Int J Softw Tools Technol Transfer (2011) 13:167–179
DOI 10.1007/s10009-010-0181-7

WSE 2009

Automated maintenance of service compositions with SLA violation detection and dynamic binding

Adina Mosincat · Walter Binder

Published online: 6 November 2010

Abstract Web service compositions need to adapt to changes in their constituent web services, in order to maintain functionality and performance. Therefore, service compositions must be able to detect web service failure and performance degradation resulting in the violation of service-level agreements. Automated diagnosis and repair are equally important. However, existing standards and languages for service compositions, such as BPEL, lack constructs for web service monitoring and runtime adaptability, which are prerequisites for diagnosis and repair. We present a solution for transparent runtime monitoring, as well as automated performance degradation detection, diagnosis, and repair for service compositions expressed as BPEL processes. Our solution uses lightweight monitoring techniques, supports customizable diagnosis and repair strategies, and is compatible with any standards-compliant BPEL engine.

Keywords Dynamic adaptability · SLA violation detection · Statistical tests · BPEL processes · Performance monitoring

1 Introduction

Service compositions, often expressed as BPEL [8] processes,[1] allow reusing existing functionalities to build new applications. The quality-of-service (QoS) parameters of the process depend on the QoS provided by the composing web services. In this article, we focus on the performance-related QoS, such as maximum response time for the execution of a composition [29]. The web is a dynamic environment in which services change and service performance can alter. This dynamic environment makes the tasks of adaptive maintenance [18,37] very difficult for service compositions, because, in the absence of runtime adaptability, the developer has to continuously be aware of the changes and take corrective actions, such as replacing some of the constituent services. Indeed, the degradation of service performance might render the service composition temporarily unavailable, requiring modification of the composition in order to "keep the software product usable in a changed or changing environment" [37]. To overcome this problem, the tasks of composition maintenance need to be automated.[2] Thus, in order to fulfill its service-level agreement (SLA [30,35]), a process must be able to notice and dynamically adapt to changes [32]. BPEL does not provide any means to survey the processes' SLA fulfillment or the performance of its constituent web services. Monitoring of bound services is not supported by the BPEL specification, being left to the BPEL engine implementor which may or may not provide monitoring functionality in a proprietary way. Moreover, BPEL offers limited runtime adaptability, such that in order to replace a service used in a process, the developer needs to manually update the process and redeploy it.

We address these issues in our framework for automated dynamic update of BPEL processes, which we name ADULA. ADULA ensures maintenance of process performance through automated detection of service failures and SLA violations, diagnosis, and repair. Our framework

A. Mosincat (✉) · W. Binder
Faculty of Informatics, University of Lugano, Lugano, Switzerland
e-mail: adina.diana.mosincat@usi.ch

W. Binder
e-mail: walter.binder@usi.ch

[1] In this article we refer to a web service composition as a *process*.

[2] By automated maintenance we refer strictly to the tasks of noticing changes and replacing constituent services without requiring modification of the service composition code.

allows for monitoring of process and service performance using lightweight sampling techniques. The failure and performance degradation detection and repair strategies are customizable, leveraging statistic methods and our monitoring mechanism. ADULA supports transparent dynamic binding of services, failure recovery, and is compatible with any BPEL engine.

A BPEL process is deployed to and executed in a BPEL engine, which creates a *process instance* when one of the receive activities (a start activity) in the process is triggered, and ends the instance after completion of the corresponding reply activity. Our solution binds services used in a process instance at runtime, selecting the services to use from a group of alternative, substitutable services.

When updating a process by changing the service bindings, one issue to take into consideration is the motivation behind the choice of services. Besides technical aspects, the choice of services a developer makes is based on business rules and personal knowledge, which are criteria usually not known to an automated binding system. Therefore, we consider that the best approach is either to use a service matcher that takes business rules into consideration, or to preserve the choice of services made by the developer and to change the process bindings only when a service performs badly and the process does not meet its SLA. Moreover, for service management purposes [31], the information about the services used in the process must be available and the process evolution[3] must be traceable and visible. ADULA preserves the developer's choice of services favoring the binding of the services used initially at the process development time, and keeps a full history of the process evolution. Based on the statistics provided by ADULA, the developer can later change the choice of services.

Let us consider a loan-approval process, which is offered as a bank service through which the bank customers can request a sum of money as a loan. Usually, the customer has to pay a fee for the processing of a loan request, and he is given certain guarantees, such as the maximum time to answer the loan request. If the loan request is not processed in the guaranteed time, the bank not only looses credibility and thus customers, but might also be liable for breach of contract. Thus, it is necessary to notice a performance degradation as soon as possible and remedy it. ADULA automatically detects process performance degradation and takes corrective actions to recover from such a situation. The loan-approval process uses two services: one to assess the risk of the loan, and one to approve the loan based on the risk assessment. The bank can have a contract with a risk assessor service which it trusts and which charges a special price. Therefore, the developer chooses the risk assessor service according to the bank's business rules. If the risk assessor service suffers a downtime, ADULA replaces it ensuring the bank's service to its customers is not affected. As soon as the risk assessor service is repaired, ADULA switches to the default choice of services, thus assuring the bank profits from its contract with the risk assessor service.

This article addresses the following three issues. First, we introduce a framework for automated dynamic update of process service bindings through performance degradation (SLA violation) detection, diagnosis, and repair. The framework leverages a lightweight runtime process monitoring mechanism and allows for customizable detection and repair strategies. Second, we present an approach to automated process evolution that allows for traceability and visibility, providing process and service statistics. Third, we evaluate the framework in different settings, showing the benefits of automated process binding update as well as the cost in terms of overhead.

This article extends the authors' prior work presented in [26] with the following original contributions: (1) it explores the effectiveness of null-hypothesis statistical significance tests [34] for detecting SLA violations, (2) it refines the selection strategy for replacing services that have caused SLA violations, and, (3) it extends the maintenance capabilities of the framework by providing the developer with comparative information on the performance of different process bindings.

This article is structured as follows. Section 2 introduces the concepts used in this article and presents the architecture of our infrastructure. In Sect. 3, we describe the functionalities through which automated maintenance of processes is achieved. We evaluate our approach in Sect. 4 and discuss related work in Sect. 5. Section 6 concludes this article.

2 System description

This section describes the overall architecture of our framework and details the components that play an important part in the automated process maintenance.

2.1 Terminology

The following terminology will be used throughout the article:

- *Process Name* : Unique identifier of a process in the system.
- *Partner Link* : Unique identifier of a service binding in a process (represents the partner link name in a process).
- *Endpoint Reference* (ER): Unique identifier of a service in the system (service endpoint reference [40]).

[3] We consider that the process evolves when composing services are replaced, and thus the process uses different service bindings. We refer to the usage of different service bindings in the process as *process evolution.*

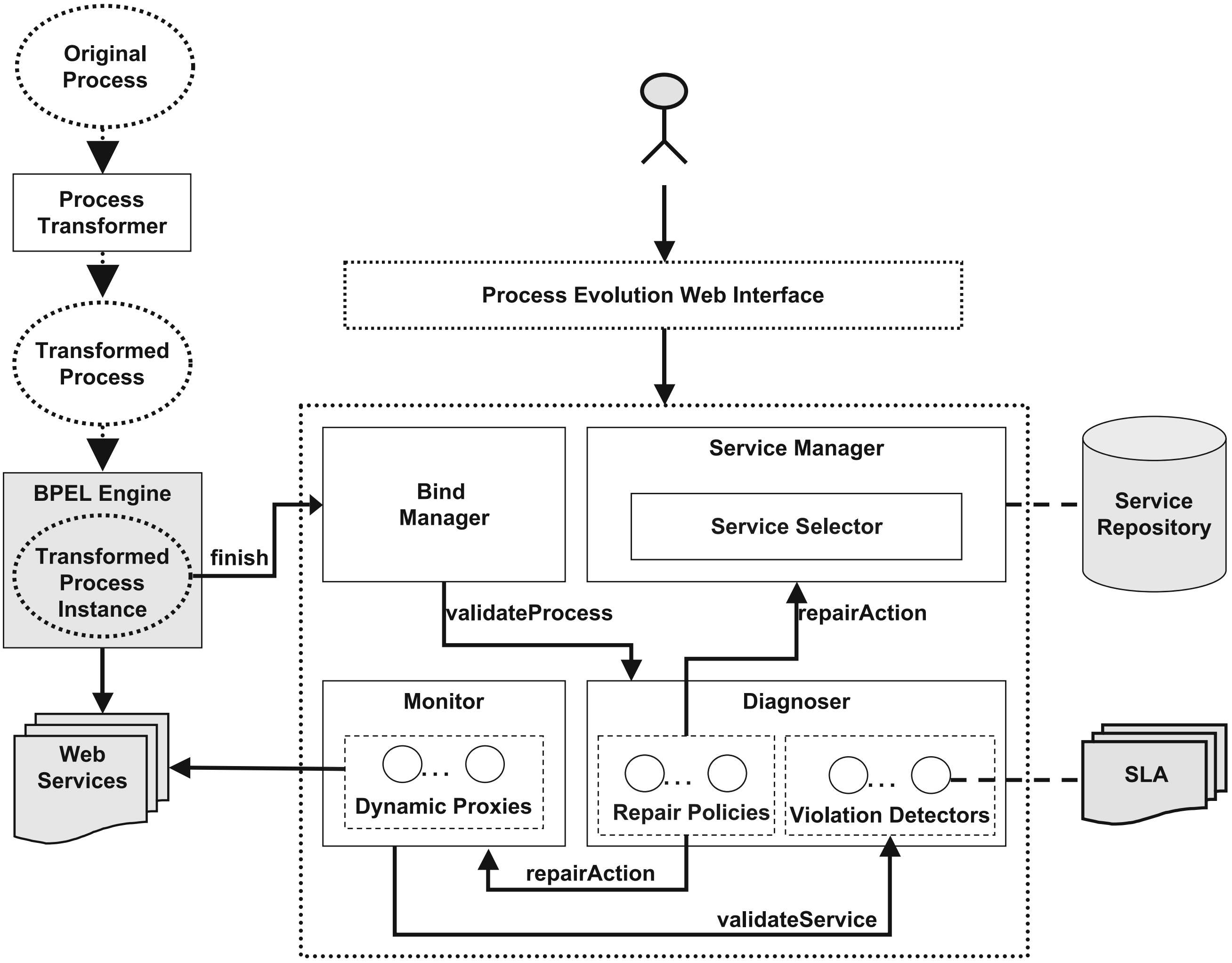

Fig. 1 ADULA architecture

– *Bindings*: A set of ⟨*Partner Link*, **ER**⟩ pairs, specifying the services bound in a process instance. Bindings must not include multiple pairs with the same partner link.

2.2 Process transformation and failure recovery

The ADULA framework is based on our former system introduced in [25], which transparently supports dynamic binding of services and failure recovery for BPEL processes. In this section, we briefly summarize the features of our former system that are used by ADULA, as well as the improvements of these features in ADULA. This section presents important background for understanding ADULA.

Dynamic binding in ADULA is achieved by automatically (and transparently for the developer) transforming processes before deployment in the BPEL engine. Upon transformation, the new process is registered in the ADULA system, which assigns a unique process name. Every partner link used in the original process is changed into a dynamic partner link and it is assigned the endpoint reference selected by ADULA at runtime. In contrast to our former system, ADULA preserves the original process bindings; we call them the *default bindings*. The default bindings are used for each new process instance, unless some services are excluded from selection.

The transformed process interacts with ADULA in several ways. Upon process instance start, the transformed process requests bindings from ADULA.[4] The transformed process also notifies ADULA on completion of a process instance. Upon failure of a stateless service, ADULA is notified and an alternative binding is returned in order to retry the invocation with a different service. Upon failure of a stateful service, ADULA is notified and the whole process instance is restarted, acquiring new bindings in the beginning. Process

[4] While in our prior work [25] bindings were requested before each service invocation, ADULA obtains the bindings for all partner links occurring in the process at once. Hence, ADULA reduces the number of messages exchanged with the transformed process, reducing overhead.

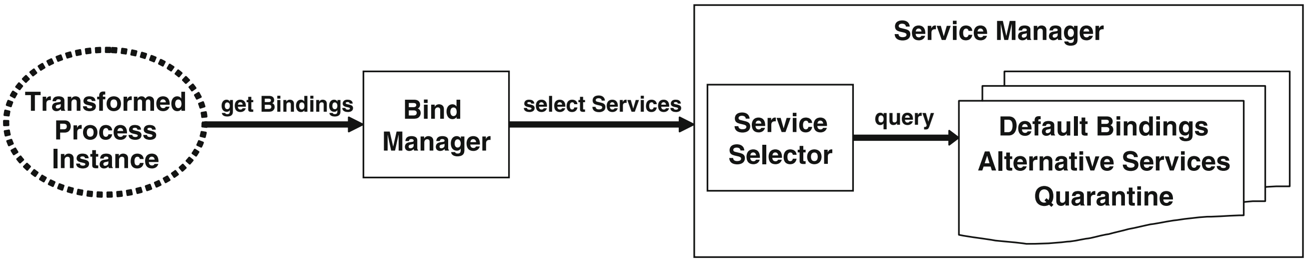

Fig. 2 Dynamic binding in ADULA

restart is achieved by a wrapping technique. For details, we refer to [25].

2.3 ADULA

Figure 1 presents the overall architecture of the ADULA framework. We will describe only the components that are important for automated process maintenance.

The *Bind Manager* (BM) constitutes ADULA's interface to the process. The *Service Manager* is responsible for selecting the services to bind for every process instance and managing the service quarantine. All services that fail or perform badly are put in quarantine. All services that are in quarantine are excluded from selection until the quarantining period has elapsed. The quarantining period is a configurable parameter.

The *Diagnoser* detects SLA violations and takes corrective actions. The main responsibility of the *Diagnoser* is to manage *Violation Detectors* and *Repair Policies*. A *Violation Detector* performs tests to detect if a certain constraint has been violated. The constraint can be a guarantee specified in the process SLA. A *Repair Policy* specifies the actions to be taken in case of violation. *Violation Detectors* and *Repair Policies* are further explained in Sect. 3.2.

The *Monitor* provides statistics on service performance and intervenes in case of a service timeout (the service exceeds a given response time limit). It has the following responsibilities:

- Creation of *Dynamic Proxies*. A *Dynamic Proxy* can measure the response time of any service, if the service is invoked through the *Dynamic Proxy*.
- Collection of measurements from the *Dynamic Proxies*.
- Aggregation of measurements and provision of service response time statistics to other parts of the system.
- Management of timeout constraints.
- Request of service quarantine in case of service timeout.

The *Process Evolution Web Interface* constitutes the user interface. It provides statistics on the process evolution as well as on service performance.

In case of normal execution, the process instance has two interactions with ADULA: first, when a process instance is created, it requests the service bindings from the *Bind Manager*. Second, on completion, the process instance notifies the *Bind Manager*, triggering the start of the tests implemented by the corresponding *Violation Detector* and storing information on process instance execution.

3 Automated maintenance

There are five important functionalities through which ADULA achieves automated process maintenance: service selection, violation detection, diagnosis, repair, and monitoring.

3.1 Service selection

Figure 2 shows the selection of service bindings on request from the process instance. The *Default Bindings* are the bindings created at transformation time which bind the services used in the original process; we call them the *default services*. The *Alternative Services* are all available replacement services, and the *Quarantine* are the quarantined services. The *Alternative Services* provide a group of replacement services for every service, by using a substitutability relation. The substitutability relation is reflexive and transitive, but not necessarily symmetric. Thus, every default service can be substituted by the services provided by the *Alternative Services*, but it is not mandatory that the services are functionally exactly equivalent. For instance, a new, backward-compatible version of a service can offer an extended interface, but it can still be used as a replacement for the old service. The old service, on the other hand, can only be replaced by the new one, but it cannot be a replacement service for the new service.

The pseudo-code in Fig. 3 shows the service selection logic. `SelectedServices` is the set of ⟨*Partner Link, ER*⟩ pairs representing the selected services to be bound by the process instance.

The *Service Selector* favors the selection of the default services used in the original process. To this end, the *Service Selector* reads the process default bindings that were stored at process transformation time. Default services that are in quarantine are replaced and new bindings are created for the

```
selectedServices ← ∅;
defaultBindings
←getDefaultBindings(processName);
forall <partnerLink, er> ∈ defaultBindings do
begin
    if er ∈ Quarantine then
        newER ← selectReplacementService(er);
        selectedServices ←
         selectedServices ∪ {<partnerLink,
        newER>};
    else
        selectedServices ←
         selectedServices ∪ {<partnerLink, er>};
    end
end
```

Fig. 3 Service selection: creating the set of service bindings for new instances of process *processName*

process. The replacement service is selected from the available replacement services which are not in quarantine. If no replacement service is available, the default service is used and taken out of the quarantine.

When selecting a replacement service, the *Service Selector* takes into consideration monitoring statistics and favors the service with the best performance. Let's consider there are n replacement services for a service. The *Service Selector* ranks the services according to the formula:

$$\text{ranking}_i = 1/\text{avg}(\text{rt}_i), \quad 1 \leq i \leq n.$$

The function $\text{avg}(\text{rt}_i)$ represents the average monitored response time of the service i. If there are no statistics for the service, i.e., the service has never been monitored, and the SLA of the service is provided, the response time specified in the service SLA is used. The *Service Selector* chooses the service i with $\text{ranking}_i \geq \text{ranking}_j, \forall i, j, 1 \leq i, j \leq n$.

3.2 Violation detection, diagnosis, and repair

The labeled arrows in Fig. 1 illustrate the interactions between the components of the ADULA framework when the violation detection action is triggered.

The process instance notifies the *Bind Manager* on completion (`finish`). The *Bind Manager* stores the execution time of the process instance and triggers the violation detection action (`validateProcess`). The violation detection performed for a process instance is an asynchronous action, and the process instance does not wait for the violation detection to finish. The *Diagnoser* uses the *Violation Detector* corresponding to the process name. If a violation is detected, the *Repair Policy* corresponding to the process name first reaches a diagnosis based on available information, such as monitoring statistics, and then dictates the action to be taken (`repairAction`). Depending on the repair action, the *Monitor* can be requested to perform violation detection actions on individual services (`validateService`).

Determining if a violation which has occurred is done through tests implemented by *Violation Detectors*. Typically, *Violation Detectors* test the fulfillment of SLA guarantees using statistic methods, such as statistical hypothesis testing [34], or Bayesian inference [7].

Below we give two examples of possible *Violation Detectors*.

For SLAs that require a process response time below a given threshold, a simple *Violation Detector* reports an SLA violation whenever the process response time exceeds the given threshold. This *Violation Detector* does not depend on any measurements of previous process instances.

As another example, for SLAs that require the average (arithmetic mean) process response time not to exceed a given threshold, we can apply a null-hypothesis statistical significance test [34]. The null-hypothesis is that the SLA is not violated, i.e., we assume that the average process response time is smaller or equal to the threshold specified in the SLA. The statistical test tells us whether the samples, i.e., the measured response times for a given process using specific bindings, are unlikely to have occurred by chance given the truth of the null-hypothesis. In that case, the null-hypothesis is rejected and an SLA violation is reported. The *Violation Detector* chooses the significance level α, the probability of committing a type-I error, i.e., the probability of rejecting the null-hypothesis when it is true. Since the impact of wrongly quarantining well-behaving services is limited in time (until the quarantining period expires), *Violation Detectors* may choose unusually high α values (e.g., $\alpha = 0.25$). In this example, the well-known one-sample Student t test can be applied. *Violation Detectors* using null-hypothesis statistical significance tests need access to some history of previous samples (i.e., process response times for process service bindings); ADULA provides that information. As a drawback of this technique, a rather high number of samples is needed; e.g., for the one-sample Student t test and unknown distribution and variance of the samples, more than 30 samples are required. Consequently, at the startup of the system, SLA violations may be detected with some delay.

We verify that the response times (the samples) follow a normal distribution using the Anderson–Darling normality test [2]. For the Anderson–Darling normality test, the null-hypothesis is that the samples follow a normal distribution. We choose the same α value that we use for the t-test. The Anderson–Darling test rejects the null-hypothesis if the computed test statistic is greater than the critical value computed for the normal distribution and the chosen α value. If the Anderson–Darling test is positive, i.e., the distribution is normal, we use the Student t test to determine if a violation has occurred. In case the Anderson–Darling test is negative, i.e., the distribution is not normal, we use the Wilcoxon signed-rank test [39] for one sample.

```
suspectServices←
   {er | ∃<partnerLink, er> ∈ violatingBindings};
validatedServices←
   getValidatedServices(processName,
                                violatingBindings);
forall er ∈ suspectServices - validatedServices do
begin
    quarantine(er);
end
forall er ∈ suspectServices ∩ validatedServices do
begin
    monitor(er);
    setTimeout(er, true);
    setViolationDetector(er, serviceVD);
end
```

Fig. 4 Repair Policy example pseudo-code. *violatingBindings* represents the bindings used by the process instance causing the violation

The Wilcoxon signed-rank test is a non-parametric test that can be used as an alternative to the Student t test when the distribution cannot be assumed to be normal. The Wilcoxon test computes the difference between the response time value for each sample and the average response time guaranteed by the process SLA. It then ranks the differences taking into consideration the sign (positive or negative) giving to each difference a value according to its rank. The test statistic is represented by the minimum value between the sum of positive values and the sum of negative values. The test statistic is verified against the critical value for the chosen α and the null-hypothesis is rejected if the test statistic value is less than the critical value.

When a violation is detected, the *Diagnoser* instantiates the *Repair Policy* mapped to the process name. The responsibility of the *Repair Policy* is to diagnose the cause of the violation and dictate repair actions. Typically, a repair action affects the service selection by quarantining one or more services and thus enforcing the services' replacement in subsequent process instances. A *Repair Policy* can combine diagnosis and repair actions of different complexity. For e.g., a simple *Repair Policy* can diagnose all services in the violating bindings as causing the violation and quarantine all of them.

Figure 4 shows the pseudo-code of a more complex *Repair Policy* which leverages the monitoring capabilities of ADULA. The policy narrows the set of services that are most likely to have caused the violation (`suspectServices`), taking into account the process instances that have recently used the services and that have good performance. Function `getValidatedServices` makes use of the process statistics stored by ADULA to check if any of the services used in the violating bindings have been recently used in other bindings which have completed with good performance. The bindings taken into consideration must belong to different processes than the process causing the violation (identified by `processName`). The function returns the set of found services, `validatedServices`.

The *Repair Policy* puts all `suspectServices` which have not been validated, and thus are more likely to have caused the violation, in quarantine (`quarantine`). Then, the policy marks for monitoring the services that were used in the process instance, but have been validated, and dictates the following repair actions for these services:

1. Services are monitored and put in quarantine if they cause timeout (`setTimeout`).
2. If there is an SLA for each individual service, services are checked for SLA violations using the `serviceVD`, which is a detector chosen by the *Repair Policy* from existing detectors (`setViolationDetector`). The services causing a violation are put in quarantine.

Figure 5 shows the dynamics of the system for service selection. We consider an example process using a loan service and an approver service. In the first step, the *Bind Manager* returns the default bindings for the process instance to use. In the second step, the *Violation Detector* detects a violation of the process response time (`violation (processName)`), and the *Repair Policy* quarantines the services used by the current bindings. When a subsequent request for bindings arrives, in the third step, the *Bind Manager* asks the *Service Selector* to provide alternative services for the services in quarantine.

3.3 Monitoring

The *Monitor* plays two roles: (1) it provides service performance statistics and (2) it interrupts a service on timeout.

To gather data for service performance statistics, we use sampling-based monitoring that allows for effective monitoring, while reducing the cost in terms of overhead. Every service is monitored with a given configurable probability. The monitoring probability for services marked for monitoring by a *Repair Policy* is 100%. *Dynamic Proxies* inside the monitor measure service response time. For every service invoked through a *Dynamic Proxy*, the *Monitor* computes an average response time value. This value can be used to detect performance degradation in a service, in case the service SLA is not available or does not specify the normal service response time.

Monitoring can be used as repair action in two ways. First, using the timeout mechanism through which a service that takes too long to respond is interrupted (`setTimeout` in Fig. 4), causing a service invocation failure in the process instance. Subsequently, the process instance will request a replacement service using ADULA's failure recovery feature [25]. Second, the *Repair Policy* can dictate a *Violation Detector* to be used to validate the service response time (`setViolationDetector` in Fig. 4). In order to determine a service timeout, the *Monitor* makes use of the service

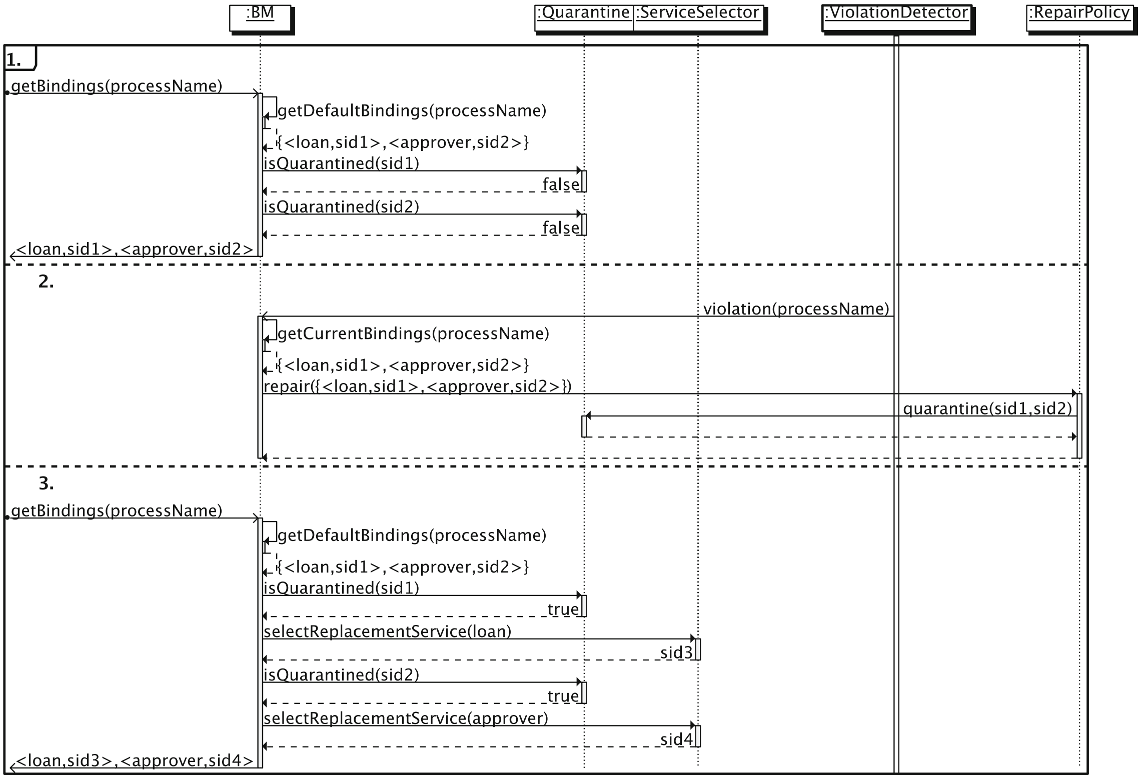
:BM
:Quarantine
:ServiceSelector
:ViolationDetector
:RepairPolicy
1.
getBindings(processName)
getDefaultBindings(processName)
{<loan,sid1>,<approver,sid2>}
isQuarantined(sid1)
false
isQuarantined(sid2)
false
<loan,sid1>,<approver,sid2>
2.
violation(processName)
getCurrentBindings(processName)
{<loan,sid1>,<approver,sid2>}
repair({<loan,sid1>,<approver,sid2>})
quarantine(sid1,sid2)
3.
getBindings(processName)
getDefaultBindings(processName)
{<loan,sid1>,<approver,sid2>}
isQuarantined(sid1)
true
selectReplacementService(loan)
sid3
isQuarantined(sid2)
true
selectReplacementService(approver)
sid4
<loan,sid3>,<approver,sid4>

Fig. 5 System dynamics for service selection

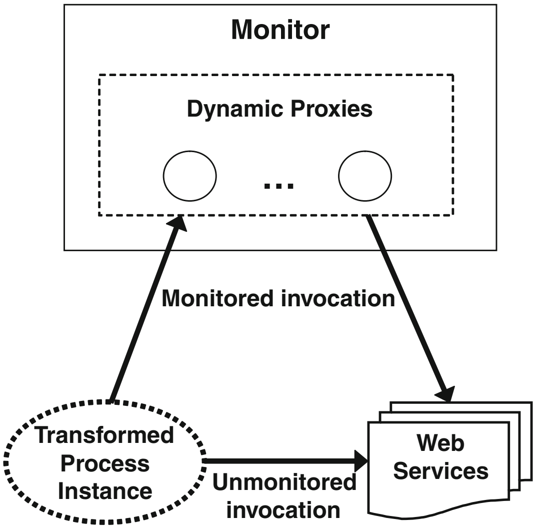

Fig. 6 Monitored and unmonitored service invocation

statistics. A service that has been interrupted because of timeout or has violated the constraint tested by the *Violation Detector* is put in quarantine.

Figure 6 shows the monitored and unmonitored service invocations from a process instance. In case of a monitored invocation, the service binding returned by the *Bind Manager* to the process instance upon process instance start refers to a *Dynamic Proxy*, while in case of an unmonitored invocation, the service binding refers to a service. The service binding represents a service endpoint reference. If a selected service is to be monitored, the *Bind Manager* embeds the endpoint reference of the selected service within the endpoint reference of the *Dynamic Proxy* as *ReferenceParameters* [40]. The *Bind Manager* then returns the endpoint reference of that *Dynamic Proxy* to the process instance.

When a *Dynamic Proxy* receives a request, it first inspects the header of the SOAP [36] message, building the endpoint reference of the service to invoke from the *ReferenceParameters*. Then the *Dynamic Proxy* invokes the service while measuring its response time.

3.4 History of process evolution

There are two important issues concerning process maintenance when dynamically replacing service bindings: first, process performance, which is the main reason driving the replacement; and second, traceability of replacements, and thus of process evolution. The knowledge gained during the execution of process instances, respectively, through monitoring, is useful for the future development of processes, as well as for business metrics (e.g., computing process rentability, service reliability). It is important that information is stored and provided on request. ADULA preserves process and service statistics that are available through the *Process Evolution Web Interface* to users.

The process statistics represent the history of the process evolution determined by SLA violations and obtained through automated process maintenance. Service statistics are obtained through monitoring and represent information on service performance.

The statistics preserved in the system are:

– All bindings that have been used for a process. The default bindings and the bindings that are currently in use are highlighted. The history of the process is presented as a list of bindings and the timestamps when the updates took place.
– For each bindings, the total number of process instances that have used the bindings.
– For each bindings, the minimum, maximum, average execution time of instances using the bindings, as well as the standard deviation.
– Violations that have occurred. For each violation, the bindings that have caused the violation and the time when the violation occurred are shown, as well as the repair actions taken.
– Quarantine: services in quarantine and for each service the time left until exiting the quarantine.
– For each service: the number of process instances that have used the service; the violations in which the service has been part of the diagnosis; the number of times the service was put in quarantine and total time of quarantining; the number of timeouts the service has caused; service performance statistics.

The statistics provide an overview on service performance and reliability, and on process usage. Thus, the statistics allow the developer to track the evolution of the process, as well as help selecting the services to use in future processes.

ADULA provides a performance comparison test that can be used to determine if there are bindings that perform on average better than the default bindings. The developer can tune the process performance by changing the default bindings using the performance comparison test provided by ADULA.

The comparison test makes use of the Mann–Whitney nonparametrical statistical test [21] to determine the chances of obtaining better performance using the default bindings versus alternative bindings. The Mann–Whitney test is the Wilcoxon signed-rank test variant for two independent samples. The null-hypothesis is that the performance of the process instances using the default bindings is not worse than the performance of the process instances using the alternative bindings. The alternative hypothesis is that the performance of the process instances using the alternative bindings exceeds the performance of the process instances using the default bindings. As samples, the test uses the monitored response times of the process instances. The alternative bindings are the ones used as replacement when one of the default services has been put in quarantine.

4 Evaluation

In this section, we evaluate the ADULA framework. First, we explore the response time of processes using ADULA compared to processes that statically bind services when the performance of composing services degrades. We consider two different kinds of SLA guarantees, which are validated with different techniques, and explore the effectiveness of using statistical tests to detect violations. Second, we measure the overhead introduced by ADULA for different service response times when no SLA is violated.

We present the common settings we use in our evaluation, the implementation details, and the evaluation results for violation detection and overhead.

4.1 Common settings

Our evaluation is based on two processes, one of them extending the loan-approval sample process (included in the BPEL specification [8]) and interacting with three different services. The other process is interacting with only one service. The loan-approval process takes as input a sum of money and a currency and outputs the response on approving the loan. It interacts with three services: the convertor (to convert the sum of money from the given currency in euro), the loan assessor (to assess the risk of the loan), and the loan approver (to decide if the loan is approved or not). The second process just invokes a loan assessor service. Both processes use the same default loan assessor service, which we will call *assessor1*. There are three different available substituting services for each service functionality.

We developed a testbed which models web service performance with discrete time Markov chains [23]. The testbed includes web services, client workload generators, as well as performance measurement tools.

For measuring response time, we use two different settings in which the *assessor1* service follows a service model with five states, as depicted in Fig. 7. All other services are in the fast state (500 ms) and do not change state. In the first setting, the *assessor1* service changes state on every time slot[5] until it reaches the slowest state (2,500 ms) and then remains in the same state; in the second setting the *assessor1* service changes state on every time slot.

[5] A *time slot* is the moment in time when a decision is made randomly based on the current state and the transition probability to change or to keep state.

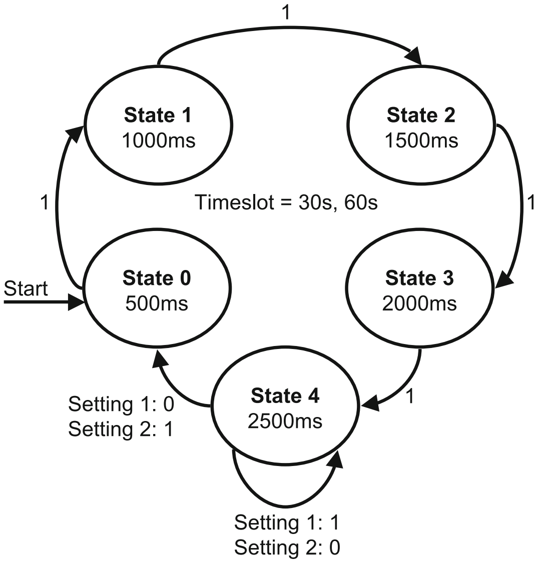

Fig. 7 Service performance model: discrete Markov chains with 5 states

Our implementation uses Java 5, Apache Axis 1.4, and BPEL 2.0; as BPEL engine we use ActiveBPEL 4 [1]. Both ADULA and the BPEL engine are deployed in an Apache Tomcat 4.1.24 installation. The violation detectors using statistical tests are implemented with the Apache Commons Mathematics 2.0 library. Our measurement machine is an Intel Core 2 Duo (2.4 GHz, 2GB RAM) running Mac OS X v10.4. All measurements were repeated 15 times and we report the median of these measurements.

4.2 Violation detection

4.2.1 SLA limits the maximum process response time

In the following evaluation (Figs. 8, 9), we assume the loan-approval process SLA guarantees that the response time does not exceed 3,000 ms, and the simple assessor process SLA guarantees that the response time does not exceed 2,000 ms. For Figs. 8 and 9 we use a simple *ViolationDetector* that tests whether the response time of the process is below a given threshold. If a violation occurs, all services in the violating process instance are quarantined. The quarantining period is 30 s. The time slot for our service model is 30 s.

Figure 8 shows the response time of the transformed[6] loan-approval process compared to the original loan-approval process when the performance of the *assessor1* service degrades and remains poor (setting 1). Once ADULA detects an SLA violation, it replaces the *assessor1* service allowing the transformed process to recover, while the original process continues violating the SLA because of the degradation of the *assessor1* service. The peaks in the transformed process response time correspond to the elapsing of quarantining time for service *assessor1*. Because *assessor1* is the default service, ADULA tries to use it as soon as it exits the quarantine. As the performance of *assessor1* is still causing a process SLA violation, the service is quarantined again. Thus, ADULA ensures that the process will return to the default service bindings after the service causing the violation has been repaired, ensuring the control over the process evolution and the preservation of the developer's original choices.

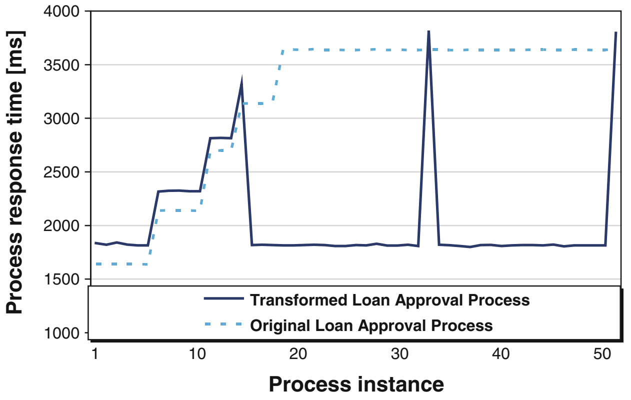

Fig. 8 Response time of the original and transformed loan-approval processes. SLA guarantees a maximum process response time ≤3,000 ms. Response time of *assessor1* degrades from 500 to 2,500 ms and remains slow (setting 1 in Fig. 7). Time slot: 30 s

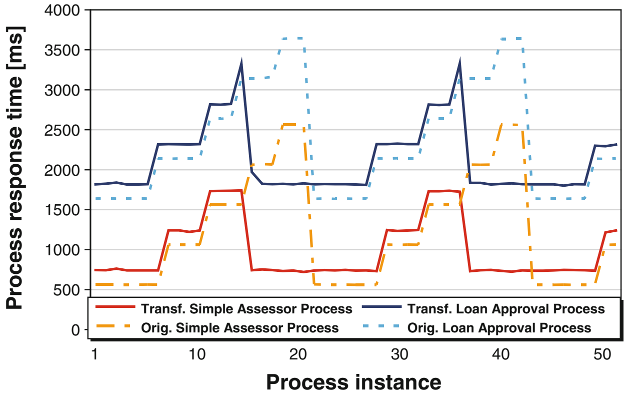

Fig. 9 Response time of the original and transformed loan-approval and simple assessor processes. SLA guarantees a maximum process response time ≤3,000 ms for the loan-approval process, respectively ≤2,000 ms for the simple assessor process. *assessor1* repeatedly changes state between 500 and 2,500 ms (setting 2 in Fig. 7). Time slot: 30 s

Figure 9 shows the response time of the transformed processes compared to the original processes when the performance of *assessor1* fluctuates (setting 2). Although the performance degradation of *assessor1* does not yet cause an SLA violation for the simple assessor process, when *assessor1* is put into quarantine (because it causes the violation of the loan-approval process' SLA), it is replaced in all

[6] We refer to the process that uses ADULA as the *transformed process*.

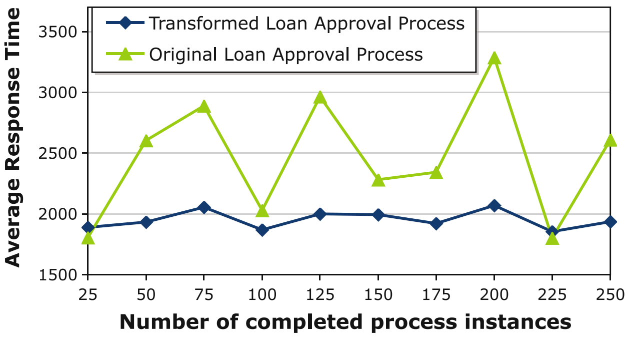

Fig. 10 Average response time (calculated for each 25 process instances) of the original and the transformed loan-approval processes. SLA guaranteeing an average process response time $\leq$2,000 ms. Violation detector with Student *t* test, $\alpha = 0.0005$. *assessor1* repeatedly changes state between 500 and 2,500 ms (setting 2 in Fig. 7). Time slot: 60 s

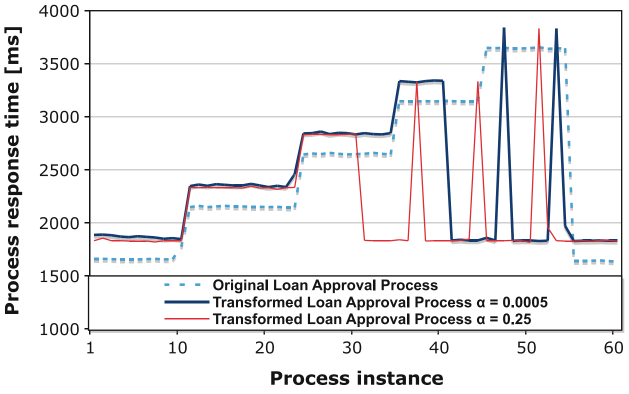

Fig. 11 Response time of the original and the transformed loan-approval processes. SLA guaranteeing an average process response time $\leq$2,000 ms. Violation detector with Student *t* test, $\alpha = 0.0005$ and $\alpha = 0.25$. *assessor1* repeatedly changes state between 500 and 2,500 ms (setting 2 in Fig. 7). Time slot: 60 s

processes that use it, hence also in the instances of the simple assessor process. Thus, ADULA prevents an SLA violation in other processes, the simple assessor process in our case, which would be caused by the service performance degradation. When service performance is constant, the increase in response time of the transformed process compared to the original process is caused by the overhead introduced by the interactions with ADULA and monitoring.

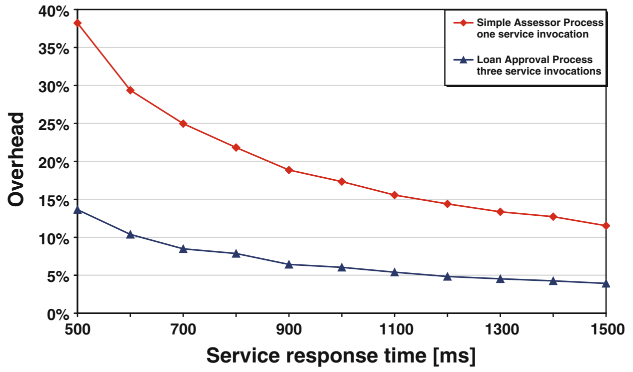

Fig. 12 Overhead for processes with one and three service invocations for service response time between 500 and 1,500 ms

4.2.2 *SLA guarantees an average process response time*

In the following evaluation (Figs. 10, 11), we assume the loan-approval process SLA guarantees that the average process response time does not exceed 2,000 ms. The *ViolationDetector* relies on the Student *t* test; we consider both a very low α-value ($alpha = 0.0005$) and a very high α-value ($alpha = 0.25$).[7] If an SLA violation is detected, all services in the violating process instance are quarantined. The quarantining period is always 30 s. In the service performance model the time slot is 60 s.

Figure 10 shows the average response time for the original and the transformed loan-approval processes when using the Student *t* test for SLA violation detection with $\alpha = 0.0005$; the average response time is computed after each 25 process instances. In this scenario, we use a service performance model according to setting 2 in Fig. 7. The overall average response time is 2,461 ms for the original process, respectively, 1,952 ms for the transformed process. Despite the very low α-value, the transformed process reacts to SLA violations in a timely way, avoiding any long-term performance degradation.

Figure 11 illustrates the effectiveness of the *ViolationDetector* using the Student *t* test in terms of how quickly violations are detected. As expected, with $\alpha = 0.0005$, a larger number of samples with long response time are needed until a violation is detected (i.e., until the null-hypothesis can be rejected), whereas with $\alpha = 0.25$ violations are detected very quickly. However, a high α-value increases the risk of false positives, that is, spurious violations may be detected, resulting in unnecessarily frequent quarantining of services. Nonetheless, if we assume that there is a sufficient number of replacement services, the negative impact of false positives is mitigated. Hence, for this particular use of the Student *t* test, we consider a relatively high α-value appropriate.

4.3 Overhead

We present the overhead of the transformed processes relative to the execution time of the original processes. We consider the relative overhead for varying service response time between 500 ms and 1,500 ms. Figure 12 shows the overhead

[7] The α-value is the probability of committing a type-I error; for details, see Sect. 3.2.

introduced by ADULA. The overhead is mainly caused by the two interactions with the *Bind Manager*: requesting the service bindings at process instance start, respectively, notifying process instance completion. The percentage of overhead decreases with the increase of process complexity and process response time; for a process with three service invocations and service response time of more than 600 ms, the overhead introduced by ADULA is below 10%.

In summary, our evaluation confirms that ADULA effectively and efficiently detects, diagnoses, and repairs processes, keeping the process evolution traceable and visible. The cost in terms of overhead introduced when there is no SLA violation is moderate, and is compensated by the gain obtained through service replacement in case of performance degradation. Moreover, ADULA prevents subsequent SLA violations and improves performance of processes once the service causing the performance degradation is detected.

5 Related work

There is a number of approaches that deal with QoS awareness of service compositions. Some of these solutions analyze and compose the QoS of individual services to obtain the QoS of the process statically at design time [22,41]. In contrast, our solution is dynamic and we consider failures and changes in the environment. Other solutions continuously recompute the estimated QoS [9,10,42]. The re-computations can be expensive and thus affect process performance. Our solution reduces overhead by using sampling-based monitoring and asynchronous violation detection. Solutions that use regression models and machine learning techniques to compute and predict SLO values [11,12,20,38] may yield higher accuracy of prediction in the presence of unknown factors that affect the QoS parameters of the composition, but often incur higher runtime overhead. Our solution relies on statistical tests that offer sufficient accuracy with less resource usage when monitoring information is available.

Our approach to automated process update is based on dynamic binding of services. Different solutions deal with dynamic binding of services in BPEL processes [4–6,15,16,24]. Some of these solutions also use monitoring to replace a failing service [15,16,24], others [6] use different analysis techniques to detect SLA violations, but none of them take into consideration the original services used in the process, and the history of process evolution. Most of these solutions rely on a modified BPEL engine [4–6,24]. Our approach is transparent to the user and to the BPEL engine, ensures the process fulfills its SLA, provides process statistics, and controls the process evolution in such a way to preserve the choices done at development time.

An interesting solution close to our approach is presented by Canfora et al. in [9,10]. The solution introduces a tool to define new domain specific QoS parameters, a language, and an interpreter to define QoS aggregation formulae to compute the QoS of a composition based on the QoS of the composing services. The system integrator has to specify aggregation formulae for QoS and define an abstract composition. Our approach does not require defining a special composition, but automatically transforms existing processes. In the framework presented by Canfora et al., all service invocations used in the abstract composition are linked to proxies which bind services at runtime. For every abstract composition, the system determines the optimal service composition computing different genomes by making use of genetic algorithms. The estimation of the composition's QoS is re-computed at every step of the composition execution and different services can be re-bound if the estimated QoS does not meet the required one. In our approach, information about service status is shared between processes allowing to prevent SLA violations without adding the cost of QoS re-computation.

AgFlow [42] is a middleware platform that enables the quality-driven composition of web services. AgFlow computes optimal plans for the execution of the composition using integer programming. The middleware makes use of two service selection approaches, one based on local optimization that does not take into account the overall QoS, and one based on global planning that considers the composition QoS constraints rather then the QoS of individual tasks. The cost of continuously re-computing QoS can be high and cause a penalty on the process performance. While our solution does not support complex QoS constraints and the composition may not always be the optimal, we ensure low penalty on process performance.

Xiao et al. [41] present a framework for verifying SLA compliance of composition models at design time. The solution integrates a simulator engine in a composition environment and checks all execution paths of the composition. While the approach ensures that SLA requirements are met at design time, it does not take into consideration the dynamics of the services and QoS changes at runtime. Our focus is to ensure SLA fulfillment also in exceptional situations, taking into account dynamic service performance fluctuations.

Mei et al. [22] propose a solution that selects services based on their failure rates and popularity (how often they are used in compositions). The popularity of services is computed by applying link analysis on WSDL information extracted from public registries. The developer can select the highly ranked services for the compositions. The aim of the proposed solution is to reduce the number of service failures experienced by the consumers, while our aim is to ensure the process' SLA fulfillment.

A flexible approach allowing for integration of user preferences in the computation of QoS parameters that affect service selection is the LCP-nets framework [11,12]. When computing the QoS of the process the framework takes into consideration user preferences, relative importance, and tradeoffs between QoS parameters, which are expressed in linguistic terms using LCP-nets. It then compares the candidate services upon several different QoS dimensions, applying the expressed preferences to the currently measured QoS values. Our approach is targeted at individual QoS parameters, which allow for lightweight computational techniques, such as statistic tests.

Another solution using machine-learning techniques is presented in [20,38]. The solution proposes the use of regression models to predict the SLO values in the presence of unknown factors that affect the QoS parameters of the process. The QoS parameter values are recomputed at different points during the process execution, specified initially by the process developer. The solution allows for high accuracy of prediction and can react to changes that occur between the defined checkpoints, but it has to pay the price in computational expense and to rely on the developer's ability of instrumenting the process. In contrast, our solution offers complete transparency to the developer.

A different direction is taken by the Planning as Model Checking [33] approach, which allows to monitor the execution of the composition and take corrective actions in case some of the conditions are not met. The approach uses planning techniques as well as the EaGLe goal language [19], a language that allows expressing system goals such as non-functional requirements, i.e., QoS parameter requirements.

A different approach to deal with unsatisfactory QoS provided by services is SLA negotiation [13,14,27,28]. The negotiation can be manual, requiring human intervention, or automatic in which case software agents are used. Negotiation of SLA is done on a client basis, which means that a service can have a different SLA for each of its consumers. These solutions are based on agents that carry out negotiation, i.e., explore possible solutions that eventually lead to an agreement, using different algorithms and negotiation strategy models. While some approaches use optimization algorithms to speed up the negotiation process [28], finding an agreement may be a long-lasting process. In our solution, we deal with fixed SLAs that are not negotiable.

[3] takes a new view on contract (SLA) negotiation, considering the evolution of the contract based on the possible evolution of the parties involved in the contract. The solution provides a way of defining constraints on the contract, defining boundaries in which the provided service QoS can vary and what is acceptable both to provider and consumer. Therefore, a contract violation is more strictly defined in an evolving context and the need of re-negotiation is reduced.

There are different solutions that handle failure recovery of processes with the aid of dynamic binding of services. VieDAME [24] is a service monitoring and selection system that uses aspects to intercept SOAP messages and to dynamically replace services used in the BPEL process. Another aspect-based system is the *Dynamo* system [4,5]. *Dynamo* provides self-healing capabilities to the process with the aid of complex recovery strategies specified using two domain-specific languages (WSCoL, the *Web Service Constraint Language* and WSReL, the *Web Service Recovery Language*). Another approach to failure recovery is RobustBPEL2 [15,16] which makes use of a dynamic proxy to discover alternative services upon failure of services that have previously been marked for monitoring.

6 Conclusion

In this article, we described ADULA, a framework for automated maintenance of BPEL processes. ADULA automatically detects and repairs SLA violations caused by service performance degradation in a way transparent to the user and to the BPEL engine. Violation detection, diagnosis, and repair leverage a lightweight sampling monitoring technique and allow for customizable violation detection strategies and repair policies. ADULA ensures that the process evolution obtained through service replacement is traceable, providing the user with process and service statistics that can be used for further development. Our evaluation shows that ADULA maintains process performance ensuring SLA compliance with moderate cost in terms of overhead. Furthermore, the framework allows sharing of information about service performance between processes so that ADULA prevents SLA violations caused by degradation of service performance.

Our current implementation takes only service performance into account. Our future research activities include work on extending the monitoring and detection capabilities to domain-specific and functional QoS. When no replacement service with good performance is available, ADULA can detect and diagnose SLA violations, but it cannot prevent subsequent violations. To address this limitation, the framework capabilities need to be extended so as to include also repair actions at the service level, such as those presented in [17].

Acknowledgments We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "SOSOA: Self-Organizing Service-Oriented Architectures" (SNF Sinergia Project No. CRSI22_127386/1).

References

1. Active Endpoints. ActiveBPEL engine. http://www.activevos.com/
2. Anderson, T.W., Darling, D.A.: Asymptotic theory of certain "Goodness of Fit" criteria based on stochastic processes. Ann. Math. Stat. **23**(2), 193–212 (1952)
3. Andrikopoulos, V., Fugini, M., Papazoglou, M.P., Parkin, M., Pernici, B., Siadat, S.H.: Qos contract formation and evolution.

In: 11th International conference on electronic commerce and web technologies, pp. 119–130 (2010)

4. Baresi, L., Ghezzi, C., Guinea, S.: Towards self-healing composition of services. In: Contributions to Ubiquitous Computing, pp. 27–46. Springer Berlin, Heidelberg (2007)
5. Baresi, L., Guinea, S.: Dynamo and self-healing BPEL compositions. In: ICSE Companion, pp. 69–70 (2007)
6. Baresi, L., Guinea, S., Pasquale, L.: Integrated and composable supervision of BPEL processes. In: ICSOC '08: Proceedings of the 6th international conference on service-oriented computing, pp. 614–619 (2008)
7. Berger, J.: Statistical Decision Theory and Bayesian Analysis. Springer, Berlin (1999)
8. BPEL: BPEL 2.0 standard specification. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf
9. Canfora, G., Di Penta, M., Esposito, R., Perfetto, F., Villani, M.L.: Service composition (re)binding driven by application-specific QoS. In: ICSOC '06: Proceedings of the 4th international conference on service-oriented computing, pp. 141–152 (2006)
10. Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: A framework for QoS-Aware binding and re-binding of composite web services. J. Syst. Softw. **81**(10), 1754–1769 (2008)
11. Châtel, P., Malenfant, J., Truck, I.: Qos-based late-binding of service invocations in adaptive business processes. In: ICWS '10: Proceedings of the 2010 IEEE international conference on web services, pp. 227–234. IEEE Computer Society, Washington, DC, USA (2010)
12. Châtel, P., Truck, I., Malenfant, J.: Lcp-nets: a linguistic approach for non-functional preferences in a semantic SOA environment. J. Univ. Comput. Sci. **16**(1), 198–217 (2010)
13. Chhetri, M.B., Lin, J., Goh, S., Zhang, J.Y., Kowalczyk, R., Yan, J.: A coordinated architecture for the agent-based service level agreement negotiation of web service composition. In: ASWEC, pp. 90–99 (2006)
14. Comuzzi, M., Pernici, B.: An architecture for flexible web service QoS negotiation. In: EDOC, pp. 70–82 (2005)
15. Ezenwoye, O., Masoud Sadjadi, S.: RobustBPEL2: transparent autonomization in business processes through dynamic proxies. In: Proceedings of the 8th international symposium on autonomous decentralized systems (ISADS 2007), pp. 17–24. Sedona, Arizona, March (2007)
16. Ezenwoye, O., Sadjadi, S.M.: A proxy-based approach to enhancing the autonomic behavior in composite services. JNW **3**(5), 42–53 (2008)
17. Gmach, D., Krompass, S., Scholz, A., Wimmer, M., Kemper, A.: Adaptive quality of service management for enterprise services. ACM Trans. Web **2**(1), 1–46 (2008)
18. ISO/IEC 14764: Software engineering—software life cycle processes—maintenance. http://www.iso.org/iso/catalogue_detail.htm?csnumber=39064 (2006).
19. Lago, U.D., Pistore, M., Traverso, P.: Planning with a language for extended goals. In: The 16th AAAI conference on artificial intelligence, pp. 447–454 (2002)
20. Leitner, P., Michlmayr, A., Rosenberg, F., Dustdar, S.: Monitoring, prediction and prevention of SLA violations in composite services. In: ICWS '10: Proceedings of the 2010 IEEE international conference on web service, pp 369–376 (2010)
21. Mann, H., Whitney, D.: On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Stat. **18**(1), 50–60 (1947)
22. Mei, L., Chan, W.K., Tse, T.H.: An adaptive service selection approach to service composition. In: ICWS '08: Proceedings of the IEEE international conference on web services, pp. 70–77 (2008)
23. Meyn, S.P., Tweedie, R.L.: Markov Chains and Stochastic Stability. Springer, London (1993)
24. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for WS-BPEL. In: WWW '08: Proceeding of the 17th international conference on World Wide Web, pp. 815–824. ACM, New York, NY, USA (2008)
25. Mosincat, A., Binder, W.: Transparent runtime adaptability for BPEL processes. In: ICSOC '08: Proceedings of the 6th international conference on service-oriented computing, pp. 241–255 (2008)
26. Mosincat, A., Binder, W.: Automated performance maintenance for service compositions. In: WSE '09: The 11th IEEE international symposium on web systems evolution, pp. 131–140 (2009)
27. Ncho, A., Aïmeur, E.: Building a multi-agent system for automatic negotiation in web service applications. In: AAMAS '04: Proceedings of the third international joint conference on autonomous agents and multiagent systems, pp. 1466–1467. IEEE Computer Society, Washington, DC, USA (2004)
28. Nitto, E.D., Penta, M.D., Gambi, A., Ripa, G., Villani, M.L.: Negotiation of service level agreements: an architecture and a search-based approach. In: ICSOC '07: Proceedings of the 5th international conference on service-oriented computing, pp. 295–306 (2007)
29. O'Brien, L., Bass, L., Merson, P.: Quality attributes and service-oriented architectures. Technical Report CMU/SEI-2005-TN-014, CMU - Software Engineering Institute, Pittsburgh, PA, September (2005)
30. Open Grid Forum. WS-Agreement specification. http://www.ogf.org/documents/GFD.107.pdf
31. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: state of the art and research challenges. IEEE Comput. **40**(11), 38–45 (2007)
32. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: a research roadmap. Int. J. Co-Op. Inf. Syst. **17**(2), 223–255 (2008)
33. Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., Traverso, P.: Planning and monitoring web service composition. In: The 8th international conference on artificial intelligence: methodology, systems, applications, pp. 106–115 (2004)
34. Romano, J.: Testing Statistical Hypotheses. Springer, Berlin (2005)
35. Skene, J., Lamanna, D.D., Emmerich, W.: Precise service level agreements. In: Proceedings of ICSE '04, pp. 179–188. IEEE Computer Society (2004)
36. SOAP. SOAP specification. http://www.w3.org/TR/soap12-part1/
37. Swanson, E.B.: The dimensions of maintenance. In: ICSE '76: Proceedings of the 2nd international conference on software engineering, pp. 492–497. IEEE Computer Society Press, Los Alamitos, CA, USA, (1976)
38. Wetzstein, B., Leitner, P., Rosenberg, F., Brandic, I., Dustdar, S., Leymann, F.: Monitoring and analyzing influential factors of business process performance. In: EDOC, pp. 141–150 (2009)
39. Wilcoxon, F.: Individual comparisons by ranking methods. Biometrics Bull. **1**, 80–83 (1945)
40. WS-Addressing. WS-Addressing standard specification. http://www.w3.org/Submission/ws-addressing/
41. Xiao, H., Chan, B., Zou, Y., Benayon, J.W., O'Farrell, B., Litani, E., Hawkins, J.: A framework for verifying SLA compliance in composed services. In: ICWS '08: Proceedings of the IEEE international conference on web services, pp. 457–464 (2008)
42. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware middleware for web services composition. IEEE Trans. Softw. Eng. **30**(5), 311–327 (2004)