Replacement Policies for Service-Based Systems

Khaled Mahbub and Andrea Zisman

Department of Computing, City University London, Northampton Square,
London EC1V 0HB, UK
{k.mahbub,a.zisman}@soi.city.ac.uk

Abstract. The need to change service-based systems during their execution time has been recognized as an important challenge in service oriented computing. There are several situations that may trigger changes in service-based systems such as unavailability or malfunctioning of services; changes in the functional, quality, or contextual characteristics of the services; changes in the context of the service-based system environment; emergence of new services; or changes in the requirements of the system. However, in order to support dynamic changes in service-based systems, it is necessary to have *replacement policies* describing *what* needs to be changed, and *how* and *when* the changes should be executed. In this paper, we describe replacement policies to support dynamic changes in service-based systems. These replacement policies are used in our service discovery framework that supports proactive identification of services in parallel to the execution of the system. A prototype tool has been implemented in order to illustrate and evaluate the framework. The results of some initial evaluation are also described in the paper.

Keywords: service discovery, replacement policies, service adaptation, queries.

1 Introduction

Dynamic changes in service-based systems are considered a major research challenge for service oriented computing [18][19]. There are several situations that may trigger the need to change service-based systems during execution time. Examples of these situations are: (i) changes in the context of the service-based system environment or their participating services, (ii) changes in functional and quality aspects of services participating in service-based systems, (iii) failures in or unavailability of services participating in service-based systems, (iv) emergence of new services, or even (v) changes in or emergence of new requirements. To deal with the above situations, service-based systems need to be (a) *self-configuring*, systems that are able to automatically identify, select, and combine new services with which to interact; (b) *self-optimizing*, systems that can select the best services with which to interact in order to become more efficient; and (c) *self-healing*, systems that can detect and react to violations of functional and quality requirements, failure and unavailability of services, or changes in its context environment.

In order to provide support for dynamic changes in service-based systems, it is necessary to use *replacement policies* specifying what needs to be changed (*what*), the

A. Dan, F. Gittler, and F. Toumani (Eds.): ICSOC/ServiceWave 2009, LNCS 6275, pp. 345–357, 2010.

ways that the changes need to be executed (*how*), and the moment that the changes should be performed in the systems (*when*).

Changes in a service-based system can range from the replacement of a service by another service, or a composition of services, to changes in the execution process (e.g., conditions, loop statements, variables, exception handlers). The changes in a service-based system can be performed by stopping the system, making the necessary changes, and resuming the system [2]. Other approaches can be used when replacing a service by another service in a system such as binding partner links during execution time of the system [12]; using proxy services as place holders for the services in a composition, instead of having concrete services referenced in the system [3][4][14]; or even using an adaptation layer based on aspect oriented programming with information about alternative services [17].

Furthermore, the moment to execute changes in a service-based system should also be considered in order to avoid (1) making changes when it is not really necessary, or (2) making changes that may cause the system to behave incorrectly. As an example of case (1), consider the situation when a service S1 is replaced by a service S2, but the execution process of the system never accesses the execution path that contains S1. For an example of case (2), consider the scenario in which a service-based system currently uses a direct debit service (Sa) to assist with payments of purchased items and a new service that supports credit card payment (Sb) becomes available when the system is debiting a user's bank account. In this case, it may be better to wait to replace service Sa with service Sb, instead of risking charging the user twice. However, if the direct debit service becomes unavailable it needs to be replaced so that the system can continue its operation.

Replacement policies should take into consideration (a) the situations that trigger changes in the system (e.g., situations (i) to (v) above), (b) the type of changes that needs to be performed in the system, and (c) if the parts in the system that require changes are being used. Recently, few approaches have been proposed to support adaptation of service-based systems [2][3][4][6][11]. Adaptation in these approaches consists of replacing services in service-based systems with alternative services, or composition of services. However, these approaches do not consider the problem that triggers the need for changes, and when and how to execute the changes.

In this paper, we describe different types of replacement policies for situations (i) to (v) above. In our work, changes in a service-based system consist of replacing a service participating in the system by another service. We use our proactive service discovery framework that allows for the identification of replacement services in parallel to the execution of the system due to cases (i) to (v) above [24]. We extend the framework to allow the use of proxy services to support changes in the system during execution time, avoiding changes in the original service-based system. In the framework we consider cases when (1) changes are required to be performed so that the system can continue its operations; (2) changes that can wait to be performed after the current execution of the system; and (3) no changes are required. A prototype tool incorporating the replacement policies and the deployment of the changes in service-based systems using proxy services has been implemented in order to illustrate and evaluate the work. Initial evaluations of the work comparing the performance of executing a service-based system without the need for changes with the performance to

execute the system when changes are required using the replacement policies being described is also presented.

The remaining of this paper is organized as follows. In Section 2, we provide an overview of our proactive service discovery framework with its extensions. In Section 3, we describe the replacement policies that we use to support changes in the system. In Section 4, we discuss implementation aspects and results of some initial evaluation of the work. In Section 5, we present related work. Finally, in Section 6, we present some conclusions and discuss future work.

2 Overview of Service Discovery Framework

Our service discovery framework allows a proactive identification of services to replace services participating in service-based systems during execution time of these systems. The discovery framework supports the identification of services based on structural, behavioral, quality, and contextual criteria represented as complex queries. The queries are specified in an XML-based query language named SerDiQueL [25][1]. The framework assumes services in registries specified as multi-faceted descriptions as proposed in the SeCSE project [20]. Figure 1 presents the architecture of our framework with its main components, namely: (i) service requestor, (ii) service matchmaker, (iii) service listener, (iv) service context server, (v) system context server, (vi) proxy server, and (vii) service registry intermediary.

The *service requestor* orchestrates the functionality offered by the other components in the framework. It (a) receives a service request from a service-based system and context information about the services and system environment, (b) prepares service queries to be evaluated, (c) organises the results of a query and returns these results to a service-based system, (d) manages subscriptions of queries and services, (e) receives information from listeners about services that become available or changes to existing services, and (f) invokes the service matchmaker to execute a query and the proxy server to execute replacement policies.

The *service matchmaker* is responsible to parse the different criteria of a query and evaluate these criteria against service specifications in the various service registries. The different criteria in a query are: (1) *structural*, describing the interface of a required service; (2) *behavioral*, describing the functionality of a required service; and (3) *constraints*, describing additional conditions of a service to be discovered. The constraints in a query can be *contextual* or *non-contextual*. A contextual constraint is concerned with information that changes dynamically during the operation of the service-based system and/or the services that the system deploys; while a non-contextual constraint is concerned with quality aspects of a required service. The non-contextual constraints can be *hard* or *soft*. A hard constraint must be satisfied by all discovered services for a query while the soft constraints do not need to be satisfied by all discovered services, but are used to rank candidate services for a query.

[1] Due to space limitation we do not describe SerDiQueL in this paper. Detailed information about SerDiQueL can be found in [25].

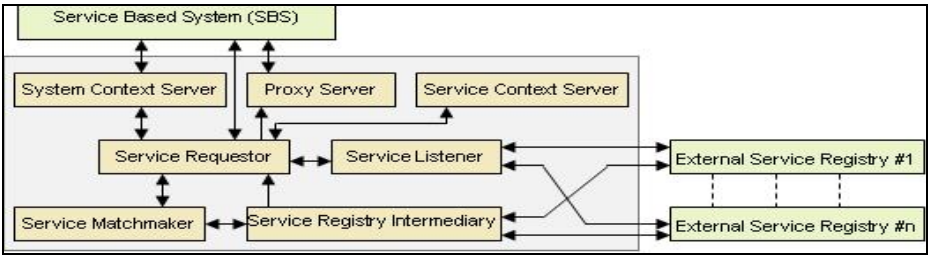

Fig. 1. Architecture overview of service discovery framework

The *service context server* and the *system context server* send updates to the service requester whenever changes of context occur in the services or system.

The *service listener* polls the external service registries at regular interval and notifies the service requester about a new service that becomes available or about changes in the characteristics of a service.

The *proxy server* supports replacement of services in service-based systems. It receives calls from the service-based system when a participating service needs to be invoked and events from the service requestor when a service needs to be replaced. The proxy server keeps track of the execution of a service-based system and, when necessary, replaces a service by using the replacement policies.

The *service registry intermediary* supports the discovery of services stored in various registries by providing an interface to access services from these registries. The framework supports services from registries that are based on a faceted structure as developed in the SeCSE project [20]. In this structure, a service is specified by a set of facets representing its different aspects such as (i) structural facets describing the operations of a service with their data types in WSDL [23], (ii) behavioral facets describing behavioral models of services in BPEL [5], (iii) quality of service facets describing quality aspects of services represented in XML-based schema, and (iv) context facets describing the types of context information for a service represented in XML-based ontologies.

In the framework, the evaluation of a query against services in the registries is based on the computation of distances. More specifically, a query is executed in a two-stage process. In the first stage (viz. *filtering stage*), hard constraints of a query are matched against services in the registries returning a set of candidate services that are compliant with these constraints. In the second stage (viz. *ranking stage*), overall distances are computed between the query and services returned by the first stage in the process.

The overall distance between a query Q and a service S is denoted as d(Q,S) and is computed by considering the average of three partial distances, namely *structural_behavioral,* soft_noncontextual, and contextual. The structural evaluation is based on the matching of the signatures of the operations in a query and services by comparing graphs of the data types of the parameters of the operations and the linguistic distances of the names of the operations and parameters. The behavioural evaluation is based on the comparison of paths representing the behavioral criteria in the query and paths extracted from state machines representing behavioral specifications of services. The *soft_noncontextual* and *contextual* partial distances are computed by evaluating

constraints in a query against service specifications. This evaluation compares context information provided by context services to context conditions in a query. Details of the computation of the overall and partial distances are described in [24].

The framework supports the execution of queries in pull and push modes. The pull mode is executed to identify services that are initially bound to a service-based system, as a first step in the push mode of query execution (to identify an initial set of candidate services), or when a client application requests a service to be discovered. The push mode is executed when the system is running and a service needs to be replaced due to cases (i) to (v) as described in Section 1. The push mode of query execution requires the system environment, the services participating in the system, and the queries associated with these services to be subscribed. In this case, candidate services for subscribed services and queries are identified in parallel to the execution of the system in a proactive way. These candidate services are maintained in an up-to-date set in which services are organized in ascending order of distances with the query. The algorithms for identifying services in a proactive way due to the different cases are describes in details in [24]. When notifications about changes in the subscribed services or system environment are pushed to the listeners, the first service from the set of candidate services is selected and may be replaced in the service-based system depending on the replacement policy.

3 Replacement Policies

The replacement policies used in our work specify when participating services should be replaced in service-based systems due to (a) unavailability or malfunctioning of services; (b) changes in the structural, functional, quality, or context of services; (c) availability of a new service; and (d) changes in the context of the system's environment, changes in requirements, or emergence of new requirements[2].

The replacement policies take into consideration the position of a service S that may need to be replaced with respect to the current execution point of the system.

There are three different positions that are considered, namely:

- *not_in_path*: when service S in not in the current execution path of the system, i.e., S appears in a different branch of the system's execution path or before the current point in the execution path;
- *current*: when service S is in the current execution point of the system;
- *next_in_path*: when service S is in the current execution path of the system, and will be invoked some time in the future.

We describe below the replacement policies for each case (a) to (d) above. The replacement policies will be executed based on events concerned with these cases.

In order to follow the description of the policies consider P the process of the service-based system being executed; S_P a subscribed service being used in P that may need to be replaced; Q a subscribed query associated with S_P that is composed of structural,

[2] Please note that in reference to the situations described in Section 1, case (a) is concerned with situation (iii), case (b) is concerned with situations (i) and (ii), case (c) is concerned with situation (iv), and case (d) is concerned with situations (i) and (v).

behavioural, quality or contextual constraints; d(Q,S) the distance between a service S and a query Q, as introduced in Section 2; dmax(Q) the threshold of acceptable distance values between services and query Q; Set_S the set of subscribed candidate services for Q, including Sp, ranked in ascending order of the distances between the services and query Q; $d_{delta}(Q)$ the threshold used to decide about the replacement of S_p in P in different situations, as explained below; *pos(S)* a function that returns the position of service S in process P. The candidate services S_i in Set_S are services that match the criteria of query Q and have distances with Q that is less or equal to the threshold distance ((d(Q,Si)) <= dmax(Q)). All distance values are within [0, 1]. The returned values of *pos(S)* can be *not_in_path, current,* or *next_in_path*, as explained above. *mark(S,P,CURRENT)* is a function that marks S for replacement when S is accessed in the current execution of P; *mark(S,P,FUTURE)* is a function that marks S for replacement when S is accessed in a future execution of P; *mark_or_replace(S1, S2, P)* is a function that replaces a service or marks a service for replacement in P. The description of *mark_or_replace* function is shown in Figure 2.

```
mark_or_replace(S1, S2, P)
  If pos(S1) == not_in_path then  mark(S1, P, FUTURE)
  If pos(S1) == current then  replace S1 by S2;
  If pos(S1) == next_in_path then mark(S1, P, CURRENT)
```

Fig. 2. mark_or_replace function

Case (a): A subscribed service S becomes malfunctioning or unavailable. In this case, S can be either a service participating in process P (S == S_p) or a service in the set of candidate services for S_p that is not being used in the process. In any case, S needs to be removed from the set of candidate services and unsubscribed.

```
E(unavailable/malfunctioning, Q, S)
If S is being used in P then //S == Sp
 If S_Set is empty then       // There are no available candidate services to replace SP
   If pos(Sp) ==not_in_path then  mark(Sp, P, FUTURE)
   If pos(Sp) == current then
       If exist_exception(P) then  execute exception handler in P;
       else throw exception(unavailable/malfunctioning,S) ;
   If pos(Sp) == next_in_path then mark(Sp, P, CURRENT);
 If S_Set is not empty then mark_or_replace(Sp, S0, P);
```

Fig. 3. Replacement policy for unavailable or malfunctioning services

Figure 3 shows the replacement policy for case (a). As shown in the Figure, in the case that S is a service in process P, it may be necessary to replace S by another candidate service from Set_S, if the set of candidate services is not empty (i.e., service S that was removed from Set_S was the only service in the set). The replacement of service S by the first service in Set_S (the one with the smallest distance), will be performed when S is currently being executed by process P (i.e., *pos(S) == current*). However, when service S is not in the current execution path (i.e., *pos(S) == not_in_path*) or S is in the current execution path, but should only be invoked in the

near future (i.e., *pos(S) == next_in_path*), S is marked to be replaced. S will be replaced by a service in Set_S, through the proxy server, when the execution process reaches the point of S in P and Set_S is not empty at this stage.

In the situation that Set_S is empty and S is currently being executed by process P, an exception will be thrown. This exception can be either from process P, when such exception exists in P, or an exception specified in the replacement policy. In the case that S is not in the current execution path, or S is in the current execution path but should only be invoked in the future, S is marked to be replaced and an attempt to replace S will be executed in the future, when the process reaches the respective point of execution. In the situation in which S is not being used in P, nothing needs to be done with respect to replacement policy.

Case (b): A subscribed service S has its structural, behavioral, quality, or contextual characteristics changed. As in case (a), S can be either a service participating in process P ($S == S_p$) or a service in the set of candidate services for Sp that is not being used in the process. A new distance between S and Q needs to be calculated and if this distance is below the threshold, Set_S is re-ordered. Otherwise, S is removed from Set_S and unsubscribed.

Figure 4 shows the replacement policy for case (b). As shown in the figure, when S is a service in process P, but the set of candidate services is empty (i.e., S was the only service available for Q, but changes in its characteristics caused it not to be suitable anymore), S is marked to be replaced when the execution process reaches the point of S in the process, for the situations in which S is neither in the execution path nor being currently executed. Otherwise, an exception is thrown.

When the set of candidate services is not empty, it is necessary to verify if S is still the best service to be used in the process. In positive case, nothing needs to be done. However, when S is not the best service anymore, it is necessary to verify how bad it will be to continue using S, instead of replacing S by the best service (i.e., service S_0).

```
E(change,Q,S):
If S is being used in P then // S == Sp
  If Set_S is empty then     // There are no available candidate services to replace SP
    If pos(Sp) == not_in_path then mark(Sp, P, FUTURE)
    If pos(Sp) == current then
          If exist_exception(P) then execute exception handler in P;
          else throw exception(no available candidate service for S) ;
    If pos(Sp) == next_in_path then mark(Sp, P, CURRENT)
  If Set_S is not empty then
       If S0==S then do nothing; // despite the changes, S is still the best service
       else
         If d(Q, S) – d(Q, S0) <= d_delta(Q) then mark(Sp, P, FUTURE)
         else //d(Q, S) – d(Q, S0) > d_delta(Q)
              mark_or_replace(Sp, S0, P)
If S is not being used in P then // S != Sp
    If S0 != S then do nothing; //after the changes, S is not the best service
    If S0 == S then //S is now best service
      If d(Q, Sp) – d(Q, S0) <= d_delta(Q) then mark(Sp, P, FUTURE);
      If d(Q, Sp) – d(Q, S0) > d_delta(Q) then mark_or_replace(Sp, S0, P);
```

Fig. 4. Replacement policy for changes in the structural, behavioral, quality, or contextual characteristics of a service

This verification is done by calculating the value of the difference of the distance between S and query Q and the distance between the current best service in Set_S (S_0) and query Q, and verifying if this value is acceptable (i.e., if the value is less or equal to $d_{delta}(Q)$). If this is the case, S should not be replaced, but it should be marked to be replaced in the future. If the difference of the distance is not acceptable (i.e., $d(Q,S) - d(Q,S_0) > d_{delta}(Q)$), S should be replaced by the current best service S_0, when S is in the current point of process execution. Otherwise, S should be marked to be replaced in the future.

In the case that S is not being used by process P (S is a service in Set_S), it is possible that S is still not the best service and, therefore, nothing should be changed. However, if S becomes the best service in Set_S, it is necessary to verify the difference of the distances of the current service in the process related to Q and service S. When this difference is acceptable, the current service in process P (S_P) is marked for replacement in a future execution of P. But, when the difference is beyond the acceptable threshold ($d_{delta}(Q)$), S_P needs to be replaced by S, when it is currently being executed in P. Otherwise, S_P is marked for future replacement.

Case (c): A new service S becomes available. This case is concerned with the situation in which a new service S is provided and inserted in the service registry or an existing service in the registry, that is not part of a set of candidate services for subscribed services and queries, has been modified. The distance between service S and query Q is calculated and if the distance is above the threshold, S is not considered as a candidate service for Q and nothing needs to be done. When the distance is below the threshold, S is added into Set_S.

E(new,S):
If S is in Set_S **then** //the new service is a candidate service
  If S_0 != S **then** do nothing; // the new service S is **not** the best service
  If S_0 == S //the new service S is the best service
    If $d(Q, S_p) - d(Q, S_0) <= d_{delta}(Q)$ **then mark**(S_p, P, FUTURE);
    If $d(Q, S_p) - d(Q, S_0) > d_{delta}(Q)$ **then mark_or_replace**(S_p, S_0, P);
If S is not in Set_S **then** do nothing; //the new service is not a candidate service

Fig. 5. Replacement policy for a new service that becomes available

Figure 5 shows the replacement policy for this situation. In the case that S was included in Set_S, but is not the best service for Q, nothing needs to be done. However, when S is the best match for query Q ($S == S_0$), it is necessary to verify the difference of the distances of the current service in the process (S_P) and service S. When this difference is acceptable, the current service in process P (S_P) is marked for replacement in a future execution of P. But, when the difference of the distances is beyond the acceptable threshold ($d_{delta}(Q)$), S_P needs to be replaced by S, when S_p is in the current execution point of P. Otherwise, S_P is marked to be replaced in the future.

Case (d): Changes in the context of the system environment, changes in the requirements, or emergence of new requirements. In this case there is a change in the criteria of query Q and a new query Q' is created. Therefore, it is not always the case

```
E(constraint,Q,Q'):
If Set'_S is not empty then
  If Sp is in Set'_S then //Sp is a candidate service for Q'
    If Sp == S0 then do nothing;
    else
      If d(Q', Sp) – d(Q', S0) <= d_delta(Q') then mark(Sp, P, FUTURE)
      If d(Q', Sp) – d(Q', S0) > d_delta(Q') then mark_or_replace(Sp, S0, P)
  If Sp is not in Set'_S then mark_or_replace(Sp, S0, P) //Sp is not a candidate service for Q'
If Set'_S is empty then //There are no available candidate services for Q'
  If pos(Sp) == not_in_path then mark(Sp, P, FUTURE)
  If pos(Sp) == current then
    If exist_exception(P) then execute exception handler in P;
    else throw exception(no available candidate service for S) ;
  If pos(Sp) == next_in_path then mark(Sp, P, CURRENT);
```

Fig. 6. Replacement policy when there are changes in the context of the system environment, changes in the requirements, or new requirements

that the current set of candidate services for query Q are still candidate services for Q', and that the subscribed service used in process P associated with Q is a candidate service for Q'. A new set of candidate services for Q' needs to be created (Set'_S), and the distance between S_P and Q' is calculated.

Figure 6 shows the replacement policy for this situation. As shown in the figure, when there are available candidate services for Q' (Set'_S is not empty) and S_P is still the best candidate service for Q', nothing needs to be done. However, when S_P is a candidate service for Q', but not the best service, and the difference in the distances of S_P and the best candidate service for Q' is acceptable, then S_P is not replaced, but it is marked to be replaced in a future execution of process P. When the difference in the distances of S_P and the best candidate service for Q' is not acceptable, then S_P is replaced by the best candidate service when S_P is in the current execution point. Otherwise, S_P is marked to be replaced in the near future. In the situation in which S_P is not a candidate service for Q', or there are no candidate services for Q' (Set_S is empty), either an exception is thrown or S_P is marked to be replaced in the future, when accessed during the execution of the process.

4 Implementation Aspects and Evaluation

A prototype tool of the framework has been implemented in Java. The tool is available as a web service and can be deployed by any client that can produce service requests in the format required by the framework. The subscription of the services is supported by WS-Eventing [22] and by an event receiver. The external service registry uses eXist [8] database. Communication with the registry is through the use of Remote Method Invocation (RMI). The proxy server has been implemented as an HTTP server using Java socket programming.

The work was initially evaluated to measure the delay in the execution process that may be caused when using the approach. More specifically, we measure the times to execute a service-based system without the need for changes and compare this value with the times to execute the system when changes are required using our replacement

policies. The times were calculated as the average of 60 executions using a Pentium 2.33 GHz with 3.23 GB RAM machine.

In the experiment we have used a *Route-Planner* service-based system specified as a BPEL [5] process that allows users to request information from a PDA about optimal routes to be taken when driving. More specifically the system offers services that (i) identify the exact current location of a user (S_Loc), (ii) allow users to find an optimal route for a certain location given the exact location of the user by using a *Global Positioning Service* (S_GPS), (iii) display colored electronic maps of the area where the user is located and the route to be taken supported by the use of e-AZ Map service (S_e-AZ), (iv) provide traffic information in the area where the user is located and in the route that the user is supposed to take to get to his/her destination by using *Road Traffic Service* (S_RT), and (v) compute new routes at regular intervals due to traffic changes (S_Route).

The query used in the experiment was specified in SerDiQueL [30] and is concerned with the identification of candidate services to replace the Global Positioning Service (S_GPS) in the system. This query has structural, behavioral, and soft quality constraints. The structural constraint is concerned with the interface of the S_GPS service, the behavioral constraint is concerned with the existence of a certain operation (e.g., get_location()), and the quality constraint specifies that the service should be available 24 hours per day.

We have executed the query for situations when (a) service S_GPS becomes unavailable; (b) there is a change in service S_GPS and this service is available only 12 hours per day, instead of 24 hours; and (c) a new better service S_GPS' becomes available. For case (c), we consider the scenario in which S_GPS initially used in the system was available only 12 hours and the new service (S'GPS) is available 24 hours. We used a distance threshold ($d_{max}(Q)$) in the experiment such that there is always an average of four services in the set of candidate services, and a threshold for replacement condition ($d_{delta}(Q)$) of value zero.

Table 1. Results of average response time in seconds

No required changes	Service unavailable	Change in service	New service
0.48	0.56	0.52	0.54

Table 1 presents the times in seconds for the experiment. The results show that although there is an increase in the average response time when using our replacement policies to support changes in the service-based system, this value is small when compared to the ideal situation when no changes are necessary to be executed in the system. Moreover, the results also demonstrate that the small penalty regarding the use of the policies and changes in the system is very similar for all the cases considered in the experiment and changes in the system is very similar for all the cases considered in the experiment, due to the similarity of the policies for each case.

5 Related Work

Several approaches have been proposed to support service discovery and adaptation of service-based systems. We present below some of these approaches.

Approaches for service discovery can be based on semantic matchmaking and use logic reasoning over terminological concept relations defined by ontologies [1][13]. Other approaches specify requests and services using graph transformation rules [10]. The approach in [13] focuses on operation signature checking based on string matching, but it is limited since it cannot account for changes in the order or names of the parameters. The approach in [9] advocates the use of (abstract) behavioural models of services to increase the precision of the discovery process. In [7], context information is represented by key-value pairs attached to the edges of a graph representing service classifications. This approach does not integrate context information with behavioural and quality matching and, context information is stored explicitly in a service repository that must be updated following context changes.

Overall, most of the proposed approaches support service discovery for only specific types of service criteria. Unlike them, our framework supports dynamic service discovery based on a comprehensive set of service and application properties including structural, functional, quality, and contextual properties. It also provides proactive service discovery mechanisms, optimising service replacement during the execution of an application.

In [15][16] mechanisms are presented for policy based adaptation of networks. In these approaches, reconfiguration of hardware (e.g. alter the queuing strategy in a router, increase the buffer size) or software (e.g. restrict unauthorized access, enable different encoding) components is suggested based on a set of policies to optimise the performance of the network. These approaches may require execution of alternate workflow depending on the adaptation policy, whereas in our approach we ensure the execution of the same workflow by amending the workflow.

Recently, a few approaches that support adaptation of service-based systems have started to appear. The dynamic binding approach described in [4][6] provides binding and reconfiguration rules to support evolution of service compositions during runtime.

The works in [2][3] propose approaches towards self-healing for services compositions based on monitoring and recovery actions. In [2], the recovery actions involve *retry* of the process task, *redo* of the process task, *substitute* the service by a candidate service, or *compensate* an executed task by a compensation action. Contrary to our work, in this approach, when a fault is detected, the process is suspended and moved to a repair mode. When the recovery actions are completed, the process is resumed.

The VieDAME framework [17] uses an aspect-oriented approach to allow adaptation of service-based systems for certain QoS criteria based on various alternative services. In the framework, a service participating in the system can be marked as replaceable to indicate that alternative services can be invoked instead of the original one, when necessary.

In [10], the authors propose PROSA, a proactive adaptation approach for service-based systems based on online testing. Although, the focus of this paper is on how to detect the need for changes in service-based systems before they occur, and not how to modify the system and when services should be replaced, this work is interesting and we intend to extend our framework to support proactive detection of necessary changes in service-based systems together with the proactive identification of candidate services to replace participating services.

Our framework complements existing approaches for adaptation of service-based systems. Contrary to existing approaches, our framework provides replacement policies

for different situations that may require changes in the system. These policies avoid replacing services unnecessarily. In addition, the services to be replaced are identified in parallel to the execution of the system in a proactive way.

6 Conclusion and Future Work

In this paper we presented replacement policies to support changes in service-based systems due to different situations such as (i) changes in functional and quality aspects of services participating in service-based systems, (ii) failures in or unavailability of services participating in service-based systems, (iii) emergence of new services, (iv) changes in the context of the service-based system environment or their participating services, or even (v) changes in or emergence of new requirements. The replacement policies consider the cases in which changes need to be performed so that the system can continue its operations; changes can wait to be performed after the current execution of the system; and no changes are required. These policies have been used in a service-discovery framework that we have developed in order to support proactive identification of services in parallel to the execution of the service-based system, in terms of structural, behavioral, quality, and contextual characteristics. A prototype tool has been implemented in order to illustrate and evaluate the work. Initial experiment of the work has shown that the use of the replacement policies does not cause an overhead in the performance when compared to the execution of a service-based system that does not require changes to be performed.

We are currently extending the replacement policies to support cases in which changes to the service-based system may be concerned with modifications of the execution process or to the replacement of a service by a composition of services. We are also executing more experimentation of the work to compare the time necessary to use the policies when changes in the system are executed by instrumentation of the BPEL [5] engine and by changing the code of the system.

Acknowledgement

The research leading to these results has received funding from (a) the European Community's Seventh Framework Programme [FP7/2007-2013] under Grant Agreement 215483 (S-Cube) and (b) the European Commission under the IST Programme as part of the project GREDIA (contract FP6-34363).

References

1. Aggarwal, R., Verma, K., Miller, J., Milnor, W.: Constraint Driven Web Service Composition in METEOR-S. In: International Conference on Services Computing (2004)
2. Ardagna, D., Comuzzi, M., Mussi, E., Pernici, B., Plebani, P.: PAWS: A Framework for Executing Adaptive Web-Service Processes. IEEE Software 24(6) (2007)
3. Baresi, L., Ghezzi, C., Guinea, S.: Towards Self-Healing Compositions of Services. Studies in Computational Intelligence, vol. 42. Springer, Heidelberg (2007)
4. Baresi, L., Di Nitto, E., Ghezzi, C., Guinea, S.: A Framework for the Deployment of Adaptable Web Service Compositions. Service Oriented Computing and Applications 1(1) (April 6, 2007)

5. BPEL4WS, http://www128.ibm.com/developerworks/library/specification/ws-bpel/
6. Colombo, M., Di Nitto, E., Mauri, M.: SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined through Rules. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 191–202. Springer, Heidelberg (2006)
7. Doulkeridis, C., Loutas, N., Vazirgiannis, M.: A System Architecture for Context-Aware Service Discovery. Electr. Notes Theor. Comput. Sci. 146(1), 101–116 (2006)
8. eXist, http://exist.sourceforge.net
9. Hall, R.J., Zisman, A.: Behavioral Models as Service Descriptions. In: International Conference on Service Oriented Computing, ICSOC, New York (2004)
10. Hausmann, J.H., Heckel, R., Lohman, M.: Model-based Discovery of Web Services. In: International Conference on Web Services (2004)
11. Hielscher, J., Kazhamiakin, R., Metzger, A., Pistore, M.: A Framework for Proactive Self-Adaptation of Service-based Applications Based on Online Testing. In: Mähönen, P., Pohl, K., Priol, T. (eds.) ServiceWave 2008. LNCS, vol. 5377, pp. 122–133. Springer, Heidelberg (2008)
12. Horrocks, I., Patel-Schneider, P.F., van Harmelen, F.: From SHIQ and RDF to OWL: The Making of A Web Ontology Language. Journal of Web Semantics 1(1), 7–26 (2003)
13. Keller, U., Lara, R., Lausen, H., Polleres, A., Fensel, D.: Automatic Location of Services. In: Gómez-Pérez, A., Euzenat, J. (eds.) ESWC 2005. LNCS, vol. 3532, pp. 1–16. Springer, Heidelberg (2005)
14. Kim, J., Lee, J., Lee, B.: Runtime Service Discovery and Reconfiguration using OWL-S based Semantic Web Service. In: Proceedings of the 7th IEEE International Conference on Computer and Information Technology (2007)
15. Lymberopoulos, L., Lupu, E., Sloman, M.: An Adaptive Policy-Based Framework for Network Services Management. Journal of Network and Systems Management 11(3) (September 2003)
16. Marshall, A., Hussain, S.A., Chieng, D., Gu, Q.: Dynamic Network Adaptation Techniques In An Open Network Environment. In: Intl. Conference on IT and Communications (AIT 2000), Bangkok, Thailand (August 2000)
17. Moser, O., Rosenberg, F., Dustdar, S.: Non-Intrusive Monitoring and Service Adaptation for WS-BPEL. In: 17th Int. World Wide Web Conference, WWW, China (April 2008)
18. Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M., Pohl., K.: A Journey to Highly Dynamic, Self-Adaptive, Service-based Applications. ASE Journal 15 (2008)
19. Papazoglou, M.P., Traverso, P., Dustdar, S., Leyman, F., Kramer, B.: Service-Oriented Computing Research Roadmap, ftp://ftp.cordis.lu/pub/ist/docs/directorate_d/st-ds/services-research-roadmap_en.pdf
20. SECSE Project, http://secse.eng.it
21. Subramanian, V., Gilberti, M., Doboli, A.: Online adaptation policy design for grid sensor networks with reconfigurable embedded nodes. In: Design, Automation & Test in Europe Conference & Exhibition, DATE 2009 (2009)
22. WS-Eventing, http://www/w3/org/Submission/WS-Eventing
23. WSDL, http://www.w3.org/TR/wsdl
24. Zisman, A., Spanoudakis, Dooley, J.: A Framework for Dynamic Service Discovery. In: IEEE Int. Conference on Automated Software Engineering, ASE, Italy (September 2008)
25. Zisman, A., Spanoudakis, Dooley, J.: A Query Language for Service Discovery. In: 4th Int. Conference on Software and Data Technologies, ICSOFT, Bulgaria (July 2009)