Vienna University of Technology
Information Systems Institute
Distributed Systems Group

Cost-Based Optimization of Service Compositions

Under Review for Publication in IEEE Transactions on Services Computing (TSC)

Philipp Leitner, Waldemar Hummer and Schahram Dustdar
lastname@infosys.tuwien.ac.at

TUV-1841-2011-01 Feb. 15, 2011

For providers of composite services preventing cases of SLA violations is crucial. Previous work has established runtime adaptation of compositions as a promising tool to achieve SLA conformance. However, in order to get a realistic and complete view of the decision process of service providers, the costs of adaptation need to be taken into account. In this paper we formalize the problem of finding the optimal set of adaptations, which minimizes the total costs arising from SLA violations and the adaptations to prevent them. We present possible algorithms to solve this complex optimization problem, and detail an end-to-end system based on our earlier work on the PREvent (prediction and prevention based on event monitoring) framework, which clearly indicates the usefulness of our model. We discuss experimental results that show how the application of our approach leads to reduced costs for the service provider, and explain the circumstances in which different algorithms lead to more or less satisfactory results.

Keywords: Service-oriented Computing, Service Composition, Optimization, Adaptation

Argentinierstr. 8/184-1
A-1040 Vienna, Austria
phone: +43 1 58801-18402
fax: +43 1 58801-18491
http://www.infosys.tuwien.ac.at/

Cost-Based Optimization of Service Compositions

Philipp Leitner *Member, IEEE,* and Waldemar Hummer *Member, IEEE,* and Schahram Dustdar *Senior Member, IEEE*

Abstract—For providers of composite services preventing cases of SLA violations is crucial. Previous work has established runtime adaptation of compositions as a promising tool to achieve SLA conformance. However, in order to get a realistic and complete view of the decision process of service providers, the costs of adaptation need to be taken into account. In this paper we formalize the problem of finding the optimal set of adaptations, which minimizes the total costs arising from SLA violations and the adaptations to prevent them. We present possible algorithms to solve this complex optimization problem, and detail an end-to-end system based on our earlier work on the PREvent (prediction and prevention based on event monitoring) framework, which clearly indicates the usefulness of our model. We discuss experimental results that show how the application of our approach leads to reduced costs for the service provider, and explain the circumstances in which different algorithms lead to more or less satisfactory results.

Index Terms—Service Composition, Service Level Agreements, Adaptation, Optimization

1 Introduction

Service-based applications have seen tremendous research activity in the last years, with many important results being generated around the world [1]. This global interest is justified by the ever increasing services industry, which is still only starting to explore the potential that new paradigms like Everything-as-a-Service (XaaS) or Cloud Computing provide [2]. However, to fully realize this potential, research and industry alike need to focus more strongly on non-functional properties and quality issue of services (generally referred to as QoS). In the business world, QoS promises are typically defined within legally binding Service Level Agreements (SLAs) between clients and service providers, represented, e.g., using WSLA [3]. SLAs contain Service Level Objectives (SLOs), i.e., concrete numerical QoS objectives which the service needs to fulfill. If SLOs are violated, agreed upon monetary consequences go into effect. For this reason, providers generally have a strong interest in monitoring SLAs and preventing violations, either by using post mortem analysis and optimization [4], [5], or by runtime prediction of performance problems [6], [7]. We argue that the latter is more powerful, allowing to prevent violations before they have happened by timely application of runtime adaptation actions [8]–[10].

However, preventing SLA violations is, in general, not for free. For instance, some alternative services usable in a composition may provide faster response times (thereby improving the end-to-end runtime of the composite service, and reducing the probability of violating runtime related SLOs), but those services are often more expensive than slower ones. Therefore, there is an apparent tradeoff between preventing SLA violations and the inherent costs of doing so. We argue that this tradeoff is currently not covered sufficiently in the literature. Instead, researchers assume that the ultimate goal of service providers is to minimize SLA violations, completely ignoring the often significant costs of doing so (e.g., [9], [10]).

In this paper we contribute to the state of the art by formalizing this tradeoff as an optimization problem, with the goal of minimizing the total costs (of violations and applied adaptations) for the service provider. We argue that this formulation better captures the real goals of service providers. Additionally, we present possible algorithms to solve this optimization problem efficiently enough to be applied at composition runtime. We evaluate these algorithms within our PREVENT (prediction and prevention based on event monitoring) framework [8].

The remainder of this paper is structured as follows. In Section 2 we motivate our work and present an illustrative example, which will guide us through the rest of the paper. Following in Section 3, we present our earlier work on prevention of SLA violations. In Section 4 we formalize the problem of cost-based optimization of service compositions. We explain possible algorithms to solve this problem efficiently in Section 5, which are evaluated in Section 6. Finally, we compare our work with the most important related scientific approaches in Section 7, and conclude the paper in Section 8.

2 Motivation

In this paper we use the scenario depicted in Figure 1 (in BPMN [11] notation) to motivate and explain our approach.

• *Philipp Leitner, Waldemar Hummer and Schahram Dustdar are with the Distributed Systems Group, Vienna Univ. of Technology, Argentinierstrasse 8, 1040 Vienna, Austria. E-mail: {lastname}@infosys.tuwien.ac.at*

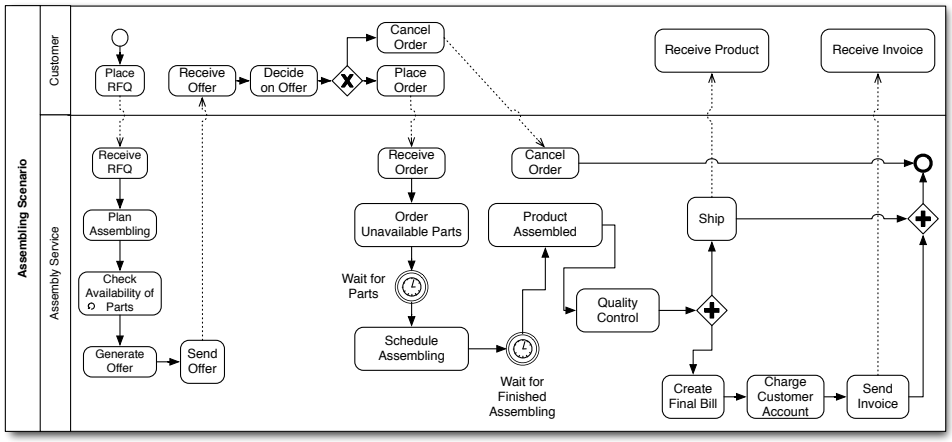

Fig. 1: Motivating Scenario

This scenario considers the case of a manufacturer of industry products. These products are constructed on-demand by assembling various parts, some of which can be produced in-house by the manufacturer, while others need to be ordered from external suppliers. The manufacturing process depicted in Figure 1 consists of two segments: firstly, the customer sends a request for quotation (RFQ), which the manufacturer responds to with an offer (consisting of estimated price and delivery time for the finished product), secondly, the customer can then order this product to the offered conditions. For reasons of brevity we concentrate on the two roles "Customer" and "Assembly Service" in the figure, even though the manufacturer interacts with many different external partners (e.g., suppliers of parts, shippers, credit card companies) to implement the described functionality. Since the manufacturer's business is based entirely on a service-based notion, the manufacturing process is implemented as a service composition, i.e., activities in the process are mapped to one or more invocations of (Web) services.

#	SLO Name	Description
1	**Time to Offer**	Time between receiving the RFQ and responding with an offer (in working days).
2	**Order Fulfillment Time**	Time between receiving the order and finishing the process (in working days).
3	**Process Lead Time**	Time between initializing the process and finishing it (excluding activities at customer side) in working days.
4	**Cost Compliance**	Cost overrun with regard to the offer in % of the offer.
5	**Product as Specified**	Product is exactly as specified.

TABLE 1: Service Level Objectives

With its key customers, the manufacturer has some established service level agreements. We provide a list of typical SLOs in Table 1. Note that these objectives can be of quantitative (SLOs #1 to #4) or of qualitative (SLO #5) nature.

#	Target Value	Costs of Violation
1	$<= 2$	Implicit costs - customer will choose a different manufacturer if offer is not received in time.
2	$<= 5$	Manufacturer grants 5% discount per 1 day delay, 20% max discount, not additive with SLO#3.
3	$<= 6$	Manufacturer grants 5% discount per 1 day delay, 20% max discount, not additive with SLO#2.
4	$<= 5$	Manufacturer cannot charge more than the offer plus 5%.
5	n/a	If wrong product is delivered, manufacturer needs to produce and ship the specified product within 7 working days and grant a 5% discount.

TABLE 2: Target Values and Penalties

All SLOs have some target values and penalties for violating these targets associated (see Table 2). Therefore, the manufacturer has a strong interest in complying to these SLOs, as long as the costs of doing so do not exceed the benefit. The manufacturer may apply a number of runtime adaptations to the process. We sketch some example adaptation actions in Table 3. The columns **+** and **-** refer to SLOs in Table 1, and indicate that the respective action has a positive (**+**) or negative (**-**) impact on this SLO. Note that these actions and impacts are just of examplatory nature, that is, while for some business cases outsourcing may reduce costs and increase the process duration (and error rate), this does not necessarily hold for all processes. Additionally, applying these actions generally also has some associated costs, which need to be taken into account (for instance, express shipping is more expensive than regular shipping). As we can see, for the manufacturer there is a tradeoff between the three dimensions duration, costs and quality, which is well-known in many fields of engineering.

Since the manufacturer business process is imple-

#	Adaptation Action	+	-
1	Use faster shipper or faster shipping option, e.g., express shipping.	#2, #3	#4
2	Order more parts instead of producing them in-house.	#2, #3	#4
3	Generate offer with higher priority.	#1, #3	-
4	Outsource assembling and quality control.	#4	#2, #3, #5
5	Skip quality assurance or do it less thoroughly.	#2, #3, #4	#5
6	Add an additional quality assurance step.	#5	#2, #3, #4

TABLE 3: Possible Adaptation Actions

mented as a service composition, applying these adaptations essentially boils down to adapting the service compositon. This can be done by either adapting the data flow of the composition (e.g., to use a different shipping option), by invoking different base services, or by changing the structure of the composition itself. In our previous work we have already shown how such adaptations can be applied at runtime [8], [9]. However, until now we have not discussed the question of how the service provider can select the actions which are economically most senseful to apply. Evidently, this question results in an optimization problem, minimizing the total costs of all SLA violations plus all costs arising from the adaptation. Furthermore, this problem needs to be solved very efficiently, since this optimization has to be repeated at runtime for every composition instance that is predicted to violate one or more SLOs.

3 Background

In order to provide some background information for this paper, we now present the PREVENT framework, which forms the basis for the research discussed here. Generally, PREVENT is a closed loop system [12] for self-optimizing service compositions. PREVENT is based on the existing SOA runtime environment VRESCO [13]. As we have sketched in Figure 2, the PREVENT framework consists of the seminal steps "monitor", "analyze", "plan" and "execute", as defined in the vision of autonomic computing [14]. We have previously presented our initial version of the PREVENT framework in [8].

Generally, the idea of PREVENT is to use event-based monitoring of composition data to generate runtime predictions of SLA violations before they have happened. Based on these predicted violations, adaptation actions are triggered with the goal of preventing the violation. In this paper we focus on the implementation of the *Cost-Based Optimizer* component in Figure 2, which we have not discussed so far in our earlier work. For every composition instance, this component receives estimations of concrete SLO values from the *Violation Predictor* component, and decides (based on these estimations as well as on knowledge of standing SLAs and available

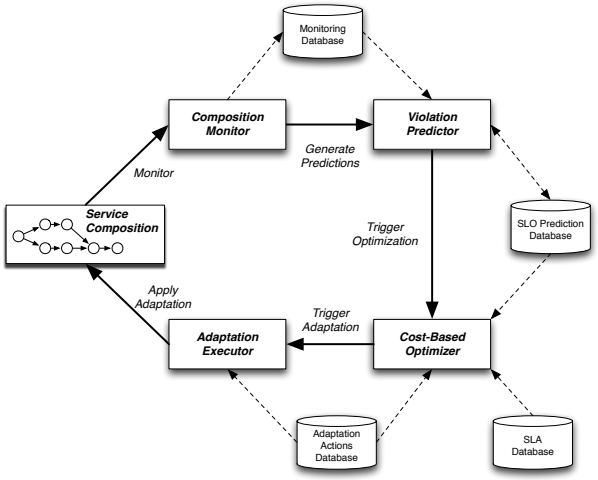

Fig. 2: Overall Framework

adaptation actions) which adaptations should be applied to a composition instance. In the following, we refer to this decision procedure as *cost-based optimization*. We use the term *optimization time* as the point in time during a composition instance's execution at which cost-based optimization happens. The interested reader may download our current version of the prototype[1].

3.1 Prediction of SLOs

Generally, the PREVENT approach to prediction of SLA violations is based on the idea of predicting concrete SLO values based on whatever monitoring information is already available at optimization time. We distinguish three different types of information. *Facts* represent data which can already be measured at optimization time. *Unknowns* are the opposites of facts. They represent data which is entirely unknown at optimization time. Evidently, unknown data cannot be used in the prediction. *Estimates* are a kind of middle ground between facts and unknowns, in that they represent data which is not yet available, but can in some way be estimated. This is often the case for QoS data, since techniques such as QoS monitoring [15] can be used to get an idea of e.g., the response time of a service before it is actually invoked. The *Violation Predictor* uses both facts and estimates from previously monitored historical service executions to train a machine learning function (we use multi-layer artificial neural networks [16] for quantitative SLOs and C4.5 decision trees [17] for qualitative SLOs), which can then be used to produce a numerical estimation of the SLO values at runtime. More details about our approach to prediction of SLOs can be found in our earlier work [6].

We have sketched this machine learning based implementation of the *SLO Predictor* in Figure 3. One model trained per SLO that needs to be predicted (even though the same model can be used if this SLO is used in

1. http://sourceforge.net/projects/vresco/

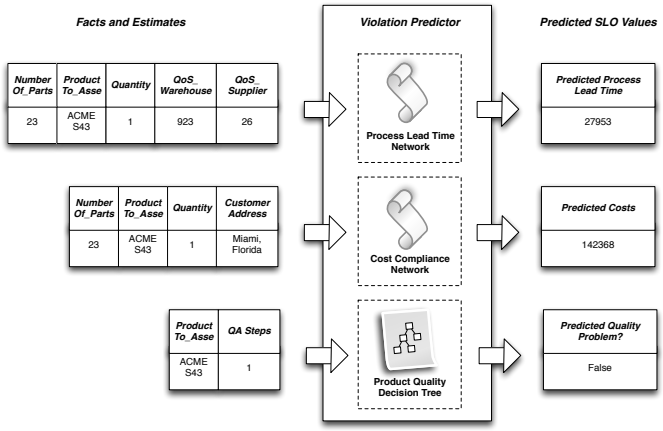

Fig. 3: Predicting SLOs Using Machine Learning

multiple customer SLAs), and every model is trained from different data. In order to identify which data should be used to train which model, some domain knowledge is necessary. However, dependency analysis [5] can be used to identify the factors which have the biggest influence on the respective SLOs. These factors of influence are also the most important data to use as basis for generating the prediction models. Apparently, some historical executions of the service composition are necessary to bootstrap the prediction. The concrete amount of instances that are necessary depend both on the expected quality of prediction (more historical information in tendency improves the prediction quality) and on the size and complexity of the service composition.

For understanding the remainder of the paper it is important to keep in mind that the machine learning models trained in the *SLO Predictor* essentially implement a set of estimator functions, which can be used for any partially known instance of the composition (i.e., an instance, whose facts and estimates are partially known, for instance a half-finished instance) to generate an estimation of the SLO value when the instance is finished. We will use these estimator functions in our modelling in Section 4.

3.2 Adaptation Actions

The PREVENT *Adaptation Executor* can execute a range of different adaptations of service composition instances. Generally, we distinguish three types of adaptations: data manipulation, service rebinding and structural adaptation. Data manipulation actions represent the most simple type of adaptation, where the composition is in fact not changed. Instead, the data flow of the composition instance is intercepted and some datum is changed (e.g., the *priority* parameter of the service invoked as part of the "ship" activity is changed to "high priority"). Service rebinding represents the common case where a different service is used to implement an activity in the composition, e.g., a faster shipping service is used in the activity "ship". For this type of adaptation we differentiate between three types, one-to-one service rebinding without interface mediation (the original and the new service have identical interfaces), one-to-one service rebinding with interface mediation (the services have different interfaces, but the same number of service invocations is needed to achieve the required functionality), and substitution with subflow (the original service invocation is not only replaced with another single service invocation, but with a whole subcomposition). This adaptation is similar to the another type of adaptation, structural adaptation. In this case not only the data or service bindings of a composition are changed, but the logical structure of the composition itself. This includes simpler cases like removing activities in an instance (e.g., skip the "quality control" activity) and more complex adaptations, where an entire subtree of the composition definition is replaced (e.g., outsource the assembling process to an external provider). Please refer to our earlier publications [8], [9] for details on how these actions are implemented. Most important for the remainder of this paper is to know how adaptation actions are defined in the PREVENT framework. We have sketched this in Figure 4.

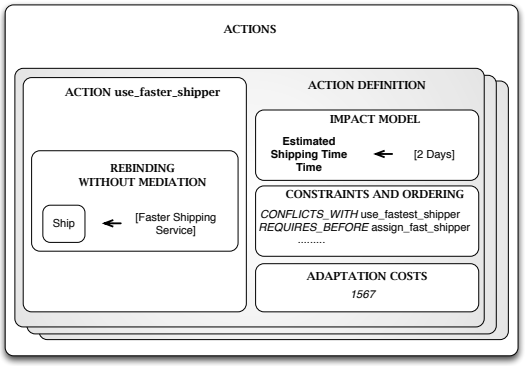

Fig. 4: Definition of Adaptation Actions

As depicted in Figure 4, we can define any number of adaptation actions which can be applied to an instance. Each of those definitions contains the description of the actual action, which can be any of the action types discussed above. In addition, the action definition also contains the impact model of the action, a list of constraints and ordering clauses and the costs of applying this action. We assume that every adaptation action has a constant, non-negative cost. For example, the cost of using a faster shipping service is the cost of using the new service minus the costs of using the original shipping service. The impact model contains a set of impact clauses. Every impact clause represents the concrete impact that applying this adaptation action has on one concrete monitorable fact or estimate. Essentially, therefore, the clauses model updates to the data used to generate predictions (see Figure 3). Every adaptation action can have any number of positive as well as negative impacts on any fact or estimate. This impact value can be determined in several ways: (1) based on measured

history data if the corresponding advice has already been used before, for example, using data mining; (2) based on SLAs with external providers, if such SLAs exist; or (3) by using QoS aggregation techniques [18]. We assume that the impact model specifies impact clauses for all metrics which the advice affects. Of course, impact clauses do not need to be exact (very often it will realistically be impossible to statically define an exact impact model before execution), however, more exact impact models lead to better predictions of SLOs after adaptation, which in turn leads to a better end-to-end performance of the PREVENT system.

4 Optimization Problem Formulation

In this section we formalize the problem of selecting the most cost-effective adaptation actions to prevent one or more predicted SLA violations in a service compositon.

We consider an interaction of the service composition with a given client, who has a given SLA with the composition provider. Let I be the set of all possible composition instances of this client, and let $i \in I$ be concrete instances that we can monitor using the PREVENT tooling. Furthermore, let $S = \{s_1, s_2, \dots s_k\}$ be the set of SLOs defined in the relevant SLA. As part of the SLO definition, a penalty function is associated with all SLOs in S. Collectively, we refer to these functions as $P = \{p_{s1}, p_{s2}, \dots p_{sk}\}$. Penalty functions define the costs for the provider based on a measured SLO value, i.e., they are functions defined as $p_s : \mathbb{R} \rightarrow \mathbb{R}, s \in S$. Similarly, the measured value of an SLO m_s is a function $m_s : I \rightarrow [0:1]$. We normalize SLO values to the interval $[0:1]$ in order to make them comparable. Putting it all together, we define the penalty function for a given SLO s and instance i as $p_s^i \stackrel{\text{def}}{=} p_s(m_s(i))$.

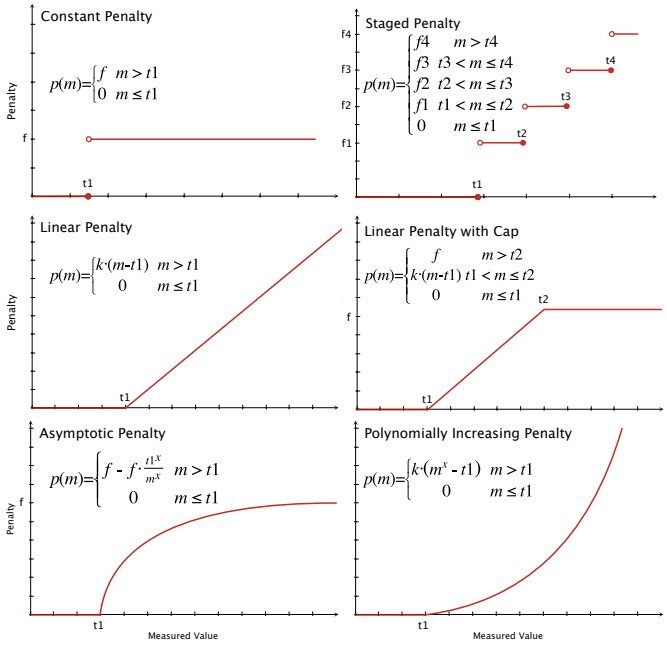

Fig. 5: Typical SLA Penalty Functions

Penalty functions for SLOs can take many different shapes. We give some examples of typical penalty functions in Figure 5. Even though this is not a complete list, these functions still incorporate the most common types of penalties: (1) *constant* penalty (a constant payment needs to be made if a certain SLO threshold value is surpassed), (2) *staged* penalty (similar to a constant penalty, but with different levels of penalty), (3) *linear* penalty (the penalty is linearly increasing with the degree of violation), (4) *linear* penalty with cap (the penalty is linearly increasing up to a maximum value), (5) *asymptotic* penalty (the penalty asymptotically converges against a maximum penalty), and (6) *polynomially increasing* penalty. Note that even though these functions apparently span many different types of mathematical functions, they share two essential characteristics. Firstly, SLA penalty functions are always monotonically increasing, i.e., $\forall p_s \in P : \forall x_1, x_2 \in \mathbb{R} : (x_1 < x_2) \implies (p_s(x_1) \leq p_s(x_2))$. This is evident, since the penalty for a higher degree of violation should never be smaller than the penalty for a lesser violation. Secondly, SLO penalty functions always have a point discontinuity in a special violation threshold point (t_1). Before (and including) t_1 the penalty is generally 0 (no violation has occured), and beyond this point a positive penalty needs to be paid ($\forall s \in S : \forall x \in \mathbb{R} : (x \leq t_1 \iff p_s(x) = 0) \wedge (x > t_1 \iff p_s(x) > 0)$). This also means that penalty functions are generally discontinuous. Furthermore, this property signifies that there is no incentive for the service provider to apply further adaptation and improve an SLO value below t_1, since all further improvements do not further reduce his costs (they are already 0 for this SLO).

To prevent violations, we are able to apply a number of possible adaptations to an instance i. We define $A = \{a_1, a_2, \dots a_l\}$ as the set of all possible adaptation actions, and $A^* \in \mathcal{P}(A)$ ($\mathcal{P}(A)$ denotes the powerset of A) as the subset of adaptation actions that are selected to be applied. We assume that all adaptations have some costs associated, defined as a cost function $c : A \rightarrow \mathbb{R}$. We assume that cost functions are constant, that is, we do not consider cross-pricing models for services [19], which would lead to non-constant costs of adaptation. Furthermore, adaptation actions have some defined impact on the composition instance i if applied. Hence, we define the transformation of i to a modified instance i' using the $\circ$ operator, defined as a function $\circ : I \times \mathcal{P}(A) \rightarrow I$. This is captured by the impact model which has to be specified as part of the action definition (see Section 3).

Selecting the most cost-effective adaptation actions means finding the adaptation actions ($A*$) that minimize the total costs for the service provider. The total costs TC are defined in Equation 1 as the sum of the costs of SLA violations after adaptation (VC) and the costs of adaptation (AC).

$$TC : \mathcal{P}(A) \rightarrow \mathbb{R}, TC(A^*) = VC(i \circ A^*) + AC(A^*) \quad (1)$$

AC is the sum of the costs of all applied adaptation actions (Equation 2).

$$AC : \mathcal{P}(A) \rightarrow \mathbb{R}, AC(A^*) = \sum_{a_x \in A^*} c(a_x) \quad (2)$$

VC is defined as the sum of all penalty functions applied to an instance (Equation 3).

$$VC : I \rightarrow \mathbb{R}, VC(i) = \sum_{s_x \in S} p^i_{sx} \quad (3)$$

Obviously, the goal of the service provider is to minimize TC. Hence, the optimization objective becomes finding the A^* that minimizes TC for a given instance i (Equation 4).

$$TC(A^*) = \sum_{s_x \in S} p^i_{sx} + \sum_{a_x \in A^*} c(a_x) \rightarrow min! \quad (4)$$

Note that we can easily calculate AC for any given A^*, but at optimization time VC is unknown (we do not know for sure which SLOs will be violated, with or without adaptation). However, the SLO Predictor provides estimations for SLOs based on instance data (see Section 3). Hence, we can assume that we have estimation functions $e_s : I \rightarrow \mathbb{R}, s \in S$ available for each SLO, which estimate the concrete penalty values in advance with a reasonably small estimation error ϵ ($\forall s \in S, i \in I : |e_s(i) - p_s(i)| < \epsilon$). Replacing VC with its estimation using e_s leads to Equation 5, which we can solve.

$$TC(A^*) \approx \sum_{s_x \in S} e^i_{sx} + \sum_{a_x \in A^*} c(a_x) \rightarrow min! \quad (5)$$

However, not all combinations of adaptation actions are legal. Some adaptation actions are mutually exclusive (e.g., `use Shipping Service DHL` and `use Shipping Service UPS`), while others depend on each other (see our earlier work [9] for details on dependencies between adaptation actions). For simplicity, we capture these additional constraints using a penalty term $v : \mathcal{P}(A) \rightarrow \mathbb{N}$. The definition of v is shown in Equation 6.

$$v(A^*) = \begin{cases} \infty & A^* \text{ contains constraint violation} \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

By incorporating this penalty term we arrive at our final target function (Equation 7).

$$TC(A^*) \approx v(A^*) + \sum_{s_x \in S} e^i_{sx} + \sum_{a_x \in A^*} c(a_x) \rightarrow min! \quad (7)$$

We have all necessary information to evaluate Equation 7 at optimization time for any set of actions A^*. However, finding the A^* that minimizes $TC(A^*)$ is still far from trivial, since Equation 7 is discrete and cannot be optimized analytically. We present algorithms to find a (near-)optimal solution in Section 5.

5 Algorithms

We will now discuss different approaches for finding solutions to this problem. These algorithms are implemented in the *Cost-Based Optimizer* component. Optimization is always triggered by a predicted violation of at least one SLO, and receives as input a list of monitored facts and estimates of the current instance.

5.1 Branch-and-Bound

Branch-and-Bound is a very general deterministic algorithm for solving optimization problems. The high-level idea of this approach is to enumerate the solution space in a "smart" way, so that at least some sub-optimal solutions can be identified and discarded prematurely, i.e., before they have been fully constructed and evaluated. We use binary encoding to represent solutions, i.e., every solution is represented as a binary vector, and an adaptation action with index j is applied iff the solution vector is 1 at index j. For example, the solution vector 00110100 encodes that the third, fourth and sixth adaptation action should be applied. Evidently, $2^{|A|}$ different solutions exist for each optimization problem, where $|A|$ is the number of possible adaptation actions (but not all combinations need to be legal). For solutions that are still being constructed we allow a third symbol, "$*$", representing an action which is still undecided (alive). We refer to solutions which contain at least one alive action as partial, and solutions which do not contain any alive actions as complete. Therefore, the vector $001101*0$ is a partial solution, where the last-but-one action is alive.

```
1  # name: bab
2  # input: partial solution p,
3  #        next alive action index i,
4  #        target function v
5  # output: optimal complete solution
6
7  bab(p,i):
8    # recursion break condition
9    if(p is complete solution)
10     return p
11
12   # check if this sub-tree can be pruned
13   if(p is pruneable)
14         forall alive_actions(p) as j
15            set p(j) = 0
16     return p
17
18   # investigate solution sub-tree with p(i)=0
19   set p(i) = 0
20   s1 = bab(p,i+1)
21
22   # investigate solution sub-tree with p(i)=1
23   set p(i) = 1
24   s2 = bab(p,i+1)
25
26   # return better solution from both subtrees
27   if(v(s1) <= v(s2))
28     return s1
29   else
30     return s2
```

Fig. 6: Branch-and-Bound Algorithm

We describe our general branch-and-bound algorithm in Figure 6. The algorithm is straight-forward and easy to

understand. What is most important is the implementation of Line 13, the rules for pruning the search tree (i.e., for prematurely discarding solutions). In our branch-and-bound approach, we prune a partial solution in two cases: (1) if the partial solution already contains at least one conflict, or (2) if the partial solution already prevents all SLA violations (the penalty function p_s is 0 for all SLOs s) without applying any more actions. Case (1) is trivial, since the target function value for all solutions in such a sub-tree will always be ∞. Case (2) lends itself to more discussion. Remember the assumption that every action has non-negative costs, and that we described SLA penalty functions as non-negative functions. Therefore, we can assure that for any solution where all penalty functions are 0, the additional application of more actions can never improve the target function value. Hence these partial solutions cannot be improved by applying more actions, and the remaining solution sub-tree can be pruned.

Note that in Listing 6, we simply iterated over all actions in the order they appeared in the solution vector (in every step, we always just investigate the next action, see Lines 18 and 22). In general, this approach is suboptimal. Even though the order in which we investigate actions has no impact on the quality of our solution (the algorithm is deterministic, i.e., we will always find the global optimum eventually), the order may have an impact on the number of solutions we are able to prune. This is illustrated in Figure 7. Assume the following simple scenario: there is only one SLO, and 3 possible adaptations. Only adaptation 3 is able to prevent the violation of the SLO. Actions 1 and 2 have costs but no relevant influence. There are no conflicts between actions. Hence, the optimal solution vector is 001. In Figure 7(a), we strictly followed the algorithm in Listing 6 and investigated the actions in the order they appear in the solution vector. Since the only "useful" action is investigated last, we extend the whole solution tree without any pruning (the worst case, equivalent to full enumeration). Now, in Figure 7(b), we investigate the actions in reverse order (from back to front). Now, the "useful" action is investigated first, and a large part of this solution tree can be pruned according to pruning case (2).

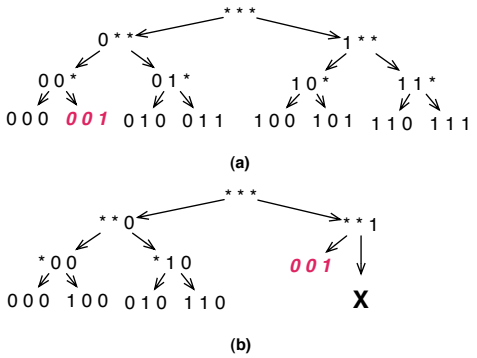

Fig. 7: Pruning of Solution Trees

Therefore, we can conclude that it is beneficial to investigate actions in a specific order that maximizes the number of solutions that can be pruned. We specify two possible criteria for this ordering: (1) the *impact* of an action on the SLOs (actions with higher total impact should be investigated first), and (2) the *utility* of an action (actions with higher utility should be investigated first). We will now define those two orderings.

Assuming we have a defined set of historical process instances available (we already used the same assumption in Section 3 to train the *Violation Predictors*) we can calculate an estimation of impact and utility of each action as follows. We use the same definitions as in Section 4. Additionally, we define the set of available historical process instances as $H = \{h_1, h_2, \dots h_q\}$, with $H \subseteq I$. We refer to the number of historical instances as $q = |H|$. Now, we are able to calculate an estimation of the overall impact of an adaptation action a on a SLO s as $\Delta_{a,s}$ (Equation 8). Simply put, the impact is the arithmetic mean of the difference between SLO value with and without applying the adaptation to each historical instance.

$$\Delta_{a,s} = \sum_{h \in H} \frac{m_s(h) - m_s(h \circ \{a\})}{q} \tag{8}$$

Note that we have already defined in Section 4 that SLA penalty functions are monotonically increasing. Hence, higher impact values are generally good. However, the impact value may also be negative (i.e., $m_s(h) < m_s(h \circ \{a\})$). In this case this action has a negative impact on one of the SLOs, which is reasonble and realistic. For instance, an adaptation which reduces the process lead time can very well have a negative impact on the SLO cost compliance. Based on $\Delta_{a,s}$, we can now define the total impact of each action (Equation 9).

$$\Delta_a = \sum_{s \in S} \Delta_{a,s} \tag{9}$$

Finally, we define u_a as the utility of an action (Equation 10).

$$u_a = \frac{\Delta_a}{c(\{a\})} \tag{10}$$

u_a is the sum of the impact values of this action on all SLOs in relation to the costs of this action. A high utility means that, in average, this action is cheap for its usefulness in preventing violations. We can now improve the branch-and-bound algorithm trivially: instead of investigating the actions in the order they are specified in the solution vector, we now investigate them either in the order of their impact Δ_a (impact-based sorting), or in order of their utility u_a (utility-based sorting). We will evaluate and discuss both alternatives in Section 6, and compare them to Branch-and-Bound with randomly ordered actions.

5.2 Local Search

While the Branch-and-Bound algorithm discussed above has the advantage of always finding the optimal set of actions for any composition instance, the execution time of the algorithm increases exponentially with the number of available actions. Even though we can reduce the runtime using impact- or utility-based sorting of actions, the complexity still remains exponential. Hence there is an evident need to find strong heuristics, i.e., non-deterministic algorithms that find "good" (even if not necessarily optimal) solutions in polynomial time.

A simple heuristic that is often used to very good ends is Local Search. Local Search is a metaheuristic, i.e., final solutions are constructed by iteratively improving a start solution. The general idea is that in each iteration the algorithm searches a specified *neighborhood* for better solutions than the current one. If at least one such solution is found the algorithm progresses to the next iteration with one of the better solutions (typically the best one in the neighborhood, equivalent to steepest descent). If no better solution can be found in the neighborhood the algorithm has converged to a local optimum and is terminated. Usually, this algorithm is repeated multiple times with different starting solutions (since, obviously, different starting solutions can lead to different local optima). This kind of algorithm typically depends on the definition of (1) a suitable neighborhood and (2) a senseful selection of starting solutions. We use the following neighborhood definition: a complete solution vector is in the neighborhood of an original vector if the two vectors have a Hamming distance of 1, i.e., if they differ in exactly one bit. We have visualized this neighborhood definition in Figure 8.

Fig. 8: neighborhood for Local Search

According to this definition, every solution vector has a neighborhood of size $|A|$. Additionally, we note that two solutions in the same neighborhood have a Hamming distance of 2.

For selecting the start solutions, we use two different approaches. The first and primitive one is to select n start solutions with m bits set to 1 at random. Alternatively, we propose to use an algorithm commonly referred to as GRASP [20] (greedy randomized adaptive search procedure). GRASP is essentially a variation of local search where the start solutions are constructed using a greedy heuristic. The idea is that GRASP can converge to a better solution than a simple local search because the start solutions are already better than random start solutions. However, some attention needs to be paid to using a greedy construction heuristic that actually generates start solutions which are both of reasonable quality and at the same time widely spread over the search space.

```
# name: grasp_init
# input: number of start solutions n,
         RCS max size r
# output: set of start solutions

grasp_init(n, r):
  G = {} // empty set of start solutions
  repeat n times:
    pa = empty_partial_solution
    while( VC(pa) > 0 ):
      rcs = construct_rcs(pa, r)
      if( empty(rcs) )
        break
      a = random(rcs)
      pa(a) = 1
    add(G, pa)
  return G
```

Fig. 9: GRASP Construction Heuristic

We have sketched the construction heuristic that we have used in our implementation of GRASP in Figure 9. Summarizing, the algorithm constructs n solutions by stepwise addition of actions selected randomly from a restricted candidate set (RCS). The heuristic is based on similar concepts that we have already used in our discussion of Branch-and-Bound: the idea is to stop adding actions if either no more SLOs are violated or no senseful actions are available anymore (the RCS is empty), and to prefer adding actions which have a high utility value (u_a). Hence, in every step the RCS consists of the r (maximum size of the RCS) actions with highest non-negative u_a, which have not yet been added and which do not lead to a conflict.

5.3 Genetic Algorithm

As an alternative to locality-based heuristics (local search, GRASP) we also present a solution based on the concept of evolutionary computation. More precisely, we use genetic algorithms [21] (GA) as a more complex, but potentially also more powerful heuristic to generate good solutions to the cost-based optimization problem. The overall idea of GA is to mimic the processes of evolution in biology, specifically natural selection of the fittest individuals, crossover, and mutation. Therefore, in GA we rather work on a population of solutions instead of a single one. We use the term "fit" to describe solutions with a good (low) target function value. Firstly, we generate a random start population. For this, we use the same primitive construction scheme as discusssed above for local search: we randomly apply m actions in every solution. Every following iteration of the algorithm (referred to as generations) essentially follows a three-step pattern.

Firstly, we *select* a set of solutions from the population to "survive" into the next generation. In our genetic algorithm implementation, the fittest solution (i.e., the one with the lowest target function value) is selected deterministically (elitism), while all remaining slots in the

next generation population are selected using a process called tournament selection. In tournament selection, t random solutions from the last generation are put into a tournament. The fittest solution of the tournament is selected into the next generation. The parameter t steers the selection pressure: low t increases the time that the population takes to converge against a solution, but high t increases the danger of converging against a local optimum instead of the global one.

Secondly, *crossover* is used to produce new solutions based on the selected ones from the last generation. The main challenge of implementing a strong crossover mechanism is to ensure that the crossover product of two fit solutions is also likely to be fit. Given the binary vector representation we use to encode solutions we can make use of a simple one-point crossover scheme. We choose a random crossover point cp from $[1 : |A| - 1]$. To construct a new child we copy the binary vector of the first solution from the start until cp, and the vector of the second solution from $cp + 1$ to the end of the vector.

This simple procedure ensures that characteristics of both original solutions are preserved. However, because of the random selection of cp it is possible that the child solution has a conflict, even if this was not the case for any of the parents. In this case, we remove one of the conflicting actions at random.

Thirdly, we use *mutation* to introduce entirely new features into the population. The need for mutation can be illustrated easily: assume that a given action a is not applied in any solution in the population. Using one-point crossover as discussed above it is not possible to create any solution that uses a. Hence, we introduce some additional randomization. After crossover, we may randomly flip every bit in every solution in the population with a very small probability. This means that most solutions in the population are not mutated, but sometimes new actions are applied which are not the product of crossover.

Name	Description	Default
Population Size	Number of solutions in every generation	150
Selection Pressure	Number of solutions to select for tournament selection	2
Crossover Probability	Probability per solution that crossover is applied	0.8
Mutation Probability	Probability per bit that mutation is applied	0.02
Break Condition	Condition for stopping the algorithm	No improvement in 20 generations

TABLE 4: GA Configuration Parameters

GAs are notorious for having many parameters to fine-tune the performance of the optimization. For illustrative purposes, we have summarized the parametrization options available in our implementation of GA in Table 4, including some values that we found to provide useful default parameters if applied to the cost-based optimization problem. Evidently, further customization would also be possible, for instance by using a different selection or crossover scheme. Unless stated otherwise, we will use the configuration described in Table 4 for experimentation in Section 6.

Unfortunately, this "canonical" GA implementation takes a significant amount of time to converge against a solution, since the solution space is searched solely through the (rather unguided and strongly randomized) means of crossover and mutation. One possibility to improve this aspect is to combine the canonical GA with local optimization as presented above. This leads us to an adapted algorithm, which we have sketched in Figure 10. In literature, such combinations of GA and local search are often referred to as Memetic Algorithms [22] (MA).

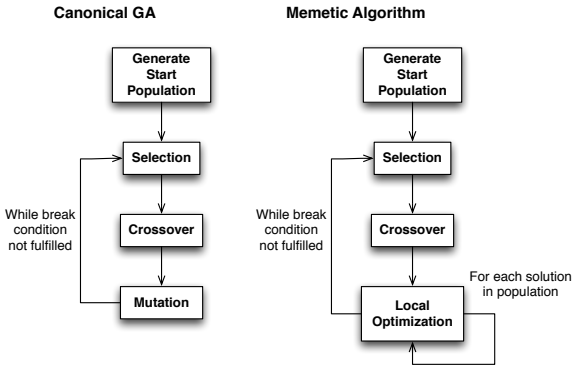

Fig. 10: Memetic Algorithm

The main changes of MA (as compared to GA) are as follows. Firstly, a new *Local Optimization* operator is introduced after crossover. Local optimization applies the local search algorithm as discussed above to each solution in the generation, basically reducing the population to a set of locally optimal solutions. Secondly, we remove the mutation operator from the algorithm (technically speaking, we set the mutation probability parameter to 0). The main reason is that given that all solutions in the population are already locally optimal, randomly mutating one bit in a solution can only lead to a worse solution. In theory it is possible that multiple bits in a single solution are mutated at the same time, and that these mutations lead to an improvement, but this corner case is very unlikely in practice. Furthermore, the main motivation for having mutation in the first place was that it is the only way of introducing new actions in the canonical GA. This is no longer the case, since local search can do the same thing.

Generally, MA is slower than GA, since more solutions are evaluated in each generation (evidently, MA executes one local search for every solution in each generation). However, the algorithm potentially converges against a very good solution in a low number of generations. Hence, we argue that in practice MA improves on the canonical form most of the time for our problem. This will be substantiated further in Section 6.

6 EXPERIMENTATION

In the following section, we will numerically validate the algorithms discussed in Section 5 based on an implementation of the scenario presented in Section 2. For reasons of brevity we only summarize the experiment setup here. More details can be found in the accompanying experimentation web page[2]. We have implemented the scenario using .NET Windows Communication Foundation[3] (WCF) technology and the VRESCo SOA runtime environment on a server running Windows 2007 SP2. The server machine was equipped with 2 2.99GHz Xeon X5450 processors and 32 GByte RAM. In order to train PREVENT we have initialized the system with a set of 9796 historical composition instances. These instances were created by executing the service composition repeatedly. In this historical data set, 3660 instances have not been adapted, while one or more adaptation actions have been applied in the remaining 6136 instances. In our experiments, we consider the case of an SLA containing up to five SLOs, similar to the previous example. Note that we have used an integer value in $[0 : 15]$ to represent product quality in this example, in order to allow for more fine-grained distinctions of different levels of product faults. In Table 5 we have sketched these SLOs and their basic statistics. μ is the mean value of the SLO without adaptation. $\mu*$ is the mean among instances to which some adaptation has been applied. σ and $\sigma*$ are the respective standard deviations. As before, t_1 is the violation threshold. Furthermore, SLO 1 is associated with a staged penalty function with 9 stages, SLO 2 and 3 are both associated with fixed penalty functions, SLO 4 is associated with a linear penalty function with cap, and SLO 5 with a linear penalty function without cap. Additionally, we have defined 49 adaptation actions that have positive and negative influences on some or all of these SLOs. Every action has been associated with a positive cost value.

As a first experiment, we analyse the suitability of different variants of the Branch-and-Bound algorithm. As all of these algorithms are deterministic, we are guaranteed to find the optimal solution to any optimization problem eventually. However, the three different versions of the algorithm (Branch-and-Bound with random action sorting, with impact-based sorting, and with utility-based sorting) may differ significantly with regard to their runtime. As an independent measure of algorithm runtime, we use the number of solutions that have to be evaluated. All results concerning algorithms with randomized elements are arithmetic means of 5 repeated runs.

Figure 11 plots the number of solutions depending on the number of adaptation actions that are available (up to a maximum of 17 actions, note the logarithmic scale on the y-axis). For reasons of comparison we also plot local search in the figure, whose runtime grows linearly with the number of actions. It was not feasible to evaluate Branch-and-Bound for more than 17 actions.

2. http://www.infosys.tuwien.ac.at/prototype/VRESCo/experimentation.html

3. http://msdn.microsoft.com/en-us/library/ms735967(VS.90).aspx

#	SLO Name	μ	μ*	σ	$\sigma*$	t_1
1	Order Fulfillment Time	38811	35560	4708	6004	37000
2	Payment Time	4187	2202	28	1124	4150
3	Shipping Time	1285	864	144	347	1300
4	Product Quality	2.6	3	1.9	2.5	3
5	Cost Compliance	851	1149	212	521	1400

TABLE 5: Case Study SLOs

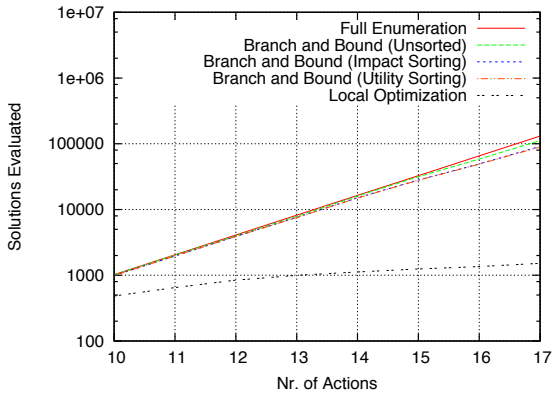

Fig. 11: Solutions Evaluated For Branch-and-Bound

As we can see, there is little difference between the three variants of Branch-and-Bound, and none is able to reduce the number of solutions that have to be evaluated significantly below full enumeration. The reason for this unsatisfying result is that, in this concrete optimization instance, very little combinations of actions can prevent the violation of all SLOs (the SLOs are conflicting), i.e., bounding condition 2 cannot be applied very often. We can see that, by relaxing the problem and disabling SLOs 4 and 5, a significant performance boost can be achieved (Figure 12) by both impact-based and utility-based sorting. The difference between impact-based and utility-based sorting is not significant in this experiment.

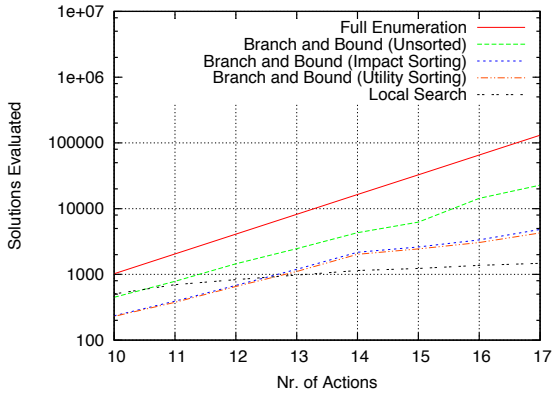

Fig. 12: Solutions Evaluated Without Conflicting SLOs

However, even though smart action sorting can reduce the solution space if there are no conflicting SLOs, the

number of solutions that need to be evaluated still grows exponentially with the number of available actions. Hence, solving the cost-based optimization problem deterministically is only possible for very small problems. If the set of possible adaptations grows, we need to fall back to heuristic optimization. For these algorithms, there are no guarantees about the quality of the solution. That means that we need to compare them in two dimensions. Firstly, and similar to before, we need to look at the number of solutions that are evaluated before the algorithm produces the final result (Figure 13), as a measure of the runtime of the algorithm. Secondly, we also need to take into account the quality of the best found solution (Figure 14).

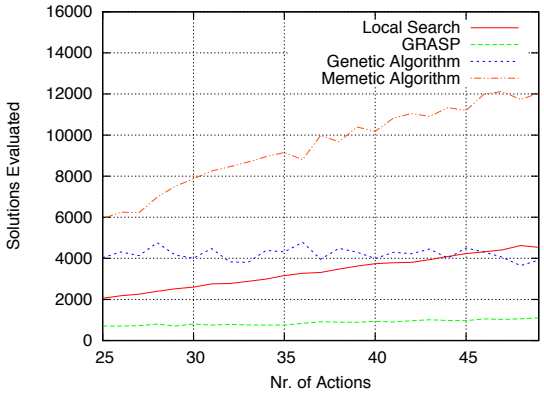

Fig. 13: Solutions Evaluated Per Heuristic Algorithm

In Figure 13 we can see that, not surprisingly, all algorithms scale much better than Branch-and-Bound (note the linear scale on the y-axis and compare with Figure 12). GRASP is very efficient, and the fastest algorithm in this experiment with an almost constant runtime. The computation of local search is also reasonably efficient, but the number of solutions that have to be evaluated increases more strongly as compared to GRASP. This is because for GRASP the start solutions are already better, hence less local search steps are necessary before a solution is reached. Note that the number of solutions evaluated for local search is directly proportional to the number of start solutions used. In this experiment we used 25 start solutions. If we had used 50 start solutions instead, the runtime of local search would have been almost on the level of MA. GA also has a relatively constant runtime, but on much higher level than GRASP. The slowest algorithm in this experiment is MA, which is due to its unique combination of local optimization and genetic algorithm.

With regards to solution quality, we observe a quite clear ordering of algorithms. GRASP and MA generally perform best. For most instances, MA is slightly better, even though this is not true for all cases. GA comes in third and local optimization with random start solutions produces, in average, solutions vastly inferior to all competitors.

Drawing conclusions from these experiments, we note

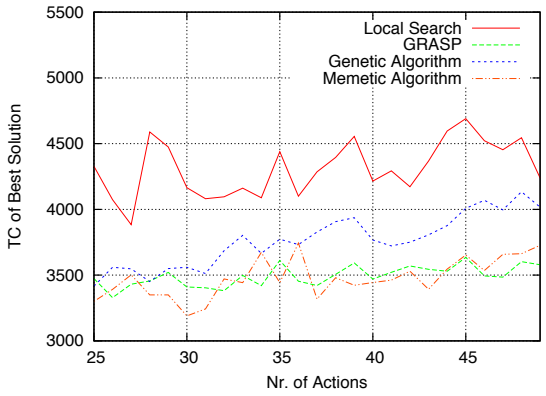

Fig. 14: Quality of Solution Per Heuristic Algorithm

that Branch-and-Bound is applicable in situations where just a small set of actions is available. In general, impact-based or utility-based sorting should be used instead of random sorting, since there is no evident disadvantage to these approaches and they may be helpful if there are no conflicting SLOs. We did not discover a significant difference in the performance of these two variants. If more actions are available, MA and specifically GRASP are interesting candidate algorithms. GRASP produces good solutions in very little time and can generally be used even for short-running compositions where adaptation decisions need to be taken in a short time frame (below 1 second). MA is very promising in case of long-running compositions, where the time necessary to find a solution is not critical. MA often produces slightly better solutions than GRASP, but takes much more time to do so.

In a second set of experiments, we now evaluate the end-to-end effectiveness of PREVENT. That is, we analyze if the system fullfills its main promise, preventing SLA violations and reducing the total costs for the service provider. Hence, we execute 500 instances of the scenario composition and monitor the actual total costs and violations (after adaptation). We compare these numbers with the number of violations and the total costs that the PREVENT SLO Predictor can predict after roughly half of the service composition if finished. We assume that these predictions reflect the violations and costs that we would end up with if we did nothing at all. Since our case study is rather short-running, but uses a relatively large set of adaptations, we use GRASP for cost-based optimization. The results of this experiments are depicted in Table 6.

Evidently, the usage of PREVENT fulfills its main promise. Using PREVENT the total number of SLO violations decreases to about 28% of the number of predicted violations. However, we can also see that PREVENT does not primarily prevent violations, but rather aims at minimizing the costs of violations. For instance, for SLO 4 and 5 the total number of violations even increases. This is because these SLOs are conflicting with the first SLOs, and SLO 1 is in general the most expensive one

	SLO 1	SLO 2	SLO 3	SLO 4	SLO 5	Total
Considering Costs						
Violations Predicted/Actual	209/129	442/1	390/42	75/104	0/41	1116/317 (-71.6%)
Avg. Costs Predicted/Actual	5207/2904	884/2	6264/558	840/1068	0/9	14923/8415 (-43.6%)
Ignoring Costs						
Violations Predicted/Actual	223/40	449/0	245/17	66/218	0/115	983/390 (-60.3%)
Avg. Costs Predicted/Actual	5521/756	898/0	4086/216	756/2364	0/29	12632/22241 (+76.1%)

TABLE 6: End-to-End Results

to violate. Hence, PREVENT happily trades violations of SLO 4 and 5 for preventing violations of SLO 1. Thereby, the total costs for the service provider can be reduced to 56% of the predicted costs. The lower part of the table validates the claim of the paper that it makes sense to incorporate the costs of adaptation into the decision process. To that end, we have modified the target function of the optimization in such a way that the costs of adaption are ignored. In this configuration the total costs after adaptation are 176% of the predicted costs. That means that in this experiment it is in fact much more expensive for the provider to prevent adaptations (in the way that optimization ignoring costs suggests) than doing nothing at all.

7 Related Work

In this paper, we discuss the problem of cost-optimal prevention of SLA violations in service compositions. To the best of our knowledge, no approaches with this exact focus have been published so far. However, there are some areas relevant or related to this problem, which we discuss in the following.

On a fundamental level, our work is based on the notion that both atomic and composite services exhibit some measurable quality (QoS). Monitoring QoS has been an active research area for some time. Different techniques proposed in this direction include monitoring based on client feedback [23], monitoring of TCP-level metrics using network analysis techniques [15] or event-based monitoring based on event-condition-action rules [24]. We use the VRESCO event engine and event-based monitoring in a manner very similar to the approach presented in [24].

The PREVENT approach aims at autonomous optimization of service compositions with regards to SLA violations and costs of adaptation. This bears a natural resemblance to the idea of QoS optimization for service compositions, as prominently described in [25]. Later approaches tried to improve on this concept by using more efficient heuristic algorithms, e.g., H1_RELAX_IP [26] (a heuristic relaxation of integer programming), WFlow [27] (based on stochastic workflow reduction) or the immune algorithm [28]. Different authors approached the problem by combining global optimization and local selection (which can be done much more efficient than global optimization). This approach can also be considered a heuristic, since the combination with local selection does not guarantee a globally optimal solution [29]. Most comparably to our work, the authors of [30] use a genetic algorithm combined with local search to efficiently solve the QoS optimization problem. The main difference of our work to all these approaches is that we do not optimize the composition with regard to global QoS goals. Instead, our optimization goal is to minimize the costs resulting from SLA violations and adaptations. Therefore, in our work, some SLAs are allowed to be violated if it is financially desirable for the provider to do so. Hence, the optimization problem we have to solve is different.

To our work, even more important than the measurement of past quality is the prediction of future QoS. One well-known approach to establishing predictable QoS levels in a composite service is QoS aggregation, i.e., the process of calculating the quality dimensions of a composite service based on the QoS of the utilized services and aggregation functions. QoS aggregation has for instance been discussed in [18]. The concept of QoS aggregation has been extended to SLA aggregation by several authors [31], [32]. As an alternative to QoS and SLA aggregation, different authors have proposed to use various machine learning techniques to predict composition QoS from monitored runtime data [6], [7]. This approach is also the one that we use in PREVENT, as explained in Section 3. The main advantage that we see in using machine learning is that it is very easy to incorporate non-QoS data (composition instance data, such as customer identifiers or ordered products) without the need to explicitly specify aggregation rules describing how this data influences the composition performance.

Generally, PREVENT is a system to monitor and prevent SLA violations. In this area, some work exist which discuss the runtime monitoring of composition quality, such as [33]. This paper is of particular interest to us, since it discusses an integrated approach towards monitoring based on events. As stated above, this is quite related to monitoring in PREVENT. These works do not attempt to explain the reasons for SLA violations, and neither do they try to prevent them. The MoDe4SLA approach [4] is a top-down approach towards identifying these influential factors of SLA violations. Research in a similar direction, but using data mining techniques instead of top-down analysis, has also been presented in [5]. Our work is different in that we do not only try to identify which parts of a service composition cause SLA violations, but actively prevent them by applying targeted adaptation actions. Therefore, our system essen-

tially implements the paradigm of self-adapting service compositions. This is related to the area of flexible service composition, as introduced in [34]. Flexible service compositions reoptimize their composition at runtime, in order to deal with unanticipated problems. Similar ideas (self-healing processes) have also been presented as part of the DISC framework [35], which implements dynamic and only partially defined processes. A different kind of self-healing processes have been discussed in [36]. In this paper, the authors present the VieDAME framework, which autonomously monitors the QoS of services used in the composition, and triggers service re-selection if the monitored QoS falls below a given threshold. This is similar to the PREVENT approach, but our system supports a wider range of adaptation actions (as discussed in our earlier work [8]). Additionally, [36] does not take the costs of adaptation into account. Another middleware for self-adapting compositions is MASC [37]. However, the authors of this paper focus more on adaptation for functional reasons, while our main goal is the optimization of non-functional aspects. Furthermore, the MASC system also does not explicitly incorporate costs of adaptation.

The core contribution of this paper is the notion that there generally is a tradeoff to consider between preventing SLA violations and the costs of doing so. Hence, a composite service provider is maximizing his own revenue by minimizing his total costs. Similar models have been investigated in many related areas before. For instance, [38] describes a model for revenue maximizing in Web services hosting using dynamic admission policies. Similarly, techniques to optimize application servers in a way to maximize the provider profit in distributed systems have been proposed in [39]. Other tradeoffs that have been discussed in the literature include the performance-security tradeoff [40] or the tradeoff between composition QoS and the costs of monitoring [41]. To the best of our knowledge, no research paper has yet considered the tradeoff between preventing violations and the costs of doing so.

8 Conclusion and Future Work

For providers of composite Web services it is essential to be able to minimize cases of SLA violations. One possible route to achieve this is to predict at runtime which instances are in danger of violating SLAs, and apply various adaptation actions to these instances only. However, it is not trivial to identify which adaptations are the most cost-effective way to prevent any violation, or if it is even possible to prevent a violation in a cost-effective way. In this paper we have modelled this problem as a one-dimensional, discrete optimization problem. Furthermore, we have presented both deterministic and heuristic solution algorithms. We have evaluated these algorithms based on a manufacturing case study, and show which types of algorithms are better suited for which scenarios.

The main current limitation is that adaptation is only considered on instance level, that is, for each composition instance separately. Aggregate SLOs, which are defined over a number of instances, are out of scope. Similarly, at the moment we do not consider 'permanent' adaptations, i.e., adaptations which are done for all future instances. We believe that the PREVENT adaptation model can be extended to this kind of SLOs and actions, but new approaches to predict violations and impact models are needed to this end.

References

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges," *IEEE Computer*, vol. 40, no. 11, pp. 38–45, November 2007.

[2] A. Lenk, M. Klems, J. Nimis, S. Tai, and T. Sandholm, "What's Inside the Cloud? An Architectural Map of the Cloud Landscape," in *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing (CLOUD'09)*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 23–31.

[3] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef, "Web Services on Demand: WSLA-Driven Automated Management," *IBM Systems Journal*, vol. 43, no. 1, pp. 136–158, January 2004.

[4] L. Bodenstaff, A. Wombacher, M. Reichert, and M. C. Jaeger, "Analyzing Impact Factors on Composite Services," in *Proceedings of the 2009 IEEE International Conference on Services Computing (SCC '09)*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 218–226.

[5] B. Wetzstein, P. Leitner, F. Rosenberg, S. Dustdar, and F. Leymann, "Identifying Influential Factors of Business Process Performance Using Dependency Analysis," *Enterprise Information Systems*, vol. 4, no. 3, pp. 1–8, July 2010.

[6] P. Leitner, B. Wetzstein, F. Rosenberg, A. Michlmayr, S. Dustdar, and F. Leymann, "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'09)*, 2009, pp. 176–186.

[7] L. Zeng, C. Lingenfelder, H. Lei, and H. Chang, "Event-Driven Quality of Service Prediction," in *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC'08)*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 147–161.

[8] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar, "Monitoring, Prediction and Prevention of SLA Violations in Composite Services," in *Proceedings of the IEEE International Conference on Web Services (ICWS'10)*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 369–376.

[9] P. Leitner, B. Wetzstein, D. Karastoyanova, W. Hummer, S. Dustdar, and F. Leymann, "Preventing SLA Violations in Service Compositions Using Aspect-Based Fragment Substitution," in *Proceedings of the International Conference on Service-Oriented Computing (ICSOC'10)*, 2010.

[10] R. Kazhamiakin, B. Wetzstein, D. Karastoyanova, M. Pistore, and F. Leymann, "Adaptation of Service-Based Applications Based on Process Quality Factor Analysis," in *Proceedings of the 2nd Workshop on Monitoring, Adaptation and Beyond (MONA+)*, 2009, pp. 395–404.

[11] "Business Process Modeling Notation Specification," Object Management Group (OMG), Tech. Rep., 2006.

[12] M. Salehie and L. Tahvildari, "Self-Adaptive Software: Landscape and Research Challenges," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 2, pp. 1–42, May 2009.

[13] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, "End-to-End Support for QoS-Aware Service Selection, Binding, and Mediation in VRESCo," *IEEE Transactions on Services Computing*, vol. 3, pp. 193–205, July 2010.

[14] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[15] F. Rosenberg, C. Platzer, and S. Dustdar, "Bootstrapping Performance and Dependability Attributes of Web Services," in *Proceedings of the IEEE International Conference on Web Services (ICWS'06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 205–212.

[16] S. Haykin, *Neural Networks and Learning Machines: A Comprehensive Foundation*, 3rd ed. Prentice Hall, 2008.

[17] J. R. Quinlan, "Induction of Decision Trees," *Machine Learning*, vol. 1, pp. 81–106, March 1986.

[18] M. C. Jaeger, G. Rojec-Goldmann, and G. Muhl, "QoS Aggregation for Web Service Composition using Workflow Patterns," in *Proceedings of the 8th International Enterprise Distributed Object Computing Conference (EDOC'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 149–159.

[19] L. Xu and B. Jennings, "A Cost-Minimizing Service Composition Selection Algorithm Supporting Time-Sensitive Discounts," in *Proceedings of the 2010 IEEE International Conference on Services Computing (SCC'10)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 402–408.

[20] T. Feo and M. Resende, "Greedy Randomized Adaptive Search Procedures," *Journal of Global Optimization*, vol. 6, pp. 109–133, 1995.

[21] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1989.

[22] N. Radcliffe and P. Surry, "Formal Memetic Algorithms," *Evolutionary Computing*, vol. 865, pp. 1–16, 1994.

[23] R. Jurca, B. Faltings, and W. Binder, "Reliable QoS Monitoring Based on Client Feedback," in *Proceedings of the 16th International Conference on World Wide Web (WWW'07)*. New York, NY, USA: ACM, 2007, pp. 1003–1012.

[24] L. Zeng, H. Lei, and H. Chang, "Monitoring the QoS for Web Services," in *Proceedings of the 5th International Conference on Service-Oriented Computing (ICSOC'07)*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 132–144.

[25] L. Zeng, B. Benatallah, A. H.H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "QoS-Aware Middleware for Web Services Composition," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 311–327, 2004.

[26] R. Berbner, M. Spahn, N. Repp, O. Heckmann, and R. Steinmetz, "Heuristics for QoS-aware Web Service Composition," in *Proceedings of the 2006 IEEE International Conference on Web Services (ICWS'06)*. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 72–82.

[27] T. Yu, Y. Zhang, and K.-J. Lin, "Efficient Algorithms for Web Services Selection with End-to-End QoS Constraints," *ACM Transactions on the Web*, vol. 1, May 2007.

[28] J. Xu and S. Reiff-Marganiec, "Towards Heuristic Web Services Composition Using Immune Algorithm," in *Proceedings of the 2008 IEEE International Conference on Web Services (ICWS'08)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 238–245.

[29] M. Alrifai and T. Risse, "Combining Global Optimization With Local Selection for Efficient QoS-Aware Service Composition," in *Proceedings of the 18th International Conference on World Wide Web (WWW'09)*. New York, NY, USA: ACM, 2009, pp. 881–890.

[30] F. Rosenberg, M. B. Müller, P. Leitner, A. Michlmayr, A. Bouguettaya, and S. Dustdar, "Metaheuristic Optimization of Large-Scale QoS-aware Service Compositions," in *Proceedings of the 2010 IEEE International Conference on Services Computing (SCC'10)*.

[31] T. Unger, F. Leymann, S. Mauchart, and T. Scheibler, "Aggregation of Service Level Agreements in the Context of Business Processes," in *Proceedings of the 12th International Enterprise Distributed Object Computing Conference (EDOC'08)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 43–52.

[32] I. Haq, A. Huqqani, and E. Schikuta, "Aggregating Hierarchical Service Level Agreements in Business Value Networks," in *Proceedings of the 7th International Conference on Business Process Management (BPM'09)*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 176–192.

[33] L. Baresi, S. Guinea, M. Pistore, and M. Trainotti, "Dynamo + Astro: An Integrated Approach for BPEL Monitoring," in *Proceedings of the 2009 IEEE International Conference on Web Services (ICWS'09)*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 230–237.

[34] D. Ardagna, M. Comuzzi, E. Mussi, B. Pernici, and P. Plebani, "PAWS: A Framework for Executing Adaptive Web-Service Processes," *IEEE Software*, vol. 24, no. 6, pp. 39–46, 2007.

[35] E. Zahoor, O. Perrin, and C. Godart, "DISC: A Declarative Framework for Self-Healing Web Services Composition," in *Proceedings of the 2010 IEEE International Conference on Web Services (ICWS'10)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 25–33.

[36] O. Moser, F. Rosenberg, and S. Dustdar, "Non-Intrusive Monitoring and Service Adaptation for WS-BPEL," in *Proceedings of the 17th International Conference on World Wide Web (WWW'08)*. New York, NY, USA: ACM, 2008, pp. 815–824.

[37] A. Erradi, P. Maheshwari, and V. Tosic, "Policy-Driven Middleware for Self-Adaptation of Web Services Compositions," in *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware (Middleware'06)*. New York, NY, USA: Springer-Verlag New York, Inc., 2006, pp. 62–80.

[38] M. Mazzucco, I. Mitrani, J. Palmer, M. Fisher, and P. McKee, "Web Service Hosting and Revenue Maximization," in *Proceedings of the Fifth European Conference on Web Services (ECOWS'07)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 45–54.

[39] D. Villela, P. Pradhan, and D. Rubenstein, "Provisioning Servers in the Application Tier for E-Commerce Systems," *ACM Transactions on Internet Technology*, vol. 7, no. 1, p. 7, 2007.

[40] S. S. Yau, Y. Yin, and H. G. An, "An Adaptive Tradeoff Model for Service Performance and Security in Service-Based Systems," in *Proceedings of the 2009 IEEE International Conference on Web Services (ICWS'09)*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 287–294.

[41] Y. Zhang, M. Panahi, and K.-J. Lin, "Service Process Composition with QoS and Monitoring Agent Cost Parameters," in *Proceedings of the 2008 10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 311–316.

Philipp Leitner has a BSc and MSc in business informatics from Vienna University of Technology. He is currently a PhD candidate and university assistant at the Distributed Systems Group at the same university. Philipp's research is focused on middleware for distributed systems, especially for SOAP-based and RESTful Web services.

Waldemar Hummer holds a BSc (Univ. of Innsbruck) and MSc (Vienna Univ. of Technology) in Computer Science, and a BSc in Business Administration (WU Vienna). Currently he is a university assistant and PhD candidate at the Distributed Systems Group, Vienna Univ. of Technology. His main topics of interest are in the area of self-optimizing service-based systems, Web service composition and Web data aggregation.

Schahram Dustdar is Full Professor of Computer Science with a focus on Internet Technologies heading the Distributed Systems Group, Vienna University of Technology (TU Wien). He is also Honorary Professor of Information Systems at the Department of Computing Science at the University of Groningen (RuG), The Netherlands.