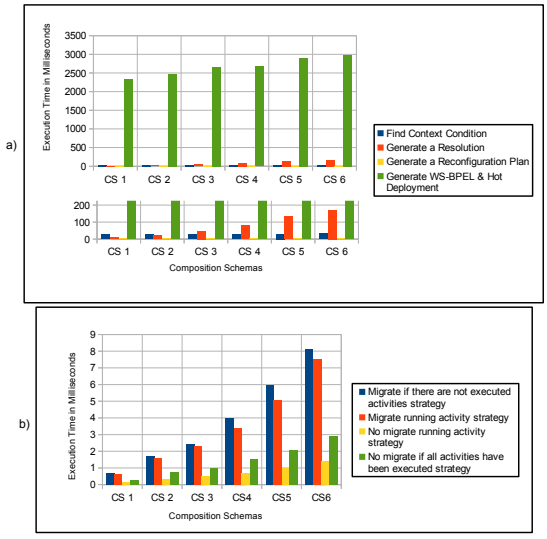


**Fig. 10.** Execution time for a) dynamic evolutions and b) dynamic adaptation strategies

Section b in Figure 10 presents the average execution times in milliseconds taken in the dynamic adaptation of the instances that run on $CS_1$ to $CS_6$. The strategies that required migrations took the longest time since instances had to be adapted. Between the

two strategies that do not require migrations, the *no migrate if all activities have been executed* strategy took a longer time since MoRE-WS had to be sure that all evolved activities had previously been executed. The execution time for dynamic adaptations grew exponentially as service operations and evolved service operations increase.

The amount of effort required by our approach during the development process was minimal. At design time, the creation of the supporting models was straightforward because it was supported by enterprise modeling tools. Also, no extra work was required to carry out the model-driven operations at runtime because MoRE-WS automatizes the dynamic adjustments. Also, since our approach is transparent to the Execution Engine, the engine does not have to be modified for hot deployment.

In summary, our solution provided a feasible performance even for large service compositions with an increasing number of evolved service operations. Specifically, the generation of proactive dynamic evolution actions were carried out without excessively affecting the execution time. The generation of WS-BPEL code from composition model versions took the longest execution time. However, this situation was mitigated by executing the dynamic adaptation strategies in parallel with the generation of WS-BPEL code and hot deployment. Also, the strategies for the dynamic adaptation of instances had good execution times in all cases. Last but not least, our approach required a minimum effort during the development process.

## 9 Related Work

Research works related to dynamic service compositions have tended to implement variability constructs at the language level. For example, SCENE [13] 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 [15] 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 [6], Web service monitoring directives and recovery strategies are expressed with two languages. We argue that implementing and managing dynamic evolutions at the language level is complex, especially in large systems. Moreover, the trend has been on reactive dynamic evolutions [4,16], which is time-consuming and can lead to unwanted results.

We highlight relevant related approaches that use models at runtime to support the dynamic evolution of service compositions. In [10], Bosloper et al. introduce a tool that offers components for monitoring and reconfiguring Web service-centric systems. However, the supporting models and their related reconfiguration mechanisms are not presented in a detailed way. In [11], Calinescu et al. present a tool-supported framework for the QoS management of self-adaptive, service-based systems. It combines formal specification of QoS requirements, model-based QoS evaluation, 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 in the formalization of these QoS requirements. In [17], Menasce et al. present a model-driven framework that provides runtime evolution of service compositions in response to changing operating conditions. Dynamic evolutions are carried out through patterns, which are not autogenerated at runtime. Thus,

a system restart is required to modify them. Moreover, it is not clear how the service coordination logic can be deployed in a WS-BPEL engine. In [20], Morin et al. propose combining model-driven and aspect-oriented techniques to support runtime variability from requirements to execution. However, this solution is not focused on Web services. In [5], Aschoff et al. describe an approach for the proactive evolution of service compositions using an *execution model*. However, to the best of our knowledge, the aforementioned approaches do not handle the dynamic evolution of multiple versions of the composition schema. Moreover, they do not describe the mechanisms for the dynamic adaptation of instances. Therefore, it is unclear how instance migrations are done.

The approaches that are presented above make evident the trend towards implementing reactive dynamic evolutions of service compositions at the language level, which can be complex and time-consuming. Model-driven approaches have made a first step in managing dynamic service compositions at a higher abstraction level. However, they lack support for the dynamic evolution of several versions of the composition schema and for the dynamic adaptation of instances. Moreover, most model-driven works propose reactive solutions for self-adjusting service compositions.

## 10 Conclusions and Future Work

In this paper, we have presented a framework that is based on easy-to-understand models at runtime to support the proactive dynamic evolution of composition schema versions, and guide the dynamic adaptation of running instances. Specifically, we focus on three aspects of dynamic evolution: the generation of evolution policies, the generation of reconfiguration plans, and the generation of WS-BPEL code and hot deployment. Dynamic adaptations are possible through a set of model-driven strategies. Autonomic behavior is achieved by a prototype based on MoRE-WS. An evaluation demonstrates the feasibility of our approach. As future work, we will look for alternatives for the transformation from BPMN composition models to WS-BPEL code to reach better execution times. One possible solution is to connect reusable code fragments to features. At runtime, these fragments can be added or removed accordingly.

## References


1. Alférez, G.H., Pelechano, V.: Context-aware autonomous web services in software product lines. In: SPLC. pp. 100–109 (2011)
2. Alférez, G., Pelechano, V.: Systematic reuse of web services through software product line engineering. In: 9th IEEE European Conference on Web Services (ECOWS). pp. 192 –199 (sept 2011)
3. Ameller, D., Franch, X.: Service level agreement monitor (SALMon). In: Proceedings of the Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008). pp. 224–227. IEEE Computer Society, Washington, DC, USA (2008)
4. Ardagna, D., Comuzzi, M., Mussi, E., Pernici, B., Plebani, P.: PAWS: A framework for executing adaptive web-service processes. IEEE Softw. 24(6), 39–46 (Nov 2007)
5. Aschoff, R., Zisman, A.: Qos-driven proactive adaptation of service composition. In: Proceedings of the 9th international conference on Service-Oriented Computing. pp. 421–435. ICSOC'11, Springer-Verlag, Berlin, Heidelberg (2011)

6. Baresi, L., Guinea, S.: Self-supervising BPEL processes. IEEE Trans. Softw. Eng. 37, 247–263 (March 2011)
7. Benavides, D., Cortés, R.A., Trinidad, P.: Automated Reasoning on Feature Models. In: The 17th Conference on Advanced Information Systems Engineering (CAiSE'05). LNCS (June 2005)
8. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. Computer 42, 22–27 (October 2009)
9. Bosch, J.: Design and use of software architectures: adopting and evolving a product-line approach. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (2000)
10. Bosloper, I., Siljee, J., Nijhuis, J., Hammer, D.: Creating self-adaptive service systems with DySOA. In: Proceedings of the Third European Conference on Web Services. pp. 95–. ECOWS '05, IEEE Computer Society, Washington, DC, USA (2005)
11. Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic QoS management and optimization in service-based systems. IEEE Transactions on Software Engineering 37, 387–409 (2011)
12. Cetina, C., Giner, P., Fons, J., Pelechano, V.: Autonomic computing through reuse of variability models at runtime: The case of smart homes. Computer 42, 37–43 (October 2009)
13. Colombo, M., Di Nitto, E., Mauri, M.: SCENE: A service composition execution environment supporting dynamic changes disciplined through rules. In: Dan, A., Lamersdorf, W. (eds.) Service-Oriented Computing – ICSOC 2006, Lecture Notes in Computer Science, vol. 4294, pp. 191–202. Springer Berlin / Heidelberg (2006)
14. Dey, A.K.: Understanding and using context. Personal Ubiquitous Comput. 5, 4–7 (January 2001)
15. Koning, M., Sun, C.a., Sinnema, M., Avgeriou, P.: VxBPEL: Supporting variability for web services in BPEL. Inf. Softw. Technol. 51, 258–269 (February 2009)
16. Lin, K.J., Zhang, J., Zhai, Y., Xu, B.: The design and implementation of service process reconfiguration with end-to-end QoS constraints in SOA. Serv. Oriented Comput. Appl. 4(3), 157–168 (Sep 2010)
17. Menasce, D., Gomaa, H., Malek, S., Sousa, J.: SASSY: A framework for self-architecting service-oriented systems. IEEE Software 28, 78–85 (2011)
18. Michlmayr, A., Rosenberg, F., Leitner, P., Dustdar, S.: End-to-end support for QoS-aware service selection, binding, and mediation in VRESCo. IEEE Trans. Serv. Comput. 3, 193–205 (July 2010)
19. Morin, B., Barais, O., Jezequel, J.M., Fleurey, F., Solberg, A.: Models@ run.time to support dynamic adaptation. Computer 42, 44–51 (October 2009)
20. Morin, B., Fleurey, F., Bencomo, N., Jézéquel, J.M., Solberg, A., Dehlen, V., Blair, G.: 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. pp. 782–796. MoDELS '08, Springer-Verlag, Berlin, Heidelberg (2008)
21. Pohl, K., Böckle, G., Linden, F.J.v.d.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
22. Rinderle, S., Reichert, M., Dadam, P.: Correctness criteria for dynamic changes in workflow systems: a survey. Data Knowl. Eng. 50(1), 9–34 (Jul 2004)
23. Song, W., Ma, X., Dou, W., Lü, J.: Toward a model-based approach to dynamic adaptation of composite services. In: Proceedings of the 2008 IEEE International Conference on Web Services. pp. 561–568. ICWS '08, IEEE Computer Society, Washington, DC, USA (2008)
24. Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features - enhancing flexibility in process-aware information systems. Data Knowl. Eng. 66, 438–466 (September 2008)