Preventing SLA Violations in Service Compositions Using Aspect-Based Fragment Substitution

Philipp Leitner[1], Branimir Wetzstein[2], Dimka Karastoyanova[2],
Waldemar Hummer[1], Schahram Dustdar[1], and Frank Leymann[2]

[1] Distributed Systems Group, Vienna University of Technology
Argentinierstrasse 8, 1040 Wien, Austria
lastname@infosys.tuwien.ac.at
[2] Institute of Architecture of Application Systems, University of Stuttgart, Germany
Universitätsstraße 38, 70569 Stuttgart, Germany
lastname@iaas.uni-stuttgart.de

Abstract. In this paper we show how the application of the aspect-oriented programming paradigm to runtime adaptation of service compositions can be used to prevent SLA violations. Adaptations are triggered by predicted violations, and are implemented as substitutions of fragments in the service composition. Fragments are full-fledged standalone compositions, and are linked into the original composition via special activities, which we refer to as virtual activities. Before substitution we evaluate fragments with respect to their expected impact on the performance of the composition, and choose those fragments which are best suited to prevent a predicted violation. We show how our approach can be implemented using Windows Workflow Foundation technology, and discuss our work based on an illustrative case study.

1 Introduction

As more and more companies shift towards a service-based model [1] of doing business, e.g., by providing coarse-grained value-added services as compositions of existing (external) Web services, management of service level agreements (SLAs) [2] is becoming increasingly important. SLAs are contractual agreements between a service provider and its customers, which govern the quality that the customers can expect. Violating SLAs is often associated with monetary penalties for the provider, i.e., the service provider generally has a strong interest in preventing SLA violations.

To this end, research in the area of SLA monitoring and compliance management [2, 3, 4] has so far mostly focused on detecting and explaining SLA violations after they have happened. While this is very useful to optimize service compositions in the long run, it does not prevent the problem in the first place. Therefore, we see the need for mechanisms to prevent violations at runtime, before they have happened. Basically, such mechanisms need both, a way

P.P. Maglio et al. (Eds.): ICSOC 2010, LNCS 6470, pp. 365–380, 2010.
© Springer-Verlag Berlin Heidelberg 2010

to predict violations ahead of time, and a means to actually adapt the problematic composition instance in such a way that the violation is prevented. The former has already been covered in earlier work [5,6].

The main contribution of this paper is a proposed solution for the latter problem. We apply the aspect-oriented programming (AOP) approach [7] to adaptation of running composition instances. Adaptations are triggered by predicted violations. Unlike in earlier work [8], aspects can contain composition fragments of arbitrary complexity, which can be applied before, after or instead of any subset of the original composition. We evaluate potential fragments based on their expected impact on SLA conformance, in order to identify the fragments which are best suited to prevent a predicted violation. Note that in this work we focus on performance-related service level objectives (SLOs). Our work is not directly applicable for most qualitative SLOs, such as security.

The rest of this paper is structured as follows. In Section 2 we present an example case, which we use as an illustrative example in the remainder of the paper. In Section 3 we present our approach to aspect-based adaptation in detail. In Section 4 we explain how our approach can be implemented using Windows Workflow Foundation [9] (Windows WF) technology. Section 5 contains an evaluation of our approach. In Section 6 we discuss related scientific work. Finally, Section 7 concludes the paper with a summary and an outlook on future work.

2 Case Study and Motivation

To illustrate the core ideas of the paper we use an order processing scenario. The scenario consists of a reseller who offers products to its customers. As shown in Figure 1 (left part of the figure), the reseller composes services from other providers (supplier, shipper, banking) to implement its process. After receiving the customer order, the list of needed parts is determined and parts which are not in stock are ordered from a supplier. After all the parts are gathered, the product is assembled and shipped to the customer. The reseller guarantees its customers a certain order processing time via an SLA. The goal of the reseller is to prevent cases of SLA violations, as this would lead to reduced customer satisfaction as well as penalty payments. The reseller can use SLA monitoring and prediction techniques as discussed in our previous work [5] to predict at process runtime whether the SLA with the customer is likely be violated, i.e., in our case, whether the order processing time will exceed the agreed value. If SLA prediction shows that the SLA with the customer will be violated, the reseller wants to adapt the service composition instance by using alternative, better performing services. Assume that in our scenario there are two alternative suppliers who offer faster delivery times, but do not provide all needed product types on their own. The full product range offered to the customer can only be realized by using both alternative suppliers in conjunction. Figure 1 shows a composition fragment consisting of a switch between those two suppliers, whereby supplier 2 is used if supplier 1 is unable to provide a certain part. Even though not the default case, this composition fragment can be used at runtime instead of the original supplier

invocation if a given instance is likely to exceed the maximum processing time as promised to the customer in the SLA.

There are two approaches for supporting the execution of alternative composition fragments: (1) the straight-forward approach is to model all alternative fragments already at design time as part of the composition, e.g., using if-else branches; (2) the approach of this paper is to model alternative fragments separately from the original composition model, and dynamically substitute them based on prediction results at composition runtime. We will now explain this approach and its advantages in detail.

3 Aspect-Based Adaptation

In this paper we use the notion of aspect-based fragment substitution to model how service composition instances can be adapted at runtime in order to prevent predicted SLA violations. Our approach is general, in the sense that it is not specific to any concrete composition model. Instead, it can be applied to many existing block-structured composition models (for instance WS-BPEL or Windows Workflow Foundation [9]). In our work we reuse well-known AOP terminology to describe adaptations of service compositions. The most important of these terms are *aspects* (cross-cutting concerns, which are turned off and on at design or run-time, e.g., logging), *advices* (business logic which implements aspects, e.g., the code to implement logging), *joinpoints* (points in the application code where advices can potentially be inserted), and *weaving* (the process of dynamically inserting advices in jointpoints). Note that in literature AOP is often discussed as both a design time and a run time technology, i.e., weaving can

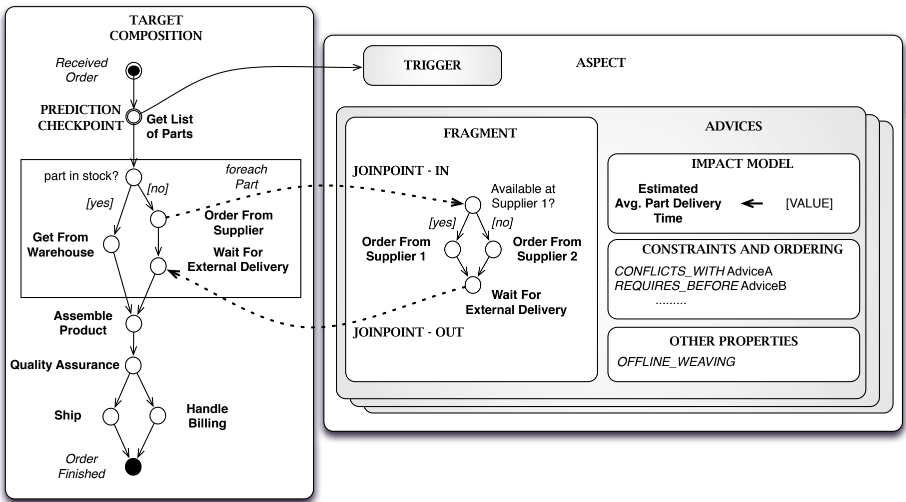


Fig. 1. Illustrative Example and Approach Overview

happen both statically or dynamically. In this paper we only consider weaving at runtime (at running instances), since our primary concern is the adaptation of composition instances, without modification of the underlying definition.

The main concepts of our work are summarized in Figure 1. *Aspects* are defined as an *adaptation trigger*, which is based on a predicted SLA violation, and any number of *advices*. Every advice in turn contains exactly one *composition fragment*, one *impact model* containing any number of impact clauses, a list of *constraints* on other advices of the same aspect, and any optional *other properties*. The fragment is linked to the service composition to adapt (denoted as *target composition* in the following) using two types of *joinpoints* – in-joinpoints mark the beginning of the composition segment to replace, while out-joinpoints mark the end of the segment. We will now discuss these components in more detail.

3.1 Adaptation Triggers

As discussed extensively in Section 1 and Section 2, our approach is motivated by the need to prevent SLA violations. Therefore, runtime adaptation is generally triggered by predictions of such violations. In the remainder of this paper we will assume that some means of prediction are available. This may be powerful prediction tooling as presented in [5] or [6], or simply some estimations provided by a human domain expert. The actual approach to prediction is out of scope of this paper, however, for the sake of completeness we give a very brief overview of our own earlier work.

We generate predictions using regression from runtime data. This data is collected using an event-based approach (i.e., components in the service-based system emit status events, which are collected and correlated). The actual regression is implemented as a black-box function, using methods from the field of machine learning, more precisely artificial neural networks [10] (ANNs). We sketch this in Figure 2. This prediction is carried out in predefined places in the service composition, the so-called checkpoints. Checkpoints should be selected in such a way that enough data is already available to generate useful predictions. Note that there is a strong relationship between the checkpoint selection and which actions can be associated with an advice – in earlier checkpoints a lot of adaptation actions are still available, while later checkpoints allow for more accurate predictions because of more data being available. The problem of selecting checkpoints is discussed in more detail elsewhere [5].

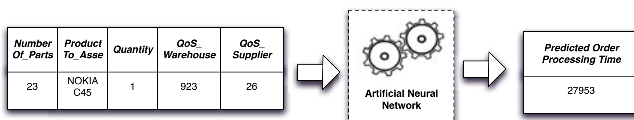


Fig. 2. Generating Predictions From Runtime Data

3.2 Composition Fragments

Composition fragments can be considered the core of our adaptation approach. In essence, fragments are full-fledged, even if usually small, service compositions. That is, fragments may contain variables, branches, Web service invocations, parallel execution, loops, scopes, fault handling, compensation, or any other construct which is legal in the composition model used. However, they do not have to follow the same syntactic and semantic rules as the target composition. For example, if WS-BPEL is used as composition model, designers of fragments may access e.g., variables defined in the target composition, even if the respective data is undefined in the fragment itself (syntactic rule). Also, they could specify a receive activity without a corresponding reply activity (semantic rule). The reason for this is that during weaving the fragment will be inserted into the composition model of the target composition, essentially becoming part of the composition itself. A fragment definition is valid if it results in an executable composition after weaving, which cannot be checked in isolation.

In addition to all activities provided by the composition metamodel, fragments may contain three additional activity types (`FRAGMENT_START`, in the following referred to as `start`, `FRAGMENT_END`, `end`, and `TRANSPARENT_BLOCK`, `transparent`) with a semantic specific to our approach. We refer to these activities as *virtual activities*, because they are never actually executed. Instead, virtual activities are dropped or replaced during weaving. Virtual activities are solely responsible for defining the joinpoints between the fragment and the target composition, marking the segment of the target composition to substitute.

Every fragment starts with exactly one `start` activity and ends with exactly one `end` activity. In-joinpoints, defined via the `start` activity, represent the start of substitution, and out-joinpoints, defined via the `end` activity, represent the end of substitution. All joinpoints can reference any activity in the service composition, either before or after the execution of the activity (i.e., both "immediately before executing Get List of Parts" and "immediately after executing Get List of Parts" are valid joinpoints). However, the in- and out-joinpoint of a fragment need to reference activities in the same sequence in the target composition, i.e., the joinpoints defined in Figure 1 are correct, but, for example, it would not be possible to move the in-joint point to the activity "Get List of Parts". The reason for this limitation is that semantic problems arise if in- and out-joinpoints are situated in different sequences. In the example, the branching activity "part in stock?" would be removed, but not the actual branches, rendering it impossible to decide which branch to execute.

It is not only possible to replace a segment of the target composition, even though this is the general case we discuss. Trivially, one may also just insert the fragment at a specific joinpoint (the in- and out-joinpoints are identical, and the fragment is non-empty), or remove a segment (substitution with an empty fragment). We refer to the sum of all joinpoints of a fragment as the *linking* of the fragment to the target composition. Figure 3 summarizes this linking. The `start` activity specifies that the fragment should be inserted before the activity "Schedule Assembling", while the `end` specifies that the end of the substitution

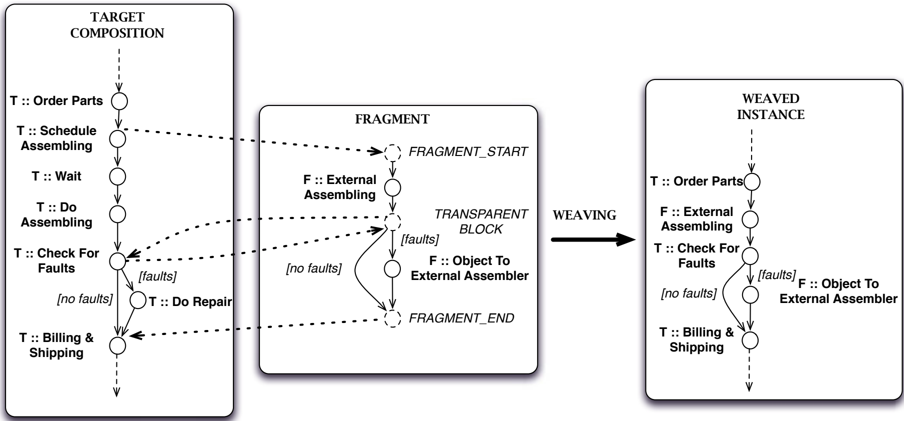


Fig. 3. Fragment Activities and Linking

segment is before the activity "Billing & Shipping". On the right-hand side, Figure 3 shows the dynamically constructed instance after the fragment has been weaved into the target composition. Activities depicted with the prefix `T` originate from the target composition, while activities with the prefix `F` are specified in the fragment.

`Transparents` are more complicated than `start` or `end` . They are a placeholder representing a part of the target composition in the fragment. This part is defined in the same way as the substitution segment, i.e., `transparents` have both out- and in-joinpoints. Additionally, the same restrictions apply (in and out-joinpoints need to reference activities in the same sequence). At runtime, `transparents` are replaced by a copy of the part that they represent. The purpose of `transparent` activities is threefold. Firstly, they allow for the definition of fragments which substitute segments, while still retaining some of this segment's functionality. One example of this usage is depicted in Figure 3, where the "Check for Faults" activity from the target composition is retained in the fragment. Note that it is not mandatory that a `transparent` references only a single activity. Secondly, transparent activities allow to essentially duplicate activities in the target composition. This is because `transparents` are in fact free to reference any part of the target composition, not only parts which are in the substitution segment (and hence removed during weaving). Additionally, many `transparents` may copy the same activities, multiplying them even further. Thirdly, `transparent` activities allow for the definition of generic fragments.

3.3 Generic Fragments

Generic fragments are (unlike the fragments discussed so far) not developed specifically for a given target composition. Instead, they can be applied to a number of compositions. Therefore, generic fragments do not contain any concrete

case-specific business logics. They are used to implement adaptation scenarios which can be useful across several concrete target compositions and domains. Figure 4 exemplifies three generic fragments. The main property of generic fragments is that they consist only of virtual activities and control flow constructs, i.e., they do not contain any concrete activities such as Web service invocations. These generic fragments are instantiated by defining the linking (i.e., all in- and out-joinpoints) to concrete target compositions. As soon as this linking is defined, the fragment stops being generic, and is as case-specific as any other fragment.

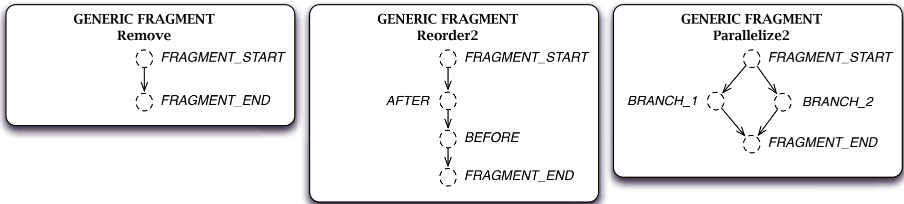


Fig. 4. Examples of Generic Fragments

The first and most simple generic fragment in Figure 4 (`Remove`) has been mentioned before – it is an empty fragment consisting only of a `start` and `end` activity. Using this generic fragment any segment of the target composition can be deleted. The second example is a generic fragment named `Reorder2`. It consists of `start`, `end`, and two `transparents` ("after" and "before"). Using this fragment two segments in the target composition can be rearranged, e.g., exchanging their order. Trivially, one can also implement similar generic fragments `ReorderX`, rearranging X segments instead of just two. Finally, `Parallelize2` consists again of `start`, `end`, and two `transparents` ("branch_1" and "branch_2"), however, this time "branch_1" and "branch_2" are executed in parallel. Using this generic fragment one can parallelize two segments from the target composition (which presumably have been executed in serial before). Of course, it is again possible to define `ParallelizeX` fragments to parallelize more than two segments at the same time.

3.4 Dynamic Weaving

At run-time, one or more previously selected fragments are weaved into the running instance of the target composition. The selection procedure will be discussed in Section 3.5. As we have sketched in Figure 5, the general weaving algorithm is a simple 2-step procedure. Firstly, the fragment is pre-processed, i.e., for each `transparent` in the fragment the linking to the target composition is resolved, and the `transparent` is replaced by a deep copy of the segment that it represents. Secondly, the linking of the fragment itself is resolved, and the

```
1  # input: instance, fragment, mode # output: weaved instance
2
3  if(mode == "OFFLINE") suspend(instance)
4
5  # step 1 - fragment preprocessing
6  foreach transparent in fragment
7       linking = resolve_linking(transparent,instance)
8       copy = copy_segment(linking,instance)
9       replace_fragments(fragment,transparent,copy)
10
11 # step 2 - fragment substitution
12 seqment := resolve_linking(fragment,instance)
13 remove_start_activity(fragment)
14 remove_end_activity(fragment)
15 replace_fragments(instance,segment,fragment)
16
17 if(mode == "OFFLINE") resume(instance)
18
19 return instance
```

Fig. 5. Weaving Algorithm

`start` and `end` virtual activities are removed from the fragment (they are not needed anymore). Finally, the segment of the target composition (indicated by the linking) is removed, and the fragment is inserted instead.

Weaving can be done either online or offline. For offline weaving the composition instance is halted while the adaptation is applied (see Line 4-5 in Listing 5), and resumed when the adaptation is finished (Lines 19-20). If online adaptation is used the instance continues running during weaving. This has the advantage that weaving does not introduce additional execution time overhead. However, if after weaving the running instance has already passed the entry point of the fragment (the linking of the fragment's `start` activity) the weaving fails and is rolled back. This is because our system needs to guarantee that a fragment is either executed as a whole, or not at all (which cannot be guaranteed after the instance has begun executing the substitution segment in the target composition). Our system falls back to offline adaptation as soon as at least one advice which needs to be applied requires it (i.e., if many advices are applied and only one of them requires offline weaving, we still need to suspend the composition instance before adaptation).

Generally, if more than one advice needs to be applied, we use recursive one-by-one weaving, that is, we start by weaving the first fragment into the instance (ignoring any other fragments). The result of this first weaving process is then the input to the weaving of the second fragment. This is continued until all fragments are weaved. The order in which fragments are applied is unimportant as long as all fragments are independent (i.e., as long as none of the segments indicated by any linking of either fragments or transparents overlaps). If this is not the case the user can specify a defined ordering of advices as part of the advice definition. The ordering can be defined using five different order predicates (`REQUIRES_BEFORE`, `REQUIRES_AFTER`, `IF_PRESENT_BEFORE`,

`IF_PRESENT_AFTER`, and `CONFLICTS_WITH`). `REQUIRES_[BEFORE|AFTER]` specifies that a given advice has to be applied before or after this advice (otherwise the advice cannot be applied at all). `IF_PRESENT_[BEFORE|AFTER]` specifies that if the other advice is present, it has to be applied before or after this one (but the other advice can also simply not be applied). Using `REQUIRES_BEFORE` one can specify complex fragments, whose linking does not actually point to the target composition itself, but to another fragment. This is possible since we can rely that the referenced fragment has already been weaved into the target composition before before the weaving of the dependent fragment starts. Another type of ordering predicate is `CONFLICTS_WITH`. This predicate specifies that two fragments are mutually exclusive, i.e., they cannot be applied together. At runtime, we construct a forest of directed graphs from these predicates, whose nodes are advices and whose edges are "is executed after" relationships. If the graphs in this forest are acyclic there is at least one allowed order of advices, which can be consructed using topological ordering. If the graphs contains at least one cycle the definition of advices is invalid, since the definition contains at least one cyclic dependency.

3.5 Impact Model and Advice Selection

As described briefly in Section 3.1, we build upon a predictor which estimates SLO values by assessing a set of lower level metrics. Examples include ordered product types, duration of branches of the composition, expected delivery times of suppliers and shipper, or QoS of services used. In order to being able to evaluate whether a given advice will actually help preventing the predicted SLA violation, we need to specify for each advice its impact on those lower-level base metrics (*impact model*). The impact model is used to identify which concrete advice (from all advices designed within an aspect) should be applied, i.e., which advices are best suited to prevent a predicted violation (*advice selection*).

The impact model contains a non-empty set of *impact clauses*. An impact clause relates to one base metric and specifies the expected value of that metric after adaptation (i.e., after this fragment has been applied). This value can be determined in several ways: (1) based on measured history data if the corresponding advice has already been used in past composition instances, e.g., using data mining techniques; (2) based on SLAs with external providers; or (3) by using QoS aggregation techniques as discussed in earlier research [11]. In QoS aggregation, based on the composition fragment structure, the properties of atomic activities are recursively aggregated (e.g., the duration of a sequence of activities is the sum of durations of those activities, the duration of the parallel execution of activities is given by the duration of the longest acitivity etc. [11]). The impact model should specify impact clauses for all metrics which the advice affects.

The impact model is specified as part of advice definition. Advice selection at runtime is performed as follows. If in a checkpoint a violation is predicted, we obtain the set of advices defined for this checkpoint. For each allowed combination of advices we evaluate if the usage of these advices would prevent the

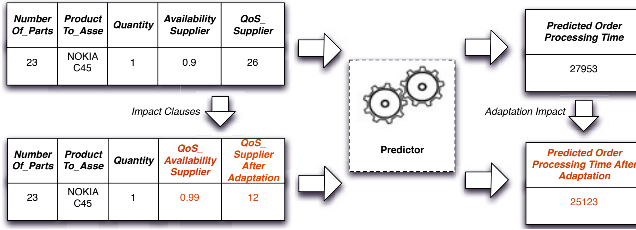


Fig. 6. Evaluation of Impact Models

SLA violation, i.e., all impact clauses are applied to the data which has originally been used to generate the prediction. The updated data (which essentially represents the state after adaptation) is then again fed to the predictor, to regenerate the prediction after adaptation. The difference between the original prediction and the new prediction is the estimated impact of applying these advices. This is sketched in Figure 6. If more than one advice should be applied at the same time, the impact clauses are applied in the same order to the data as the weaving order of the fragments would be. If the predicted value complies to the SLA, that advice or advice combination is put into a candidate set. In the next step, we then select the best alternative from the candidate set by looking at additionally specified criteria (in addition to the concrete predicted value). In this step, complex evaluations can take place, taking into account and weighting different dimensions (e.g., cost, customer satisfaction, reliability) according to a user-defined utility function, which is currently left for future work. At the moment, we simply choose the candidate which brings the SLO value closest to the target value, i.e., we apply "just as much" adaptation as necessary to prevent the predicted violation.

4 Prototype Implementation

In our prototype we consider the aspect-based adaptation of service compositions implemented using the Windows Workflow Foundation [9] (WF) composition model. More concretely, our system can be applied to WF Sequential Workflows[1]. WF Sequential Workflows are similar to e.g., Web service compositions implemented using WS-BPEL. However, unlike most WS-BPEL engines, WF is deeply integrated with Microsoft .NET (starting with version 3.0), along with strong tool support for developing compositions. Additionally, and most importantly for this paper, Microsoft .NET supports the dynamic adaptation of WF instances via an explicit API, the WorkflowChanges API[2]. This API allows us to suspend, modify, and resume any running composition instance. Additionally, activities in the composition can easily be replicated. Implementation of

[1] http://msmvps.com/blogs/theproblemsolver/archive/2006/10/11/Sequential-versus-State-workflows.aspx

[2] http://msdn.microsoft.com/en-us/library/ms734569(VS.85).aspx

the weaving algorithm as discussed in Section 3 is, therefore, straight-forward. Another important advantage of building the prototype based on WF is that we can reuse the tooling integrated in Visual Studio to support the development of fragments.

We have implemented the approach discussed in this paper within the larger VRESCo SOA runtime environment project. VRESCo is discussed in detail elsewhere [12], and will not be covered here. To trigger adaptations as briefly discussed in Section 3.1, we utilize our earlier work on prediction of violations, as discussed in [5]. The prototype has been designed to fit into PREvent , an autonomous system for prevention of SLA violations [13]. The interested reader may download a recent snapshot of our VRESCo prototype, which includes an implementation of the case study used in this paper[3].

5 Evaluation

We will now evaluate our approach in two different ways. Firstly, we will qualitatively analyse the expressiveness of our approach by comparison with previously published adaptation patterns [14]. Secondly, we will have a look at performance implications. This is done by monitoring the weaving time in our prototype system, as well as comparing the execution time of dynamically weaved and statically defined composition instances.

5.1 Coverage of Adaptation Patterns

In order to discuss the expressiveness of our approach we have used the adaptation patterns defined in [14]. In this work, 14 patterns of structural changes in processes are identified. Using our approach 9 of these patterns are fully supported.

We have summarized the coverage of adaptation patterns in Table 1. The patterns **AP1**, **AP2** and **AP4** are the core feature of our approach, and can be implemented trivially. For **AP2** and **AP5** we specifically described a generic fragment in Section 3. Similarly, all of **AP3**, **AP5**, **AP8**, **AP9**, **AP10** **AP14** can be implemented rather elegantly using `transparents`. The patterns concerning subprocesses (**AP6** and **AP7**) cannot be implemented since our approach does not support linking to more than one composition at the same time. **AP11** and **AP12** are in simple cases implementable using `transparents`, but our approach does not provide any explicit support for it, making the implementation rather cumbersome. Similarly, **AP13** can be implemented by replacing the branching node as a whole, but we do not consider this solution as in line with the idea of this pattern.

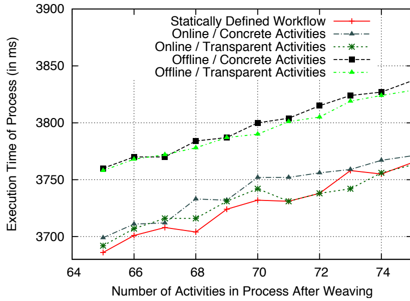
5.2 Performance Analysis

In a second step we have evaluated the runtime implications of our prototype. For this, we monitored the average execution time of dynamically weaved composition instances with an increasing number of activities, and compare them to

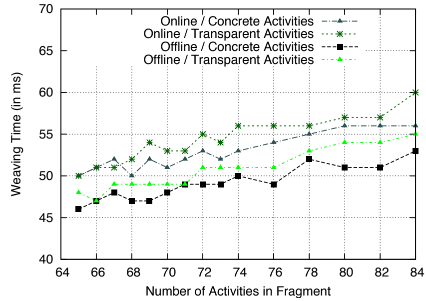
[3] `http://sourceforge.net/projects/vresco/`

Table 1. Coverage of Adaptation Patterns

ID	Pattern Name	Covered
AP1	Insert Fragment	✓
AP2	Delete Fragment	✓
AP3	Move Fragment	✓
AP4	Replace Fragment	✓
AP5	Swap Fragment	✓
AP6	Extract Sub Process	✗
AP7	Inline Sub Process	✗
AP8	Embed Fragment in Loop	✓
AP9	Parallelize Activities	✓
AP10	Embed Fragment in Conditional	✓
AP11	Add Control Dependency	✗
AP12	Remove Control Dependency	✗
AP13	Update Condition	✗
AP14	Copy Fragment	✓



(a) Execution Time



(b) Runtime Weaving

Fig. 7. Performance Analysis Outcomes

the same instance defined statically. We also compare online and offline weaving, and distinguish between fragments defined using `transparent` activities and fragments defined without. For simplicity, all compositions and fragments are sequences of "wait" activities. Using different types of activities would not have an impact on the evaluation outcome, since our adaptation approach handles all non-virtual activities the same way, i.e., weaving an "invoke" activity has a similar overhead than weaving a "wait" activity. To mimimize external influences all results are the average of 50 independent test runs. We have also repeated the evaluation multiple times to make sure that the outcome is reproduceable. The outcomes of these experiments are summarized in Figure 7(a).

As can be seen, online weaved compositions exhibit very little overhead as compared to statically defined compositions. Of course, offline weaving introduces some overhead, which stems from the time necessary to select the fragments, to implement the actual weaving, and to suspend and unsuspend the composition. In our experiments, the largest part of these factors is the actual

weaving time. Therefore, we have further analyzed this factor in Figure 7(b). We depict the weaving time depending on the number of activities to weave. Generally, concrete activities are faster to weave than `transparent` activities (since the logics of weaving `transparents` is more complicated), and offline weaving is faster than online weaving (since, in the online case, some additional sanity checks are done by the Windows Workflow runtime). In general, this increased weaving time for online weaving does not matter too much, since the online weaving time does not directly impact the execution time of the process. Overall, the overhead introduced by weaving is relatively constant in [45 : 80] ms, even for large fragments (more than 80 activities).

Summarizing, we can see that dynamic weaving does not introduce a big overhead, especially if online weaving is possible. If offline weaving has to be used, an additional weaving overhead, which is generally in [45 : 80] ms, is introduced. We argue that for most application areas this overhead is still far from being dramatic. Even though the concrete numbers are specific for our prototype implementation, they still show that implementing our ideas efficiently is well possible.

6 Related Work

In this paper we apply the AOP paradigm to adaptation of service compositions. On the level of atomic services earlier work in this direction has been presented by Kongdenfha et al [15]. In this work, they use the AOP paradigm to adapt the implementation of atomic Web services. A comparable approach has also been presented by Song et al. [16], who use the AOP approach to weave cross-cutting concerns, such as security, into all atomic Web services in a composition. A similar track has also been followed by Narendra et al., who used AOP-based adaptation of services in a composition to propagate changes in non-funtional properties through the composition [17]. Of course, all of these approaches assume that the developer has access to the implementation of these atomic services.

The general scope of our work is similar to work presented by Gmach et al. [18]. However, the focus of our contribution is purely on adaptation of service compositions, while Gmach et al. adapt on service infrastructure level (i.e., by moving services to different hosts, or by re-scheduling requests in the service bus). Finally, adaptation with the explicit goal of preventing SLA violations has been discussed by various authors, e.g., our own earlier work on PREvent [13] or recent work by Metzger et al. [19]. The concrete execution of adaptation of compositions has in the past been covered by research in various directions. Earlier approaches often did not consider the adaptation of the composition structure at all, instead focusing solely on service rebinding. In a simplistic manner such adaptations can in fact be carried out using WS-BPEL alone, by using the *Dynamic Partner Link* feature. However, practical problems such as finding the right service to bind to (often based on QoS), or the need to resolve interface differences, demand for more sophisticated service rebinding approaches. Examples of such work include the WS-Binder [20] or the PAWS [21] frameworks.

More advanced service rebinding was also one of the contributions of Moser et al. in [22]. Finally, some work on service rebinding (dealing also with stateful services) has been presented by Mosincat and Binder in [23].

An early approach towards structural adaptation of compositions has been discussed in [24]. However, in this work no free-form adaptation is possible. Instead, predefined parameterizations are applied if certain conditions hold. Arguably, the AOP paradigm can provide a more powerful abstraction for adaptation in compositions. This idea has first been introduced by Charfi et al. [25,26]. However, unlike our work, Charfi et al. focus on the traditional AOP idea of weaving crosscutting concerns into the composition, while we apply the AOP paradigm with a different goal (adaptation for SLA compliance) in mind. Using aspects for runtime adaptation in WS-BPEL has been covered by the BPEL'n'-Aspects framework [8]. Our main contribution over this work is that in our case aspects can be composition fragments, while BPEL'n'Aspects supports only single Web service invocations as aspects. Work with similar goals, but specific to the telecommunications domain, has been presented Niemöller et al. [27]. An approach which deals with process fragment composition is presented by Eberle et al. [28]. Their idea is to exploit the redundancy in separately modeled composition fragments and use those redundant overlapping fragment parts to merge the fragments. In our approach we model how fragments should be merged explicitly by using virtual activities.

7 Conclusions and Future Work

In this paper we have presented an approach to runtime adaptation of service compositions for preventing SLA violations. The adaptation is based on composition fragments which are dynamically substitued at runtime using AOP techniques. Composition fragments are modeled separately and are explicitly linked into the original composition using virtual activities. In addition to their process logic, fragments specify also their expected impact on the composition performance. This is necessary in order to be able to choose the best fitting fragments for preventing a predicted SLA violation at composition runtime. We have implemented the approach using Windows Workflow Foundation technology and experiments show that the performance impact of dynamic weaving is acceptable.

While the current status of the approach is promising, there are still some open issues left for future work. Firstly, we do not take into account that adaptation which prevents the violation of one SLA metric could easily lead to the violation of another. In particular, we currently do not take the costs of adaptations into account (e.g., increased costs by using more expensive services in the weaved fragment) which in some cases could be higher than the gain of not violating the SLA. Therefore, we will extend the impact model and its evaluation in our future work. Secondly, we currently assume that the number of possible combinations of advices to apply is small, so that finding the best combination via full enumeration is possible. In future work we plan to embrace heuristic optimization for cases where full enumeration is not feasible.

Acknowledgments

The research leading to these results has received funding from the European Community's 7th Framework Programme under the Network of Excellence S-Cube (Grant Agreement no. 215483).

References

1. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. IEEE Computer 40(11) (2007)
2. Dan, A., Davis, D., Kearney, R., Keller, A., King, R., Kuebler, D., Ludwig, H., Polan, M., Spreitzer, M., Youssef, A.: Web Services on Demand: WSLA-Driven Automated Management. IBM Systems Journal 43(1), 136–158 (2004)
3. Bodenstaff, L., Wombacher, A., Reichert, M., Jaeger, M.C.: Analyzing Impact Factors on Composite Services. In: Proceedings of the 2009 IEEE International Conference on Services Computing (SCC 2009) (2009)
4. Wetzstein, B., Leitner, P., Rosenberg, F., Brandic, I., Leymann, F., Dustdar, S.: Monitoring and Analyzing Influential Factors of Business Process Performance. In: Proceedings of the 13th IEEE EDOC Conference (EDOC 2009) (2009)
5. Leitner, P., Wetzstein, B., Rosenberg, F., Michlmayr, A., Dustdar, S., Leymann, F.: Runtime Prediction of Service Level Agreement Violations for Composite Services. In: Proceedings of the 3rd Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing, NFPSLAM-SOC 2009 (2009)
6. Zeng, L., Lingenfelder, C., Lei, H., Chang, H.: Event-Driven Quality of Service Prediction. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 147–161. Springer, Heidelberg (2008)
7. Miller, F.P., Vandome, A.F., McBrewster, J.: Aspect-oriented Programming. Alphascript Publishing (2010)
8. Karastoyanova, D., Leymann, F.: BPEL'n'Aspects: Adapting Service Orchestration Logic. In: Proceedings of 7th IEEE International Conference on Web Services, ICWS 2009 (2009)
9. Shukla, D., Schmidt, B.: Essential Windows Workflow Foundation. Microsoft.Net Development Series (2006)
10. Haykin, S.: Neural Networks and Learning Machines: A Comprehensive Foundation, 3rd edn. Prentice-Hall, Englewood Cliffs (2008)
11. Jaeger, M.C., Rojec-Goldmann, G., Muhl, G.: QoS Aggregation in Web Service Compositions. In: Proceedings of the 2005 IEEE International Conference on eTechnology, eCommerce and eService, EEE 2005 (2005)
12. Michlmayr, A., Rosenberg, F., Leitner, P., Dustdar, S.: End-to-End Support for QoS-Aware Service Selection, Binding and Mediation in VRESCo. IEEE Transactions on Services Computing, TSC (2010)
13. Leitner, P., Michlmayr, A., Rosenberg, F., Dustdar, S.: Monitoring, Prediction and Prevention of SLA Violations in Composite Services. In: Proceedings of the 2010 IEEE International Conference on Web Services, ICWS 2010 (2010)
14. Weber, B., Reichert, M., Rinderle-Ma, S.: Change Patterns and Change Support Features - Enhancing Flexibility in Process-Aware Information Systems. Data and Knowledge Engineering 66(3), 438–466 (2008)
15. Kongdenfha, W., Saint-Paul, R., Benatallah, B., Casati, F.: An Aspect-Oriented Framework for Service Adaptation. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 15–26. Springer, Heidelberg (2006)

16. Song, H., Yin, Y., Zheng, S.: Dynamic Aspects Weaving in Service Composition. In: Proceedings of the International Conference on Intelligent Systems Design and Applications (2006)
17. Narendra, N.C., Ponnalagu, K., Krishnamurthy, J., Ramkumar, R.: Run-time adaptation of non-functional properties of composite web services using aspect-oriented programming. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 546–557. Springer, Heidelberg (2007)
18. Gmach, D., Krompass, S., Scholz, A., Wimmer, M., Kemper, A.: Adaptive Quality of Service Management for Enterprise Services. ACM Transactions on the Web 2(1), 1–46 (2008)
19. Metzger, A., Sammodi, O., Pohl, K., Rzepka, M.: Towards Pro-Active Adaptation With Confidence: Augmenting Service Monitoring With Online Testing. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2010 (2010)
20. Penta, M.D., Esposito, R., Villani, M.L., Codato, R., Colombo, M., Nitto, E.D.: WS Binder: a Framework to Enable Dynamic Binding of Composite Web Services. In: Proceedings of the International Workshop on Service-Oriented Software Engineering, SOSE 2006 (2006)
21. Ardagna, D., Comuzzi, M., Mussi, E., Pernici, B., Plebani, P.: PAWS: A Framework for Executing Adaptive Web-Service Processes. IEEE Software 24(6), 39–46 (2007)
22. Moser, O., Rosenberg, F., Dustdar, S.: Non-Intrusive Monitoring and Service Adaptation for WS-BPEL. In: Proceedings of the 17th International Conference on World Wide Web, WWW 2008 (2008)
23. Mosincat, A., Binder, W.: Transparent Runtime Adaptability for BPEL Processes. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 241–255. Springer, Heidelberg (2008)
24. Karastoyanova, D., Leymann, F., Nitzsche, J., Wetzstein, B., Wutke, D.: Parameterized BPEL Processes: Concepts and Implementation. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 471–476. Springer, Heidelberg (2006)
25. Charfi, A., Mezini, M.: AO4BPEL: An Aspect-oriented Extension to BPEL. World Wide Web 10(3), 309–344 (2007)
26. Charfi, A., Dinkelaker, T., Mezini, M.: A Plug-in Architecture for Self-Adaptive Web Service Compositions. In: Proceedings of the 2009 IEEE International Conference on Web Services, ICWS 2009 (2009)
27. Niemöller, J., Levenshteyn, R., Freiter, E., Vandikas, K., Quinet, R., Fikouras, I.: Aspect Orientation for Composite Services in the Telecommunication Domain. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC-ServiceWave 2009. LNCS, vol. 5900, pp. 19–33. Springer, Heidelberg (2009)
28. Eberle, H., Unger, T., Leymann, F.: Process Fragments. In: Meersman, R., Dillon, T., Herrero, P. (eds.) OTM 2009. LNCS, vol. 5870, pp. 398–405. Springer, Heidelberg (2009)